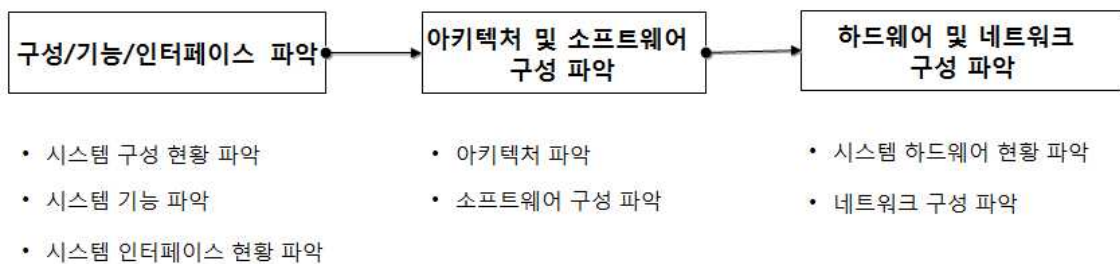


□ 현행시스템의 정의와 목적

1. 사용하고 있는 소프트웨어와 하드웨어는 무엇인지 알아본다.
2. 사용하고 있는 네트워크는 어떻게 구성되었는지 알아본다.
3. 다른 시스템과 어떤 정보를 주고 받는지 알아본다.
4. 향후 개발하고자 하는 시스템의 개발범위 및 이행방향성 설정에 도움을 주는 것이 목적이다.
5. 현행시스템이 제공하고 있는 기능과 기술을 알아본다.

□ 현행시스템의 파악절차이다.



□ 현행시스템 분석시 고려사항

1. 현행시스템 구성현황 작성 시 고려사항은 조직 내 존재하는 모든 시스템 현황파악을 위해 단위업무정보시스템의 명칭과 주요기능을 명시한다.
2. 인터페이스 현황도 작성 시 고려사항은 단위업무 시스템과 주고받는 데이터종류(xml, 가변, 고정 포맷등), 통신규약(TCP/IP, X.25등), 연계유형(EAI, FEP등)은 무엇인지 명시한다.
3. 소프트웨어 구성도 작성 시 소프트웨어 비용이 적지 않기 때문에 라이선스의 적용기준과 약이 중요하다.
4. 하드웨어 구성도는 서버의 주요 사양(CPU처리속도, 메모리 크기, 메모리용량 등)과 이중화가 적용되었는지 나타나 있어야 한다.
5. 네트워크 구성도 작성을 통해 서버의 위치, 서버간의 네트워크 연결방식을 파악할 수 있다.

□ 개발 기술 환경을 결정하기 위해 고려하지 않아도 되는 것을 고르시오

1. 운영체제 관련 요구사항을 식별할 때 운영체제의 '신뢰도', '성능', '기술지원', '주변기기', '구축비용'을 고려해야 한다.
2. DBMS 종류로는 Oracle, IBM DB2, MicroSoft DB Server, MySql, SQLight, Redis등이 있다.
3. 운영체제와 소프트웨어 애플리케이션 사이에 위치하는 미들웨어는 운영체제가 제공하는 서비스를 추가 및 확장하여 제공하는 컴퓨터 소프트웨어를 말한다.
4. 오픈소스는 소스코드를 공개해 제한 없이 사용할 수 있는 오픈소스 라이선스를 만족하는 소프트웨어를 말함으로 라이선스의 종류, 사용자 수, 기술의 지속 가능성을 고려한다.

□ 소프트웨어 지식체계에서 정의한 요구 사항 개발 프로세스이다.



□ 운영체제 주요 특징 및 고려 사항

<표 1-9> 운영체제 종류 및 특징

종류	저작자	predecessor	비용 및 라이선스	주요 용도
Windows	Microsoft	OS/2, MS-DOS	유상, 다양한 라이선스 정책	중소 규모 서버, 개인용 PC, Tablet PC, Embedded System
UNIX	IBM(AIX), HP(HP-UX), SUN(Solaris)	UNIX System V, SunOS	유상, 다양한 라이선스 정책	대용량 처리, 안정성이 요구되는 서버, Server, NAS, Workstation
Linux	Linus Torvalds	Linux kernel	무료, GNU GPLv2	중대 규모 서버
iOS	Apple	OS X NeXTSTEP, BSD	하드웨어에 번들 (Buddle)	스마트폰, 태블릿 PC, Music 플레이어 등
Android	Google	Linux	무료, Apache 2.0, GNU GPLv2	스마트폰, 태블릿 PC

관련 사이트 참고(https://en.wikipedia.org/wiki/Comparison_of_operating_systems)

<표 1-10> 운영체제 관련 요구사항 식별을 위해 고려해야 할 사항

구분	내용
신뢰도	장기간 시스템을 운영할 때 운영체제 고유의 장애 발생 가능성 특정 응용프로그램의 메모리 누수로 인한 성능 저하 및 재기동 운영체제의 보안상 허점으로 인한 반복적인 패치 설치를 위한 재기동 운영체제의 버그 등으로 인한 패치 설치를 위한 재기동
성능	대규모 동시 사용자 요청 처리 대규모 및 대량 파일 작업 처리 지원 가능한 메모리 크기(32bit, 64bit)
기술 지원	공급 벤더들의 안정적인 기술 지원 다수의 사용자들 간의 정보 공유 오픈 소스 여부(Linux)
주변 기기	설치 가능한 하드웨어 다수의 주변 기기 지원 여부
구축 비용	지원 가능한 하드웨어 비용 설치할 응용프로그램의 라이선스 정책 및 비용 유지 및 관리 비용 총 소유 비용(TCO)

□ DBMS 주요 특징 및 고려 사항

<표 1-11> DBMS 종류 및 특징

종류	저작자	비용 및 라이선스	주요 용도
Oracle	Oracle	상용	대규모, 대량 데이터의 안정적인 처리
IBM DB2	IBM	상용	대규모, 대량 데이터의 안정적인 처리
Microsoft SQL Server	Microsoft	상용	중소 규모 데이터의 안정적인 처리
MySQL	MySQL AB, Oracle	GPL 또는 상용	오픈 소스 RDBMS
SQLite	D. Richard Hipp	Public Domain 저작권 보호 만료	스마트폰, 태블릿 PC 등의 임베디드 데이터베이스(Embedded Database) 용
MongoDB	MongoDB Inc.	GNU AGPL v3.0	오픈 소스 NoSQL 데이터베이스
Redis	Salvatore Sanfilippo	BSD	오픈 소스, 메모리, 키-값(Key-Value) 데이터베이스

