Team Report

Introduction to OOAD using UML Tools

과 목 소프트웨어 공학 개론

담당교수 유 준 범 교수님

제 출 일 2010. 10. 25

학 과 컴퓨터공학부

4 학 년 200714170 모진종

200714172 배보람

200511350 장범석

200511311 김진수

목차

1. OOAD

(1) Analysis and Design

OOAD (Object Oriented Analysis and Design) 에 설명하기에 앞서, Analysis 와 Design 에 대해 먼저 설명하자면, Analysis는 Investigation of Problem. 여기서 말하는 Problem 이란 풀어나갈 과제에 대한 것으로 즉 Analysis는 문제를 정의하는 것이라고 할 수 있습니다. Design이란 앞서 Analysis에서 정의한 문제에 대한 Logical Solution으로 Source Code를 생성하는 것이 Design의 목표라고 할 수 있습니다.

Analysis는 Requirement Analysis를 포함하는데 이는 고객과 System 간의 합의점을 찾는 것이라고 할 수 있으며 functional requirement 뿐만 아니라 non-functional requirement 역시 고려해야 합니다. 또한 Analysis 과정에서는 problem domain에 대한 model 을 생성해야 하는데, 이 때에는 고객과의 대화뿐만 아니라 현실 세계에 대한 지식이 반드시 수반되어야 합니다.

Design은 Analysis 에서 생성했던 Application Concept 을 Computer Concept로 변경하고, System의 전반적인 Architecture를 설계하는 등의 역할을 합니다.

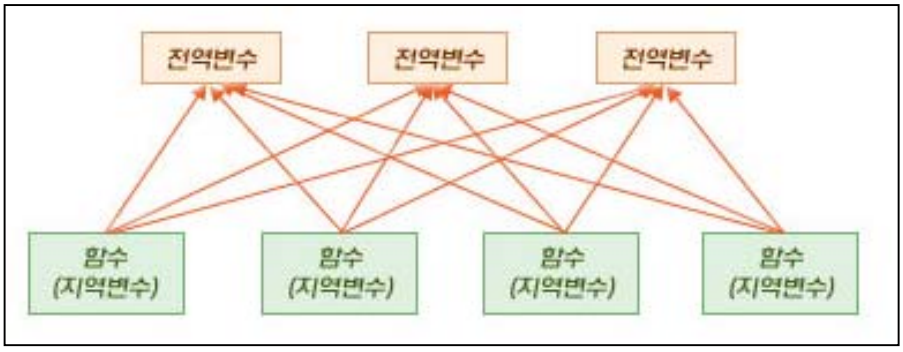
(2) Object Oriented Program

1) Object Oriented Approach

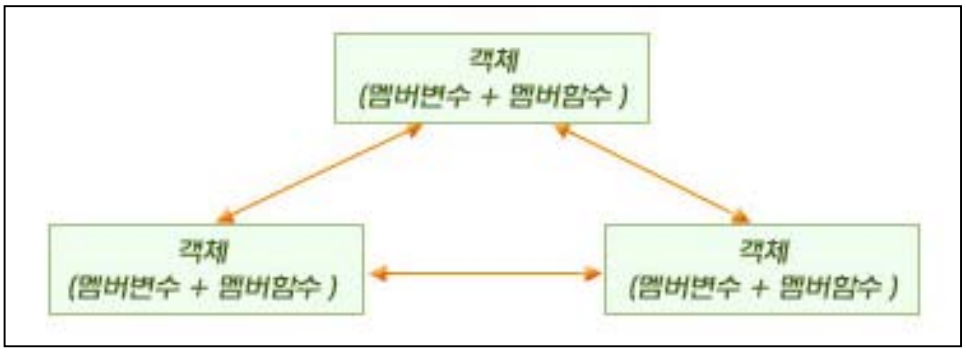
프로그래밍 기법은 순차적, 구조적, 객체지향적인 순서로 발전해 왔으며, 이를 나누는 기준은 프로그램이 다루는 주요 단위에 따라 결정됩니다.

프로그래밍 기법	주요 단위	내 용	관련 언어
Sequential Programming	문장	명령어들을 차례로 나열함	COBOL, FORTRAN
Structured Programming	함수	문장들을 함수 단위로 묶어 체계적으로 관리함	PASCAL, C
Object Oriented Programming	객체	관련된 데이터와 함수를 객체로 묶어서 관리함	C++, Java, C#

Structured Programming에서는 아래와 같이 데이터를 함수 안에 속해 있는 지역변수와 함수 밖에 있는 전역변수로 나누어서 취급합니다.



반면, Object Oriented Program에서는 객체 안에 속하는 멤버 변수로 데이터를 구현하게 됩니다.

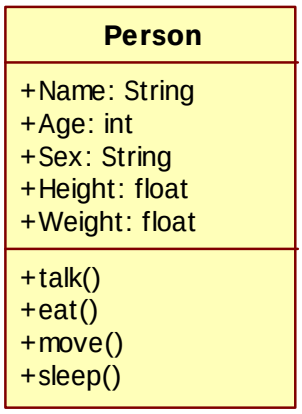


이 같은 Object Oriented 접근 방법은 현실 세계를 모형화하여 컴퓨터 구조에 맞게 옮겨오는 데에 있어 Structured 접근 방법보다 더욱 효과적입니다.

2) Terminology

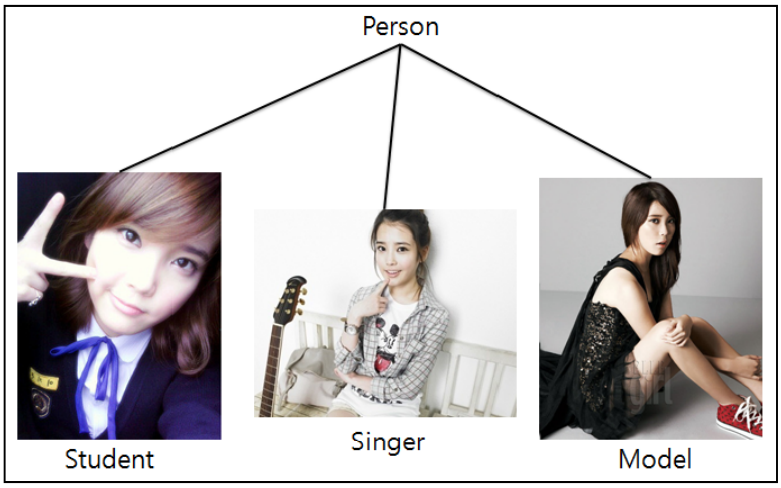
Object : 속성과 메소드를 묶어 둔 단위로의 객체라는 의미로 이야기 할 수도 있고, 메모리에 할당된 Class의 Instance를 의미할 수도 있습니다. Object는 State, Behavior, Identity 등의 개념을 가지고 있는데, State는 "변할 수 있는 상태"를 의미하며, Behavior는 "어떠한 입력에 대응하여 객체가 취할 수 있는 행동"을 의미합니다. Identity는 "객체를 유일하게 만드는 특성"을 의미합니다.

Class : Object에 대해 속성과 메소드를 정의하는 원형(Template) 이며, Class를 구현하여 실제로 만든 하나의 Instance를 Object라고 할 수 있습니다.



<Class Example>

Inheritance : 기존에 정의된 Class의 데이터와 함수를 이어 받고, 새로운 데이터와 함수를 추가할 수도 있는 기능입니다. Inheritance를 효율적으로 사용하면 Reusability를 높일 수 있습니다.



<Inheritance Example>

Polymorphism : 동일한 명칭의 함수가 상황에 따라 다르게 동작하는 기능입니다. 이는 Overloading과 Overriding을 통해 구현할 수 있습니다.

Encapsulation : 데이터와 함수를 Class로 묶어두고, 데이터를 Class 내부의 함수에 의해서만 접근이 가능하도록 구현하여 외부로부터 보호하는 기능을 말합니다. 효율적인 Encapsulation은 프로그램의 모듈화를 구현할 수 있습니다.

(3) Object Oriented Analysis

Object Oriented 개념으로 좋은 Software를 만들기 위해서는 ①고객의 needs를 확실하게 파악하고, ②이것을 Object-Oriented에 맞게 정의하고, ③Maintainability와 Reusability등을 신경 쓰며 Design 하는 등 크게 3가지 정도에 유념해야 합니다.

Object Oriented Analysis 과정에서는 개발 과정 추후에 실제 구현에 대해서는 신경 쓰지 않고 문제를 정의하는 데 초점을 맞춥니다. 앞서 말했듯이 Analysis 과정에서는 Problem Domain으로부터 Conceptual Model을 뽑아내기 위한 과정을 포함합니다.

1) Problem Domain - Requirement

Software Development에는 고객, 사용자, 개발 팀 등의 여러 Stakeholder가 관계를 맺고 있습니다. Requirement란 고객이 Software에 기대하는 것으로, Stakeholder 간에 적절한 동의를 얻어가며 기능과 제한 조건들을 포함한 Requirement를 잘 정의해야 합니다. 잘 명세 된 Requirement는 고객과 개발자 간에 빠르고 쉬운 이해를 돕습니다.

Requirement는 Functional Requirement와 Non-Functional Requirement로 나누어지는데, Functional Requirement는 추후에 Use-Case Model로 표현되며 Vision을 달성하기 위한 System Function List를 포함합니다. 아래 표와 같이 자연 언어로 List를 만들 수 있으며, Evident(반드시 수행되어야 하는 것), Hidden(수행 과정이 사용자에게 보여지지 않는 것), Frill(선택적 사항) 등으로 나눌 수 있습니다. Non-Functional Requirement는 대부분 제한 조건을 명시하게 되며, Usability, Reliability, Performance, Supportability 등과 같은 속성을 Use-Case Model이나 Supplementary Specification에 추가로 기술할 수 있습니다.

Ref#	Function	Category
R1.1	사용자는 ID와 PASSWORD의 입력을 통해 로그인 하여 MESSENGER에 접속한다.	Evident
R1.2	사용자는 친구들의 접속 여부를 계속적으로 반영하는 접속 사용자 명단을 볼 수 있다.	Evident
R1.3	서버는 지정된 클라이언트끼리 지속적인 통신 연결을 보장해야 한다.	Hidden
R1.4	서버는 클라이언트가 입력하는 문자열을 연결된 클라이언트에 전송한다.	Hidden
R1.5	연결된 클라이언트로부터 받은 문자열을 화면에 보여준다.	Evident
R1.6	상호 클라이언트 thread 간의 통신 메커니즘을 제공해야 한다.	Hidden
R1.7	기존 접속되어있는 ID 로 로그인 요청을 받는 경우, 기존에 접속된 연결을 끊고 최근 접속을 연결한다.	Evident

<Requirement Example>

2) Problem Domain - Supplementary Specification

Supplementary Specification은 추가 명세서로, Use Case 외의 다른 요구사항을 식별해야 하며, Quality에 관련된 속성들이 표현됩니다. 문제 영역 또는 비즈니스의 동작 규칙을 명세한 Domain (Business) Rules를 포함하며, Domain에 관련된 정보 등을 첨부합니다.

3) Use Case Model

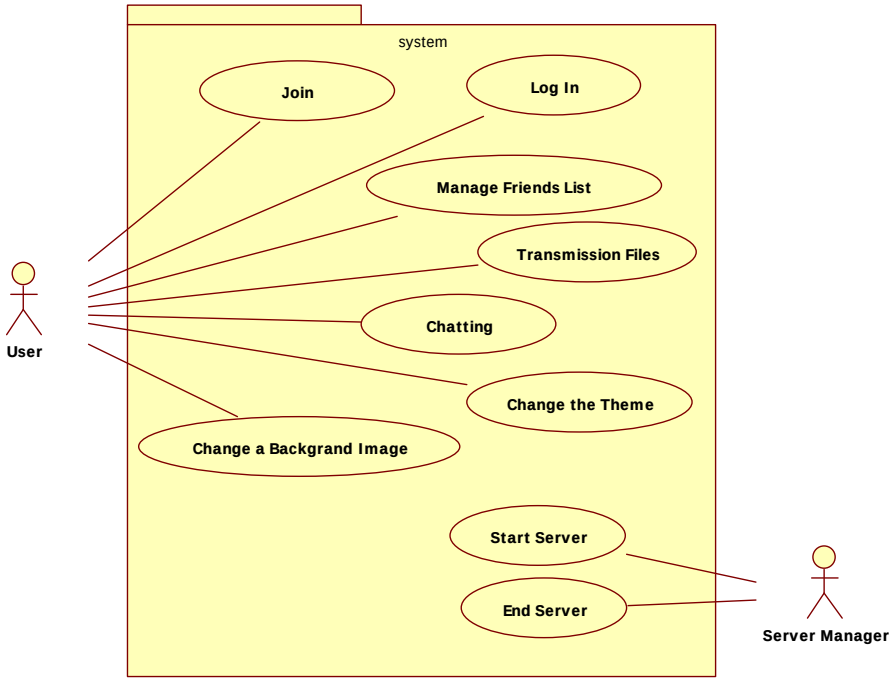
Use Case는 Process들을 상세하게 표현한 것으로, Software Project의 범위와 요구사항을 정의하며, 세부 기능과 업무 영역을 분석하고 이해할 수 있습니다. Use Case는 Use-Case를 정의하고 기술하는 과정과 Use-Case Interaction 분석, Use-Case Diagram을 그리는 과정이 있습니다. Use Case에서는 System 밖에 위치하는 Actor와 System 내에서 작동하는 Process를 표현하는 Use-Case, 그리고 이 둘 사이의 관계를 표현하는 Communication으로 표현할 수 있습니다 Use Case 별로 Actor가 System을 사용하는 과정에서 나타나는 Event를 순차적으로 표현합니다.

