Introduction to UML

201014184 김도윤
201111347 김태호
201111367 여승훈

1 UML

1.1 UML 정의

UML 이란 Unified Modeling Language 의 약자로 객체지향 분석(Analysis)와 설계(Design)를 위한 modeling Language 이다. 객체 기술에 관한 국제 표준화 기구인 OMG(Object management Group)에서 1997 년에 표준으로 채택한 모델링 언어이다. UML 은 모델링 언어일 뿐 Method 는 아니다. Method 는 프로세스에 대한 정의와 각각의 업무들에 대한 지침과, 업무들 간의 순서를 명시해야 하는 반면, 모델링 언어는 표기법만을 제시하는 것이다. 따라서 UML 은 소프트웨어 개발에 사용하기 위한 여러 다이어그램들을 정의하고 있으며, 또 다이어그램들의 의미들에 대해 정의하고 있다.

.

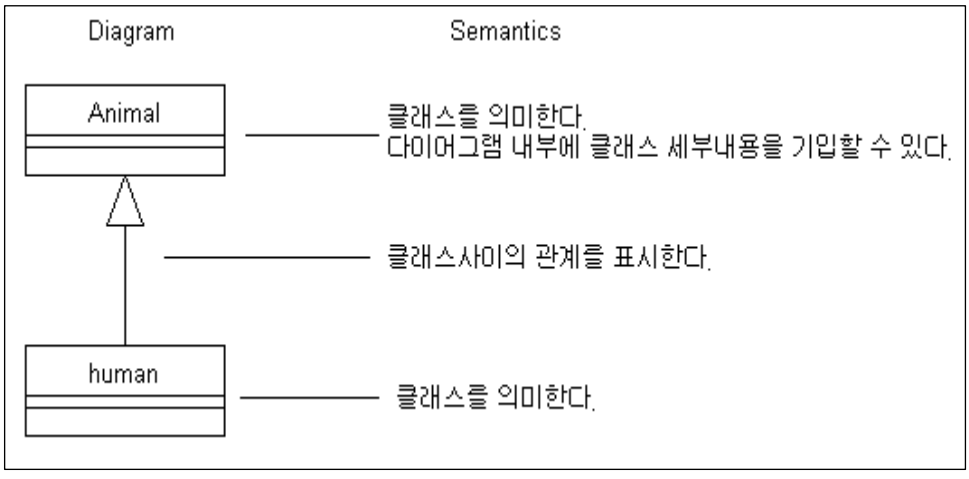
1.2 UML 역사

UML 은 Rational 사의 Grady Booch, James Rumbaugh 에 의해 1994 년 10 월에 처음 개발에 착수되었다. 이후 1995 년 년 10 월에 Unified Method 0.8 의 명칭으로 OOPSLA'95 에서 발표되었으며, 이후 Ivar Jacobson 이 UML 개발에 함께 협력하면서 1996 년에 버전 0.9 를 발표하였고, 1997 년 11 월에 UML 1.1 이 OMG 에 의해 표준으로 채택되었다.

1.3 UML 표기 및 의미

UML 에서는 표기하려는 대상을 diagram 을 사용하여 나타내고 그 대상에 의미를 부여한다.

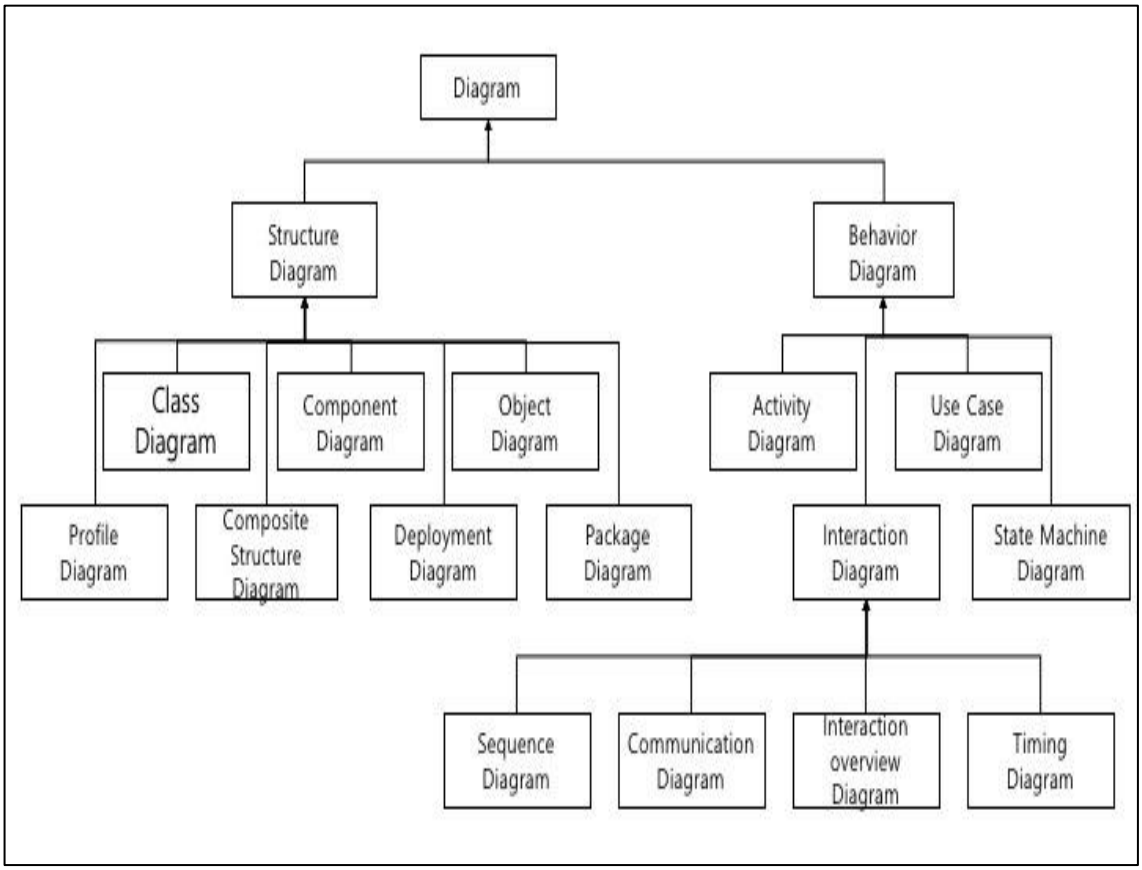
Example)



2 UML 구성

2.1 구성

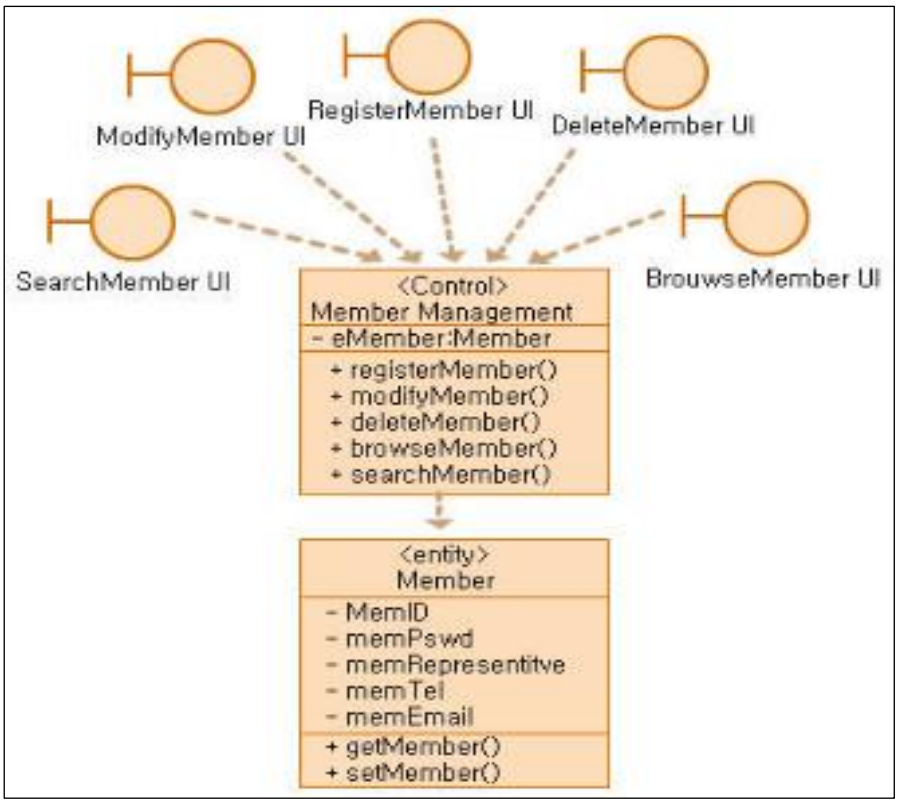
UML 은 Structure Diagram 7 개, Behavior Diagram 7 개로 총 14 종류의 다이어그램이 있다. Structure Diagram 은 시스템의 개념, 관계 등의 측면에서 요소들을 나타내고 각 요소들의 정적인 면을 보기 위한 것이고, Behavior Diagram 은 각 요소들 혹은 요소들간의 변화나 흐름, 주고받는 데이터 등의 동작을 보기 위한 것이다.



2.2 Class Diagram

클래스 내부의 정적인 내용이나 클래스 사이의 관계를 표기하는 다이어그램으로 시스템의 일부 또는 전체의 구조를 나타낼 수 있다. Class Diagram 은 의존 관계를 명확히 보게 해주며, 순환 의존이 발생하는 지점을 찾아내서 어떻게 이 순환 고리를 깨는 것이 가장 좋은지 결정할 수 있게 해줍니다 즉 클래스들을 표현하고 그 클래스들의 정적인 관계(associated, dependent, specialized, packaged)를 표현한다. 이러한 정적인 요소는 시스템의 life cycle 과 수명을 같이하며 하나의 시스템은 여러 개의 class diagram 으로 표현이 가능하다.

Example)



· 제일 위 쪽의 클래스들은 boundary class 로 UI(User Interface) 화면을 제공한다.

· Boundary class 들과 Member Management 클래스는 종속 관계(Dependency)에 있다.

· Member Management 클래스는 Member 클래스와 종속 관계가 있다.

· 즉 Boundary class 들이 Member Management 클래스에 처리를 의뢰하고 Member Management 클래스는 Member 클래스들에게 처리를 의뢰하는 관계를 모델링 한 내용이다.

2.3 Component Diagram

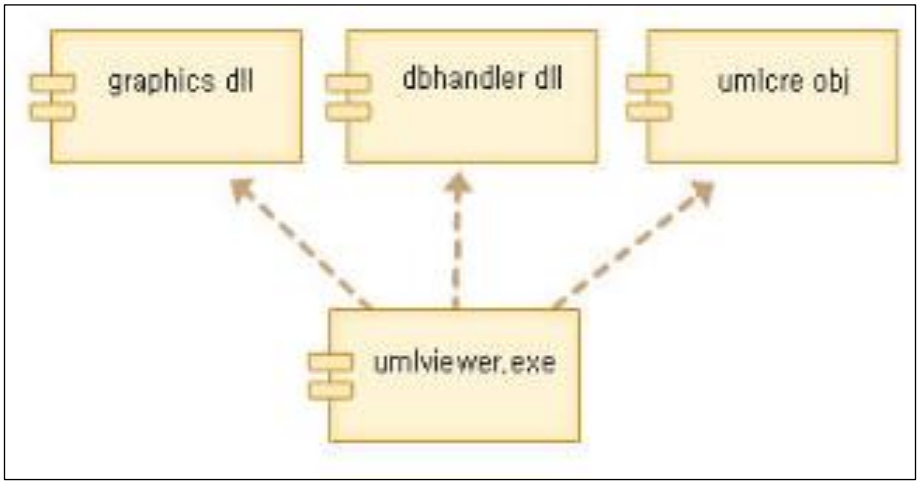
현대의 소프트웨어 개발 추세는 컴포넌트 중심으로 되어가고 있다. 팀 단위 프로젝트라면 특히 중요한 것이 컴포넌트이다.