<표 1-12> DBMS에 관하여 고려할 사항

구분	내용
가용성	장기간 시스템을 운영할 때 장애 발생 가능성 DBMS의 버그 등으로 인한 패치 설치를 위한 재기동 백업 및 복구 편의성 DBMS 이중화 및 복제 지원
성능	대규모 데이터 처리 성능(분할 테이블의 지원 여부) 대량 거래 처리 성능 다양한 튜닝 옵션 지원 비용 기반 최적화 지원 및 설정의 최소화
기술 지원	공급 벤더들의 안정적인 기술 지원 다수의 사용자들 간의 정보 공유 오픈 소스 여부
상호 호환성	설치 가능한 운영체제 종류 다양한 운영체제에서 지원되는 JDBC, ODBC
구축 비용	라이선스 정책 및 비용 유지 및 관리 비용 총 소유 비용(TCO)

관련 사이트 참고

(<http://www.dbguide.net/knowledge.db?cmd=view&boardUid=126000&boardConfigUid=19>)

□미들웨어의 주요 특징 및 고려 사항

<표 1-13> 웹 애플리케이션 서버(WAS: Web Application Server) 종류 및 특징

종류	벤더	비용 및 라이선스	주요 용도
GlassFish	GlassFish Community	CDDL, GPL	NetBeans 개발 툴과 연동하여 사용
JBoss	Red Hat	LGPL	JBoss 기반 오픈 소스 제품들을 이용하는 경우
Jetty	Eclipse Foundation	Apache 2.0, EPL	빠른 처리 속도가 요구되는 경우
JEUS	TmaxSoft	상용	대량의 안정적인 거래 처리가 요구되며 적시의 기술 지원이 필요한 경우
Resin	Caucho Technology	GPL, 상용	빠른 처리 속도가 요구되는 경우
WebLogic	Oracle Cor.	상용	대량의 안정적인 거래 처리가 요구되는 경우
WebSphere	IBM	상용	

관련 사이트 참고(https://en.wikipedia.org/wiki/Comparison_of_application_servers)

<표 1-14> 웹 애플리케이션 서버(WAS: Web Application Server)에 관하여 고려할 사항

구분	내용
가용성	장기간 시스템을 운영할 때 장애 발생 가능성 안정적인 트랜잭션 처리 WAS의 버그 등으로 인한 패치 설치를 위한 재기동 WAS 이중화 지원
성능	대규모 거래 요청 처리 성능 다양한 설정 옵션 지원 가비지 컬렉션(GC: Garbage Collection)의 다양한 옵션
기술 지원	공급 벤더들의 안정적인 기술 지원 다수의 사용자들 간의 정보 공유 오픈 소스 여부
구축 비용	라이선스 정책 및 비용 유지 및 관리 비용 총 소유 비용(TCO: Total Cost of Ownership)

□ 요구사항 정의 에 관한 내용이다

1. 요구사항 분석 시 소프트웨어 범위를 파악하며 시스템 요구사항을 정제한다.
2. 요구사항 명세를 통해 체계적으로 검토,평가 승인 되도록 문서를 작성한다.
3. 요구사항 확인 시 문서가 회사 표준에 적합하고 이해 가능 하며 일관성 있고 완전한지 검증한다.

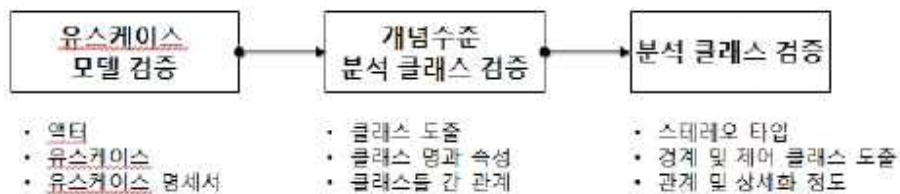
□ 요구사항 분석 기법

1. 요구사항 분류 기준은 기능/비기능, 요구사항의 범위, 제품/프로세스 등이 있다.
2. 개념 모델링은 소프트웨어 요구사항 분석의 핵심이며 문제발생시 상황을 이해하며 해결책을 제시한다.
- 3.요구사항 할당은 아키텍처 구성요소를 식별하는 것이다.
- 4.두명의 이해관계자가 서로 상충되는 내용을 요구하거나 , 요구사항 내용이 상충될 때 어느 한 쪽을 지지하기보다는 적절한 트레이드 오프 지점에서 합의가 중요하다.
5. 씨맨틱을 지닌 언어로 요구사항을 표현하는 것을 정형분석이라고 한다.

□ 요구사항 확인 기법

- 1.요구사항 검토시 검토자들이 에러 , 잘못된 가정, 불명확성, 표준과의 차이등르 찾아 내도록 한다.
- 2.요구사항 검토는 시스템 정의서, 시스템 사양서, 소프트웨어 요구사항 명세서를 완성한 시점 등에서 이루어진다.
- 3.프로토타이핑은 사용자 인터페이스의 동적인 행위가 잘 이해되어진다. 요구사항이 잘못된 경우 피드백이 원활하게 이루어진다.
- 4.분석단계에서 개발된 모델의 품질을 검증할 필요가 있다.
- 5.요구사항을 만족 시키는지 확인 하기 위해 인수테스트가 필요하다.

□ 분석모델 검증 절차이다



□ 유스케이스 모델 검증시 점검 대상




액터 유스케이스 유스케이스 명세서

□ 개념 수준의 분석 클래스 검증

1. 시스템 전체를 대상으로 작성한다.
2. 도출된 클래스의 이름과 설명이 명확하게 작성한다.
3. 클래스들 간에 순환적 관계가 불필요하게 정의 되어 있는지 검토한다.
4. 클래스 간의 관계에서 다중성이 정의되었는지 검토한다.

□ 분석 클래스의 스테레오 타입이다.

〈표 3-3〉 분석 클래스의 스테레오 타입

역할구분	스테레오 타입	내용
경계 Boundary	<<boundary>> 	시스템과 외부 액터와의 상호작용을 담당하는 클래스
엔터티 Entity	<<entity>> 	시스템이 유지해야 하는 정보를 관리하는 기능을 전담하는 클래스
제어 Control	<<control>> 	시스템이 제공하는 기능이 로직 및 제어를 담당하는 클래스

□ 분석모델의 시스템화 타당성 검토 내용

- 1.요구사항을 만족시키는 분석모델을 따라 시스템을 구현할 때 성능 및 용량이 타당한지 분석한다.
- 2.분석모델이 시스템의 기술적인 위험을 분석한다.
- 3.분석 모델이 과거의 문제를 해결하고 많이 사용하는 트렌드에 부합되도록 한다.
- 4.분석모델을 이용하여 시스템간 상호 정보 및 서비스를 교환 가능 한지 검토한다.

