소프트웨어분석및설계(정진홍)

Lee Jun Hyeok (wnsx0000@gmail.com)

December 12, 2025

목차

1. 서론

1.1. SAD

1.1.1. SAD

1. SE

SE(Software Engineering, 소프트웨어 공학)은 좋은 품질의 소프트웨어를 최적의 시간과 비용으로 협업하여 개발하는 체계적인 절차 또는 방법론으로, 경험적으로 정해진 개발 convention이나 가이드라인임.

소프트웨어는 입력 데이터를 처리하여 출력 결과를 만들어내는 프로그램, 또는 그런 프로그램의 개발/운용/유지보수에 필요한 자료 등을 의미함. software 개발에서는 그 특성상 품질 평가가 어렵고, 여러 사용자에게 다양한 기능을 제공해야 하고, efficient한 설계 구조가 필요로 한다는 어려움이 있음.

하드웨어 성능이 발달하고 소프트웨어의 수요가 늘어남에도, 소프트웨어가 가진 특성에 따라 그 개발 속도가 수요를 따라가지 못하는 것을 소프트웨어 위기라고 함. 이에 따라 소프트웨어는 그 자체의 기능이나 성능뿐만 아니라 개발/운용/유지보수 과정에서의 efficiency 또한 중요함. 즉, 좋은 품질의 소프트웨어를 빠르게 만들 수 있어야 함.

SE의 측면에서 소프트웨어 프로젝트는 다음과 같이 분석, 설계, 모델링, 구현, 테스트, 관리 등의 절차를 거쳐 수행되는데, 본 수업에서는 특히 분석 및 설계에 대해 알아봄.

1) 분석: 고객의 비즈니스 환경과 문제, 필요기술, 우선순위를 파악함.
2) 설계: 요구사항을 어떻게 달성할 것인지 계획 및 결정함.
3) 모델링: 소프트웨어의 동작 또는 구성을 심볼, 수식 등의 표현으로 구체화함.
4) 구현: high-level 설계를 특정 프로그래밍 언어로 구현함.
5) 테스트: 목표 품질 달성을 위한 테스트 및 검증을 수행함.
6) 관리: 소프트웨어 개발 프로세스를 관리함.

소프트웨어 프로젝트의 절차 중 SAD(Software Analysis and Design)는 최적의 비용으로 원활히 협업하기 위해 필수적인 과정임.

2. Software Analysis

Software Analysis는 문제 상황을 이해하고, 요구사항을 구체화하는 단계임. 이때 동작만 기술하고 구체적인 구현은 기술하지 않음. 즉, what과 how 중 what에만 집중함.

1) 도메인 분석: 문제를 이해하기 위한 관련 도메인 및 배경 지식에 대해 분석함.
2) 문제 정의: 해결하려는 문제를 범위를 좁혀 구체화함.
3) 요구 추출: SW가 무엇을 해야하는지 요구사항을 취합함.
4) 요구 분석: SW가 요구사항에 따라 실제로 무엇을 해야하는지 결정함.
5) 요구 명세화: SW가 어떻게 작동하는지에 대한 자세한 사항을 기술함.

3. Software Design

Software Design은 요구사항에 대한 구현을 정하는 단계임. 이때 구체적인 구현보다는 요구사항에 대한 개념적 해결책에 중점을 둠.

1) 시스템 설계: 시스템의 각 부분을 하드웨어/소프트웨어적으로 어떻게 구현할지 결정함.
2) 아키텍쳐: 시스템을 서브시스템으로 분할하고 서브시스템 간의 상호작용을 결정함.
3) 상세 설계: 각 서브시스템의 상세 사항(자료구조, class 등)을 결정함.
4) 인터페이스 설계: 사용자와 시스템 간의 상호작용을 결정함.
5) 자료 설계: 자료를 어떻게 저장할지 결정함.