Component Diagram 은 시스템을 물리적으로 볼 수 있도록 한다. Component Diagram 의 목적은 소프트웨어가 시스템의 다른 소프트웨어 Component 들에 대해 소프트웨어가 가지고 있는 종속관계를 보여준다.

Component Diagram 은 고급레벨에서 볼 수 있거나, Component 패키지 레벨에서 확인 할 수 있다. 위 그림에 나타난 기호는 어색하다는 반응이 많아. UML 2.0 부터는 스테레오 타입으로 표기법을 바꿨다.

Component Diagram 은 시스템의 소프트웨어 구성, 파일구성, 내부 물리구성을 표현하므로 소프트웨어 모듈간의 의존관계나 소프트웨어에 할당되는 리소스 등이 명확해진다.

Example)

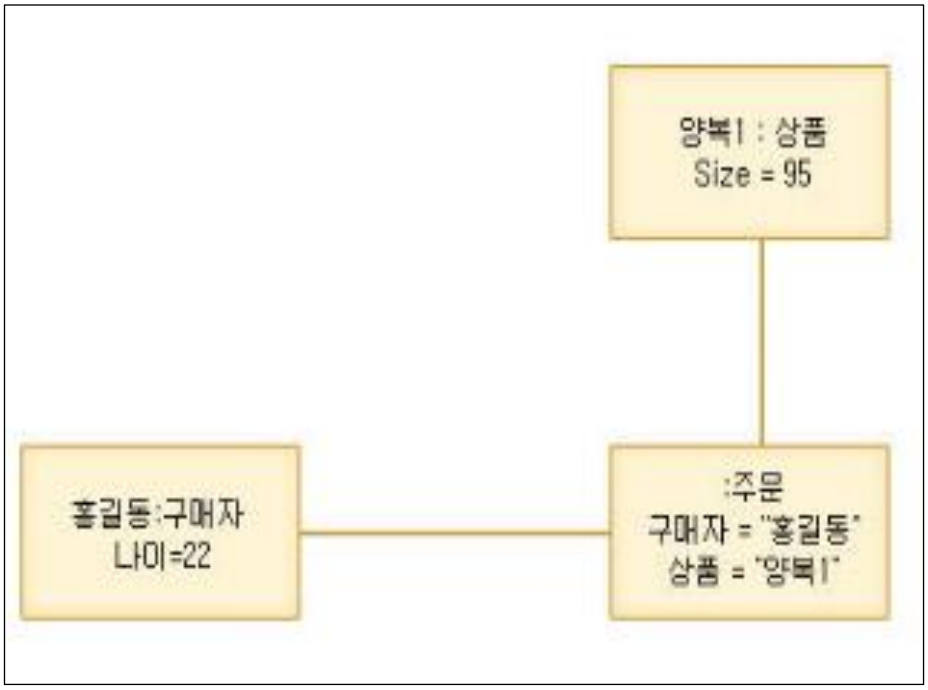


- UML Viewer 라는 시스템의 Component 구조를 표현한 것이다.
- umlviewer.exe 라는 실행 모듈이 동작하면서 graphics.dll, dbhandler.dll, umlcore.obj 등의 Component 들에게 서비스의 실행을 요청한다는 것을 모델링 한 것이다.

2.4 Object Diagram

Object diagram 과 class diagram 의 변형으로 class diagram 과 거의 같은 표기법을 사용한다. 하지만 class diagram 과의 차이점은 표현되는 것이 클래스 대신 실제 인스턴스화 된 객체를 표현한다. 이것은 클래스 다이어그램의 실행 시에 나타나는 하나의 예를 나타낸 것이다

Example)



· 구매자, 주문, 상품의 클래스들이 instance 를 생성한 Object Diagram 이다.

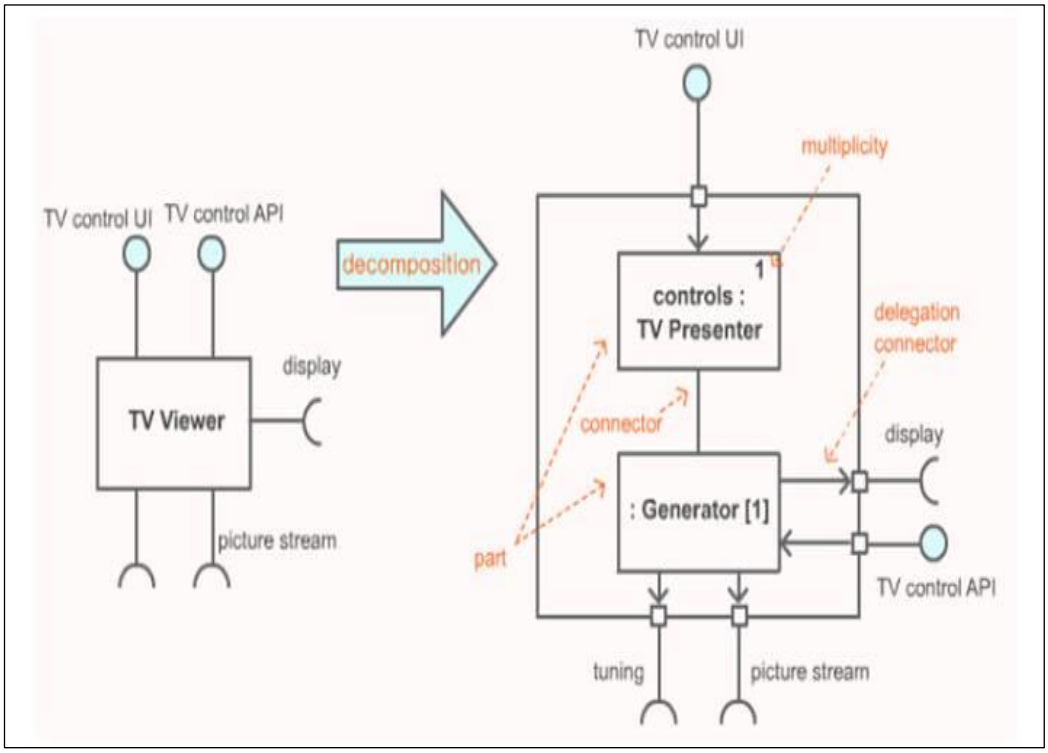
· 객체를 표현하였기 때문에 객체가 갖는 구체화된 Value 들이 나타나고, 이는 하나의 '시점'으로 고정된다.

· Object Diagram 은 객체와 그 관계의 Snapshot 을 나타낸다.

2.5 Composite Structure Diagram

하나의 클래스를 내부 구조로 분해할 수 있도록 한다. 복잡한 객체를 분할하여 여러 부분으로 나누어 표현한다. 또한 Run-time Grouping 을 나타내며 하나의 Component 가 어떻게 내부적으로 분할 되는가를 표현하는데 사용되므로 Component Diagram 과 공통적으로 사용하는 표기법들이 많다.

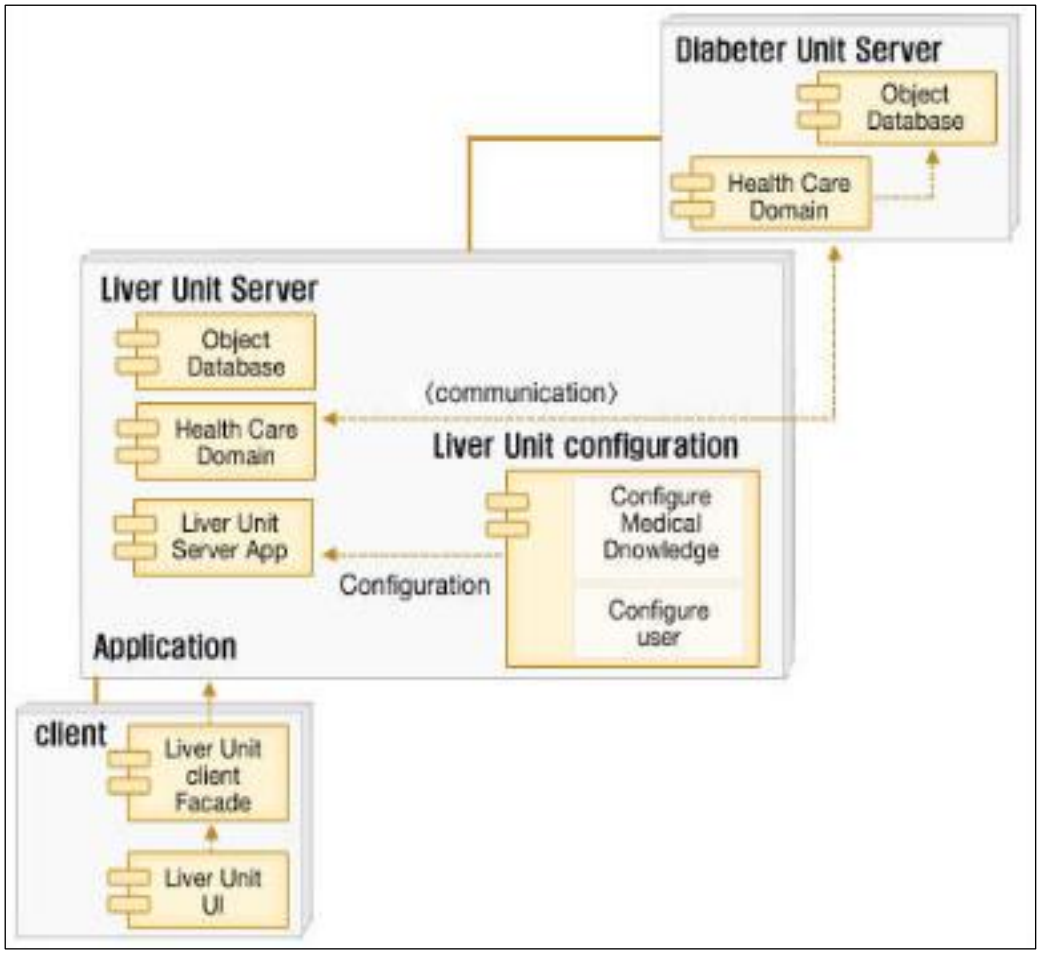
Example)



2.6 Deployment Diagram

하드웨어 시스템들은 각각 고유의 특성을 가지고 있다. 이런 환경에서 분산환경에서의 시스템 구축은 시스템마다의 고유 특성을 담고 시스템간의 관계를 표현한 diagram 을 필요로 하게 된다. 이 때 필요한 것이 각 시스템마다의 하드웨어, 소프트웨어 컴포넌트들의 관계를 나타낸 deployment diagram 이다. Deployment diagram 은 node 라는 notation 으로 computation unit(대부분 하드웨어적인 부분)을 나타낸다.

Example)



· 시스템의 물리적 실행환경이 Client, Liver Unit Server, Diabetes Unit Server 의 세가지로 구성됨을 표현한 Deployment Diagram 이다.