<참조내용>

□ 유스케이스 정의

정의

- 행위자(Actor) 중심의 시스템 구상
- 소프트웨어 시스템의 기능적 요구사항에 대한 베이스라인
- 사용자의 시각에서 소프트웨어 시스템의 범위와 기능을 정의한 모델
- 행위자(Actor)가 어떤 기능 사용할 수 있는지 보여줌

작성시기

- 소프트웨어 프로젝트의 개발범위를 정의하는 단계
- 소프트웨어에 대한 요구사항을 정의하는 단계
- 소프트웨어의 세부기능을 분석하는 단계
- 소프트웨어가 아닌 업무영역을 이해하고 분석하는 단계

구성요소

- 요소 : 행위자(Actor), 유스케이스(Use Case)
- 관계 : 커뮤니케이션(Communications), 포함(Include), 확장(Extend), 일반화(Generalization)

1) 행위자(Actor)

- 시스템이 외부에 존재하면서 시스템과 교류 혹은 상호작용 하는 것
- 시스템이 서비스를 해주기를 요청하는 존재
- 시스템에게 정보를 제공하는 대상

(1)사용자 액터

기능을 요구하는 대상이나 시스템이 수행결과를 통보받는 사용자 혹은 기능을 사용하게 될 대상으로 시스템이 제공해야하는 기능인 유스케이스의 권한을 가지는 대상, 역할이 된다.

ex) 학생이 도서를 구매한다.

|학생(Actor)| -> |도서구매(Use Case)|

(2)시스템 액터

사용자 액터가 사용한 유스케이스를 처리해주는 외부의 시스템으로 시스템의 기능 수행을 위해서 연동이 되는 또 다른시스템 액터

ex) 학생이 구매한 도서를 처리해주는 |도서구매(Use Case)| --> |도서주문시스템(Actor)|

2) 유스케이스(Use Case)

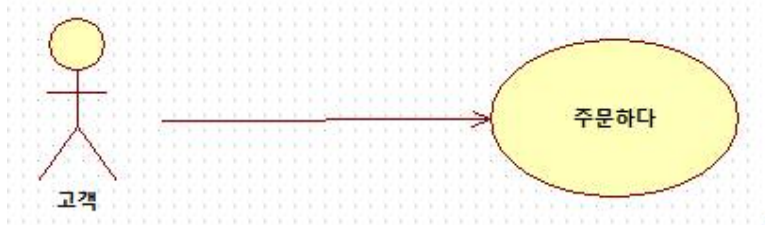
유스케이스(Use Case)는 사용자 입장에서 바라본 시스템의 특성을 설명한 구조로서, 행위자(actor) 즉, 사람, 시간의 흐름, 또는 다른 시스템에 의해 개시되는 시나리오 집합의 형태를 갖추고 있다.

- 시스템이 제공하는 서비스 혹은 기능
- 시스템이 행위자(Actor)에게 제공하는 사용자 관점의 기능단위
- 행위자(Actor)의 요청에 반응하여 원하는 처리를 수정하거나 정보를 제공
- 행위자(Actor)와 한 번 이상의 상호작용을 통한 의미있는 묶음의 시스템 행위
- 의미있는 자기완결형의 서비스 단위
- 사용자관점에서의 정의가 필요

3) 커뮤니케이션 (Communication)

- 행위자(Actor)와 유스케이스 사이에 정의되는 세계
- 일반 상호작용 관계가 존재하는 것을 의미
- 행위자(Actor)는 정보를 통보받거나 요구

- 유스케이스는 정보를 제공



상호 교류 관계시 연결된다.

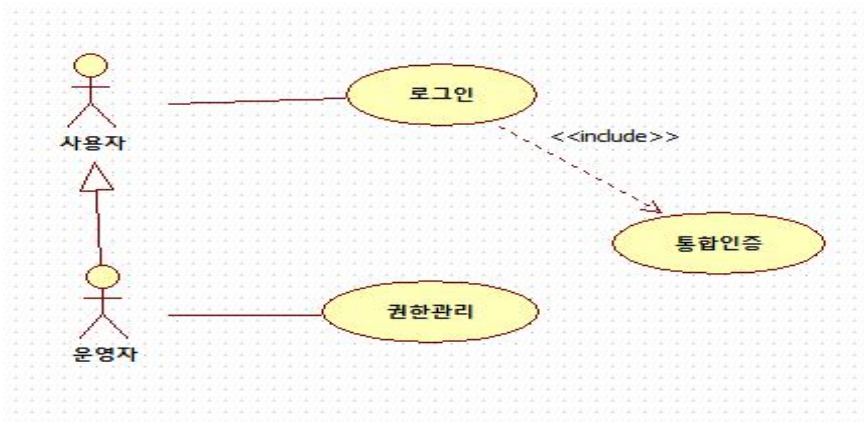
--> 커뮤니케이션을 받는 쪽에 화살표

4) 일반화(Generalization)

- 행위자(Actor)와 행위자(Actor), 유스케이스와 유스케이스 사이의 정의
- 두 개체가 일반화 관계에 있음 의미
- 보다 보편적인 것과 보다 구체적인 것 사이의 관계 (is-a 관계)
- 상속의 특성을 지님

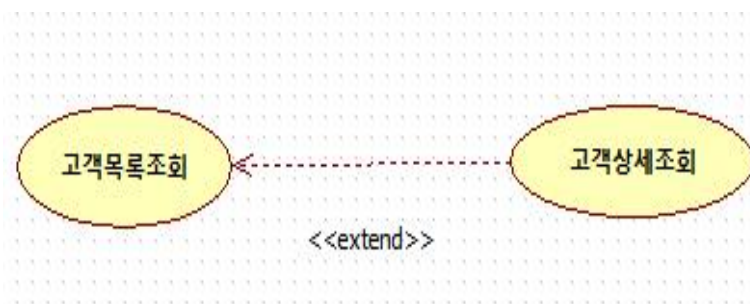
5) 포함(Include)

- 유스케이스와 유스케이스 사이에 정의되는 관계
- 한 유스케이스가 다른 유스케이스의 서비스 수행을 요청하는 관계
- 즉, 한 유스케이스가 자신의 서비스 수행 도중에 다른 유스케이스의 서비스 사용이 필요할 때 정의 (서비스는 반드시 사용이 되어야 함)
- 포함되는 유스케이스는 공통 서비스를 가진 존재



6) 확장(Extend)

- 유스케이스와 유스케이스 사이에 정의되는 관계
- 포함관계와 동일하게 서비스 수행을 요청하는 관계
- 포함관계와 달리 서비스가 수행되지 않을 수 있음
(선택적 유스케이스 관계이다. 필수적이지않고 옵션이라고 할수 있다.)
- 수행 요청 조건을 확장한 Extention Point이라고 함
- 표시방법은 포함(include)와 반대이다.