4. 모델링

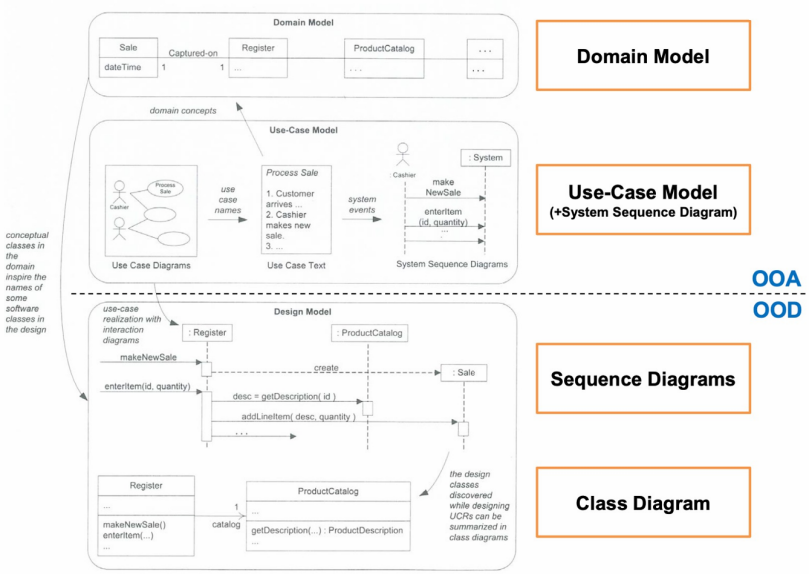
Modeling(모델링)은 sw의 동작과 구조를 그림으로 나타내는 단계, modeling 과정에서 시스템이나 현상을 단순화하여 추상적으로 표현한 것을 Model(모델)이라고 함. 다음과 같은 모델링들이 존재함.

1) Use-case modeling : 사용자의 시스템 use-case를 순서대로 나타냄.
2) Static modeling : sw에 존재하는 class나 object를 나타냄.
3) Dynamic modeling : 시스템의 상태 및 작업 수행 과정을 나타냄.

1.1.2. OOAD

1. OOAD

현대 소프트웨어의 대부분은 객체지향으로 구현됨. 이에 따라 소프트웨어 분석과 설계는 OOAD(Object Origented Analysis and Design)로 구체화됨. OOA(Object Oriented Analysis)는 문제와 관련된 개념을 domain concept이나 object로 표현하고 기술함. OOD(Object Oriented Design)는 object를 구체적으로 정의하고(static), object들이 상호작용하며 요구사항을 만족시키는지 설계함(dynamic).



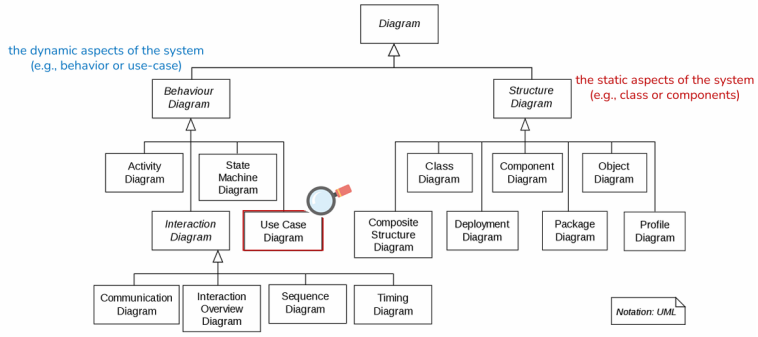
OOA에서는 requirment로부터 domain object를 파악하고 사용자와 시스템의 상호작용을 분석함. OOD에서는 그런 domain object와 상호작용을 실제로 어떻게 구현할지를 design함. 즉, OOA에서는 무엇(what)을 할지를, OOD에서는 그걸 어떻게(how) 할지를 결정함.

본 수업에서는 OOAD와 관련하여 UML과 design pattern도 다룸.

2. UML

UML(Unified Modeling Language)은 OOAD에 의한 내용을 모델링하는 데에 사용되는 표준 다이어그램 표기법 또는 modeling 언어임. 당연하게도 UML은 그 자체로 개발 방법론이나 프로그래밍 언어는 아님.

UML에는 다양한 diagram에 대한 notation이 정의되어 있지만, 물론 모든 다이어그램을 항상 그리는 건 아니고 프로젝트별로 필요할 때 취사선택해서 활용함.