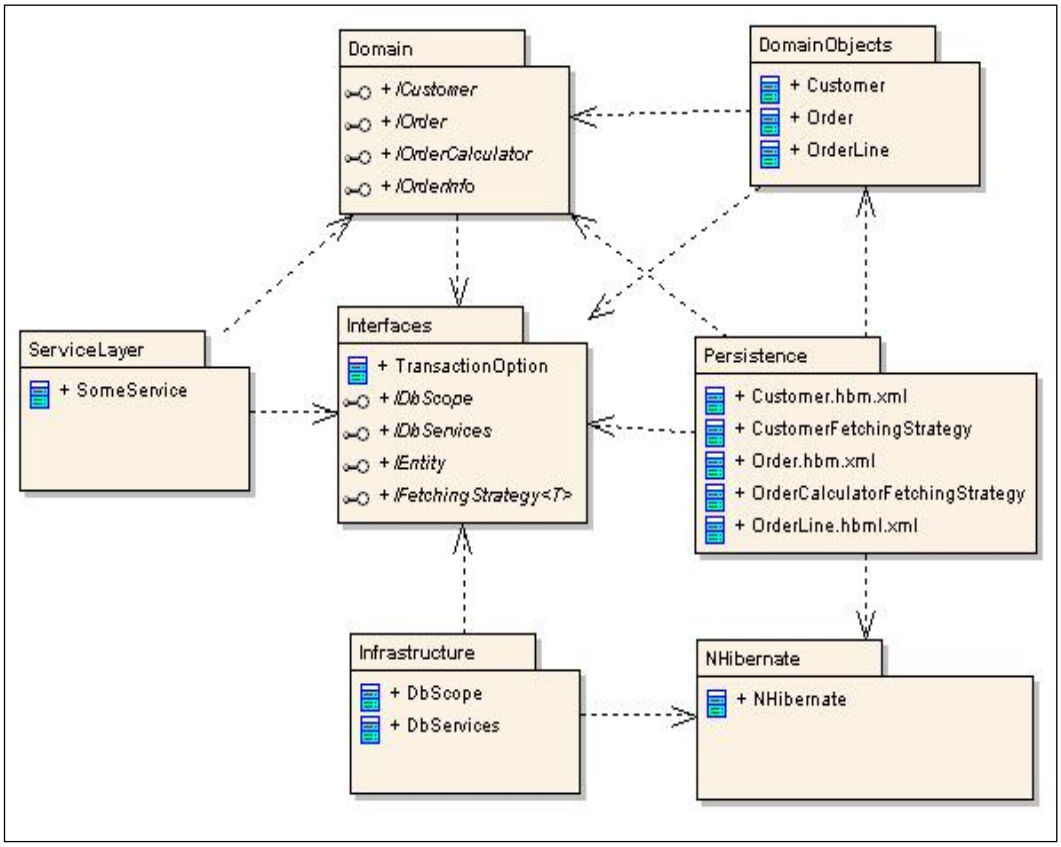
· 각 서버 별로 SW 컴포넌트의 배치 상황을 모델링 한다.

2.7 Package Diagram

"거대한 소프트웨어를 어떻게 작게 나눌 것인가?" 이 말은 소프트웨어 방법론에서 문제시하고 있는 가장 오래된 질문 중 하나이다. 이 문제의 해결을 위해 초기 구조적 방법론에서는 기능적으로 분해(functional decomposition)하는 방법을 사용하였다. 시간이 지남에 따라 여기에 process 와 data 의 분리현상이 일어났고 이것이 기능적인 분해의 방법에도 적용이 되었다. 하지만 클래스란 개념의 등장으로 이러한 방법을 대치하게 되었고 클래스들을 상위개념으로 묶는 것이 문제시되었다. UML 에서는 이러한 상위 개념으로의 그룹화 방법으로 package 란 방법을 제시하였다.

Package 란 하나의 클래스가 아닌 system 에서의 하나의 modeling element 가 된다. 쉬운 예로 하나의 element 를 modeling 하기 위해 하나의 클래스 diagram 을 작성하였다면 이 class diagram 이 하나의 package 가 된다. Package diagram 은 이러한 package 와 package 들 사이의 의존관계(dependency)를 나타낸다.

Example)

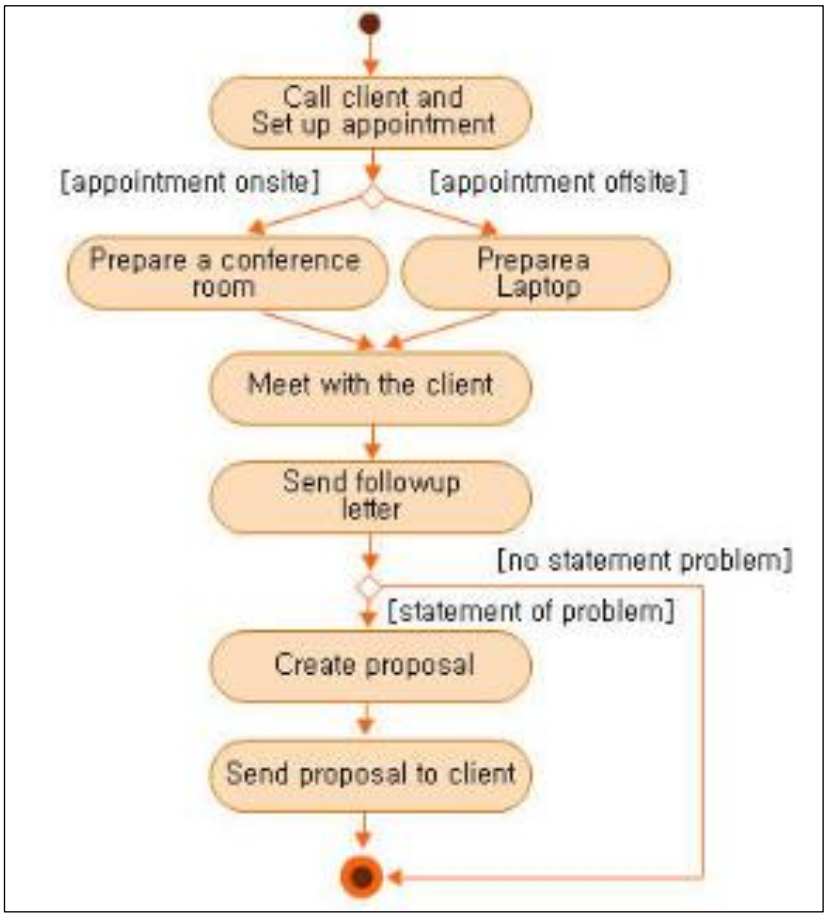


2.8 Activity Diagram

Activity diagram 은 기존의 다른 diagram 처럼 기존의 전통을 가진 것이 아니다. Jim Odell 의 event diagram, SDL state modeling techniques, Petri nets 등의 여러 가지 이론이 섞여서 만들어 진 것이다. 이런 다이어그램은 순서도나 병렬적인 처리를 요하는 행위를 표현할 때 사용하면 유용한 것이다.

Activity diagram 은 순서에 따른 activity 를 나타내는 것으로 모델링하고 있는 diagram 에서의 activity 의 의미를 파악하는 것이 중요하다. activity 의 의미로 개념적인 다이어그램에서는 activity 는 인간이나 컴퓨터에 의해 수행이 필요한 어떠한 업무(task)를 의미하고, 상세화(specification)하는 다이어그램이나 구현(implementation)을 위한 다이어그램의 경우 activity 는 class 의 method 가 된다.

Example)



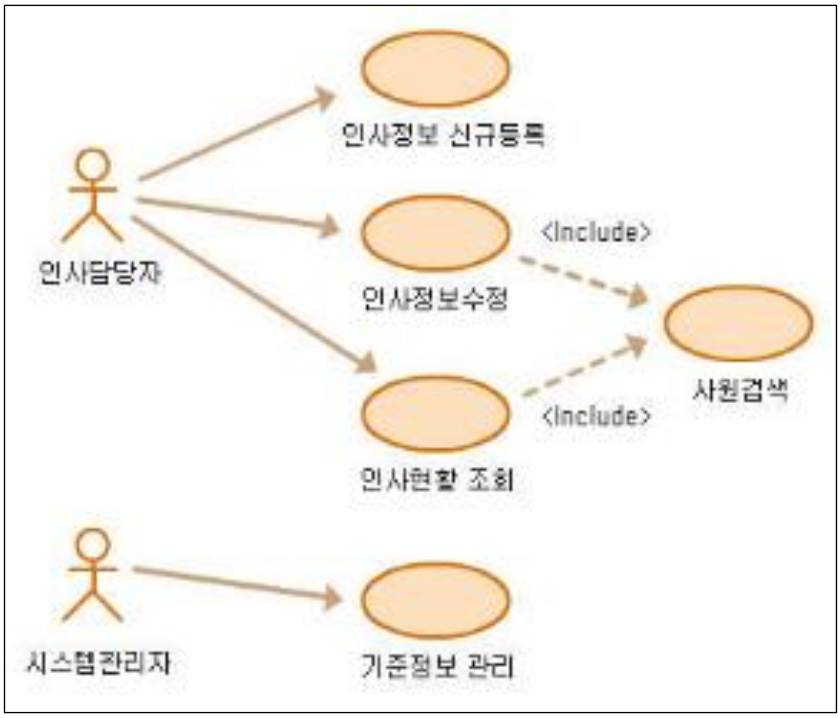
· 어느 회사 영업사원의 영업 프로세스를 정의한 Activity Diagram 이다.

· 영업사원이 고객을 만나서 미팅을 하고, 제안서를 고객에게 보내기까지의 업무 처리 순서를 모델링 한 것이다.

2.9 Use Case Diagram

말 그대로 컴퓨터 시스템과 사용자가 상호작용을 하는 하나의 경우이다. 예를 들어 보험처리 프로그램의 경우에 "고객이 보험증권에 sign 한다.", "보험 판매원이 판매통계량을 종합한다." 등이 use case 가 된다. 이러한 use case diagram 은 위 development process 에서 언급한 바와 같이 요구 사항 분석의 단계에서 주로 사용자의 요구를 기술하는데 사용된다. 여기서 알아둘 것은 처음부터 완벽하게 use case diagram 을 작성하려 하지 말라는 것이다. Use case 의 작성이 필요할 때 마다 작성하는 것이 바람직하다.

Example)

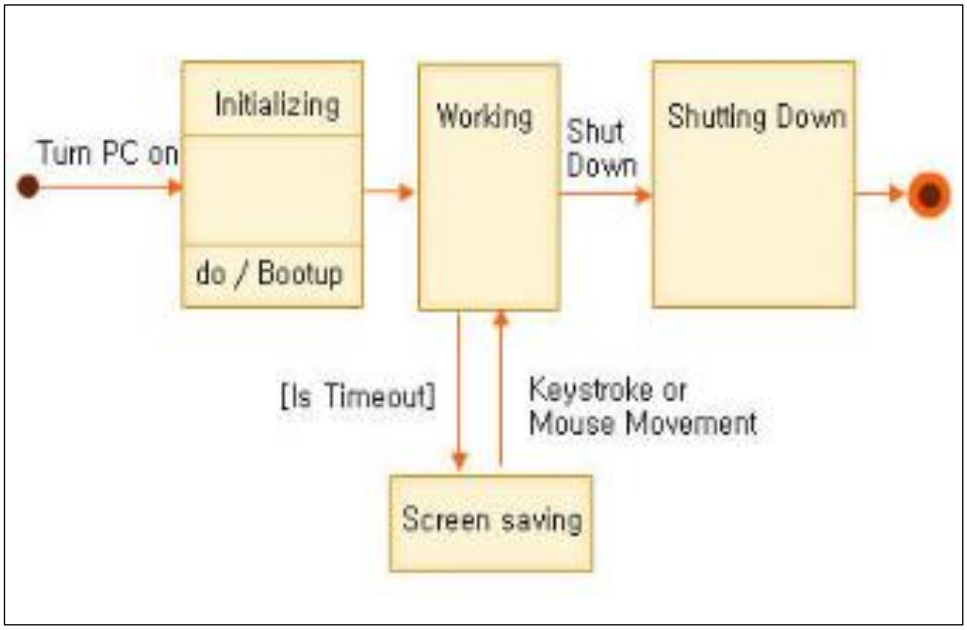


- 회사의 간단한 인사관리 시스템이다.
- 시스템의 사용자인 Actor 는 인사 담당자와 시스템 관리자이다.
- 인사 담당자에게 시스템은 인사 정보를 신규 등록하고 수정하고 인사 현황 조회를 서비스한다.
- 이를 위해 사원 검색 서비스를 공통 서비스로 분리하여 정의한다.
- 시스템 관리자는 기준 정보를 관리한다.