Use Case Diagram 작성단계

① 행위자(Actor) 식별

- 모든사용자 역할 식별
- 상호작용하는 타 시스템 식별
- 정보를 주고받는 하드웨어 및 지능형 장치 식별

② 유스케이스 식별

- 행위자(Actor)가 요구하는 서비스 식별
- 행위자(Actor)가 시스템과 상호작용하는 행위를 식별

③ 관계정의

- 행위자(Actor)와 행위자(Actor) 관계분석 정의
- 행위자(Actor)와 유스케이스 관계분석 정의
- 유스케이스와 유스케이스 관계분석 정의

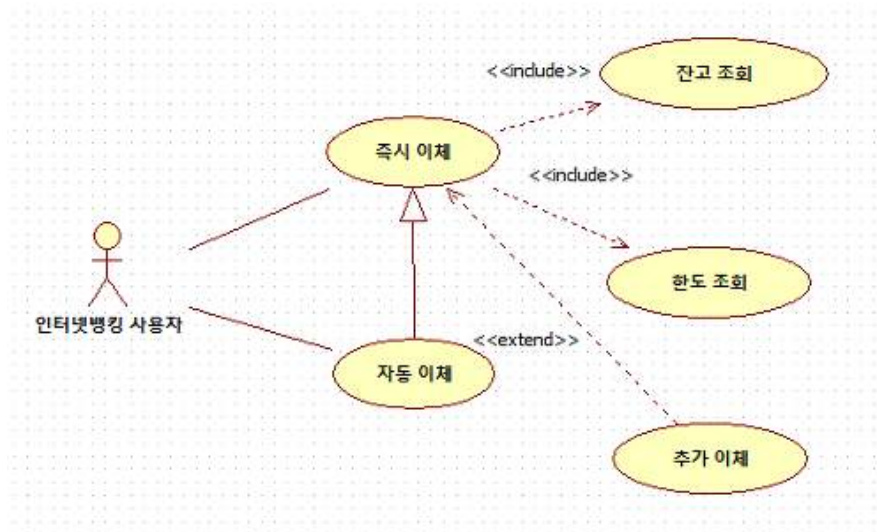
④ 유스케이스 구조화

- 두 개 이상의 유스케이스에 존재하는 공통서비스 추출
- 추출된 서비스를 유스케이스 정의
- 조건에 따른 서비스 수행부분 분석하여 추출
- 추출된 서비스를 유스케이스로 정의

정리 및 응용

- 시스템 여역 표시는 잘 사용하지 않으며, 불필요한 경우가 많다.

- 행위자(Actor)와 유스케이스(Use Case)간의 관계를 연관(Association)이라고 한다.
- 화살표의 의미가 불분명한 경우가 많으므로, 사용하지 않는 편이 낫다.
- 포함(Include) 관계는 하나의 유스케이스가 다른 유스케이스를 항상 포함한다는 의미이다
- 확장(Extend) 관계는 하나의 유스케이스로부터 다른 유스케이스로 기능이 확장될 수 있다는 의미이다.
- 확장 관계의 화살표 방향은 확장 대상이 되는 유스케이스로부터 확장점을 가진 유스케이스로 향한다.
- 확장 관계는 상속이 아니다.
- 기능의 상속은 일반화(Generalization)라고 하며, 삼각형 머리를 가진 실선으로 표시한다.



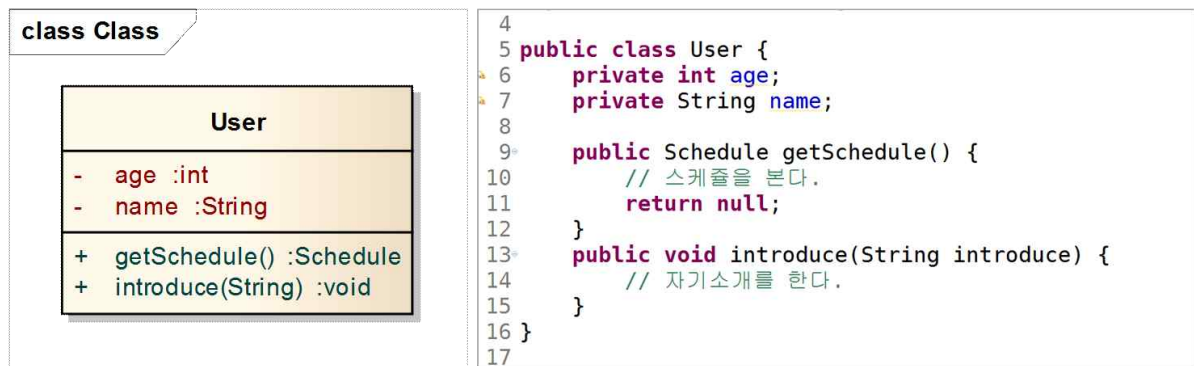
- 위의 예에서 행위자는 [인터넷뱅킹 사용자]이고 [즉시 이체]라는 유스케이스와 연관 관계를 갖고 있다.
- [즉시 이체]를 수행할 경우, 출금 계좌의 잔고를 확인하고, 이체 한도를 조회하는 기능을 먼저 수행하게 되므로, [잔고 조회] 유스케이스와 [한도 조회] 유스케이스는 [즉시 이체] 유스케이스의 **포함 관계** 대상이 된다.
- [즉시 이체] 유스케이스로부터 [추가 이체] 유스케이스를 실행할 수도 있으므로, [즉시 이체]와 [추가 이체]는 **확장 관계**가 된다.
- [자동 이체]는 [즉시 이체]의 구체적인 유스케이스 개념이므로 일반화 및 상속 관계를 표현할 수 있다.