Use Case	Join
Actors	User
Type	Primary
Preconditions	User가 메신저에 등록할 수 있는 계정을 추가하려는 상태
Description	User는 사용할 ID와 password를 입력하고, 간단한 기본정보인 이름, 생년월일, 성별, 핸드폰번호를 입력하여 계정을 등록한다. Server System은 ID 중복 확인 후, 중복이 안된 경우, 회원 가입을 수락하고 Server에 계정을 등록, 저장한다.

<Use Case Model Example – Use Case>

Actor Action	System Response
1. user는 계정 추가를 요청한다.	
	2. 계정 추가 Form을 띄운다.
3. 계정 추가 Form에서 요구하는 user의 정보들을 입력한다.	
	4. user가 입력한 정보들을 Database에 등록하여 계정을 추가한다. 5. log-in이 처음 메신저 상태로 전환한다.

<Use Case Model Example – Use Case Interaction>

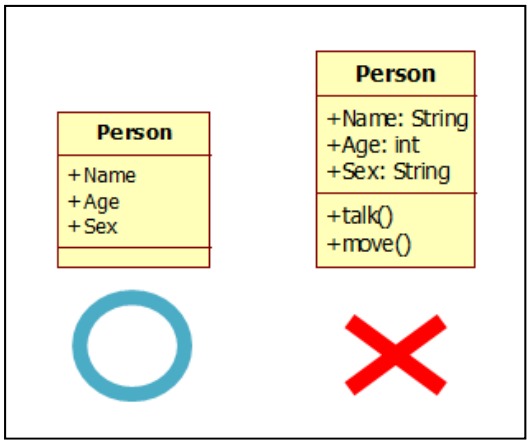


<Use Case Model Example – Use Case Diagram>

또한, Actor에 대해 Sequence Diagram으로도 Use Case를 표현하여 Process의 Sequence를 더욱 자세하게 볼 수 있습니다.

4) Domain Model (Conceptual Model)

Domain Model은 Requirement와 관련된 개념적 Class를 식별합니다. Domain 내의 개념적 Class와 현실 세계의 Object 들을 가시적으로 표현합니다. 이는 Object Oriented Analysis 과정에서 가장 중요한 결과물 중에 하나입니다. 앞서 분석한 Requirement 문맥 내에서 중요한 개념들을 추출하여 Object로 표현할 수 있습니다. Domain Model은 Software Component Model이 아니므로 Method를 포함하지 않습니다. 즉, Domain Model은 개발자와 고객 사이의 이해를 돕기 위한 목적을 포함하므로 Computer Based Model로 표현하지 않습니다.



<Domain Model Example>

Domain Model은 곧 Conceptual Class로 복잡한 문제를 이해 가능한 단위로 나누어 Domain 내의 사물이나 객체를 찾습니다. 이를 위해서 앞서 보았던 Use-Case에서 명사구를 식별하여 이를 개념적 Class나 속성의 후보로 간주하는 방법을 사용합니다.

5) Activity Diagram

Activity Diagram은 각종 Process나 업무에 대해 순서에 입각하여 그 흐름을 정의한 Model 입니다. Activity Diagram은 Software 분야 뿐만 아니라 다양한 분야에서 응용될 수 있습니다. Design 단계에서 Activity Diagram은 프로그램의 Process 흐름에 대한 사양을 정의할 수 있으며, Analysis 단계에서 Use-Case Diagram을 비슷하게 정의하는 경우도 많습니다.

(4) Object Oriented Design

Design은 Software를 통해 Analysis에서 정의한 Problem을 어떻게 풀어나갈 것인지 계획을 수립하는 활동입니다. Design은 개발 팀에 있어서 Solution을 어떻게 구현할 것인지 가이드라인이 되어 줍니다. 여기서 고려해야 할 점은, 이해하기에 쉬워야 하고, 구성에 변화를 주기에 쉬워야 하며 무엇보다 Requirement를 충족시킬 수 있어야 합니다.

Object Oriented Design은 Analysis에서 나온 Conceptual Model을 확장하여 각 Class에 대해 자세한 사항을 기입합니다. 예를 들어, attribute의 이름, 타입, 접근 지정자 등을 정하고, method의 이름, 반환 타입, 인자 등을 정해야 합니다. 또한 관련이 있는 Class 간에 상호작용이나 관계를 정의해야 합니다. 뿐만 아니라 System의 지원 사양-개발 환경, 프로그래밍 언어 등-에 맞춰 제약 조건들을 고려해야 합니다.

1) Assigning Responsibility

Architecture를 Design 할 때에는, Software의 구성 요소들을 책임과 역할에 따라 분류하고, 각 요소들 간의 상호 작용에 대해 정의하게 됩니다. Software Development에서 효율적인 Architecture는 Maintainability와 Reusability를 높일 수 있어 Software의 생산성이 좋아진다고 할 수 있습니다. Object Oriented Design은 이러한 Assigning Responsibility에 좋은 방법론 입니다.

2) Class Diagram

Static Model. Analysis 단계에서는 Method 등을 포함하지 않는 Class Diagram을 그렸지만 Design 단계에서는 바로 실제 구현 단계에서 Source Code로 쓸 수 있을 정도로 나타냅니다.

또한, 멤버 변수의 Visibility, Scope를 지정해야 하고, 각 Class 간의 관계를 나타냅니다.

3) Sequence Diagram

Interaction Diagram. Use-Case를 실제 설계한 Class와 Class의 Method를 사용하여 실현하는 Model로, 또한 Class의 Interface를 설계한 후에 이에 관련한 설계도 나타낼 수 있습니다. Diagram을 통해 객체간의 동작을 별도의 설명 없이 한 눈에 볼 수 있는 장점이 있습니다. 만약에 한 객체에 대하여 여러 Use-Case에 대한 변화를 나타내려면 State Diagram을 사용합니다. Sequence Diagram은 Sequence를 보여주기 위한 목적으로 그리는 것이기 때문에 시간 혹은 순서가 중요시 되는 경우에 사용합니다.

4) Collaboration Diagram

Object 간의 연관성에 대해 나타내는 Diagram으로, 내용이 중요한 경우에 이용합니다. 객체 간에 '연결'이라는 관계를 맺고, 각 개체가 주고 받는 메시지를 순서를 나타내는 숫자와 함께 명시합니다.

5) Component Diagram

Component는 실행 모듈을 의미하며, Component Diagram은 System의 구현 관점에서 Component를 정의하고 Component 간의 정적인 상호 작용을 정의합니다. 여기서 Component는 Interface를 통해 그 기능이 독립적으로 사용되는 모듈입니다. Component Diagram은 System 구축 단계에서 어떤 식으로 사용, 배치할 것인지 정의하는 데 사용할 수 있습니다. 단순히 실행 모듈 뿐만 아니라 Source Code, Database 등의 상호 작용 역시 나타낼 수도 있습니다.

이러한 Component Diagram은 보통 Design의 막바지 단계에서 모든 Class와 Class 간의 관계에 대해 명확한 정의가 이루어진 후에 작성될 수 있습니다. Component Diagram 작성 시, Component의 내부는 기능적으로 강한 유사성을 갖는 단위로 구성이 되어야 응집도가 높아지며, 다른 Component에 대해 약하게 종속이 되어야 낮은 결합도를 가질 수 있습니다. 또한 각 Component를 설계할 때에 크기가 너무 차이가 나지 않는 것이 좋습니다.

6) Deployment Diagram

Deployment Diagram은 Hardware 자원 간의 연결 관계를 표현한 뒤, 그에 대한 Software Component를 배치하는 Diagram 입니다. Hardware 자원을 명시적으로 작성하긴 하지만 Hardware 사양보다 Software 관점에서 Software가 동작하기 위한 Hardware 자원을 정의한다 라는 개념으로 작성하는 것 입니다.

Deployment Diagram은 System의 Hardware 구성을 개념적으로 한눈에 보여주는 역할을 합니다. Deployment Diagram을 그릴 때에는 모든 장비를 나타내지 않고, 목적과 용도에 부합하는 장비들만을 나타내는 것이 좋습니다. 모든 장비를 나타내는 경우 오히려 복잡해져서 의미를 파악하는데 방해가 될 수 있습니다.

2. UML

UML 은 컴퓨터 애플리케이션을 모델링 할 수 있는 통합 언어이다. 주요 작성자들은 Jim Rumbaugh, Ivar Jacobson, Grady Booch 이고 이들은 원래 그들만의 꽤 괜찮은 방식(OMT, OOSE, Booch)을 갖고 있었다. 결국 이들은 힘을 합쳤고 개방형 표준을 만들었다. UML 이 표준 모델링 언어가 된 한 가지 이유는 이것이 프로그래밍 언어에 독립적이라는데 있다. 또한 UML 표기법 세트는 언어이지 방법론이 아니다. 언어인 것이 중요한 이유는 방법론과는 반대로 언어는 기업이 비즈니스를 수행하는 방식에 잘 맞는다.

UML 은 방법론이 아니기 때문에 어떤 형식적인 작업 생성물들이 필요 없다. 하지만 정해진 방법론 안에서 쓰이면, 애플리케이션을 개발할 때 애플리케이션을 쉽게 이해할 수 있도록 도와주는 여러 가지 유형의 다이어그램을 제공한다. 이 다이어그램은 현재 사용하고 있는 것의 언어와 원리를 잘 소개하고 있다. 사용중인 방법론에서 생긴 작업 생산품들에 표준 UML 다이어그램을 배치하여 UML 에 능숙한 사람들이 프로젝트에 쉽게 참여하여 생산성을 높일 수 있도록 한다. 가장 유용한 표준 UML 다이어그램은 유스케이스 다이어그램, 클래스 다이어그램, 시퀀스 다이어그램, 스테이트 차트 다이어그램, 액티비티 다이어그램, 컴포넌트 다이어그램, 전개 다이어그램 등이 있다.

(1) 유스케이스 다이어그램

유스케이스는 시스템에서 제공한 기능 단위를 설명한다. 유스케이스 다이어그램의 주요 목적은, 다른 유스케이스들 간 관계 뿐만 아니라 주요 프로세스에 대한 "액터(actors)" (시스템과 인터랙팅하는 사람)들과의 관계를 포함하여, 개발 팀들이 시스템의 기능적 요구 사항들을 시각화 하는 데 있다. 유스케이스 다이어그램은 유스케이스 그룹들을 보여준다. 완전한 시스템에 대한 모든 유스케이스거나 관련된 기능을 가진 특정 유스케이스 그룹(예를 들어, 보안 관리에 관련된 유스케이스 그룹)의 유스케이스일 수도 있다. 유스케이스 다이어그램에 대한 유스케이스를 보여주려면 다이어그램 중간에 타원을 그려서, 타원의 중앙 또는 아래에 유스케이스 이름을 적어놓는다. 유스케이스 다이어그램에 액터(시스템 사용자)를 그리려면 다이어그램의 왼쪽이나 오른쪽에 사람 모양을 그려 넣는다. 액터와 유스케이스들간 관계는 그림 1 에 나타나있다.

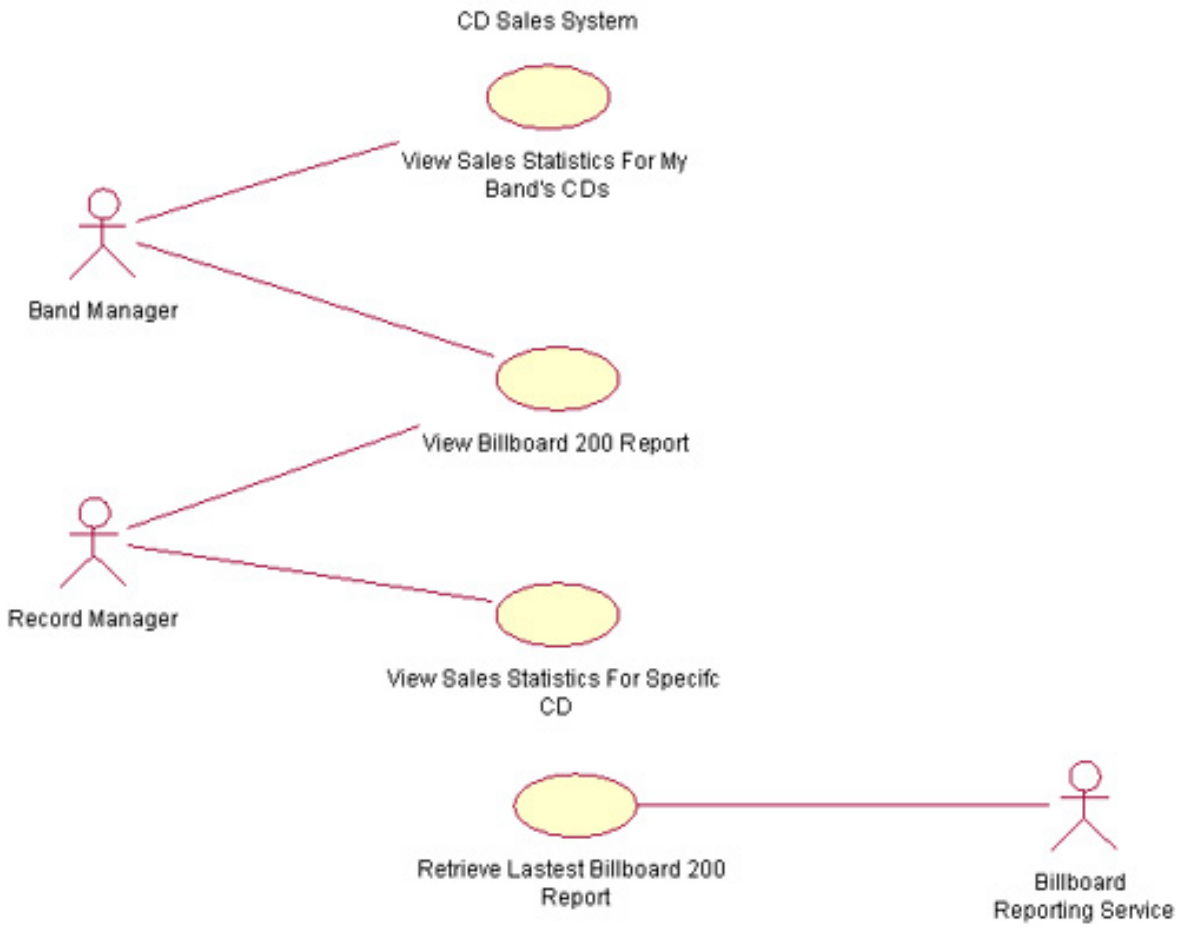


그림 1: 유스케이스 다이어그램

유스케이스 다이어그램은 시스템의 고급 기능과 시스템의 범위를 설명하는데 사용된다. 그림 1 의 유스케이스 다이어그램을 통해, 시스템이 제공하는 기능을 쉽게 표현할 수 있다. 이러한 시스템에서는 밴드 매니저가 밴드가 발매한 CD 에 대한 판매 통계 리포트와 Billboard 200 보고서를 볼 수 있다. 또한 레코드 매니저는 특정 CD 에 대한 판매 통계 보고서와 Billboard 200 보고서를 볼 수 있다. 이 다이어그램에서는 Billboard Reporting Service 라고 하는 외부 시스템에서 우리 시스템이 Billboard 리포트를 전달하고 있다는 것도 볼 수 있다.