2.10 State Machine Diagram

State diagram 은 오브젝트가 가질 수 있는 모든 상태와 어떠한 event 를 받았을 때 결과로 어떠한 상태로 변화하는지를 나타낸다. 객체가 생성되어 소멸될 때까지 가질 수 있는 모든 상태를 분석하고, 표현하는 다이어그램이다. 하나의 객체를 대상으로 작성하며, 보통 Embedded 에서 적용된다.

Example)



· 컴퓨터라는 객체의 상태를 분석한 State Diagram 이다.

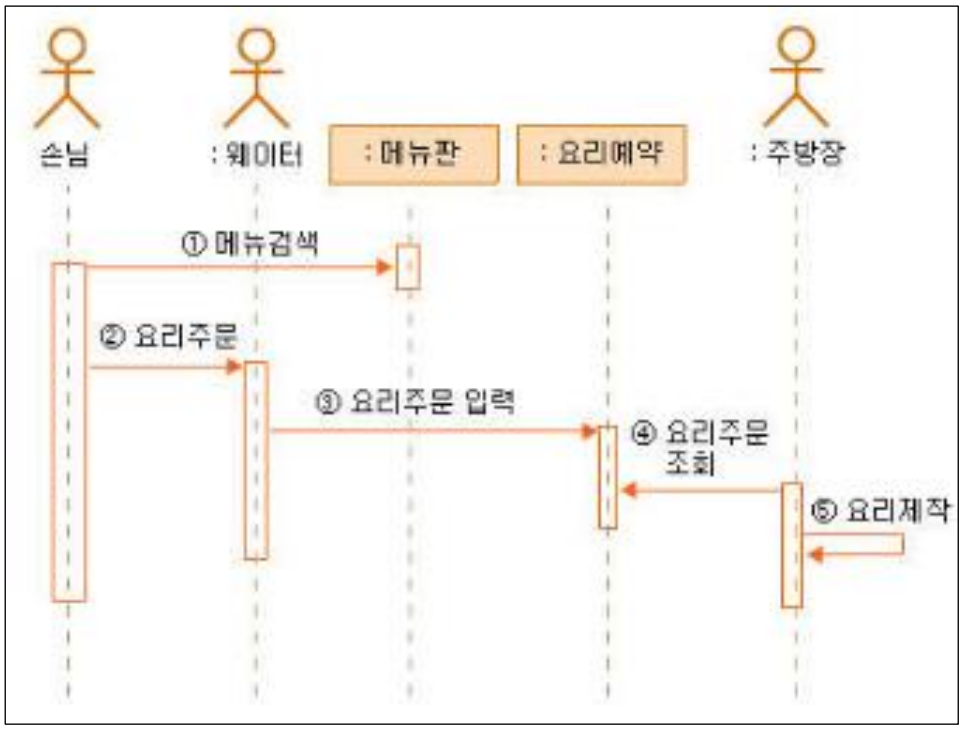
· 화살표의 방향으로 상태가 전이될 수 있고, 전이에 촉발 이벤트가 없는 경우는 자동적으로 다음 상태로 전이된다.

· 종료 상태에 이르면 객체는 소멸된다.

2.11 Sequence Diagram

Class diagram 은 시스템의 정적인 구조를 보여준다. 하지만 여기 sequence diagram 은 collaboration diagram 과 함께 시스템의 동적 구조, 즉 객체와 객체 그룹 사이, 객체와 객체 사이, 객체 그룹과 객체 그룹 사이의 동적인 행위를 기술하게 된다. 특히 sequence 는 종 좌표축으로 시간개념을 도입하고 횡 좌표축으로 객체들을 나열하여 그 사이의 상호작용을 표시하였다.

Example)



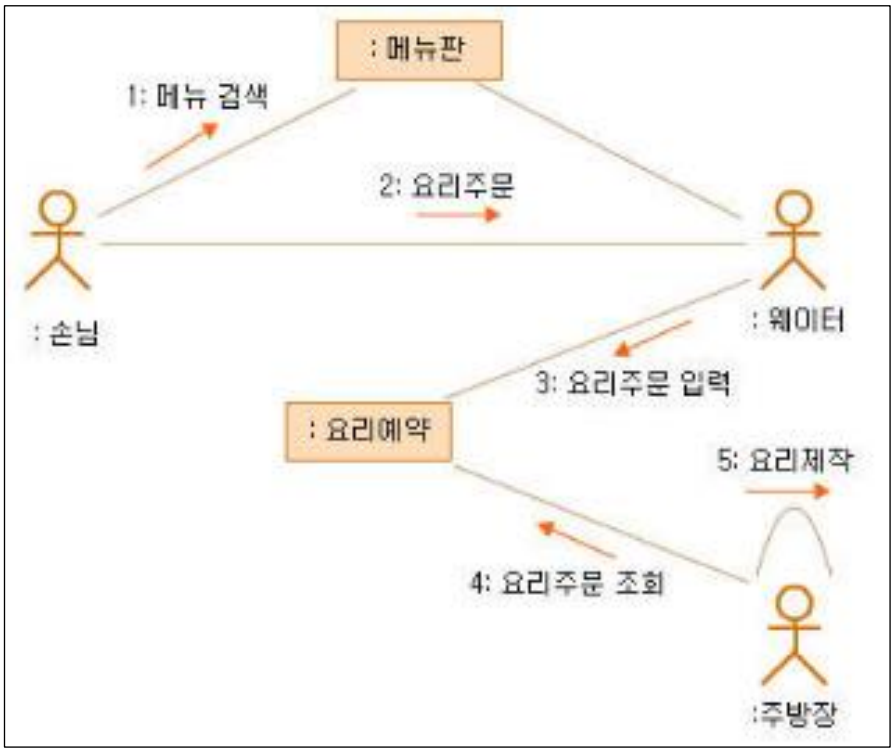
· 어느 음식점의 요리 주문에 대한 Sequence Diagram 이다. (Sequence Diagram 은 위에서 아래 방향으로 갈수록 시간이 증가하는 것을 가정한다.)
· 왼쪽의 시퀀스 다이어그램은 분석 단계에서 작성된 논리 모델에 해당한다.
· 손님이 메뉴판을 보고 메뉴를 고른다. (메뉴판은 객체)
· 손님이 웨이터에게 메뉴를 정해 알려준다.
· 웨이터는 주문된 요리를 입력한다.
· 주방장이 요리 예약을 조회한다.
· 주방장이 요리를 제작한다.

2.12 Communication Diagram

이전 1.X 버전에서는 Collaboration diagram 이라고 불렸던 Communication Diagram 의 경우 객체들 사이의 행위를 나타내는 것은 sequence diagram 과 동일하다. 둘의 차이점은 sequence diagram 이 시간에 따른 행위의 흐름에 역점을 두고 표현하지만 collaboration diagram 의 경우 객체들 사이의 정적인 구조에 더 역점을 두고 있다.

Sequence diagram 과 communication diagram 이 모두 객체들 사이의 상호작용(interaction)을 나타내므로 interaction diagram 이라 통칭하기도 한다. 이런 interaction diagram 이 필요한 경우는 하나의 use case 내부에 포함된 객체들 사이의 행위를 보일 때 필요하다. 하지만 객체 상호간의 관계에 역점을 두었기 때문에 객체 하나의 행위를 정확하게 표현하기에는 부적절하다.

Example)



· 어느 음식점의 요리 주문에 대한 Communication Diagram 이다. (Sequence Diagram 과 같은 시간 증가에 대한 전제는 없다.)

· 손님이 메뉴판을 보고 메뉴를 고른다.

· 손님이 웨이터에게 메뉴를 정해 알려준다.

· 웨이터는 주문된 요리를 입력한다.

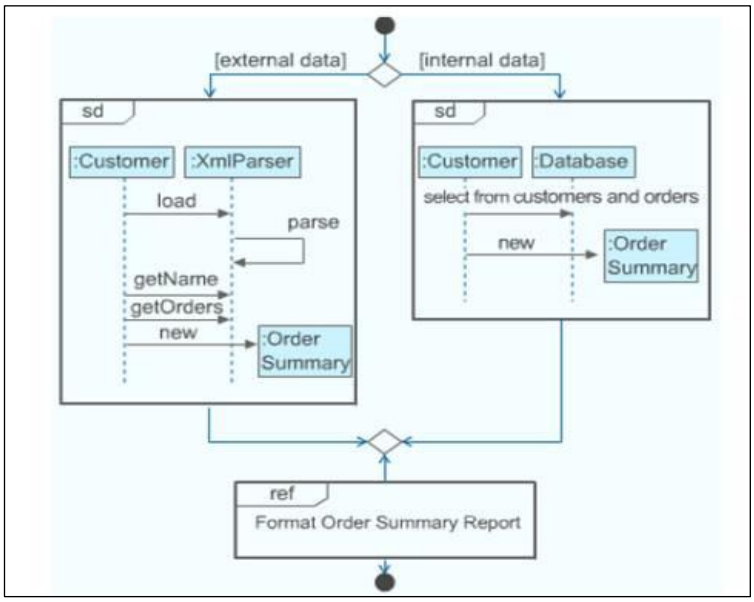
· 주방장이 요리 예약을 조회한다.

· 주방장이 요리를 제작한다.

2.13 Interaction overview Diagram

Activity 다이어그램에서 Activity 대신 작은 Sequence Diagram 을 그린 것이다. Sequence Diagram 의 제어 흐름을 보여주기 위하여 Activity Diagram 표기법에 따라 분할한다.

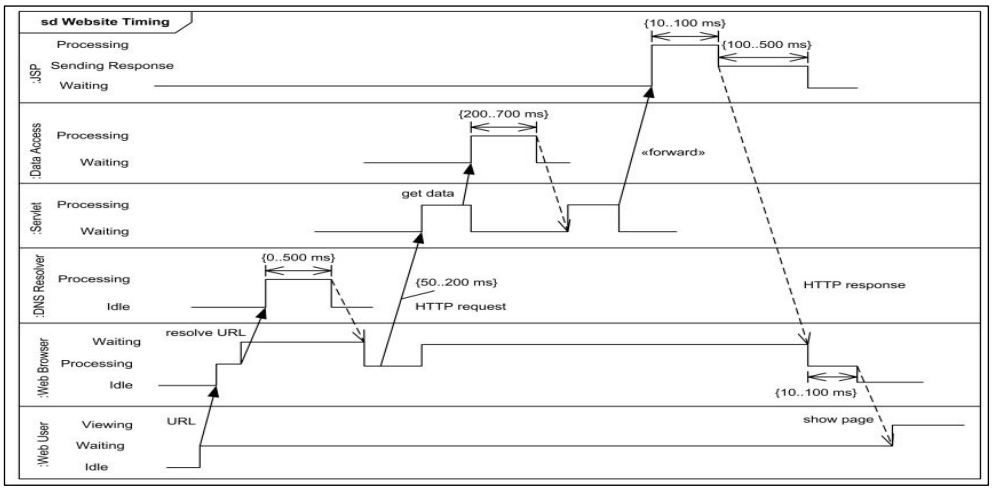
Example)



2.14 Timing Diagram

Sequence Diagram 에서 표현하지 못했던 걸리는 시가에 대한 내용을 다루는 표기방법이다. Sequence Diagram 과의 차이점은 시간이 왼쪽에서 오른쪽으로 증가하고, 라이프 라인이 수직으로 배열 된 별도의 구획에 도시되도록 축이 반전된다.