<참고문서 2>

1. 클래스 다이어그램의 요소(Element)

Class (클래스)

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

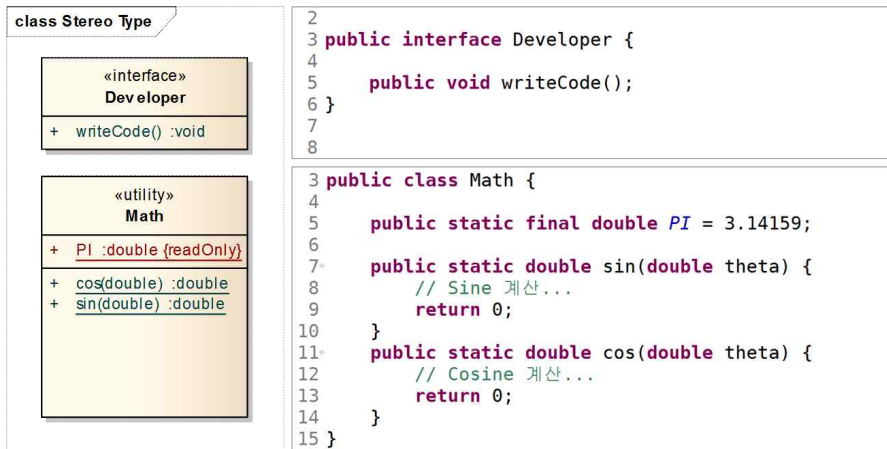


[그림 3] 클래스

클래스의 세부사항은 필드와 메서드의 Access modifier(접근제한자), 필드명(메서드명), 데이터타입, parameter(매개변수), 리턴 타입 등을 나타낼 수 있습니다. 클래스의 세부사항들을 상세하게 적는 것이 유용할 때도 있지만, UML 다이어그램은 필드나 메서드를 모두 선언하는 곳이 아니기 때문에 다이어그램을 그리는 목적에 필요한 것만 사용하는 것이 좋습니다. 추가로 보통 3개의 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 합니다.

Stereo Type (스테레오 타입)

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

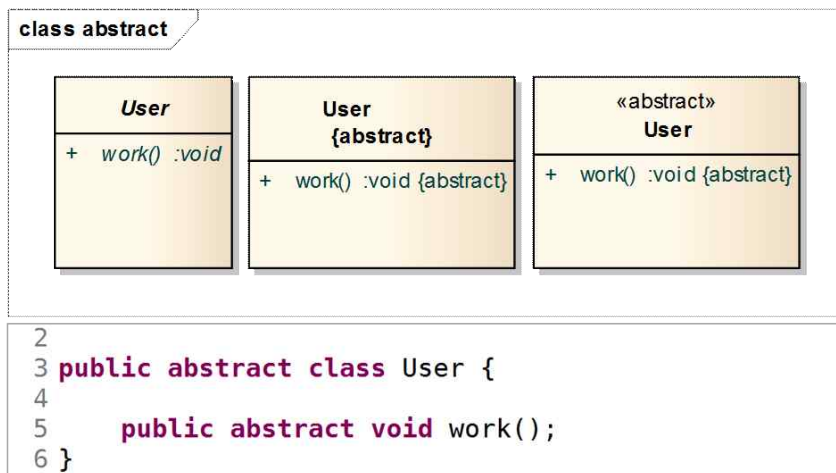


[그림 4] 스테레오 타입

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

Abstract Class/Method (추상 클래스 / 메서드)

추상클래스란 1개 이상의 메서드가 구현체가 없고 명세만 존재하는 클래스를 말합니다.



[그림 5] 추상클래스

추상 클래스의 이름과 메서드는 italic체나, {abstract} 프로퍼티를 사용하여 표기합니다. UML 툴에서는 italic체로 표기하는 것이 많지만 종이나 칠판에 그릴 때는 힘들게 italic체로 기울여서 적는 것 보다는 {abstract} 프로퍼티로 표기하는 것이 쉽고 명확할 것 같습니다. 또한 공식적인 것은 아니지만 스테레오타입을 사용하여 추상 클래스를 표기하기도 합니다.

2. 클래스간의 관계

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

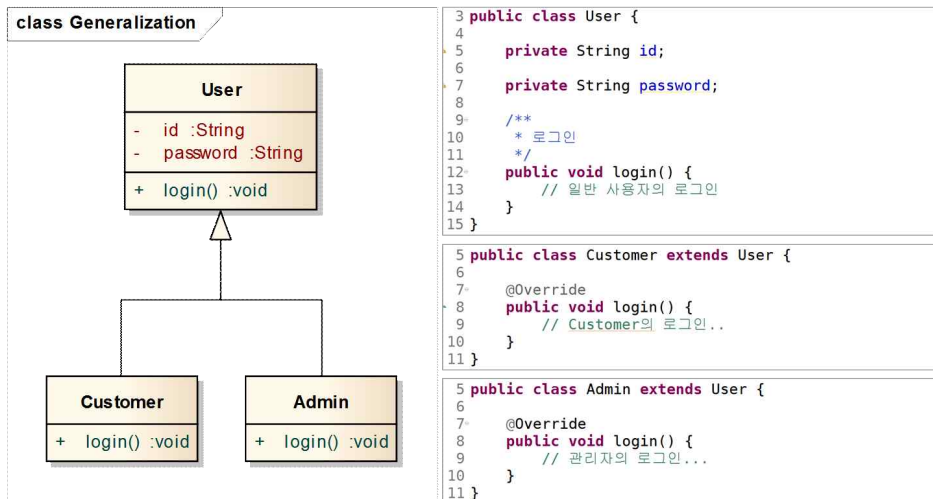
관계	UML 표기
Generalization (일반화)	
Realization (실체화)	
Dependency (의존)	
Association (연관)	
Directed Association (직접연관)	
Aggregation (집합, 집합연관)	 
Composition (합성, 복합연관)	 

[그림 6] 클래스 관계 종류

위의 그림은 클래스간의 관계들의 종류와 표기법을 나타낸 것입니다. 이 외에 다른 표기법도 있습니다만 많이 사용된다고 생각하는 것들만 나타냈습니다.