또한, 이 다이어그램에 유스케이스가 없다는 것은 시스템이 수행하지 않은 일을 보여주고 있는 것이다. 예를 들어, 이 다이어그램은 밴드 매니저가 Billboard 200 의 다른 앨범들에 수록된 노래를 듣는 방식은 나와있지 않다. Billboard 200 에서 Listen to Songs 라는 유스케이스에 대한 어떤 레퍼런스도 볼 수 없다. 이것은 중요한 포인트이다. 그와 같은 다이어그램에 제공된 명확하고 간단한 유스케이스 설명을 통해 프로젝트 스폰서는 시스템에 필요한 기능이 존재하는지 여부를 쉽게 볼 수 있는 것이다.

(2) 클래스 다이어그램

클래스 다이어그램은 다른 엔터티들(사람, 제품, 데이터)이 서로 어떻게 관계를 맺고 있는지를 보여준다. 다시 말해서, 이것은 시스템의 정적 구조라고 할 수 있다. 클래스 다이어그램은 록밴드, 씨디, 라디오 연주를 논리적 클래스로 나타내는데 사용될 수 있다. 또는 대출, 주택 저당 대출, 자동차 대출, 이자율을 나타내는데도 쓰일 수 있겠다. 클래스 다이어그램은 주로 프로그래머들이 다루는 구현 클래스들을 보여주는데 쓰인다. 구현 클래스 다이어그램은 논리적 클래스 다이어그램과 같은 클래스를 보여준다. 하지만 이 구현 클래스 다이어그램은 같은 애트리뷰트로는 그릴 수 없다. Vectors 와 HashMaps 같은 것에 대한 레퍼런스를 갖고 있기 때문이다.

그림 2 에서는 세 개의 섹션으로 클래스 다이어그램을 설명하고 있다. 위 섹션은 클래스의 이름을, 중간 섹션은 클래스의 애트리뷰트를, 가장 아래 섹션은 클래스의 연산("그림 2 에서는 세 개의 섹션으로 클래스 다이어그램을 설명하고 있다. 위 섹션은 클래스의 이름을, 중간 섹션은 클래스의 애트리뷰트를, 가장 아래 섹션은 클래스의 연산 ("메소드")을 보여주고 있다.

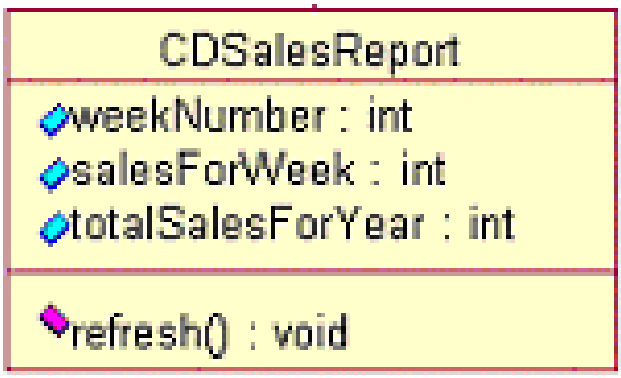


그림 2: 클래스 다이어그램의 클래스 객체

내 경험으로는 거의 모든 개발자들은 이 다이어그램이 무엇을 하고 있는지를 안다. 하지만 대부분의 프로그래머들은 관계도를 잘못 그리고 있다. 그림 3 과 같은 클래스 다이어그램의 경우 상속 관계를 그릴 때에는 화살표 방향을 위로 향하게 하여 수퍼 클래스를 지시하게 하면서 화살표 모양은 *완벽한 삼각형*이 되도록 해야 한다. 상관 관계는 두 클래스들이 서로를 인식하고 있다면 일직선이 되어야 하고, 클래스의 한 편만 알고 있는 관계라면 화살표 표시가 되어있는 선을 그어야 한다.

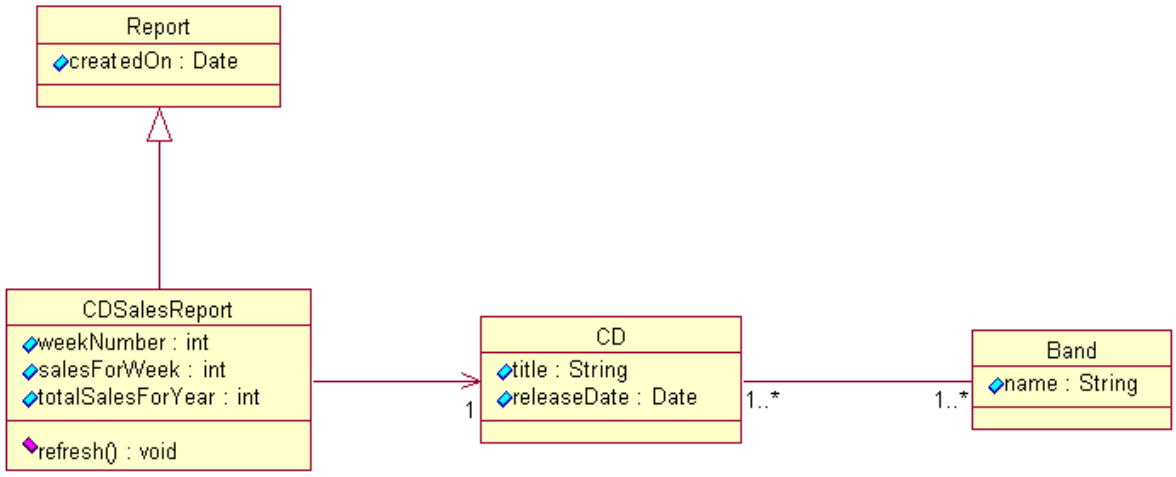


그림 3: 그림 2 의 클래스 객체가 포함된 클래스 다이어그램

그림 3 에서, 상속 관계와 두 개의 상관 관계를 보았다. CDSalesReport 클래스는 Report 클래스에서 상속을 받고, CDSalesReport 는 한 개의 CD 와 관련이 되어 있지만, CD 클래스는 CDSalesReport 에 대해 아무것도 모르고 있다. CD 와 Band 클래스는 서로에 대해 알고 있고, 두 클래스는 서로 연관되어 있다.

(3) 시퀀스 다이어그램

시퀀스 다이어그램은 특정 유스케이스에 대한 상세한 흐름이나 심지어는 특정 유스케이스의 일부분 까지도 보여준다. 대부분이 설명을 포함하고 있다. 시퀀스에서 다른 객체들 간의 호출관계를 보여주고 있고, 다른 객체들로의 다른 호출까지 상세하게 보여줄 수 있다.

시퀀스 다이어그램은 2 차원으로 그려진다. 수직 차원은 발생 시간 순서로 메시지/호출 시퀀스를 보여주고 있다. 수평 차원은 메시지가 전송되는 객체 인스턴스를 나타내고 있다.

시퀀스 다이어그램은 그리기가 매우 간단하다. 다이어그램의 상단에 각 클래스 인스턴스를 박스 안에 놓아 클래스 인스턴스(객체)를 구분한다. (그림 4) 박스 안에 클래스 인스턴스 이름과 클래스 이름을 스페이스/콜론/스페이스 " : "로 분리시킨다. (예, myReportGenerator : ReportGenerator) 클래스 인스턴스가 메시지를 또 다른 클래스 인스턴스로 보내면 클래스 인스턴스를 받는 곳을 가리키는 화살표를 긋는다. 그 라인 위에 메시지/메소드 이름을 적는다. 중요한 메시지의 경우는 원래의 클래스 인스턴스를 다시 향하도록 점선 화살표를 그릴 수 있다. 점선 위에 리턴 값을 라벨링한다. 개인적으로는 리턴 값을 포함하곤 하는데 상세한 부분을 읽기 쉽기 때문이다.

시퀀스 다이어그램을 읽기는 매우 간단하다. 시퀀스를 시작하는 "드라이버(driver)" 클래스 인스턴스가 있는 왼쪽 상단 코너부터 시작한다. 그런 다음, 다이어그램 아래쪽을 각 메시지를 따라간다. 그림 4 의 시퀀스 다이어그램 예제에서 전송 메시지에 대한 리턴 메시지는 선택적인 것임을 기억하라.

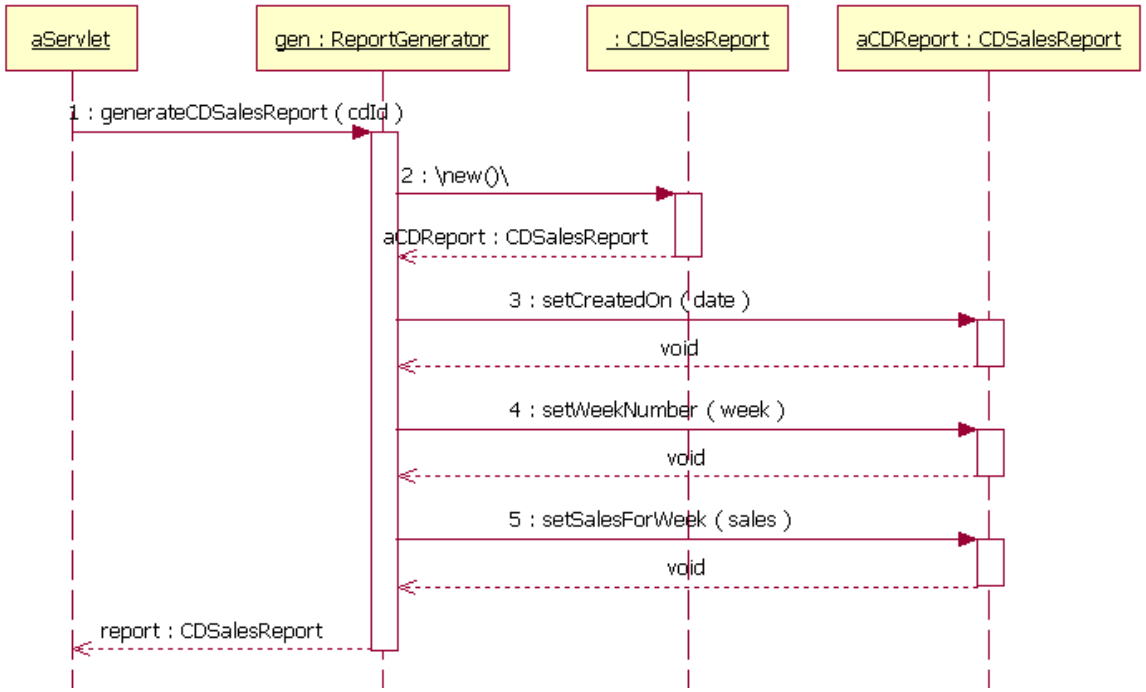


그림 4 : 시퀀스 다이어그램

그림 4 의 시퀀스 다이어그램을 읽다 보면 CD Sales Report 가 어떻게 만들어지는지를 알 수 있다. aServlet 객체가 우리의 드라이버 예제이다. aServlet 은 메시지를 gen 이라고 하는 ReportGenerator 클래스 인스턴스로 보낸다. 이 메시지는 generateCDSalesReport 라는 라벨링이 붙는다. ReportGenerator 객체가 이 메시지 핸들러를 구현한다는 의미이다. 자세히 보면, generateCDSalesReport 메시지 라벨은 괄호 안에 cdId 가 있다. gen 인스턴스가 generateCDSalesReport 메시지를 받으면 CDSalesReport 로 연속 호출을 하고 aCDReport 라고 하는 CDSalesReport 의 실제 인스턴스가 리턴 된다. gen 인스턴스는 리턴된 aCDReport 인스턴스에 호출하면서 여기에 각 메시지 호출에 대한 매개변수를 전달한다. 시퀀스의 끝에서 gen 인스턴스는 콜러였던 aServlet 에 aCDReport 를 리턴한다.

그림 4 의 시퀀스 다이어그램은 전형적인 시퀀스 다이어그램을 상세히 설명한 것이다.

(4) 스테이트 차트 다이어그램

스테이트 차트 다이어그램은 클래스가 개입된 다양한 상태(state)를 모델링 하고 그 클래스가 상태간 어떻게 이동하는지를 모델링 한다. 모든 클래스는 상태를 갖고 있지만 각각의 클래스가 스테이트 차트 다이어그램을 가질 수 없다. "중요한" 상태, 말하자면 시스템 작동 중 세 개 이상의 잠재적 상태가 있는 클래스일 경우만 모델링 되어야 한다.