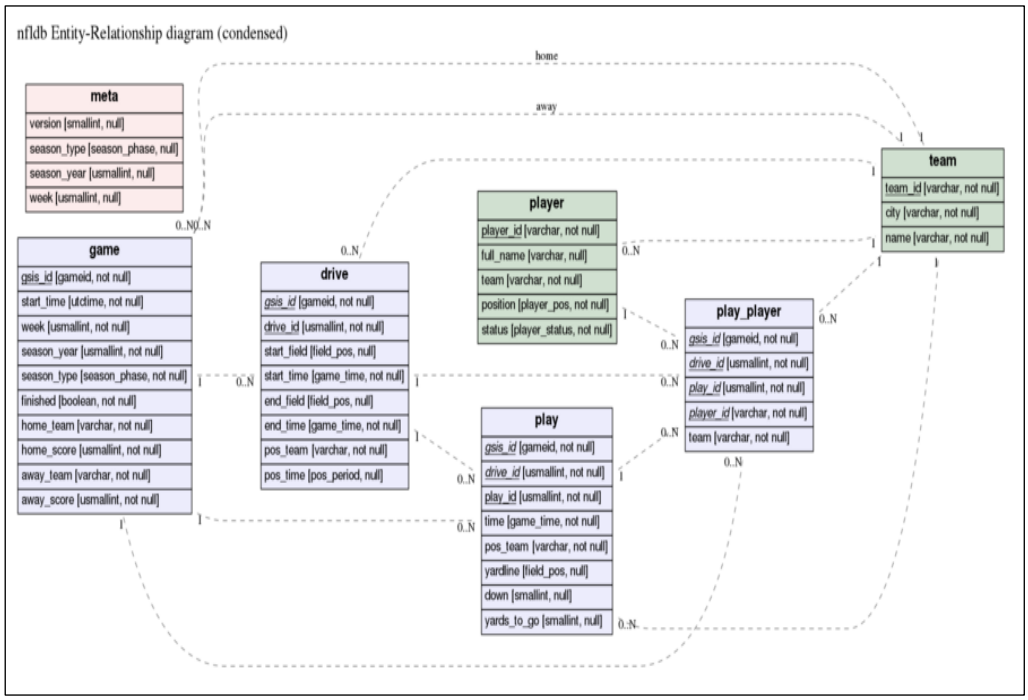
Example)



2.15 E-R Diagram

E-R Diagram 은 개체-관계 모델이라고 한다. 개체-관계 모델이란 구조화된 데이터에 대한 일련의 표현이다. 구조화된 데이터를 저장하기 위해 데이터베이스를 사용한다. 이 데이터의 구조 및 그에 수반한 제약 조건들은 다양한 기법에 의해 설계될 수 있다. 그 기법 중 하나가 개체-관계 모델링(Entity-Relationship Modelling)이다. 줄여서 ERM 이라고 한다. ERM 프로세스의 산출물을 가리켜 개체-관계 다이어그램(Entity-Relationship Diagram)이라 한다. 줄여서 ERD 라 일컫는다. 데이터 모델링 과정은 데이터 모델을 그림으로 표현하기 위해 표시법을 필요로 한다. ERD 는 개념적 데이터 모델 혹은 시맨틱 데이터 모델의 한 타입이다. 정보 시스템을 디자인 해나가는 데에 위와 같은 모델들을 사용하여 시스템이 필요로 하는 정보를 기술한다든가 데이터베이스에 저장되어야 할 정보의 타입(type)이 무엇인가 분석해 나갈 수 있다. 특히 요구사항 단계에서 말이다. 어떤 도메인 오브 디스코스(관심 대상이 되는 부분)의 온톨로지를 기술할 때, 데이터 모델링 테크닉이 사용될 수 있다. (사용된 텀들(terms)과 그들과의 관계를 정의하고 분류하는 일이다.) 데이터베이스에 기반한 정보 시스템(information system)을 디자인하고자 하는 경우에는 나중에 가서는(논리 설계 단계) 개념 데이터 모델은 논리적 데이터 모델로 맵핑 된다. 논리적 데이터 모델은 관계 형 모델 같은 것이다; 뒤이어 논리적 데이터 모델은 물리 설계 단계에서 물리적 디자인 모델로 다시 맵핑 된다. 단, 이와 같은 단계를 모두 뭉뚱그려 "물리 설계"라고 일컫기도 한다.

Example)



3 UML 표기법

3.1 클래스

클래스는 Usecase 와 함께 객체지향 분석 설계 시 가장 중요한 요소이다.

Notation: 사각형 3 부분으로 나누어 클래스 명, 멤버변수, 메소드로 표기한다.

클래스 이름
변수명:변수타입
함수명(파라미터명:파라미터 타입):리턴타입

Attribute : 객체의 특성을 표시한다.

→ 속성 이름 타입 (예 : name : String)

Operation : 객체가 수행하는 서비스

→ Operation 명(파라미터) : return type (예 : getNamed(id : String) : String)

Public(+), protected(#), private(-) 펴기 기능 → 아이콘화 혹은 visibility

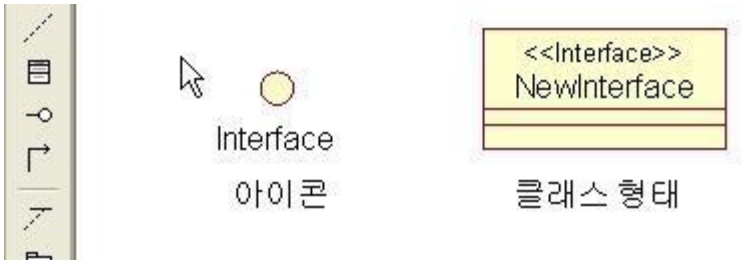
PrivateCar
velocity # wheelNum - carName
+ privateCar() + speedup() + speedDown() + stop()

3.2 인터페이스

클래스가 외부에 제공하는 서비스의 집합이다.

인터페이스는 Body, 즉 구현부위가 없다.

Notation : 아이콘 혹은 클래스 형태 표기 가능하다.



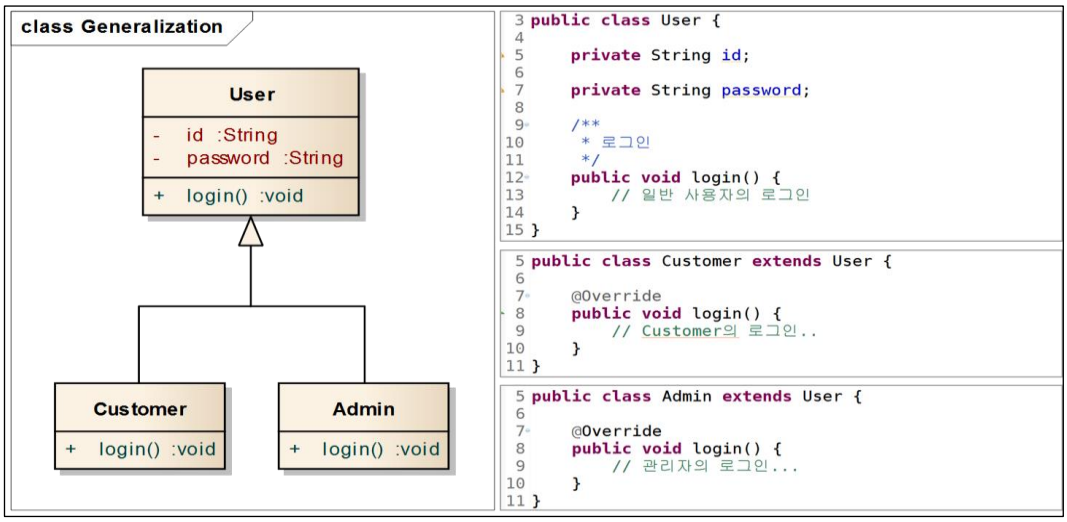
객체 : 클래스의 instance

3.3 클래스들의 관계

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

3.4 Generalization(일반화)

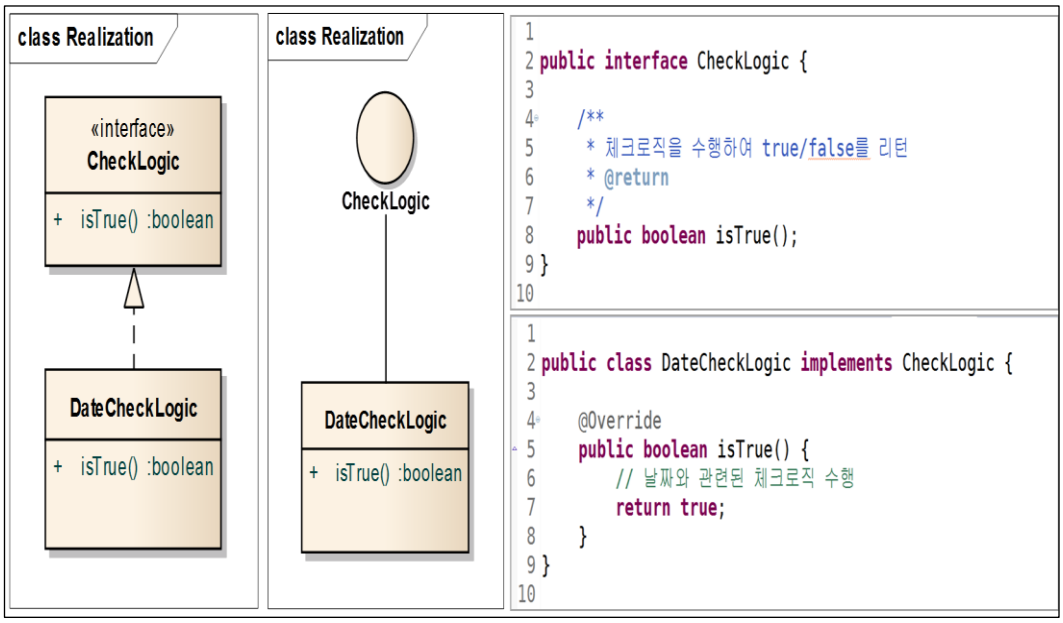
Generalization 은 부모클래스와 자식클래스간의 Inheritance(상속) 관계를 나타낸다. 여기서 Generalization 이란 서브 클래스가 주체가 되어 서브 클래스를 슈퍼 클래스로 Generalize 하는 것을 말하고 반대의 개념은 슈퍼 클래스를 서브 클래스로 Specialize(구체화) 하는 것이다. 상속은 슈퍼 클래스의 필드 및 메서드를 사용하며 구체화 하여 필드 및 메서드를 추가 하거나 필요에 따라 메서드를 overriding(오버라이딩)하여 재정의 한다. 슈퍼 클래스가 추상 클래스인 경우에는 인터페이스의 메서드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



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

3.5 Realization(실체화)

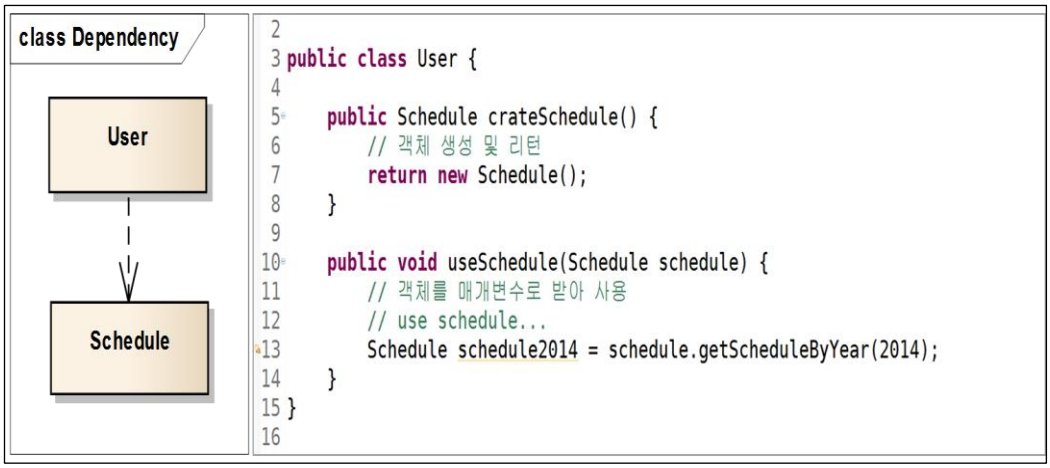
Realization 은 interface 의 명세, 명의만 있는 메서드를 오버라이딩 하여 실제 기능으로 구현 하는 것을 말합니다



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 명시한다. 그리고 인터페이스와 클래스 사이의 관계는 실선으로 연결한다. 자바에서는 위와 같이 implements 키워드를 사용하여 인터페이스를 구현한다.