Generalization (일반화)

Generalization은 슈퍼(부모)클래스와 서브(자식)클래스간의 Inheritance(상속) 관계를 나타냅니다. 여기서 Generalization이란 서브 클래스가 주체가 되어 서브 클래스를 슈퍼 클래스로 Generalize 하는 것을 말하고 반대의 개념은 슈퍼 클래스를 서브 클래스로 Specialize (구체화) 하는 것입니다. 상속은 슈퍼 클래스의 필드 및 메서드를 사용하며 구체화 하여 필드 및 메서드를 추가 하거나 필요에 따라 메서드를 overriding(오버라이딩) 하여 재정의 합니다. 또는 슈퍼 클래스가 추상 클래스인 경우에는 인터페이스의 메서드 구현과 같이 추상 메서드를 반드시 오버라이딩 하여 구현하여야 합니다.

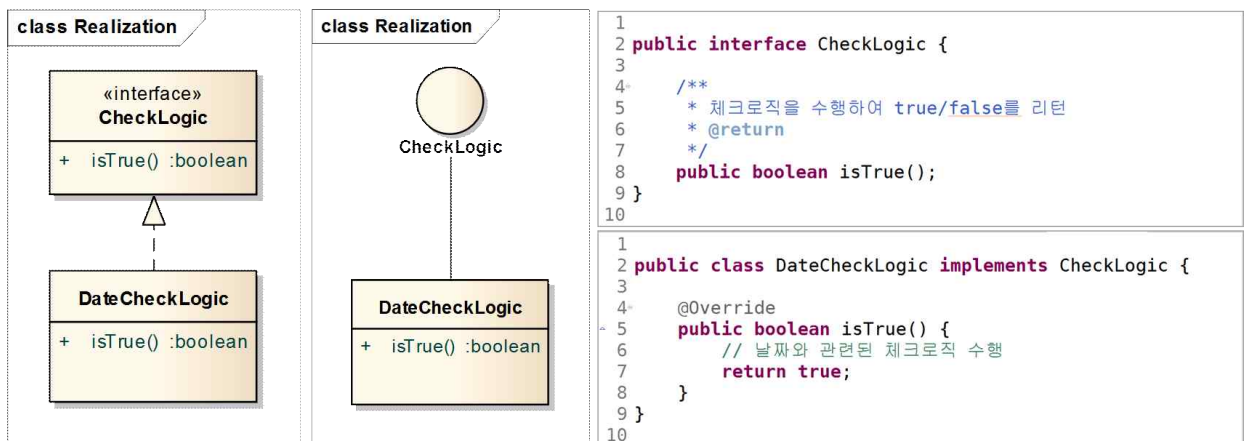


[그림 7] Generalization

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

Realization (실체화)

Realization은 interface의 spec(명세, 정의)만 있는 메서드를 오버라이딩 하여 실제 기능으로 구현 하는 것을 말합니다.



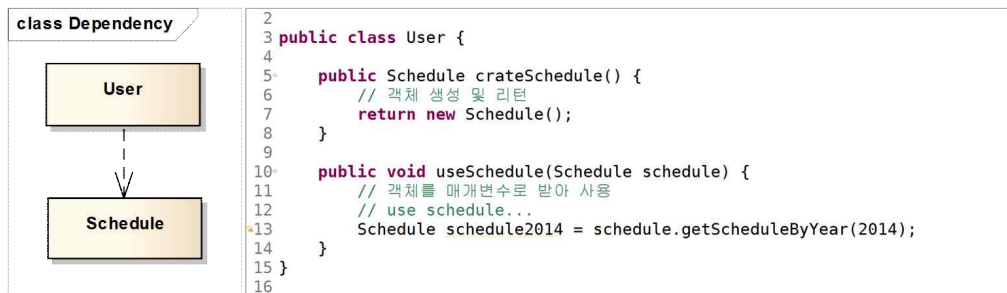
[그림 8] Realization

Realization을 나타내는 표기법은 2가지가 있습니다. 첫 번째는 인터페이스를 클래스처럼 표기하고 스테레오 타입 «interface»를 추가합니다. 그리고 인터페이스와 클래스 사이의 Realize 관계는 점선과 인터페이스 쪽의 비어있는 삼각형으로 연결합니다. 두 번째는 인터페이스를 원으로 표기하고 인터페이스의 이름을 명시합니다. 그리고 인터페이스와 클래스 사이의 관계는 실선으로 연결합니다.

자바에서는 위와 같이 implements 키워드를 사용하여 인터페이스를 구현합니다.

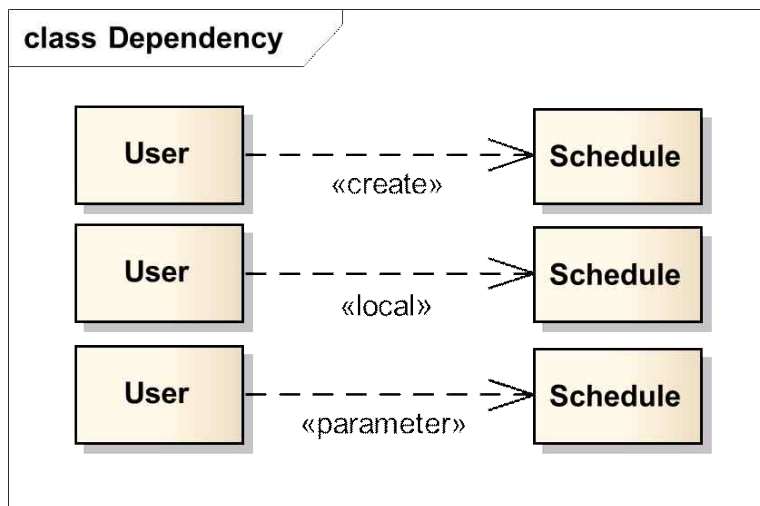
Dependency (의존)

Dependency는 클래스 다이어그램에서 일반적으로 제일 많이 사용되는 관계로서, 어떤 클래스가 다른 클래스를 참조하는 것을 말합니다.



[그림 9] Dependency

위의 그림은 자바에서 참조하는 형태에 대해 코드를 보여주고 있습니다. 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 유지하지는 않습니다.