그림 5 에서 보듯, 스테이트 챠트 다이어그램에는 다섯 개의 기본 엘리먼트들이 사용된다. 시작점(짙은 원), 스테이트 간 이동(화살표), 스테이트(모서리가 둥근 직사각형), 결정 포인트(속이 비어있는 원), 한 개 이상의 종료점(원 내부에 짙은 원이 그려져 있음)이 바로 그것이다. 스테이트

챠트 다이어그램을 그리려면 시작점과 클래스의 초기 상태를 향하는 화살표로 시작한다. 다이어그램 어디에나 이 스테이트를 그릴 수 있고 스테이트 이동 라인을 사용하여 연결한다.

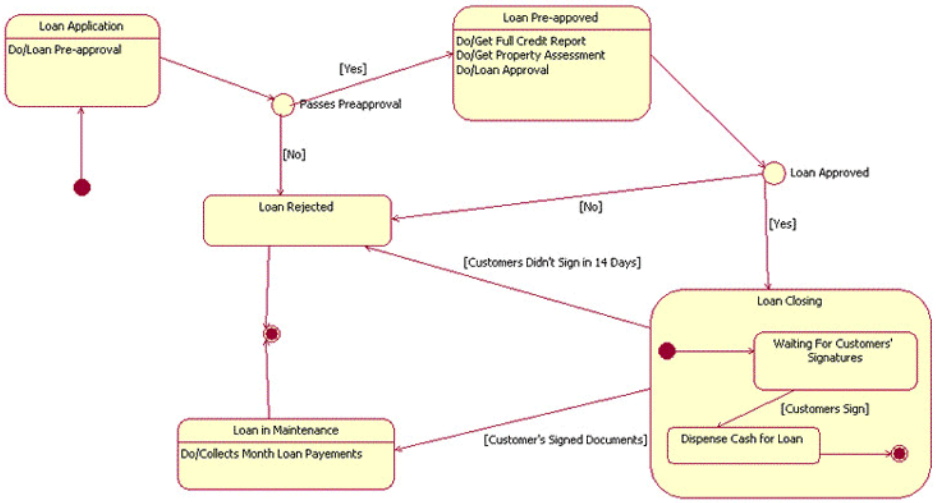


그림 5 : 시스템 작동 중 클래스가 실행되는 다양한 상태를 보여주는 스테이트 차트 다이어그램

그림 5 의 스테이트 차트 다이어그램은 중요한 정보를 보여주고 있다. 예를 들어, 대출 프로세스가 Loan Application 상태에서 출발한다고 말할 수 있다. 결과에 따라 사전 승인 프로세스가 완료되면 Loan Pre-approved 상태나 Loan Rejected 상태로 옮겨간다. 이동하는 동안 내린 결정은 결정 포인트로 보여진다. 이동 라인 상의 비어있는 원이 바로 그것이다. 이 예제를 보면 Loan Closing 상태를 거치지 않고는 대출이 Loan Pre-Approved 상태에서 Loan in Maintenance 상태로 갈 수 없음을 알 수 있다. 또한, 모든 대출이 Loan Rejected 상태 또는 Loan in Maintenance 상태에서 끝난다는 것도 알 수 있다.

(5) 액티비티 다이어그램

액티비티 다이어그램은 액티비티를 처리하는 동안 두 개 이상의 클래스 객체들 간 제어 흐름을 보여준다. 액티비티 다이어그램은 비즈니스 단위 레벨에서 상위 레벨의 비즈니스 프로세스를 모델링 하거나 저수준 내부 클래스 액션을 모델링 하는데 사용된다. 내가 경험한 바로는 액티비티 다이어그램은 기업이 현재 어떻게 비즈니스를 수행하는지, 또는 어떤 것이 비즈니스에 어떤 작용을 하는지 등의 고차원 프로세스를 모델링 할 때 가장 적합하다. 액티비티 다이어그램은 언뜻 보기에 시퀀스 다이어그램 보다는 덜 기술적이기 때문에 비즈니스 마인드를 가진 사람들이 빠르게 이해할 수 있다.

액티비티 다이어그램의 표기법은 스테이트 차트 다이어그램과 비슷하다. 스테이트 차트 다이어그램과 마찬가지로 액티비티 다이어그램은 초기 액티비티에 연결된 실선으로 된 원에서 시작한다. 이 액티비티는 모서리가 둥근 직사각형을 그려 그 안에 액티비티 이름을 적어 넣으면서 모델링 된다. 액티비티들은 이동 라인을 통해 다른 액티비티들에 연결되거나 결정 포인트의 조건에 제약을 받는 다른 액티비티들에 연결하는 결정 포인트로 연결될 수 있다.

모델링 된 프로세스를 종료하는 액티비티는 (스테이트 차트 다이어그램에서처럼) 종료 포인트에 연결된다. 이 액티비티들은 수영 레인으로 그룹핑 될 수 있다. 이것은 실제로 액티비티를 수행하는 객체를 나타내는데 사용된다. (그림 6)

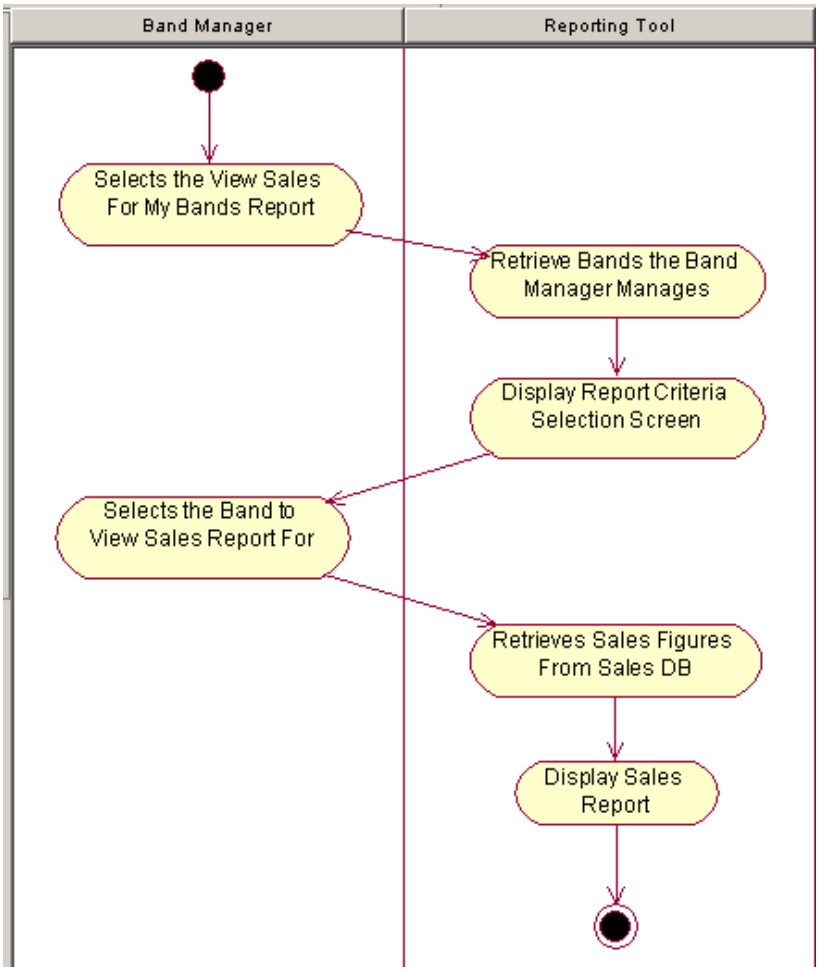


그림 6: 두 개의 객체(밴드 매니저와 리포팅 툴)에 의한 액티비티 제어를 나타내는 두 개의 수영 레인으로 되어있다.

그림 5 의 스테이트 차트 다이어그램은 중요한 정보를 보여주고 있다. 예를 들어, 대출 프로세스가 Loan Application 상태에서 출발한다고 말할 수 있다. 결과에 따라 사전 승인 프로세스가 완료되면 Loan Pre-approved 상태나 Loan Rejected 상태로 옮겨간다. 이동하는 동안 내린 결정은 결정 포인트로 보여진다. 이동 라인 상의 비어있는 원이 바로 그것이다. 이 예제를 보면 Loan Closing 상태를 거치지 않고는 대출이 Loan Pre-Approved 상태에서 Loan in Maintenance 상태로 갈 수 없음을 알 수 있다. 또한, 모든 대출이 Loan Rejected 상태 또는 Loan in Maintenance 상태에서 끝난다는 것도 알 수 있다.

액티비티 다이어그램 예제는 두 개의 객체(밴드 매니저와 리포팅 툴)에 의한 액티비티 제어를 나타내는 두 개의 수영 레인으로 되어있다. 프로세스는 한 밴드에 대한 판매 리포트를 보는 밴드 매니저로 시작한다. 리포팅 툴은 사람이 관리하는 모든 밴드들을 검색하여 디스플레이하고 이중

한 개를 고를 것을 요청한다. 밴드 매니저가 한 밴드를 선택하면 리포팅 툴은 판매 정보를 검색하여 판매 리포트를 디스플레이 한다.

(6) 컴포넌트 다이어그램

컴포넌트 다이어그램은 시스템을 물리적으로 볼 수 있도록 한다. 이것의 목적은 소프트웨어가 시스템의 다른 소프트웨어 컴포넌트들(예를 들어, 소프트웨어 라이브러리)에 대해 소프트웨어가 갖고 있는 종속 관계를 보여주는 것이다. 이 다이어그램은 매우 고급 레벨에서 볼 수 있거나 컴포넌트 패키지 레벨에서 볼 수 있다.

컴포넌트 다이어그램의 모델링은 이 예제에 잘 설명되어 있다. 그림 7 은 네 개의 컴포넌트인 Reporting Tool, Billboard Service, Servlet 2.2 API, JDBC API 를 보여주고 있다. Reporting Tool 에서 출발하여 Billboard Service, Servlet 2.2 API, JDBC API 로 가는 화살표는 Reporting Tool 이 이들 세 개의 컴포넌트에 종속되어 있음을 나타낸다.

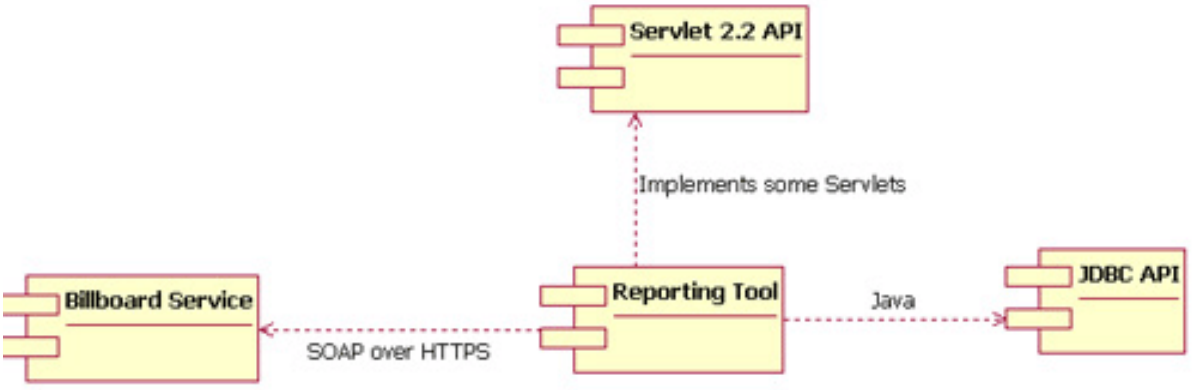


그림 7: 컴포넌트 다이어그램은 다양한 소프트웨어 컴포넌트들 간 종속관계를 보여준다.

(7) 전개 다이어그램

전개 다이어그램은 하드웨어 환경에 시스템이 물리적으로 어떻게 전개되는지를 보여준다. 목적은 시스템의 다양한 컴포넌트들이 어디에서 실행되고 서로 어떻게 통신하는지를 보여주는 것이다. 다이어그램이 물리적 런타임을 모델링 하기 때문에 시스템 사용자는 이 다이어그램을 신중하게 사용해야 한다.

전개 다이어그램의 표기법에는 컴포넌트 다이어그램에서 사용되던 표기법 요소들이 포함된다. 이외에 노드 개념을 포함하여 두 가지 정도 추가되었다. 노드는 물리적 머신 또는 가상 머신 노드(메인프레임 노드)를 표현한다. 노드를 모델링 하려면 3 차원 큐브를 그려 큐브 상단에 노드 이름을 적는다. 시퀀스 다이어그램에서 사용되던 네이밍 규칙([instance name] : [instance type]) (예, "w3reporting.myco.com : Application Server").