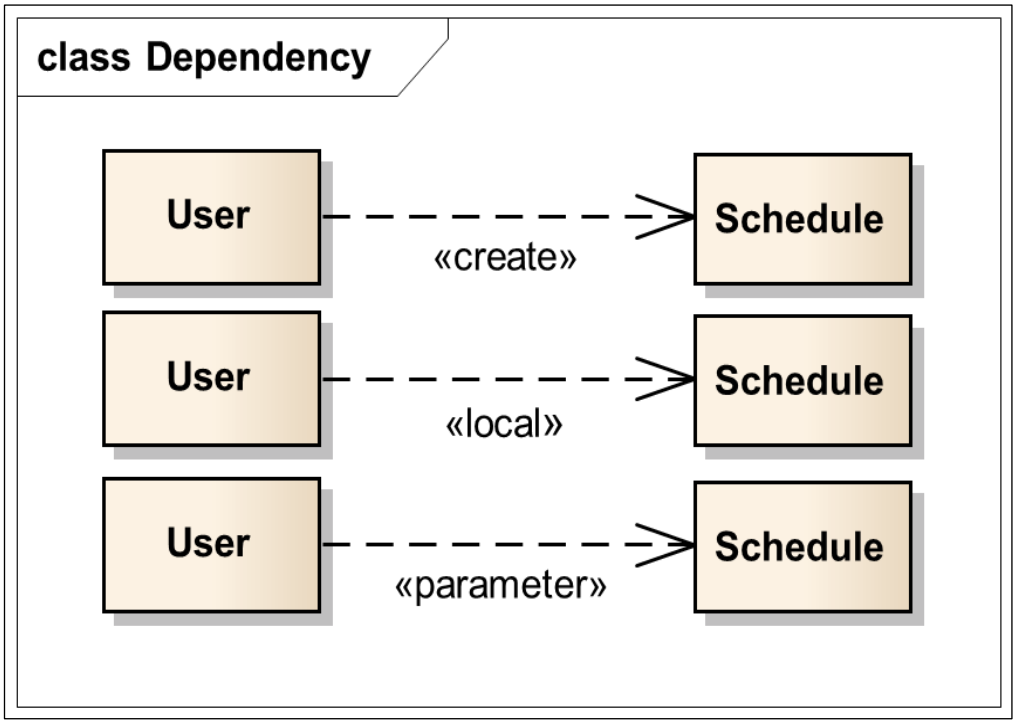
3.6 Dependency(의존)

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




```
2
3 public class User {
4
5     public Schedule crateSchedule() {
6         // 객체 생성 및 리턴
7         return new Schedule();
8     }
9
10    public void useSchedule(Schedule schedule) {
11        // 객체를 매개변수로 받아 사용
12        // use schedule...
13        Schedule schedule2014 = schedule.getScheduleByYear(2014);
14    }
15 }
16
```

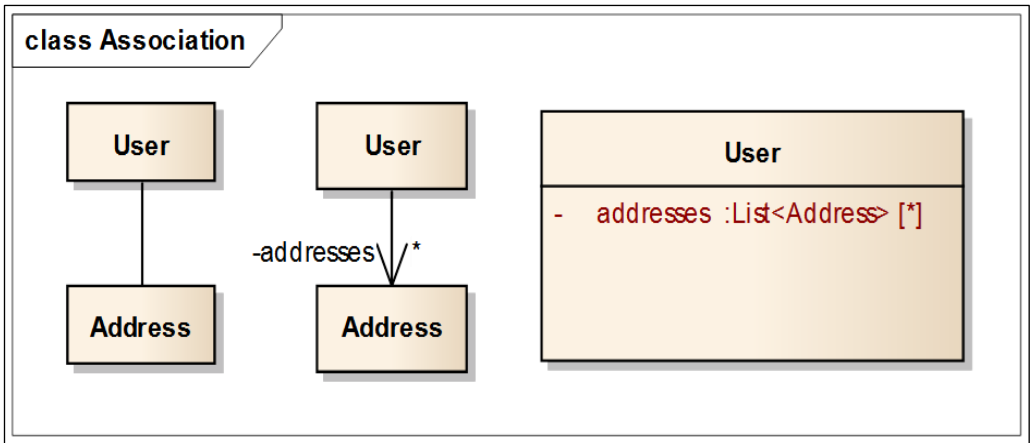
위의 그림은 자바에서 참조하는 형태에 대해 코드를 보여주고 있다. 참조의 형태는 메서드 내에서 대상 클래스의 객체 생성, 객체 사용, 메서드 호출, 객체 리턴, 매개변수로 해당 객체를 받는 것 등을 말하며 해당 객체의 참조를 계속 유지 하지는 않는다.



추가로 위와 같이 스테레오 타입으로 어떠한 목적의 Dependency 인지 의미를 명확히 명시 할 수도 있는데 Dependency 의 목적 또는 형태가 중요할 경우 사용할 수도 있다.

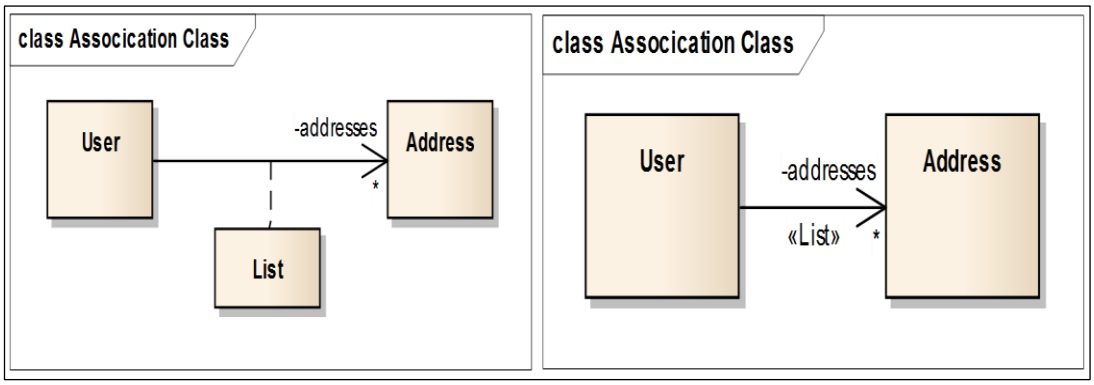
3.7 Association(연관), Directed Association(직접 연관)

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



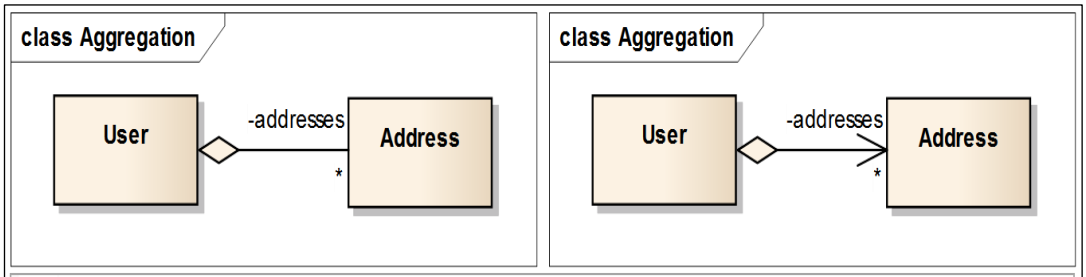

```java
import java.util.List;


public class User {
    private List<Address> addresses;
}

```

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 참조하는 쪽과 참조 당하는 쪽을 구분 한다. 두번째 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 다 일 수도 있는 것을 의미한다. 화살표 옆에 있는 –address 는 roleName(역활명)을 나타내고 Address 가 User 클래스에서 참조될 때 어떤 역할을 가지고 있는지를 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.



3.8 Aggregation(집합, 집합연관)

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

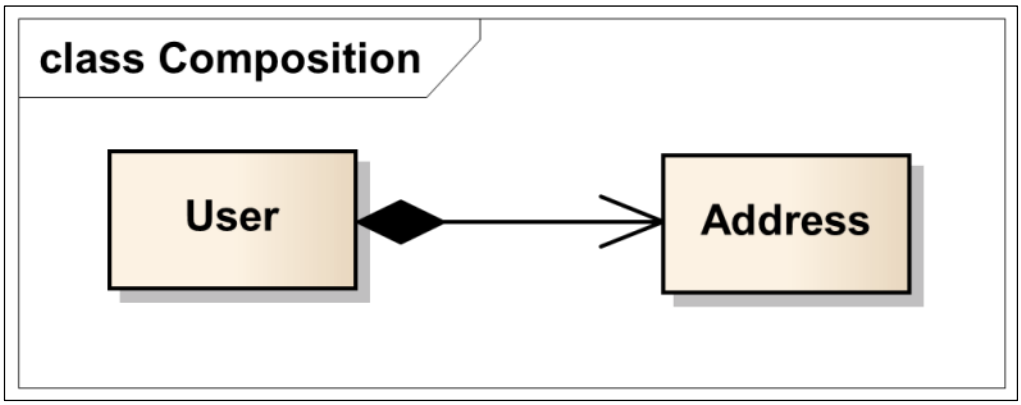



```java
import java.util.List;


public class User {
    private List<Address> addresses;
}

```

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

3.9 Composition(합성 복합연관)

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



Composition 의 표기법 또한 위와 같이 Aggregation 과 비슷하지만 다이아몬드의 내부가 채워져 있다는 점만 다르다

4 UML 을 통한 JAVA Design Pattern

4.1 MVC Pattern

4.1.1 MVC 패턴이란 ?

어플리케이션을 크게 Model, View, Controller 의 세 영역으로 구분하고 영역 간의 결합도를 최소화 한 개발의 구조적 패턴이다. 주로, 웹 어플리케이션에서 사용되며 일반 어플리케이션에서도 널리 사용된다.

MVC 패턴의 가장 중요한 장점 중 하나는 사용자의 인터페이스로부터 비즈니스 로직을 분리하여 어플리케이션의 시각적 요소나 그 이면에서 실행되는 비즈니스 로직을 서로 영향 없이 쉽게 고칠 수 있는 어플리케이션을 만들 수 있다는 것이다. 이에 따라, 디자이너와 개발자들이 각자의 영역에 집중할 수 있게 되었다.

MVC 에서 모델을 어플리케이션의 정보(데이터)를 나타내며 View 는 텍스트, 체크박스 항목등과 같은 사용자 인터페이스 요소를 나타내고 컨트롤러는 데이터와 비즈니스 로직 사이의 상호동작을 관리한다.

4.1.2 MVC 패턴의 구조

1) Model

어떠한 동작을 수행하는 동작을 말한다. 모델영역은 순수하게 public 함수로만 이루어진다. 몇몇의 함수들은 사용자의 질의(query)에 대해 상태 정보를 제공하고 나머지 함수들은 상태를 수정하는 함수이다.

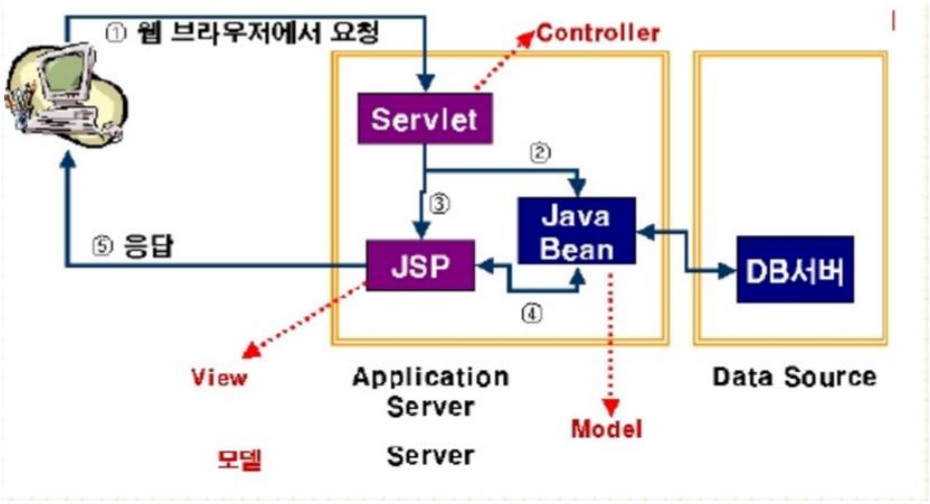
2) View

MVC 에서 모델은 여러 개의 뷰(View)를 가질 수 있다. View 는 모델에게 질의(query)를 하여 모델로부터 값을 가져와 사용자에게 보여준다.