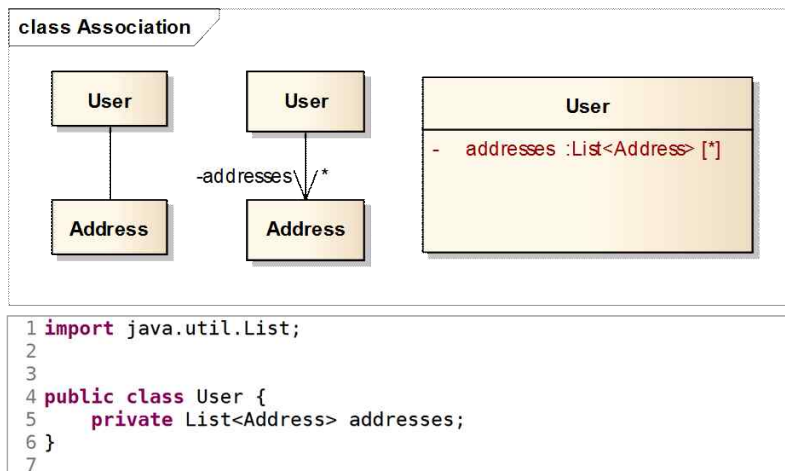


[그림 10] Dependency Stereo Type

추가로 위와 같이 스테레오 타입으로 어떠한 목적의 Dependency인지 의미를 명확히 명시할 수도 있는데 Dependency의 목적 또는 형태가 중요할 경우 사용할 수도 있을 것입니다.

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

클래스 다이어그램에서의 Association은 보통 다른 객체의 참조를 가지는 필드를 의미합니다. 아래 클래스 다이어그램은 두 가지 형태의 Association을 나타내고 있습니다.



[그림 11] Association

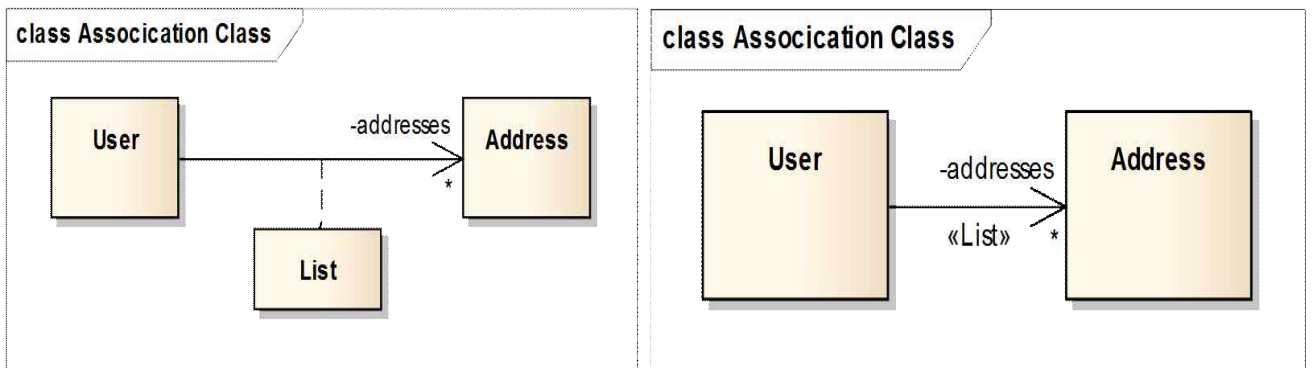
첫 번째 다이어그램은 일반적인 Association으로 단지 실선 하나로 클래스를 연결하여 표기하고 두 번째 다이어그램은 Directed Association으로 클래스를 실선으로 연결 후 실선 끝에 화살표를 추가합니다. Association과 Directed Association의 차이는 화살표가 의미하는 navigability(방향성)인데 이것에 따라 참조 하는 쪽과 참조 당하는 쪽을 구분합니다. 두 번째 다이어그램은 User에서 Address 쪽으로 화살표가 있으므로 User가 Address를 참조하는 것을 의미합니다. Navigability가 없는 Association은 명시되지 않은 것으로 User가 Address를 참조할 수도, Address가 User를 참조할 수도, 또는 둘 다일 수도 있는 것을 의미합니다. 화살표 옆에 있는 -addresses는 roleName(역할명)을 나타내고 Address가 User 클래스에서 참조될 때 어떤 역할을 가지고 있는지를 의미합니다.

*는 Multiplicity(개수)을 나타내는데 대상 클래스의 가질 수 있는 인스턴스 개수 범위를 의미합니다. 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 의미합니다. 이전에는 Multiplicity가 아닌 Cardinality로 불렸는데 “특정 집합 또는 다른 그룹에 있는 요소의 수”라는 Cardinality의 사전적 의미가 실제 인스턴스의 수가 아닌 가질 수 있는 범위를 지정하는 클래스 다이어그램에서의 의미에 적합하지 않다는 이유로 바뀌었습니다.

세 번째의 다이어그램은 두 번째의 다이어그램과 비슷한 의미를 가지고 있지만 다른 형태인 속성 표기법으로 나타낸 것입니다. 여기서 보는 바와 같이 roleName은 보통 클래스의 필드 명이 됩니다. 속성 표기법이 두 번째 클래스 다이어그램과 조금 다른 점은 여러 개의 객체에 대한 Container(컨테이너)가 List라는 것까지 알려주고 있습니다.

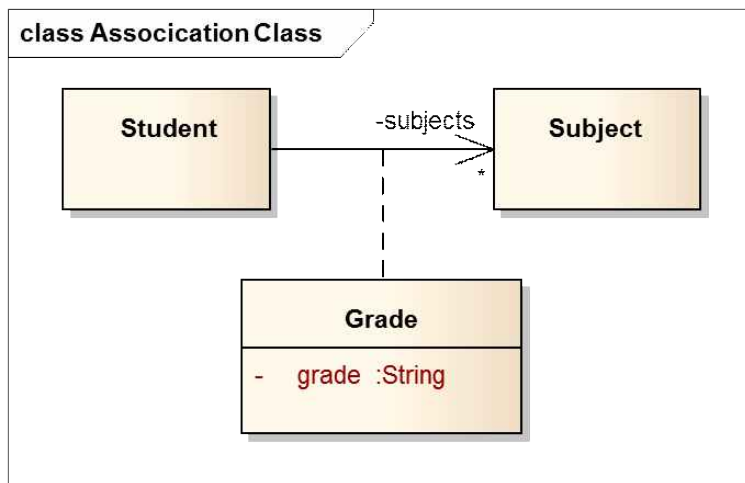
그럼 두 번째와 세 번째의 다이어그램이 완전히 동일한 의미를 가지게 하려면 어떻게 해야

할까요?