3. Object Modeling

OOD의 object modeling은 class, package, attribute, operation 등을 정의하는 Static Object Modeling과, 동작, method 등을 정의하는 Dynamic Object Modeling으로 나뉨. 이 두 modeling은 동등하게 중요하며, unified process에서는 서로를 참고하며 반복적으로 그리고 개선시킴.

static object modeling에서는 class 등 시스템의 구조와 구성 요소를 정적인 측면에서 그림.

dynmaic object modeling에서는 class의 instance로 생성된 object가 특점 시점에 다른 object와 어떻게 상호작용하는지를 그림. 이에 따라 object, 시간, 호출 순서, 논리적 흐름이 나타나게 됨.

static modeling에서의 diagram은 Structure Diagram, dynamic modeling에서의 diagram은 Behaviour Diagram이라고 함. 뒤에서 각각에 대해 어떤 방식들이 있는지를 자세히 알아봄.

Design Pattern은 sw 설계 과정에서 자주 등장하는 문제의 해결책을 재사용하기 쉽게 패턴화한 것임.

1.1.3. Object Oriented SW Development Process

1. Waterfall and Iterative Model

Waterfall Model은 한 번에 모든 것을 계획하고 각 단계를 한 번씩만 순차적으로 실행하는 sw 개발 모델임. 반면 Iterative 또는 Agile Model은 각 단계를 한 번에 완전히 끝내는 게 아니라, 단계를 짧은 주기로 반복 수행하고, 피드백해서 발전시켜 나가는 sw 개발 모델임.

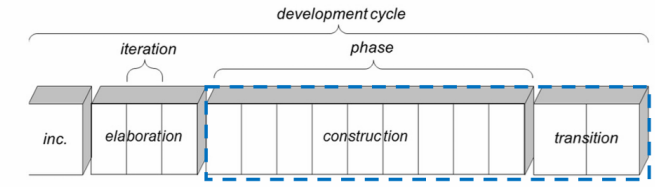
waterfall model은 구조가 단순하고 이해하기 쉽지만 유연성이 매우 낮음. iterative model은 요구사항 변경에 유연하게 대처할 수 있지만 전체적인 계획을 잘 설계해야 하고 충분한 소통이 필요함.

2. UP

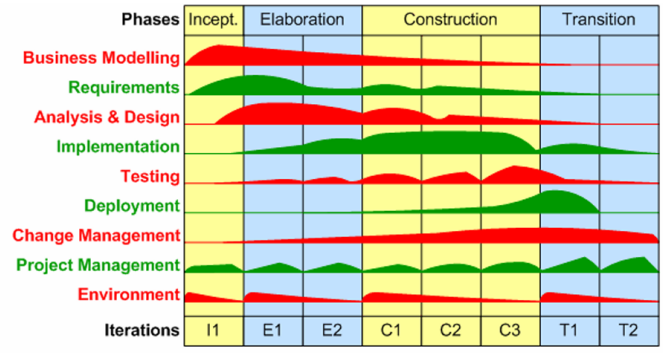
UP(Unified Process)는 object oriented sw를 개발할 때 de-facto(사실상 표준)로 사용되는 iterative model로, 고정된 기간 내에 작은 waterfall 과정을 반복적으로 수행함. UP는 다음과 같은 4개의 phase(단계)로 구성됨.

1) Inception(도입부): 프로젝트의 방향, 요구사항, use-case, 범위, 비용 등을 대략적으로 파악함.
2) Elaboration(세분화): inception 단계의 계획과 내용을 자세히 확장하여 초기 계획을 확립하고 risk를 관리함.
3) Construction(구축): 이전에 design한 내용을 바탕으로 sw를 개발함.
4) Transition(전환): 개발한 sw를 사용자들이 사용할 수 있도록 테스트, 배포, 유지보수 등을 수행함.

4개의 단계. phase로 나뉘어져 있다. incept, elaboration, constructoin, transition. 각 단계별로 여러 iteration이 존재한다. 이런 iteration마다 requirment/OOA에 대한 OOD/구현이 수행됨.



UP는 아래와 같이 9개의 discipline(분야)로 구성되고, 이는 병렬적으로 수행됨. 본 수업에서는 business modeling, requirments, analysis&design에 대해 중점적으로 다룸. 각 반복에서는 거의 모든 discipline이 동시에 수행되지만, 단계나 반복에 따라 각 discipline 별 비중이 다름.



초기 단계에서는 각 작업의 risk와 value를 평가하고, 핵심 아키텍처를 정의하고, UML로 modeling함. 또한 팀원의 분야에 따라 역할과 책임을 부여하고, 지속적으로 평가와 피드백 과정을 거침. 이런 과정에 따라 다양한 산출물(artifact)이 도출됨.