3) Controller

View 에서 들어온 Input 을 Model 에 전달하고 Model 에서 처리한 결과를 View 에 전달시켜 준다.

4.1.3 MVC 패턴을 이용한 클라이언트 요청순서



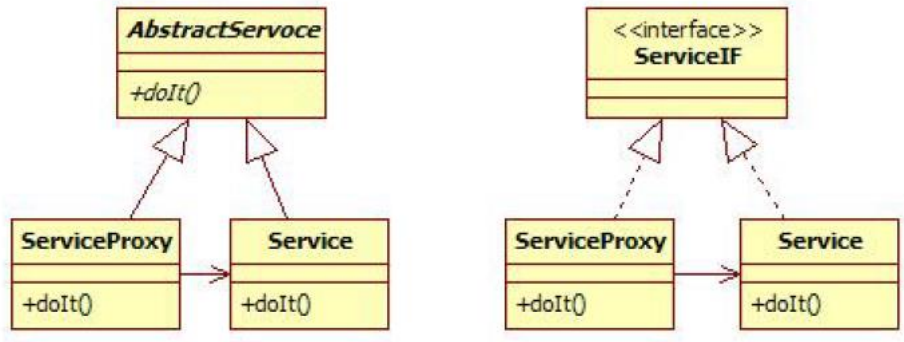
4.2 Proxy Pattern

4.2.1 Proxy 패턴이란?

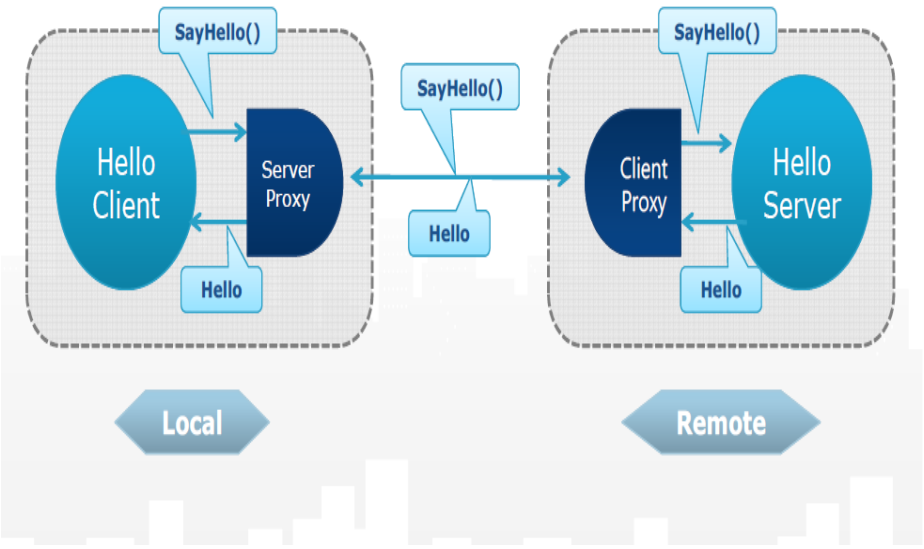
1) 다른 객체의 역할을 대신하기 위하여, 서비스를 사용하는 클래스(Client)와 실제 서비스를 제공하는 클래스(Service) 사이에서 서비스 역할을 대신하는 서비스를 제공하는 대리자 (ServiceProxy) 클래스 역할을 한다.
2) 서비스(Service) 와 서비스 대리자(ServiceProxy)는 동일한 인터페이스(ServiceIF)를 사용함으로, 실제로는 서비스 대리자(ServiceProxy)를 사용하지만 서비스를 사용하는 클래스(Client)로 하여금 실제 서비스(Service)를 사용하는 것처럼 보여준다. (Proxy 를 통합 간접 호출)
3) 주로 독자적으로 사용되기보다는, 다른 패턴에서 응용하여 사용되는 패턴이다.

4.2.2 Proxy 패턴의 구조

Proxy 패턴의 구현은 Abstract Class 및 Interface 모두 사용 가능하다.
중요한 것은 서비스와 서비스 프록시가 같은 인터페이스를 가지고 접근해야 한다는 것이다.



4.2.3 패턴을 이용한 클라이언트 요청순서



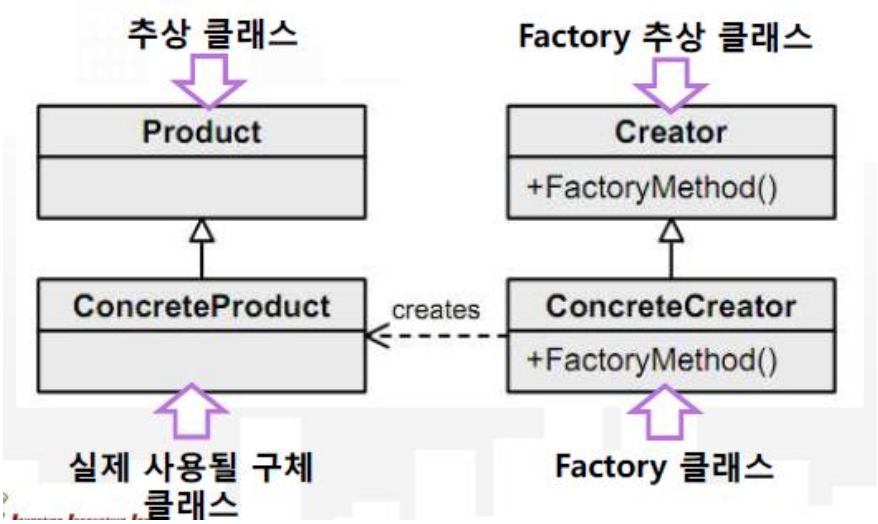
4.3 Factory Method

4.3.1 Factory Method 패턴 이란?

1) Factory Method 패턴은 주어진 입력 데이터에 따라 여러 개의 반환 가능한 클래스들 중 하나의 인스턴스를 반환한다.
2) 보통 반환되는 클래스들은 공통의 부모(parent) 클래스와 공통의 메소드들을 가지고 있지만, 각각 다른 작업(task)을 수행하며 다른 종류의 데이터에 최적화 되어있다.
3) 어떠한 서브 클래스를 생성할지 결정된 후에 실제 클래스 인스턴스가 생성된다.

4.3.2 Factory Method 패턴 구조

1) 구체 클래스와 다른 구체 클래스의 객체 생성을 위임하는 추상 클래스를 같이 제공한다.
2) 추상 클래스가 구체 클래스의 생성을 담당하기 위해서는 공통의 같은 인터페이스로 구현되어야 한다.



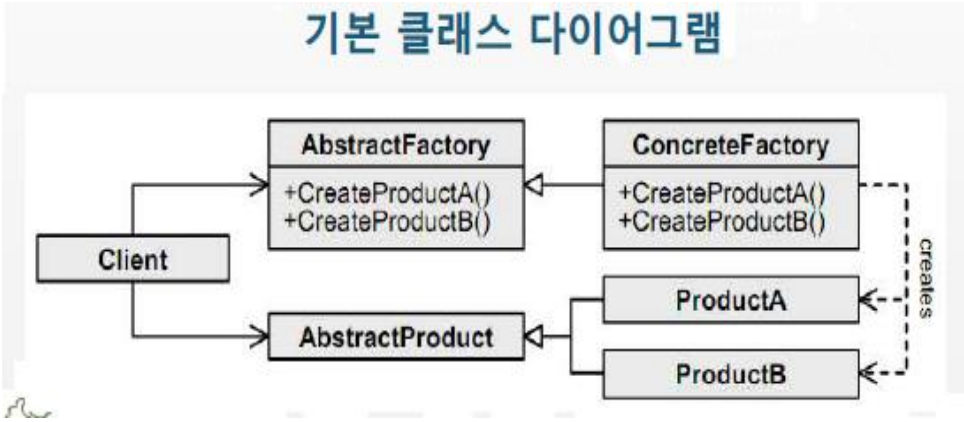
4.4 Abstract Factory

4.4.1 Abstract Factory 패턴이란?

1) Factory Method 패턴보다 좀더 높은 추상화 단계이다.
2) Abstract Factory 여러 클래스 그룹 중의 하나를 되돌려 받는 Factory 객체라고 할 수 있다.
3) 다양한 구성요소 별로 '객체의 집합' 을 생성할 때 유용하다. 즉, 서로 다른 객체들을 하나의 팩토리에서 생성 및 관리한다고 보면 된다.
4) 하나의 추상 클래스에서 객체의 생성을 처리하고, 다양한 성격의 객체를 하나의 군으로 형성 그것을 객체 단위로 취급하여 생성을 해야 할 때 유용한 패턴이다.

4.4.2 Abstract Factory 패턴 구조

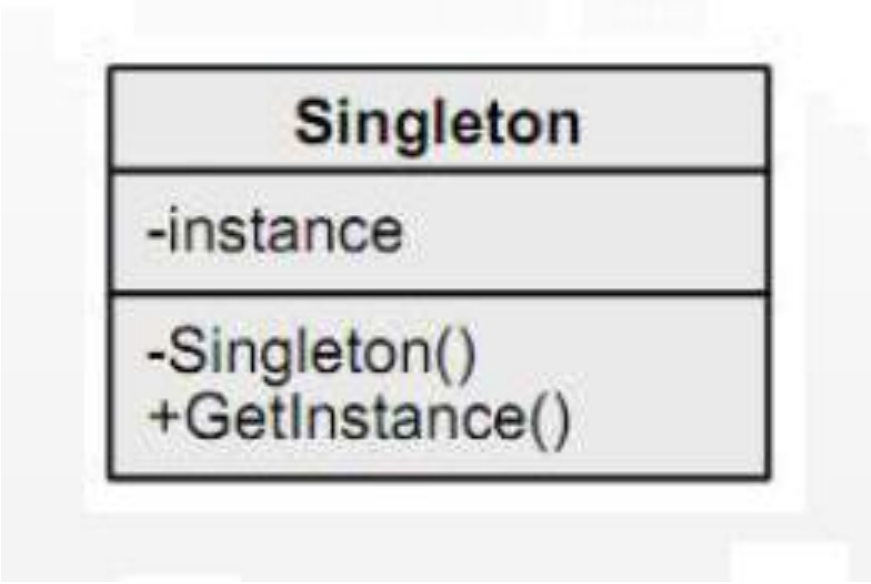
1) 다양한 제품군을 사용하는 시스템과 세부 제품 생성과 관련된 작업은 독립적이다.
2) 클라이언트는 AbstractFactory 를 통해 하나 이상의 제품들로 이루어진 제품군을 사용할 수있다.
3) Abstract Factory 인터페이스를 따르는 클래스(Concrete Factory)의 인스턴스는 반드시 제품(Product)과 함께 사용한다.
4) 클라이언트는 구체적인 제품에 대해 알지 못한다.
5) 시스템은 쉽게 확장 가능해야하며, 추가적인 제품 및 제품군의 추가/삭제는 약간의 코드 수정으로만 가능하다.