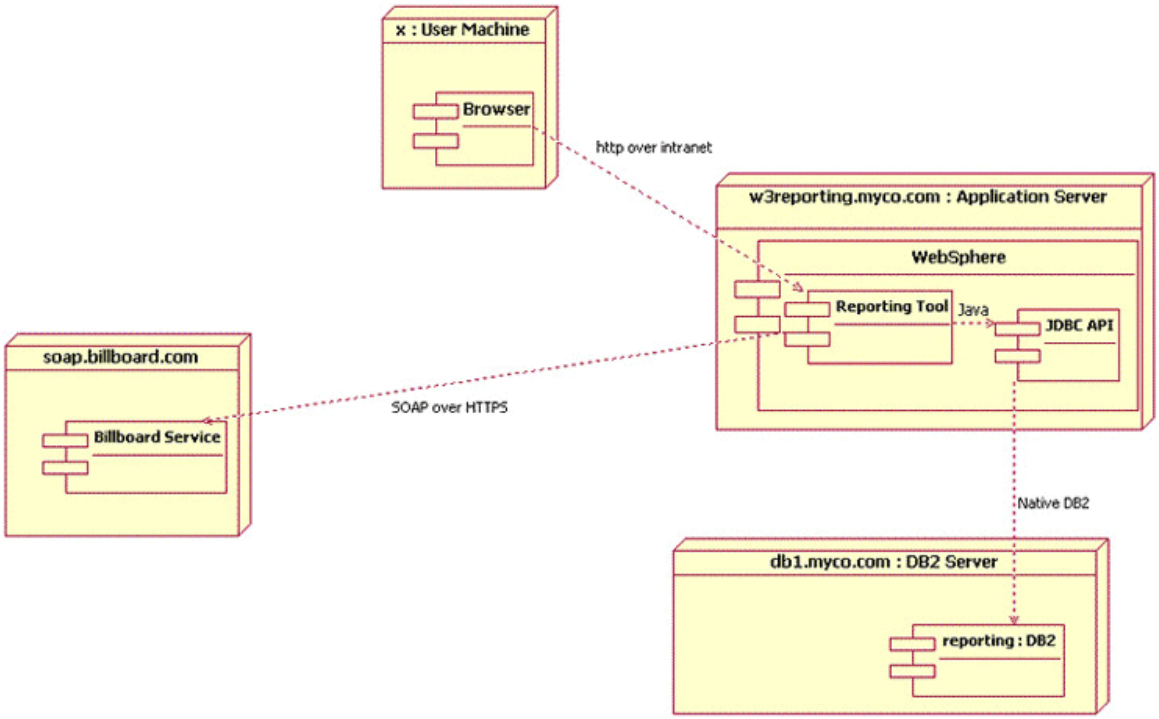


그림 8: 전개 다이어그램.

Reporting Tool 컴포넌트가 WebSphere 내부에서 그려지기 때문에(w3.reporting.myco.com 노드의 내부에서 그려짐), 사용자들은 로컬 머신에서 실행되는 브라우저를 통해 Reporting Tool 에 액세스 하면서 기업 인트라넷을 통해 HTTP 에 연결할 수 있다는 것을 알 수 있다.

그림 8 의 전개 다이어그램은 사용자가 로컬 머신에서 실행되고 기업의 인트라넷에서 HTTP 를 통해 Reporting Tool 에 연결되는 브라우저를 사용하여 Reporting Tool 에 접근하는 것을 보여주고 있다. 이 툴은 물리적으로 w3reporting.myco.com 이라고 하는 Application Server 에서 실행된다. 이 다이어그램은 IBM WebSphere 내부에서 그려진 Reporting Tool 컴포넌트를 보여준다. 이것은 결과적으로 node w3.reporting.myco.com 에서 그려지게 되어있다. Reporting Tool 은 자바를 사용하여 리포팅 데이터베이스를 IBM DB2 의 JDBC 인터페이스에 연결하여 원시 DB2 통신을 사용하는 db1.myco.com 서버상에서 실행되는 실제 DB2 데이터베이스와 통신한다. 리포팅 데이터베이스와 통신하는 것 외에도 Report Tool 컴포넌트는 SOAP over HTTPS 를 통해 Billboard Service 와 통신한다.

3. Design Patterns

- ✓ 디자인 패턴은 반복적으로 발견하게 되는 설계 문제에 대한 반복적인 솔루션이다.
- ✓ 소프트웨어 개발 영역에서 일정한 작업을 수행하는 방법을 설명하는 규칙의 집합이다.

(1) 왜 디자인 패턴인가?

디자인 패턴은 객체지향 언어에서 클래스와 클래스의 Method 를 사용하는 패턴, 즉, 의도를 추구하는 방법을 의미한다. 보통, 개발자들은 프로그래밍 언어를 배워서 한동안 코드를 작성해 보고 나서야, 디자인(설계)에 관해 고민하기 시작하는 경우가 많다. 가끔 남이 작성한 코드가 자신의 코드보다 더 심플하면서도 더 잘 동작하는 것을 보고는, 어떻게 이 사람은 이렇게 코드를 심플하게 작성했을까 하고 궁금해 한 적이 있을 것이다. 디자인 패턴은 코드의 수준을 한 단계 높여 주고, 적은 수의 클래스로 원하는 목적을 달성할 수 있도록 해준다. 또 유지 보수의 비용을 적게 하고 직관적이기 때문에 이해가 쉽고 유연하게 잘 돌아가는 그런 클래스 설계가 바로 디자인 패턴이라 할 수 있습니다.

(2) 객체지향의 기본 개념

객체 지향에서는 인터페이스와 구현의 분리를 아주 중요하게 여깁니다. 지금 사용하고 있는 컴퓨터를 예로 들어 설명하자면 컴퓨터 본체의 인터페이스를 살펴보면 전원, Reset 버튼, CD-ROM 버튼 등의 버튼들과 모니터 연결 케이블, 전원 케이블 등의 케이블이 있습니다. 다음으로 컴퓨터의 구현을 살펴보자면 많은 수의 내부 부품과 그 부품들 간의 통신이 복잡하게 연결되어 작동합니다. 그러나 유저들은 아주 쉽게 사용합니다. 소위 컴맹이라는 사람들도 컴퓨터를 켜고 끄는 것이나, CD-ROM 을 문제 없이 사용합니다. 이것이 바로 구현과 인터페이스 분리의 첫 번째 이점입니다.

1) 인터페이스와 구현의 분리 이점

① 사용하기 쉽게 해준다.

구현은 감추고 꼭 필요한 인터페이스만 유저에게 보여줌으로써 에러를 줄이고 사용을 쉽게 하는 것입니다. 컴퓨터 케이스를 벗긴 후 컴맹에게 컴퓨터를 켜보라고 하면 난감해 할 수도 있습니다.

Class 설계도 이것과 비슷합니다. Class 에 불필요한 함수들이 많고, Member 변수들까지 public 에다 놓는 것은 케이스 없는 컴퓨터를 설계하는 것과 같은 효과를 냅니다. 게다가 모든 멤버 변수와 함수를 public 에다 넣는 것은 구조적 프로그래밍을 하시는 분들에게는 프로그래밍을 편하게 하겠지만 그 프로그래머 외에는 누구에게도 쓰기 편하지 않을 것입니다. 게다가 이것은 클래스들간의 결합을 강하게 만드는 결과를 내게 됩니다. 여러 클래스들이 서로의 멤버 변수에 접근하고 구현 함수에 접근을 하는 상황은 본체와 모니터의 껍질을 벗겨놓고 무수히 많은 선으로 연결 해 놓은 상황과 같습니다.

② 객체 간의 결합을 약하게 해줌으로써 유지 보수의 비용을 줄인다

만약 본체와 모니터가 수많은 선을 통해 연결이 되어 있다면 모니터를 새로운 기종으로 바꾸는 데에만 많은 시간을 할애해야 할 수도 있습니다. 잘못 연결해서 사용 할 수 없는 경우도 있겠지만 반면 수많은 선을 통해 직접 연결되기 때문에 단일 케이블로 연결되는 것보다 속도가 더 빠를 수도 있습니다.

프로그래밍에서도 마찬가지로 객체 지향 방식과 구조적 프로그래밍 방식의 속도 차이는, 차이가 없다고 봐도 무방합니다.

2) 인터페이스와 구현의 분리 방법

① 최소한의 완전한 인터페이스를 구축한다

Effective C++에 나온 말입니다. 클래스에서는 public 이 인터페이스를 대표한다고 할 수 있습니다. public 에 불필요한 함수를 포함시키지 말라는 말입니다. 즉, 필요한 함수는 모두 넣되 쓸데 없는 것은 모두 빼고 최소화 하라는 것입니다. 그러나 이것이 실제로는 쉽지 않습니다. 필요한 함수일 것 같은데 나중에 보면 쓸데없는 경우도 있습니다. 그래서 필요한 것이 Refactoring(재분해) 입니다.

먼저 이번에 구현할 기능을 정하고, 그것을 위해 꼭 필요한 함수만 만듭니다. 그리고 다음 기능 구현할 때 필요한 함수를 또 추가하고, 바뀌어야 될 함수는 바꾸고 해서 점진적이고 반복적으로 설계를 발전시켜 나가는 것입니다.

② 멤버 변수를 public 에다 넣지 않는다

마찬가지로 Effective C++에 나온 말입니다. 멤버 변수를 public 에다 넣는 것이 좋지 않다는 것은 위에서 설명을 했듯이 좋지 않습니다.

③ 디자인 패턴을 이용한다

디자인 패턴이 이것을 도와줍니다. 유지 보수의 비용을 적게 하고 직관적이기 때문에 이해가 쉽고 유연한 class 설계가 바로 '디자인 패턴'이라 할 수 있습니다.

(3) Design Pattern

1) 디자인 패턴 적용 시 중요한 세가지 규칙

① Implementation class 가 아닌 interface 를 이용

② Inheritance 가 아닌 Delegation 을 사용

```
public interface AA {
   public String getInfo();
}
public class Super implements AA {
   public String getInfo() {}
}
public class Sub {
   AA delegator;
```

```
    public void test() {
        delegator.getInfo();
    }
}
```

cf) white-box reuse: Inheritance vs. black-box reuse: Delegation

③ Coupling 의 최소화

- Coupling: 어느 하나의 기능 변화가 전체 시스템의 구조에 영향을 미치는 정도
- 이것이 커질수록 잘못된 디자인
  - Refactoring 과 design pattern
    : 실제 문제 분석이 종료된 이후 좀더 효율적인 코드로 변화시키는 것을 refactoring 이라 한다. 이 과정에서 디자인패턴이 적용되는 경우가 많다.
    구현하기 이전에 보지 못했던 디자인상의 문제점들을 실제 구현에 들어가 이를 개선하는 것
  - Anti-Pattern
    : S/W를 만드는 과정에서 쉽게 도출되는 바람직하지 않은 방법들
    : 종류
      - 관리상의 안티
      - architecture 상의 안티
      - 개발상의 안티
    : 대표적인 anti-pattern
      - Spaghetti code: 개발과정에서 나타남. 기능의 추가로 인해 코드의 복잡도가 증가, 꼬이는 현상
      - Stovepipe System: architecture의 문제, 확실한 추상화가 없이 하나로 묶어 제품을 만드는 경우 신뢰성, 유지보수가 모두
      - Analysis Paralysis: 분석단계에서 너무 완벽함을 추구하다 보면 개발 진행 자체가 마비
      - Swiss Army Knife: architecture 작성시 하나의 객체에 너무 많은 기능/인터페이스를 넣는 경우 복잡, 유지보수 난이, 커플링 증가
      - Ambiguous Viewpoint: 불명확한 상태에서 모델링을 진행 → 객체의 불명확

2) 디자인 패턴 적용 시 중요한 세가지 규칙

① Facade pattern

- 여러 객체들이 가지는 logic 을 하나의 객체로 집중시키는 형태
- helper, wrapper라고도 한다.
- client side에서 접근해야 할 클래스가 많은 경우 앞 단에 Facade class를 만들어 이를 통해 접근
- client 와 내부 구현 class 와의 coupling 최소화
- logic 에서 jdbc api에 의존적이지 않음으로 인해 sqlj , oodb 등의 지원을 독립적으로 수행 가능.

- java.net.URL class 가 대표적인 예

```
public interface JDBCHelper { }
public class JDBCStatementHelper implements JDBCHelper{  }
public class JDBCPrepStatementHelper implements JDBCHelper{  }
```

② Factory method pattern

- logic 측의 요청에 따라 서로 다른 객체를 생성하는 Factory 를 정의
- 하나의 facade class 내의 계속적인 기능추가는 로직의 복잡성과 runtime error 유발 가능성 고조
- factory pattern 의 사용시 기능 확장 및 추가에 따라 facade interface 를 상속/구현한 클래스들의 추가 가능해짐
- 또한 하단에서 상속받은 메소드의 사용하지 않음을 별개로 결정가능
  ex) Statement 의 경우 setType() 을 비롯한 메소드들의 사용 금지

```
public class JDBCHelperFactory {
   JDBCHelper createJDBCHelper(){ }
}
JDBCHelper helper = factory.createJDBCHelper( ip, port, sid,
      JDBCHelperFactoryIF.HELPER_STATEMENT );
JDBCHelper helper1 = factory.createJDBCHelper( ip, port, sid,
      JDBCHelperFactoryIF.HELPER_PREPARED_STATEMENT );
```

③ Strategy pattern

- client 의 다양한 action 처리를 필요로 하는 경우
- 복잡한 switch 문 을 하나의 클래스로 독립 -> 보다 readability 가 증가한다.
- 처음 application 수행 시 종류가 결정됨
- OS, DB 별로 서로 다른 처리를 하려 할 때
  - 한글 converting 작업을 수행하는 별개의 클래스 도입
  - 이를 JDBCHelper 즉 facade 에서 보유
  - 코드변환 시 적절히 호출
  - 기능 추가, 변경이 용이해짐.

④ Cache Management Pattern

- cf) Cache
- 작은 용량의 빠른 기억장치를 사용, 대용량의 느린 기억장치의 access 속도 개선
- 데이터베이스의 경우 data 들의 일부를 server상에 memory cache 기법을 사용 일부 보관하는 방법을 말한다.