물론 이런 모델과 방식에는 정답이 있는 것은 아니고, 관습적으로 괜찮다고 여겨지는 것들임.

1.2. OOP

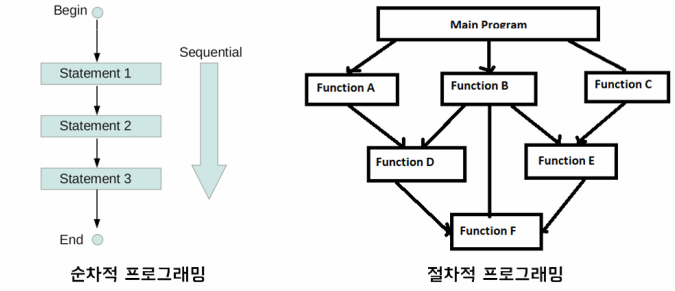
본 수업에서는 OOP를 java를 활용해 설명함.

1.2.1. OOP

1. OOP

OOP(Object Oriented Programming)는 sw를 obejct라는 기본 단위로 나누고, sw의 동작을 obejct 간의 상호작용으로 구성하는 프로그래밍 패러다임임. 즉, 데이터와 함수를 하나의 덩어리인 object로 구성하고, 이 object를 조합해 큰 문제를 해결하는 방식(bottom-up)임.

초기의 프로그래밍 방식은 주로 sequential/procedural programming이었음. Sequential Programming(순차적 프로그래밍)은 입력을 순차적으로 처리하는 방식임. PP(Procedural Programming, 절차적 프로그래밍)은 sw를 여러 개의 작은 procedure로 나누고, sw의 동작을 procedure의 단계적 호출(top-down)로 구성하는 방식임. PP는 직관적이지만 데이터 처리 방식만 구조화한 것으로 데이터에 대한 구조화는 잘 하지 못하고, 유지보수가 어려움.



순차적 프로그래밍 절차적 프로그래밍

반면 OOP는 한번 class를 잘 정의해 두면 유지보수가 굉장히 편리함.

2. PP vs. OOP

PP와 OOP는 서로 반대되는 개념이라기보단 프로그래밍 패러다임이며, 각각 장단점이 존재함. 다음과 같이 특징을 정리할 수 있음.

	절차적 프로그래밍 (PP)	객체지향 프로그래밍 (OOP)
접근 방식	Top-down	Bottom-up
구현 관점	전체 기능 동작 고려 ->각 기능 구현	객체 설계 -> 객체 상호 작용 설계
구성 요소	함수	객체
장점	객체 관련 오버헤드를 줄여 빠르게 동작	코드의 중복을 줄이고 재활용성이 높음 구조 파악 및 디버깅이 쉬움
단점	구조가 복잡해지고 코드 중복이 심해짐 순서를 지켜야 해서 재활용성이 떨어짐	설계에 많은 시간이 요구됨 객체가 많아지면 오버헤드가 발생 할 수 있음
용도	한정된 자원/특수 목적 (보안, 속도)	대규모 협업 프로젝트 (생산성 중시)

1.2.2. Object와 Class

1. Object와 Class

객체 모양을 선언하는 틀을 class, class에 따라 생성된 실체를 Object 또는 Instance라고 함. object는 상태와 행위를 가지고 있으므로, 이 둘을 하나로 묶어서(encapsulation) 관리할 수 있음.

코드 측면에서 class는 field(멤버 변수, 상태, 속성)와 method(멤버 함수, 행위)로 구성되고, 다음과 같이 정의함. java에서 class는 type임.

```
public class Animal
{
    public String name;     // 필드
    public int age;

    public int getAge()         // 메소드
    {
        ...
    }
}
```

객체는 항상 아래와 같이 new 키워드를 사용하여 생성함. 이때 new 키워드는 항상 객체의 생성자를 호출하기 때문에 반드시 ()를 붙여줘야 함. 생성자를 작성하지 않았더라도 기본생성자가 삽입되어 정상적으로 동작함.