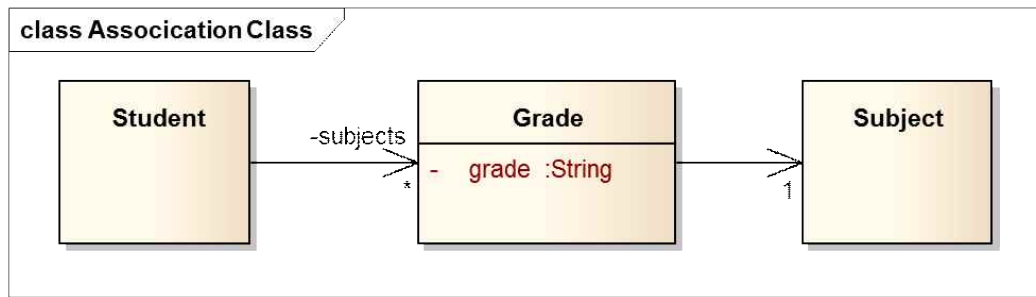
[그림 12] Association Class

위와 같이 Association Class(연관클래스)를 사용하여 어떤 종류의 컨테이너 클래스가 사용되는지 까지 나타낼 수 있습니다. 하지만 사실 특이한 클래스도 아니고 자바에서 기본으로 제공되는 List를 이렇게 표기하는건 조금 귀찮은 일일 수 있을 것 같습니다. 그래서 스테레오 타입으로 표기할 수도 있습니다. 이로써 [그림11] 세 번째 다이어그램의 속성 표기법으로 표현된 것과 모든 의미가 같아졌습니다. 보통은 클래스의 속성이 기본 제공 클래스가 아니거나 중요 또는 강조하고 싶을 때 Association 관계로 나타냅니다. Association Class는 조금 다른 의미로도 사용될 수 있습니다. 예를 들어 학생과 수강과목 클래스가 Association 관계를 가지고 있는데 단순 Association 관계가 아니라 각 관계마다 해당 과목의 학점이라는 속성이 필요하다면 어떻게 나타낼 수 있을까요? 이럴 때도 Association Class를 사용하여 나타낼 수 있습니다.



[그림 13] Association Class

물론 grade라는 값을 Subject 클래스 자체의 속성으로 할 수도 있지만 Subject의 속성이기 보다는 Student와 Subject 사이의 관계에 대한 속성이라는 관점에서 위의 다이어그램처럼 Association Class로 나타낼 수 있습니다.



[그림 14] Association Class Vs Association

또한 [그림13]의 Association Class를 풀어서 위처럼 Association 관계만으로도 나타낼 수 있습니다. [그림13]은 Student와 Subject의 관계를 나타내고 싶은데 그 관계에 대한 속성 값으로 Grade가 있는 것이고 [그림14]는 단순 3개 클래스를 Association 관계로 나타낸 것처럼 의미는 조금 다를 수 있으나 구현되는 코드는 같을 것입니다.

```

3 import java.util.List;
4
5 public class Student {
6
7     private List<Grade> subjects;
8
9 }

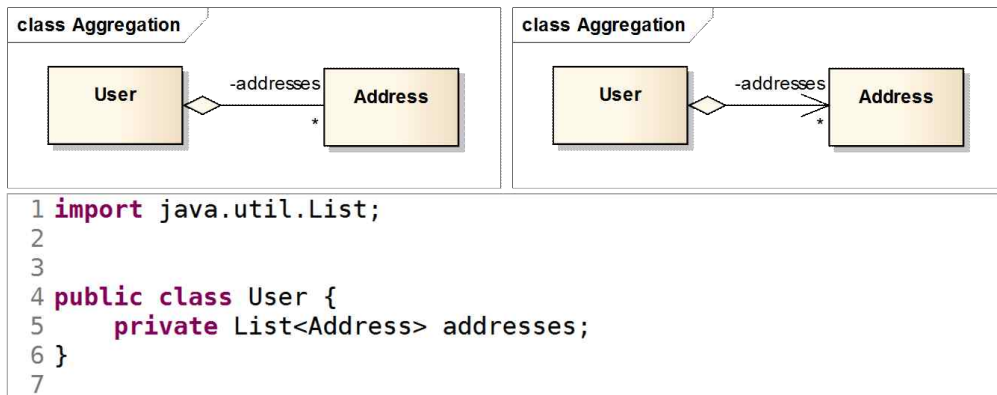
3 public class Grade {
4
5     private Subject subject;
6
7     private String grade;
8 }

3 public class Subject {
4
5     private String name;
6 }
  
```

[그림 15] Association Class Code

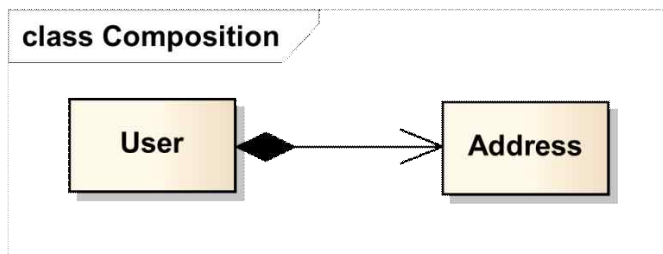
Aggregation (Shared Aggregation, 집합)

Aggregation은 Shared Aggregation이라고도 하며 Composition(Composite Aggregation)과 함께 Association 관계를 조금 더 특수하게 나타낸 것으로 whole(전체)와 part(부분)의 관계를 나타냅니다. Association은 집합이라는 의미를 내포하고 있지 않지만 Aggregation은 집합이라는 의미를 가지고 있습니다.



[그림 16] Aggregation

표기법은 위와 같이 whole과 part를 실선으로 연결 후 whole쪽에 비어있는 다이아몬드를 표기합니다. Part쪽에는 화살표를 명시하여도 되고 명시하지 않아도 됩니다. Aggregation의 다이아몬드가 이미 navigability의 방향을 표현하고 있기 때문입니다. 그런데 코드를 보시면 위에서 보았던 Association의 코드와 똑같습니다. Association과 Aggregation은 집합이라는 개념적인 차이는 있지만 코드에서는 이 차이를 구분하기 힘듭니다.



[그림 19] Composition

Composition의 표기법 또한 위와 같이 Aggregation과 비슷하지만 다이아몬드의 내부가 채워져 있다는 점만 다릅니다. 그림 Composition의 개념과 코드에서는 Aggregation과 어떤 차이가 있는지 보겠습니다. Composition은 Aggregation보다 강한 집합이라고 했습니다. 여기서 강한 집합이란 part가 whole에 종속적이어서 part가 whole의 소유입니다. 반면 Aggregation은 part가 whole에 대해 독립적이어서 whole이 part를 빌려 쓰는 것과 비슷합니다. 이러한 의미 때문에 Aggregation과는 다르게 명확하게 나타나는 점이 있습니다.

예)