- Cache 의 구현 시 필요한 기능
    1) memory repository 구현
    2) repository management 기능 구현
    3) Kick-off 의 구현: 저장소 size 초과시 하나의 객체를 저장소에서 다시 제거
- class 구성
    - Object Key
    - CacheManager
    - ObjectCreator
    - Cache

⑤ Producer-Consumer pattern
- Event 의 발생을 EventQueue 에 저장
- 이를 각 Event Consumer 에게 전달하고 이 Event object를 ActionManager 가 받아서 수행한다
- 단일 Producer, Consumer 구조가 아님.
- class 구성
    - Producer
    - EventQueue
    - EventListener
    - Consumer
    - ActionManager
- java api 에서 구현 예
    - java.io.PipedInputStream, PipedOutputStream
    - Produer: PipedOutputStream
        - PipedInputStream 을 내장
        - write() 를 이용해 Queue에 데이터 produce
        - 이때 queue 의 역할인 PipedInputStream.receive() 를 호출한다
    - Queue + Consumer : PipedInputStream
        - read() 를 이용해 queue에 저장되어 있는 데이터를 읽어낸다
        - 이때 blocked i/o 이므로 데이터가 없을 경우 계속 대기한다
        - 이를 해결하기 위해서 size(), available() 등의 메소드를 구현할 것

⑥ Dynamic Connection Pattern
- 초기화 시에 처리할 각 동작객체들을 해시테이블에 key, value 의 형태로 저장
- 처리해야 할 시점에서 해시테이블로부터 처리객체를 얻어낸다
- 처리객체는 abstract class 를 상속받아 만들고 실제 처리하는 쪽(클라이언트)에서는 abstract class 내의 method를 통해 각 처리객체들을 공통적으로 수행시킨다
- 서버의 확장성 구체화
    - 기능 추가 시 flexible

- action은 preprocess, do, postprocess로 나누어 처리하는 편이 좋다

- 구성
  - ActionAbstract class
  - ActionImpl class...
  - ActionManager: Hashtable 보유

⑦ Command pattern

- client/server 간에 수행해야 할 작업을 전달할 때 command 를 별도로 정의하고 이를 처리할 command처리기를 디자인

- 구성
  - AbstractCommand
  - ConcreteCommand class
  - CommandManager

⑧ Guarded Suspension pattern

- 실제 실행하는 부분의 처리 프로세스의 개수를 제한
- 어떤 전제조건이 만족되기 전까지는 수행을 계속하고 벗어나는 경우는 계속 대기하는 형태
- 주의사항: 적절한 notify 가 없을 경우는 deadlock 이 발생할 수 있다
- 구성
  - CommandQueue
    - singleton pattern
    - 처리될 command 객체들을 담을 Queue를 가짐
    - queue 에서 data 를 꺼내고 넣는 동작 구현 (push, pop)
    - 이때 범위를 지정하고 제어한다

⑨ Immutable pattern

- 한번 만들어진 객체는 변화시키지 않는다
- 변화가 발생할 때는 새로운 객체를 생성해서 적용시킨다
- 대표적인 예: java.lang.String

⑩ Adapter pattern

- interface + adapter + implementation class
- 여러 객체들이 하나의 interface에 의해 사용된다
- client에게는 interface만 노출되며 이의 호출이 adapter에게 전달된다
- adapter는 내부적인 수행 로직을 갖지는 않으며 로직을 갖고 있는 또 다른 클래스인 adapter를 호출 이를 처리한다
- adapter의 role: client의 interface에 의한 호출을 실제 구현객체에게 전달하는 형태
- adapter와 adaptee 의 관계는 use
- 대표적인 예: java.awt.event 내의 XXXAdapter class

⑪ Bridge pattern

- 여러 개의 추상클래스들을 서로 관계를 주어 정의하고 이를 구현한 클래스들 역시 동일한 관계를 가지는 형태
- AbstractA + AbstractB + AImpl + BImpl
- AbstractA 와 AbstractB는 상속관계
- AImpl 은 AbstractA를 BImpl은 AbstractB를 구현(use)한 객체
- AImpl 과 AbstractB와의 관계 역시 구현(use) 관계
- 대표적인 예: java.awt.Component 와 java.awt.peer.ComponentPeer, Button과 ButtonPeer
    - ComponentPeer interface
        - Button 같은 Component의 하위 클래스들이 구현될 때 요구되는 상호 운용 환경에 의존적인 method들 정의
    - ButtonPear
        - Button 에서 구현해서 사용
    - Component와 ComponentPeer 의 관계가 Button class와 ButtonPeer interface간에 동일하게 유지됨

⑫ FlyWeight Pattern

- 동일한 정보를 가지는 객체들의 재활용: pooling
- 재사용성 증대
- AbstractA + ImplA1 + ImplA2 + FactoryA
- client가 ImplA1 객체의 요청을 Factory 에게 전달
- Factory는 자신의 pool내에 동일한 정보를 가지는 객체가 있는지 체크
- 없을 경우 새로 생성, 이를 pool에 넣고 서비스
- 대표적인 예: java.lang.String - string literal 은 재사용된다. 별개의 풀링 : 이를 수행하는 메소드 String.intern() 임
    - 매번 String listeral 을 생성하라는 요청이 들어오면 intern()이 호출됨
    - 풀링을 체크, 서비스하는 기능이 native method로 구현되어 있음
    - 워드프로세서에서 문자 객체
    - HTML 의 구문처리를 위한 Element 객체 등
- immutable pattern 과 아울러서 사용되는 경우가 많다.

⑬ Template Pattern

- 공통적인 행위를 하는 여러 클래스들의 집합을 정의한 패턴
- abstract class를 정의하고 내부에 공통 행위를 abstract method로 기술한다
- 여기까지는 상속의 개념과 동일하다
    - 또한 abstract class내에는 정의된 abstract method를 이용해 처리하는 로직 메소드를 담는다
    - 이를 template method라고 한다

- template method내에는 business logic, algorithm 이 포함
- concrete class(sub class) 내에 구현되어 있을 abstract method를 호출함으로써 동작한다

- 장점
  - 코드 중복 방지
  - 기능의 추가 및 변경이 쉽다
  - 새로운 기능 추가 시 concrete class 만 새로이 정의하면 ok
- 사용 예: java.io.*
  - InputStream, OutputStream class
    - public abstract int read() throws IOException 이 abstract method임
    - template method
      - read (byte[], int, int) .... 여러 버전의 메소드
      - skip()
    - 아랫단에 이를 상속받은 FileInputStream class에서 read(byte[], 10, 20)을 사용
    - SequenceInputStream, PipedInputStream 을 비롯한 몇 가지 클래스 모두 동일한 구조

⑭ Single Threaded Execution pattern
- 단일 쓰레드의 접근만을 허용하는 패턴
- 예) java.util.Hashtable, Vector
- 이를 실제로 구현하기 위해서는 synchronized(), synchronized method를 이용한다.

⑮ Abstract Factory Pattern
- client의 interface는 ConnectionFactory 이며 실제로 이를 구현한 다양한 ConnectionFactoryImpl class 정의
- 실제 client에게 반환되는 객체는 FactoryImpl에 의해 생성된 ConnectionImpl 객체임
- JDBC의 Connection, Connector architecture에 적용
- 구성
  - ConnectionFactory: abstract factory
  - ConnectionFactoryImpl: factory
  - Connection: abstract product
  - ConnectionA: product

4. TOOL

OOAD방식으로 개발을 진행함에 있어 UML툴의 차지하는 비중은 상당한 부분입니다. 특히 설계와 디자인에 있어 시간을 단축시켜주고 소프트웨어의 이해도를 높이는데 큰 역할을 하기 때문에 많은 툴들이 시중에 나와있고 또 사용되고 있습니다.

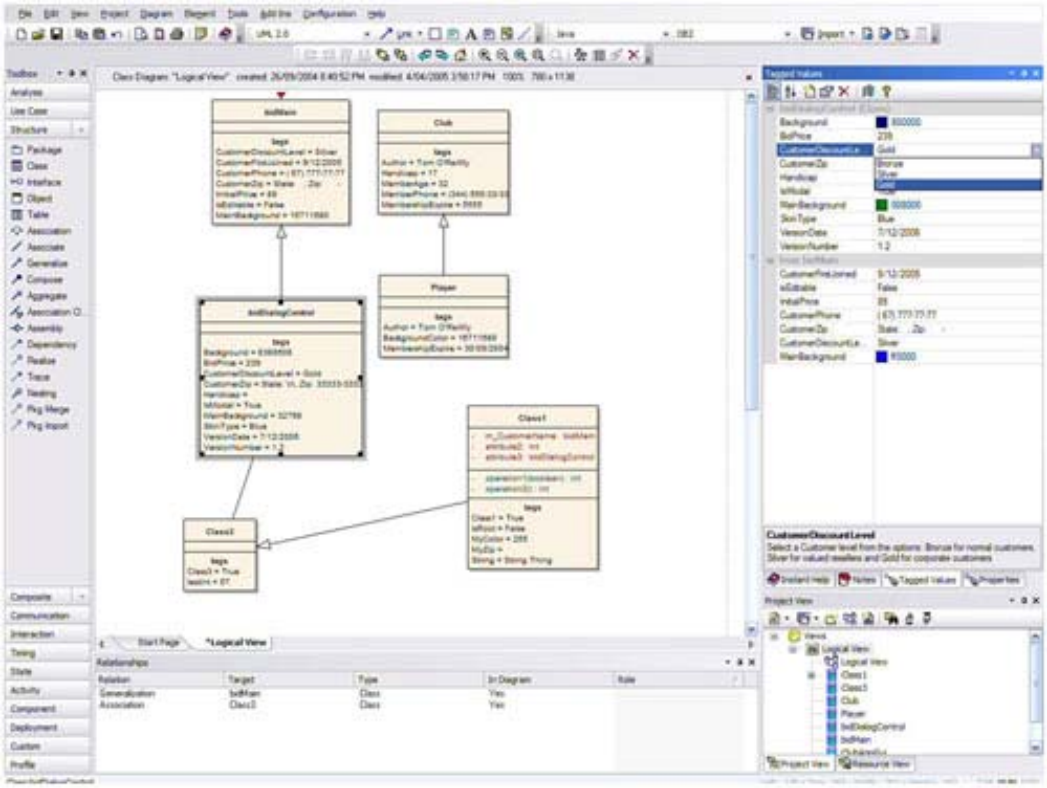
(1) 상용 툴

- IBM RSA(Rational Software Modeler/Architect)
- Together Architect/Designer/Developer (Borland)
- MagicDraw (No Magic)
- Visual Paradigm for UML (Visual Paradigm)
- EA(Enterprise Architect)
- Power Designer
- Visio

역사적으로 볼 때 가장 대표적인 도구이며 UML 툴의 대명사라고 할 수 있는 제품이 Rational사의 **ROSE**인데, IBM으로 넘어가면서 기존 Rose와 함께 **RSA(Rational Software Architect)**라는 이름으로 발전을 거듭하고 있습니다.

Borland의 Together 역시 대표적 UML 도구의 하나이며, **MagicDraw, Visual Paradigm**도 많이 알려지고 사용되는 도구입니다. **Power Designer**는 Data Modeling, BPM(Business Process Modeling), UML등을 지원하는데, CA ERwin Data Modeler와 함께 UML 보다는 Data Modeling Sofrware Tool로서 더욱 잘 알려져 있습니다. **Visio**의 경우 UML 도구라기보다는 Business Graphics Software로 분류하는 것이 더 적합했었지만, Visio 2000 이상 버전부터 UML 1.2를 지원하고 있으며, Visio 2007의 경우 UML 2.0을 지원하며 지속적으로 UML 관련 기능을 강화하고 있어 현재 여타 UML 툴들과 비교하여 손색이 없습니다.

마지막으로 **EA(Enterprise Architect)**는 고가인 다른 UML 도구들에 비해 상대적으로 저렴한데다 기능적으로도 손색이 없어 **가격대비 성능이 우수**하다고 알려져 있습니다



Enterprise Architect Screenshot

(2) 오픈소스 툴

- StarUML
- ArgoUML
- JUDE/Community
- TOPCASED-UML2 (TOPCASED Modeling Framework Open Source Project)
- MDT-UML2Tools

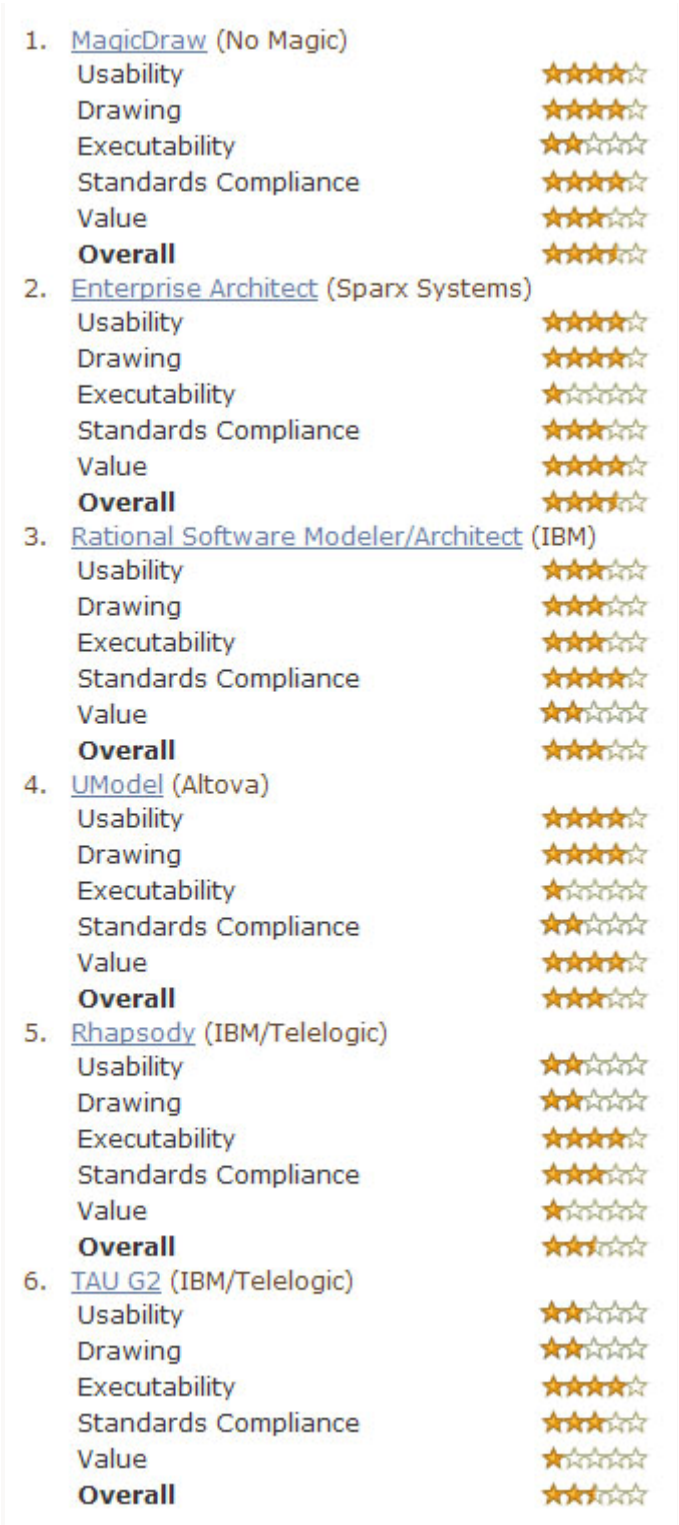
StarUML 은 국내의 플라스틱 소프트웨어(Plastic software)사에서 개발한 것으로, 이전에 "플라스틱(Plastic)" 혹은 "아고라 플라스틱(Agora Plastic)"이란 이름으로 더 잘 알려진 도구인데, 이것을 2005 년에 오픈소스로 전환한 것입니다. UML 2.0 을 지원하고 있지만 아쉬운 것은 더이상 버전업이 진행되지 예정되어 있지 않습니다..