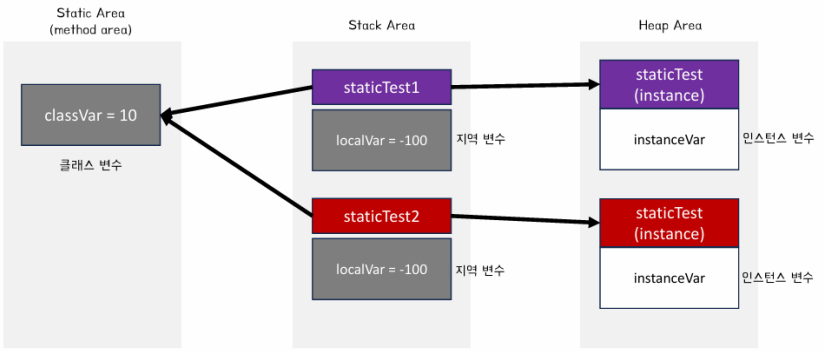
new 키워드가 사용된 수식(expression)은 해당 객체의 reference를 반환함. 이렇게 생성한 객체의 멤버에는 .으로 접근함.

```
Circle pizza;
pizza = new Circle();
pizza.getArea();
```

2. 용어 정리

object와 class 관련해서 몇 가지 용어 정리를 하고 넘어가자.

Instance Variable은 각 instance에 존재하는 변수로, 다른 object와 공유되지 않음. Class Variable은 class가 가지고 있는 변수로, class에 속하는 object들이 공통적으로 사용할 수 있음. class variable은 class에서 type 앞에 static을 붙여 정의되고, 딱 한 번만 생성됨.



Operation은 class에서 수행되어야 할 행위를 선언만 한 것으로, 구현을 고려하지 않은 것임. 즉, abstract method임. Method는 특정 class에서 실제로 구현까지 한 행위를 의미함.

PP에서는 object의 상태와 행위를 각각 구조체와 함수 등으로 표현해야 하는데, 이 경우 데이터와 행위가 분리되어 있음.

1.3. Four Pillars in OOP

OOP의 철학은 abstraction, encapsulation, inheritance, polymorphism으로 구성되는 것으로 이해할 수 있음.

1.3.1. Abstraction

Abstraction(추상화)은 대상의 중요한 특성에만 집중하고, 덜 중요한 세부 사항은 고려하지 않는 것을 말함.

design 시에는 abstaraction을 적용해 추상적인 부분과 구체적인 부분을 나눠 그 과정을 단순화할 수 있음. 즉, high-level 아이디어(what)와 전체 아키텍처를 먼저 정하고, 이후 low-level 세부 구현(how)을 정할 수 있음.

abstraction은 data와 procedure에 대한 것으로 나눌 수 있음. Data Abstraction은 여러 data type을 하나로 묶어 더 높은 단계로 generalization하는 것임. Procedure 또는 Control Abstraction은 procedure에 대한 호출 방법과 동작만 알 수 있게 하고, 내부의 세부적인 구현은 감추는 것임.

OOP에서 Generalization(일반화)은 여러 개의 subclass들의 공통점을 찾아내 하나의 super class를 도출하는 것을 말함.

abstraction은 interface, generalization, ADT 등으로 쉽게 이해할 수 있음.

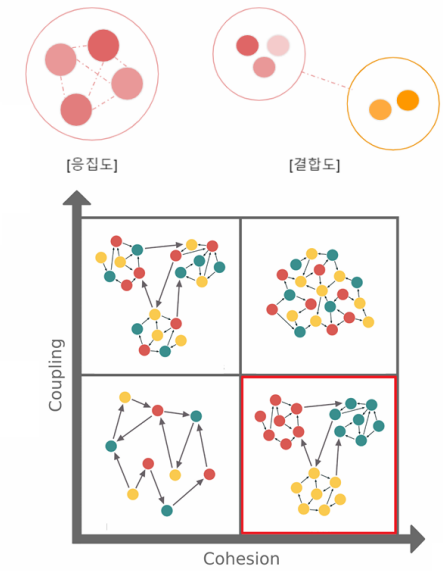
1.3.2. Encapsulation

Encapsulation은 상태와 행위를 하나로 묶어 캡슐로 만드는 추상화임. 내부의 내용은 접근 제어자(private, default, protected, public)에 의해 공개 범위를 지정함.

내부의 공개 범위를 최소화하고, 다른 class에 대한 의존도를 낮추는 것이 좋음.

encapsulatio과 관련하여 cohesion과 coupling을 고려할 수 있음. cohesion이 높고 coupling이 낮도록 design하는 것이 이상적인데, encapsulation은 이런 design을 도움.