4.5 Object Pool(Singleton)

4.5.1 Singleton 패턴이란?

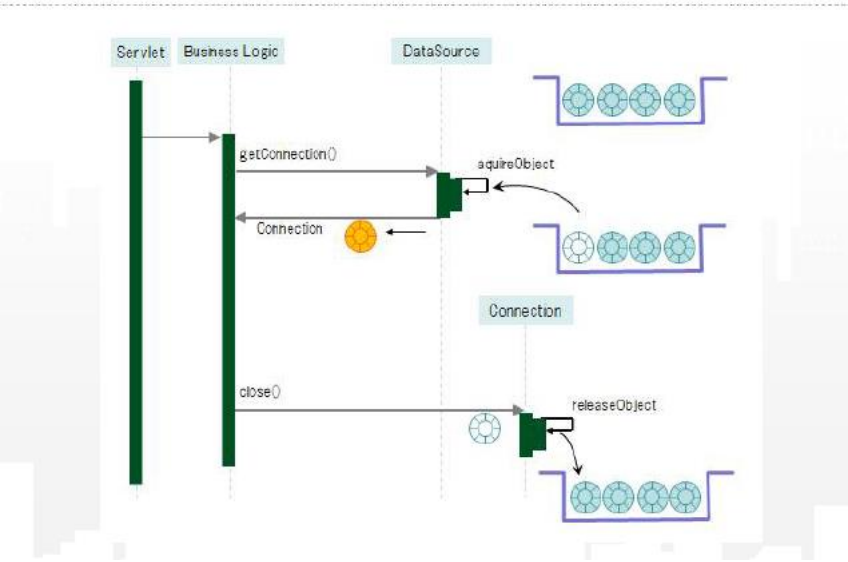
1) 하나의 클래스에서 오직 하나의 인스턴스만 생성 되도록 보장하기 위하
2) 여, 클래스 외부에서 누구도 객체의 인스턴스를 생성하지 못하게 하는 접근법을 사용한다.
3) 하나의 클래스 당 오직 하나의 인스턴스만을 가지고 작업하고 인스턴스를 제어하는 단 하나의 방법만을 제공하고자 할 때 유용하다.
   ⇨ 시스템에서 오직 하나의 윈도우 관리자를 가진다거나 혹은 하나의 프린터 스풀러, 데이터베이스를 제어하는 하나의 리소스를 가지는 경우에 유용하다.



4.5.2 Object Pool 패턴이란?

생성할 객체가 너무 시간이 오래 걸리거나 생성할 수 있는 객체 수가 제한된 경우에 객체를 컬렉션 객체(ArrayList, Vector..)에 넣어놓고 재사용 할 수 있도록 관리한다.

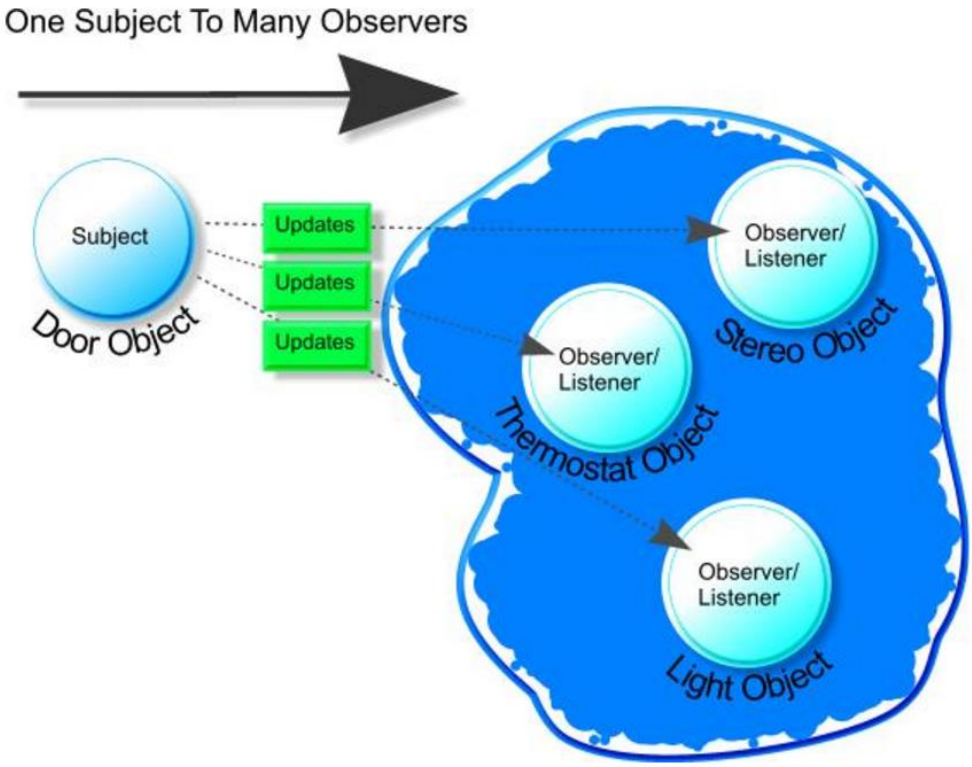
예) Thread pool , DB Connection pool



4.6 Observer Pattern

4.6.1 Observer 패턴이란?

한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들한테 연락이 가고 자동으로 내용이 갱신되는 방식으로 일대다(One-to-Many)의 의존성을 정의



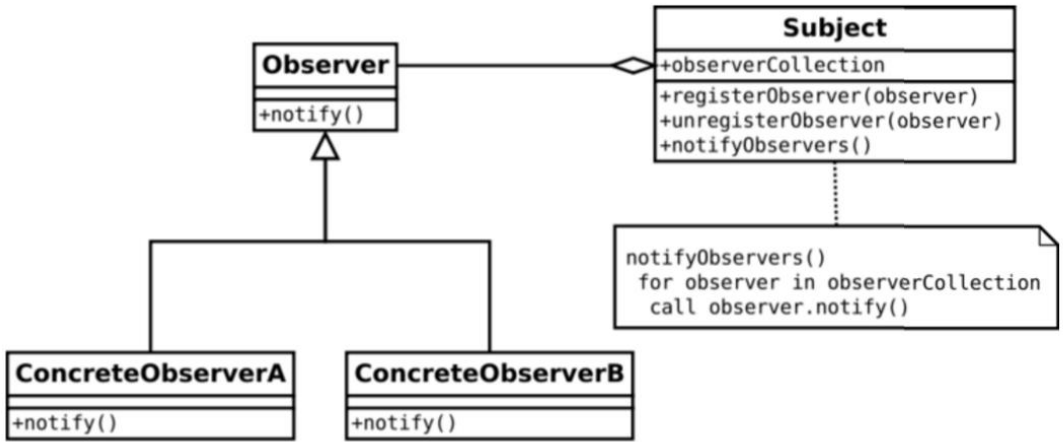
그림과 같이 Subject 에서 정보가 갱신되면 Subject 에 등록된 모든 Observer 에게 데이터를 전송한다. Observer 패턴에서 상태를 저장하고 지배하는 것은 주제객체(Subject)이므로 상태를 갖는 객체는 하나만 있으면 된다.

또한 데이터의 주인은 사실상 Subject 객체이고 옵저버 객체는 사실상 데이터가 갱신되기를 기다리는 입장이라는 특징이 있는데 이러한 구조는 여러 객체에서 동일한 데이터에 접근하는 것 보다 좀 더 깔끔한 객체지향 디자인을 만든다.

이러한 디자인은 두 객체 사이의 결합도를 느슨하게 만들 수 잇고 이 말은 주제와 옵저버가 서로 상호작용을 하기는 하지만 서로에 대해 잘 모른다는 것을 의미한다. 따라서 새로운 형식의 옵저버를 추가하더라도 Subject 를 전혀 변경할 필요도 없고, Subject 와 Observer 는 서로 독립적으로 재사용이 가능하다.

4.6.2 Observer 패턴의 구조

1) Subject 객체 – 데이터가 바뀌면 Observer 객체들에게 전달, Observer 가 될 객체의 등록 및 제거
2) Observer 객체 – 주제의 데이터가 바뀌면 갱신 내용을 전달 받음



5 UML Modeling Tools

5.1 제품군

현존하는 대부분의 UML Modeling(혹은 Design 혹은 CASE) Tool 들의 목록은 List of UML tools 페이지에서 확인할 수 있다. 제품의 가격별로 목록이 정리되어 있는 페이지도 있다.

5.2 상용제품

위에 소개한 페이지등을 참고하여 몇가지 대표적인 상용 도구들을 뽑아보면 다음 목록 정도로 정리할 수 있을 것 같다.

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 Rational 사의 ROSE 인데, IBM 으로 넘어가면서 기존 ROSE 와 함께 RSA(Rational Software Architect)라는 이름으로 발전을 거듭하고 있다.

5.3 Open Source 제품

상용 제품들과 달리 Open Source 형태의 UML 도구들도 많이 있는데 대표적인 것들은 다음과 같다.

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

이들 외에도 개발도구인 Eclipse, NetBeans 를 위한 plug-in 형태로 제공되는 UML 툴들이 많이 있으며, 고가의 상용 제품들에 비해서는 다소 부족하지만 StarUML 과 함께 오픈소스로서 비교적 강력한 UML 모델링을 지원하는 도구로 언급되는 것으로 ArgoUML 이 있다. ArgoUML 은 Java 기반으로 다양한 플랫폼을 지원한다.

5.4 UML Tool 들간의 비교분석

IBM Rational ROSE

- 모델링 도구의 표준임
- RUP , UP 방법론을 적용
- SODA를 통한 산출물 자동화 기능 제공

Borland Together Architect

- 모델링과 개발을 동시에 지원
- 강력한 Reverse Engineering
- 산출물 자동화 템플릿 지원

StarUML (Open Source)

- 오픈소스임에도 불구하고 상용 제품 수준의 기능을 보유
- UML Profile 기반의 모델링 지원
- 모델링 플랫폼을 제공

Enterprise Architect

- 초경량의 모델링 도구
- 강력한 협업 기능 제공
- 모델 파일을 DB형태로 관리

Microsoft Visio

- 범용적인 Diagram 작성 도구
- 탁월한 Diagram 작성 기능

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

5.5 StarUML 이란?

StarUML 은 Win32 플랫폼에서 빠르고, 유연하고, 확장 가능하며, 풍부한 기능을 가지고 있고, 자유롭게 이용 가능한 UML / MDA 플랫폼을 개발하는 오픈 소스 프로젝트이다. StarUML 프로젝트의 목적은 Rational rose 같은 상용 UML 도구의 강력한 대체 소프트웨어 모델링 도구 및 플랫폼을 개발하는 것이다.

UML 2.0: UML 계속 OMG 에 의해 관리 표준을 확대 (관리 그룹 개체). 최근, UML 2.0 발표와 StarUML 은 UML 2.0 을 지원하며 최신 UML 표준을 지원한다.

MDA (Model demand Architecture) : MDA 는 OMG 에 의해 도입 된 새로운 기술이다. MDA 의 장점을 얻기 위해서 소프트웨어 모델링 도구는 많은 사용자 정의 변수를 지원해야 한다. StarUML 은 MDA 를 지원하도록 설계 등으로 UML 프로파일, 접근, 모델 프레임 워크, NX (표기 확장), MDA 코드 및 문서 템플릿과 같은 많은 사용자 정의 변수를 제공합니다.

플러그인 아키텍처: 많은 사용자들이 소프트웨어 모델링 도구에 더 많은 기능을 필요로 한다. 요건을 충족하기 위해, 잘 정의 된 플러그인 플랫폼을 가져야 한다. 사람이 컴퓨터 호환 언어로 플러그인 모듈을 개발할 수 있도록 StarUML 은 간단하고 강력한 플러그인 아키텍처를 제공한다.

사용성 : 사용성은 소프트웨어 개발에서 가장 중요한 문제이다. StarUML 은 빠른 대화, 키보드 조작, 다이어그램 개요 등을 구현하였다. StarUML 은 대부분 델파이로 작성되었다. 또한 StarUML 은 다국어 프로젝트이다. 특정 프로그래밍 언어에 묶여 있지 않다. 그래서 어떤 프로그래밍 언어에서는 StarUML 을 개발하는 데 사용할 수 있다.