고가의 상용 제품들에 비해서 다소 부족하지만 StarUML 과 함께 오픈소스로서 비교적 강력한 UML 모델링을 지원하는 도구로 언급되는 것으로 **ArgoUML** 이 있다. ArgoUML 은 Java 기반으로 다양한 플랫폼을 지원합니다.

이들 외에도 개발도구인 **Eclipse** 나 **NetBeans** 를 위한 **plug-in** 형태로 제공되는 UML 툴들이 많이 있습니다.

(3) 비교분석

Usability, Drawing, Executability, Standards Compliance, Value 등의 항목별로 비교할 때, **MagicDraw**와 **Enterprise Architect**의 점수가 가장 높게 나와있습니다.



Micro Software [2007년 5월호]의 다시 꺼낸 양날의 검 UML이란 글에서 몇가지 UML 도구들에 대해서 비교하고 있는데, 해당 내용을 참고하여 특징들을 정리하자면 다음과 같습니다.

- IBM **Rational ROSE**
  - 모델링 도구의 표준임
  - RUP, UP 방법론을 적용
  - SODA 를 통한 산출물 자동화 기능 제공
- Borland **Together Architect**
  - 모델링과 개발을 동시에 지원
  - 강력한 Reverse Engineering
  - 산출물 자동화 템플릿 지원
- Plastic software **StarUML** (Open Source)
  - 오픈소스임에도 불구하고 상용 제품 수준의 기능을 보유
  - UML Profile 기반의 모델링 지원
  - 모델링 플랫폼을 제공
- Sparx Systems **Enterprise Architect**
  - 초경량의 모델링 도구
  - 강력한 협업 기능 제공
  - 모델 파일을 DB 형태로 관리
- Microsoft **Visio**
  - 범용적인 Diagram 작성 도구
  - 탁월한 Diagram 작성 기능

약간 모호하다고 생각되지만, 같은 글에서 정리한 내용을 보면 각 도구의 활용분야는 다음과 같이 분류할 수 있습니다.

Rational ROSE	RUP/UP 기반의 개발 방법론을 적용하는 중대형 규모의 프로젝트
Together Architect	Reverse Engineering 을 적극 활용할 수 있는 프로젝트
StarUML	상용이 아닌 오픈소스를 활용하고자 하며, 상용 수준의 모델링 작성이 필요한 프로젝트
Enterprise Architect	모델링 작업의 협업 및 도구의 퍼포먼스가 중요한 프로젝트
Visio	범용적인 Diagram 작성이 필요한 프로젝트

정리하자면

프로젝트에서는 상황에 따라 의지와 무관하게 툴이 선정될 수도 있고, 여건에 따라 오픈소스 형식의 툴을 사용해야만 하는 상황도 생깁니다. 가능하다면 프로젝트의 규모를 고려하고 또한 각 툴의 특징을 파악하여, 활용도를 극대화할 수 있는 도구를 선정하는 것이 필요할 것입니다.

고작해야 CBD 방법론의 산출물 구색을 맞출 용도로 앞뒤도 맞지않는 Use Case, Activity, Class, Sequence Diagram 몇 개를 그려 넣으려고 고가의 도구를 도입하는 것은 좋지 않은 예가 될 것입니다

5. STAR UML

가장 많이 쓰이는 무료 UML툴로써 STARUML이 있습니다. 국내의 개발자들이 만든 제품으로 현재는 개발이 중단되었지만 오픈소스이기 때문에 아직도 많은 개발자들이 수정에 수정을 거듭하여 발전을 거듭하고 있는 제품입니다.

고객에 적응하는 UML 도구

StarUML™은 고객의 환경에 최대한 적응할 수 있도록 설계되어 있습니다. 따라서, 고객의 소프트웨어 개발 방법론, 프로젝트의 플랫폼, 언어 등에 모두 적응할 수 있는 커스터마이징 변수들을 제공합니다.

진정한 MDA 지원 도구

소프트웨어 아키텍처는 향후 10년 이상 내다보는 매우 중요한 작업입니다. OMG에서는 MDA 기술을 통해서 플랫폼에 독립적인 소프트웨어 모델을 구성하고 그것으로부터 플랫폼에 의존적인 모델이나 코드 등을 자동으로 얻을 수 있도록 하는 것을 지향하고 있습니다. StarUML™은 UML 1.4 표준 메타모델과 2.0 표기법을 최대로 준수하면서 UML Profile 개념을 제공하여 플랫폼에 독립적인 모델을 작성할 수 있도록 지원하며, 간단한 템플릿 문서 작성만으로 고객이 원하는 산출물을 쉽게 얻을 수 있습니다.

놀라운 확장성과 유연성

StarUML™은 놀라운 유연성과 확장성을 제공합니다. 도구의 기능을 확장하기 위한 Add-In 프레임워크를 제공하고, COM Automation을 통한 모델/메타모델 및 도구의 모든 기능에 접근할 수 있으며, 메뉴 및 옵션 항목까지도 확장할 수 있도록 설계되어 있습니다. 또한 고객의 방법론에 맞도록 접근법(Approach) 및 프레임워크(Framework)를 직접 추가 작성할 수 있고 어떠한 외부 도구와도 통합이 가능합니다.

STAR UML의 최소사양은 다음과 같습니다.

- Intel® Pentium® 233MHz or higher
- Windows® 2000, Windows XP™, or higher
- Microsoft® Internet Explorer 5.0 or higher
- 128 MB RAM (256MB recommended)
- 110 MB hard disc space (150MB space recommended)
- CD-ROM drive
- SVGA or higher resolution monitor (1024x768 recommended)
- Mouse or other pointing device

STAR UML은 다음과 같은 특징이 있습니다

특징	내용
정확한 UML 표준 모델	OMG 에서 제정한 UML 의 표준 명세에 따라 소프트웨어 모델을 작성할 수 있도록 합니다. 외국산 제품과 같이 변칙적이며 벤더에 의존적인 UML 구문과 의미는 설계한 정보의 지속성을 향후 10 년이상 내다 볼 때 매우 위험한 선택이 될 수 있습니다. StarUML™은 UML 1.4 표준 구문과 의미의 준수를 극대화 하였으며, 견고한 메타모델의 기반에서 UML 2.0 의 표기법을 적극적으로 수용하였습니다.
개방적 소프트웨어 모델 포맷	독자적인 포맷으로 작성된 모델의 활용성을 크게 떨어뜨리는 외국산 제품과는 달리 StarUML™에서의 모든 파일의 포맷은 XML 로 구성됩니다. 또한 사람이 쉽게 식별할 수 있는 형태로 표현되어 있어서 누구든지 XML 파서를 이용해서 포맷을 원하는 형태로의 변환 및 사용이 가능합니다. XML 은 세계 표준인 만큼 장기적인 안목으로 본다면 10 년 이상 지속될 수 있는 소프트웨어 모델이 될 수 있습니다.
진정한 MDA 지원 도구	UML 프로파일(UML Profile)을 완벽하게 지원하여 UML 의 확장성을 극대화시킴으로써 금융, 국방, e-비즈니스, 보험, 항공우주 등 어떠한 영역의 애플리케이션도 모델링이 가능하게 됩니다. 진정한 의미의 플랫폼 독립적인 모델(PIM – Platform Independent Model)의 작성을 가능하게 하고 그로부터 플랫폼 의존적인 모델(PSM – Platform Specific Model)이나 각종 문서, 실제 실행 가능한 코드(Executable Code)를 얼마든지 자동으로 생성할 수 있습니다.
방법론 및 플랫폼의 적응성	StarUML™은 접근법(Approach)이라는 개념을 도입하여 어떠한 방법론/프로세스에도 적응할 수 있는 환경을 만들 수 있습니다. .NET, J2EE 와 같은 플랫폼의 애플리케이션 프레임워크(Framework) 모델 뿐만 아니라 소프트웨어 모델의 기본 구조(e.g. 4+1 뷰-모델 등)를 쉽게 정의할 수 있습니다.
뛰어난 확장성	StarUML™ 도구의 모든 기능이 Microsoft 의 COM 자동화(Automation)가 되어 있어 어떠한 COM 지원 언어(Visual Basic Script, Java Script, VB, Delphi, C++, C#, VB.NET, Python, ...)에서도 StarUML™을 제어하고 또한 통합된 추가 모듈을 개발할 수 있습니다.
소프트웨어 모델 검증 기능	사용자는 소프트웨어 모델링을 수행하는 동안 많은 실수를 범하게 됩니다. 이러한 실수가 최종 코딩단계로 그대로 전가될 때에는 더 큰 위험을 초래할 수 있습니다. 이를 방지하게 위하여 StarUML™은 사용자가 개발한 소프트웨어 모델을 자동으로 검증(Verification)하여 사전에 오류를 발견하게 함으로써 더욱 견고하고 완벽한 소프트웨어 설계를 수행할 수 있도록 도와줍니다.
유용한 Add-In 들	StarUML™ 은 프로그래밍 언어의 소스코드를 생성하거나 소스코드를 모델로 변환하는 기능을 제공하는 다수의 Language Add-In 들과 Rational Rose 파일 읽기, XMI 를 통한 도구간 모델링 정보 교환, 그리고 디자인 패턴 지원 등의 각각의 기능을 제공하는 유용한 Add-In 들을 빌트-인(Build-In)으로 제공합니다. 이런 Add-In 들을 활용하여 모델링한 정보의 재사용성, 생산성, 가용성, 상호 운용성을 높이십시오.

이제 STAR UML에서 손쉽게 코드가 생성되는 과정을 보겠습니다.

(1) 새 프로젝트 만들기

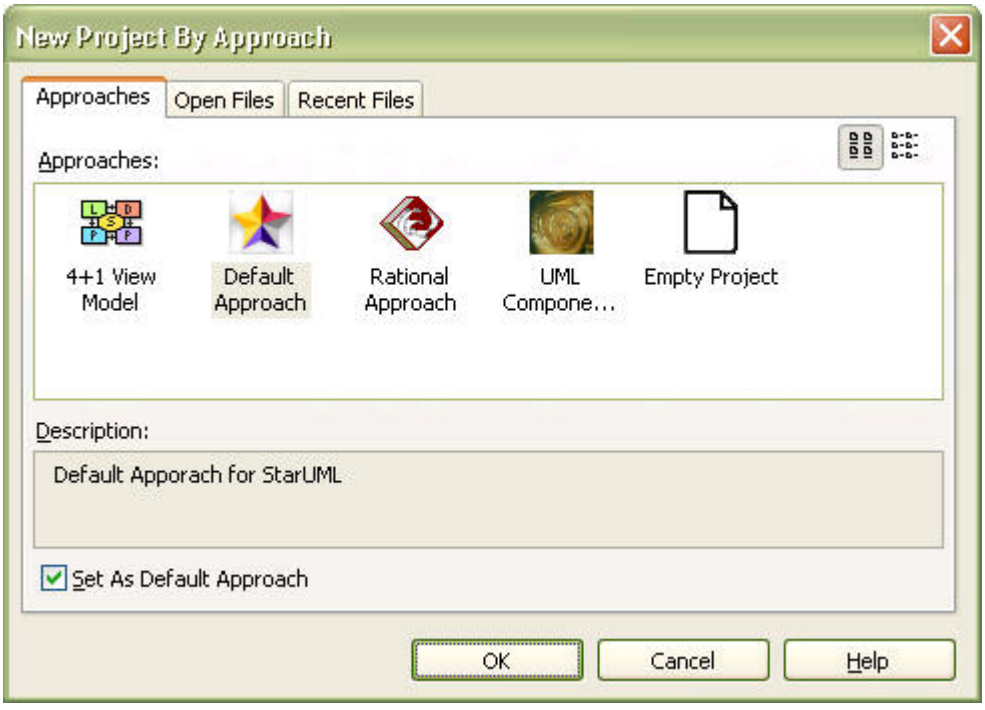
새로운 소프트웨어 프로젝트를 시작하려면 새 프로젝트를 만들어야 합니다. 새 프로젝트는 아무런 초기화도 되어있지 않은 비어있는 프로젝트를 만들거나 접근법에 따라 서로 다르게 초기화된 프로젝트를 만들 수 있습니다.