- *Cohesion(응집도)는 class 안의 요소들이 얼마나 밀접하게 관련되어 있는지를 말함.*
- *Coupling(결합도)은 다른 class에 대해 얼마나 의존적인지를 말함.*



1.3.3. Inheritance

1. Inheritacne

Inheritance(상속)은 super class에 정의된 상태와 행위를 subclass가 묵시적으로 가지도록 하는 것임. 이런 계층 구조에 의해 공통된 상태와 행위를 반복적으로 정의할 필요 없이 간단히 나타낼 수 있음.

다음과 같이 inheritance를 나타냄.

```
public class Student extends Person
{
    ...
}
```

적절한 계층 구조를 정의하기 위해 Is-A 관계를 따질 수 있음. 즉, A is kind of B가 성립하면 A는 B의

subclass가 될 수 있음.

2. Method Overriding

Method Overriding(메소드 오버라이딩)은 super class에 정의된 method를 subclass에서 재정의하여 그 정의를 덮어쓰는 것임. 이에 따라 method를 동일한 이름으로 각 subclass 별로 다르게 구현하여 사용할 수 있음.

```java
public class Vehicle { // 추상화를 통한 상위클래스 정의

    String model;
    String color;
    int wheels;

    void moveForward() {
        System.out.println("전진합니다⏩");
    }

    void moveBackward() {
        System.out.println("후진합니다⏪");
    }
}
```

```java
public class MotorBike extends Vehicle {

    boolean isRaceable;

    // 메서드 오버라이딩 -> 기능 재정의
    @Override
    void moveForward() {
        System.out.println("🏍가 앞으로 전진합니다⏩");
    }

    public void stunt() {
        System.out.println("🏍로 묘기를 부립니다.");
    }
}
```

overriding의 적용 메커니즘은 언어별로 다름. c++에서는 static binding(정적 바인딩)이 일어나 컴파일 시점에 참조 변수의 타입에 따라 호출되는 method가 결정되고, java는 dynamic binding(동적 바인딩)이 일어나 런타임에 참조 변수가 참조하고 있는 실제 object의 method가 호출됨.

```cpp
class Vehicle {
public:
    void moveForward(){
        cout << "Vehicle" << endl;
    }
};

class MortorBike : public Vehicle {
public:
    void moveForward(){
        cout << "MortorBike" << endl;
    }
};
```

```cpp
int main() {
    Vehicle* a = new MortorBike();
    a->moveForward();

    MortorBike* b = new MortorBike();
    b->moveForward();

    delete a;
    delete b;

    return 0;
}
```

출력결과:

```
Vehicle
MortorBike
```

```java
public class Vehicle {
    public void moveForward(){
        System.out.println("Vehicle");
    }
}


public class MortorBike extends Vehicle {
    @Override
    public void moveForward() {
        System.out.println("MotorBike");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Vehicle a = new MortorBike();
        a.moveForward();

        MortorBike b = new MortorBike();
        b.moveForward();
    }
}
```

출력결과:

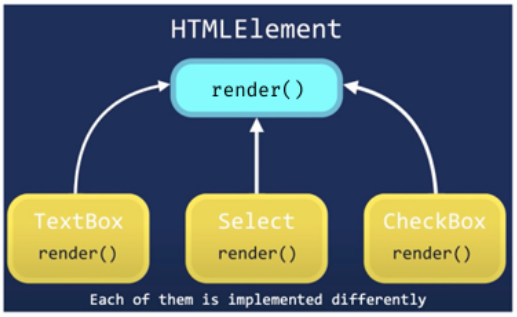
```
MortorBike
MortorBike
```

1.3.4. Polymorphism

1. Polymorphism

Polymorphism(다형성)은 어떤 object의 상태나 행위가 상황에 따라 여러 가지 형태를 가지는 것임. 이에 따라 코드를 더 간결하고 직관적으로 표현하고, 코드 수정이 용이하도록 할 수 있음.

super class type의 참조 변수는 모든 하위 type class를 가리킬 수 있으므로 polymorphism을 구현할 수 있음. 또한 상위 abstract class나 operation(abstract method)을 정의해 polymorphism을 더 직관적으로 구현할 수도 있음.