새 프로젝트를 만드는 방법 1 - 새 프로젝트:

1. **[File]->[New Project]** 메뉴를 선택합니다.
2. 사용자가 기본 접근법으로 선택한 접근법으로 초기화된 프로젝트가 바로 만들어집니다. 접근법의 종류에 따라 프로파일이 포함되고, 프레임워크가 로딩될 수 있습니다.

새 프로젝트를 만드는 방법 2 - 새 프로젝트 선택 대화상자:

1. **[File]->[New Project By Approach]** 메뉴를 선택합니다.
2. 새 프로젝트 선택 대화상자의 접근법 선택 페이지에 사용 가능한 접근법들이 목록에 나타납니다. 이 중에서 한가지를 선택하고 **[OK]** 버튼을 누릅니다.

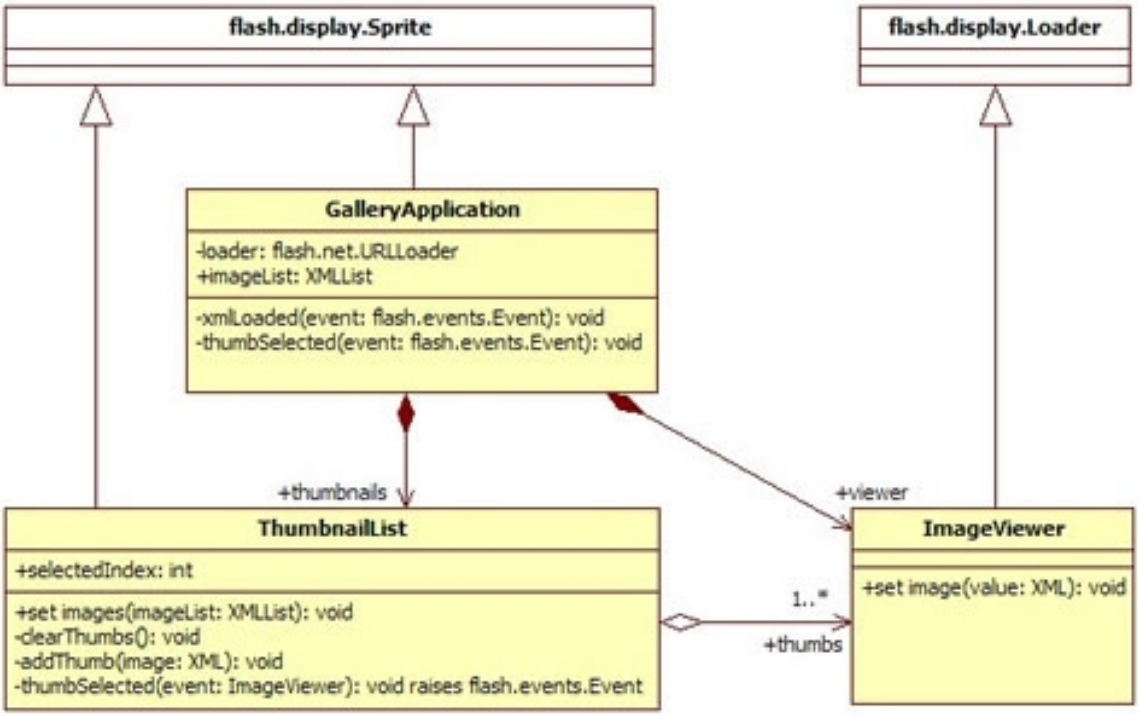


3. 선택한 접근법에 따라 프로젝트가 생성되고 초기화됩니다. 접근법의 종류에 따라 프로파일이 포함되고, 프레임워크가 로딩될 수 있습니다.

(2) 새 다이어그램 생성하기

http://staruml.sourceforge.net/docs/user-guide(ko)/toc.html

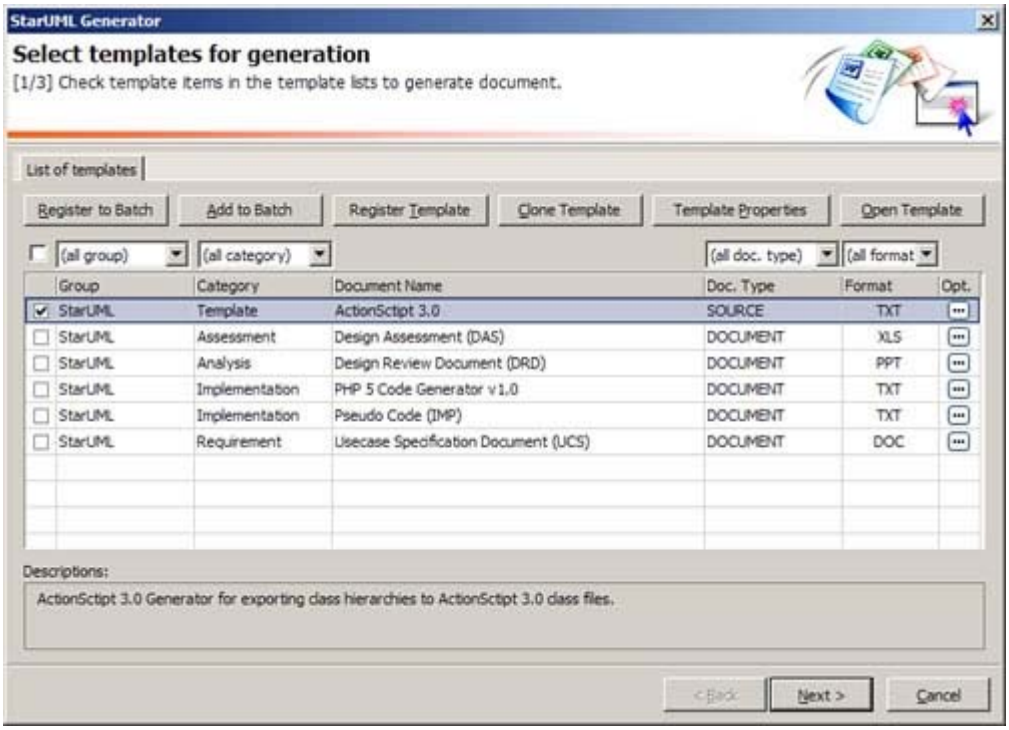
를 참조하여 아래와 같은 다이어그램을 만듭니다.



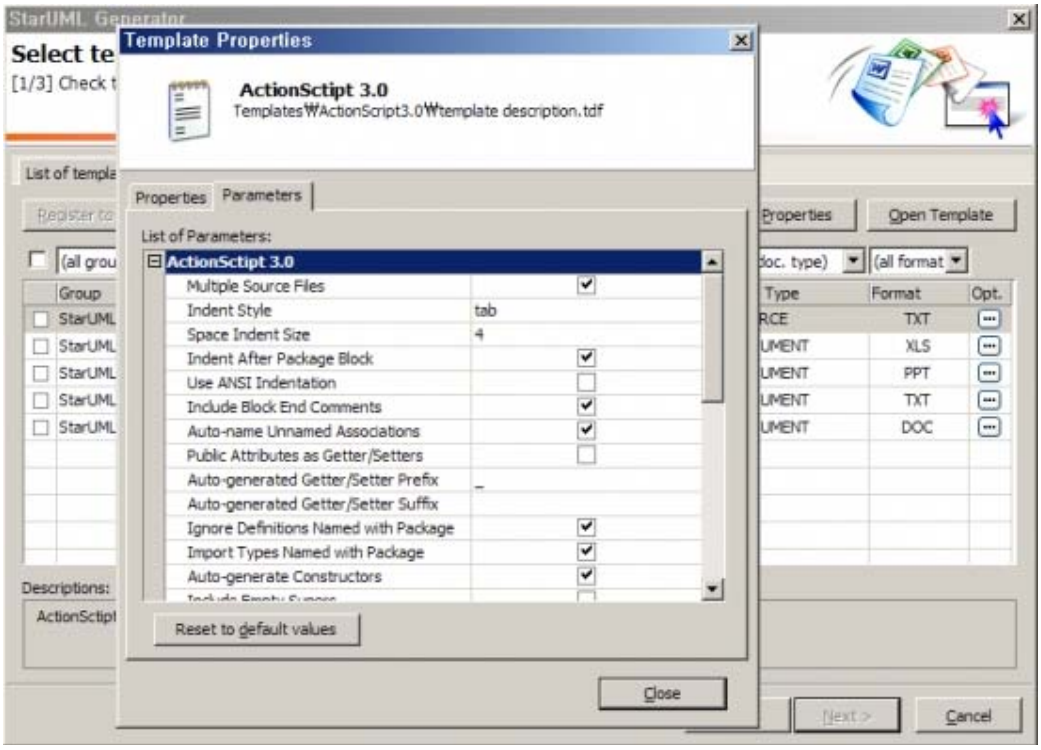
위와 같은 예제를 만들면

(3) 코드 생성준비

메뉴에서 Tools > StarUML Generator... 를 선택하면 아래와 같은 창이 나오고 여기서 ActionScript 3.0 템플릿을 선택하고 Next 버튼을 누릅니다.

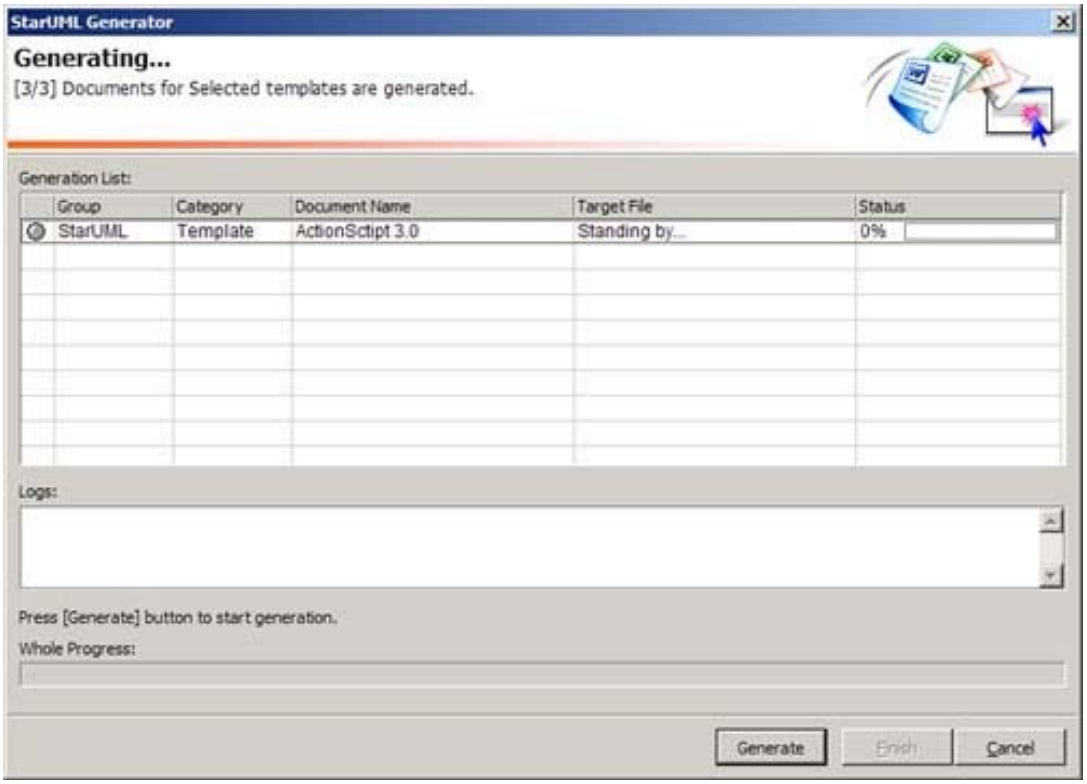


Opt.에 ...버튼을 누르면 템플릿 속성을 임의로 지정할 수 있습니다. 생성자를 기본적으로 만들 것인지 아닌지, Event 에 toString()함수를 만들 것인지 등을 선택할 수 있습니다.



(4) 코드생성

Generate 버튼을 누르면 지정된 폴더에 만든 클래스 다이어그램을 바탕으로 ActionScript 3.0 코드가 만들어집니다.



이하는 실제로 생성된 코드의 예시 입니다.

```
package com.example {
	public class ThumbnailList extends flash.display.Sprite {
		private var _selectedIndex:int;
		public function get selectedIndex():int  {
			return this._selectedIndex;
		}
		public function set selectedIndex(value:int):void  {
			this._selectedIndex = value;
		}

		private var _thumbs:Array = [];
		public function get thumbs():Array  {
			return this._thumbs;
		}
		public function set thumbs(value:Array):void  {
			this._thumbs = value;
		}

		public function ThumbnailList() {
		}

		public function set images(imageList:XMLList):void {
		}

		private function clearThumbs():void {
		}

		private function addThumb(image:XML):void {
		}

		private function thumbSelected(event:ImageViewer):void {
		}

	} // end class
} // end package
```

```
package com.example {
                    public class GalleryApplication extends flash.display.Sprite {
                    private var loader:flash.net.URLLoader;

                    private var _imageList:XMLList;
                    public function get imageList():XMLList   {
                              return this._imageList;
                    }
                    public function set imageList(value:XMLList):void   {
                              this._imageList = value;
                    }

                    private var _thumbnails:ThumbnailList;
                    public function get thumbnails():ThumbnailList   {
                              return this._thumbnails;
                    }
                    public function set thumbnails(value:ThumbnailList):void   {
                              this._thumbnails = value;
                    }

                    private var _viewer:ImageViewer;
                    public function get viewer():ImageViewer   {
                              return this._viewer;
                    }
                    public function set viewer(value:ImageViewer):void   {
                              this._viewer = value;
                    }

                    public function GalleryApplication() {
                    }

                    private function xmlLoaded(event:*):void {
                    }

                    private function thumbSelected(event:*):void {
                    }

          } // end class
} // end package
```