2. Abstract Class

abstract method를 가지는 class는 abstract class로 정의해야 함. abstract class는 object를 직접 생성할 수 없고, 해당 abstract class를 상속받은 subclass는 abstract class가 아니라면 abstract method를 overriding해서 구현해야 함.

```
public abstract class HTMLElement {

    public abstract void render(); // operation

}
public class TextBox extends HTMLElement {
    @Override
    public void render(){ // method
        // implementation here
    }
}
```

2. OOA

2.1. Requirment Analysis

2.1.1. Requirment Analysis

1. Requirment Analysis

Requirment(요구사항)는 개발할 시스템 또는 sw가 반드시 따라야 하는 기능, 성능, 조건 등을 말함. Requirement Analysis는 이런 requirment를 고안하고 정리하는 과정으로, sw 사용자와 개발자 모두가 명확히 이해할 수 있어야 함. UP에서는 이런 requirment가 반복적으로 분석/수정됨.

2. FURPS

requirments의 유형은 FURPS로 생각해 볼 수 있음. FURPS는 requirment를 다음과 같이 5가지로 분류함. 이 중 function은 Functional Requirment(기능적 요구), 나머지는 Non-functional Requirment(비기능적 요구)임.

1) Function(기능): 시스템의 외형적 기능에 대한 requirment.
2) Usablility(사용성): 사용자가 쉽고 편리하게 사용할 수 있는지에 대한 requirment. (ex. UI, document 등)
3) Reliability(신뢰성): 시스템이 안정적으로 돌아갈 수 있는지에 대한 requirment. (ex. 오류 복구 등)
4) Performance(성능): 시스템의 성능에 대한 requirment. (ex. latency, throughtput 등)
5) Supportability(지원): 시스템 유지보수에 대한 requirment. (ex. 업그레이드, 기술 지원 등)

3. Requirment 분석서 작성 방법

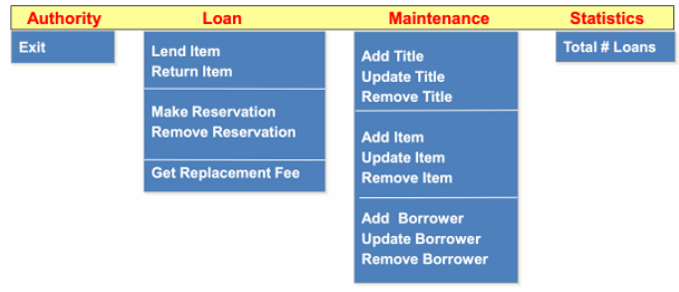
Requirment 분석서에는 아래와 같은 항목들을 작성할 수 있음.

1) 문제: 시스템으로 해결하고자 하는 문제.
2) 배경: requirment 이해에 사용되는 배경 지식.
3) 환경 및 시스템: 시스템, 서브시스템, 환경 등에 대한 내용.
4) 기능적 요구: functional requirment에 대한 use-case diagram/specification.
5) 비기능적 요구: non-functional requirment에 대한 supplementary specificaion.

functional requirment에 대해서 다음과 같이 reference 번호와 function을 정리할 수 있음.

	Function
R3.1	Add Borrower
R3.2	Remove Borrower
R.3.3	Update Borrower
R4.1	Validate System Access
R5.1	Get Total # of Loans

또한 다음과 같이 이렇게 정리한 function들을 활용해 interface prototype을 만들 수 있음.



이런 개념은 당연히 실전적이라기보단 교과서적인 내용임.

2.2. Use-case Analysis

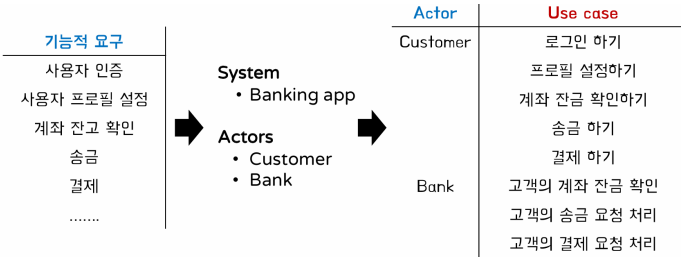
2.2.1. Use-case Analysis

1. Use-case Analysis

Use-case Analysis는 사용자 관점으로 시스템 use-case를 modeling하는 것으로, 주로 functional requirment에 대해 수행함. use-case는 시스템이 제공하는 기능으로 이해할 수 있음. 이는 다음과 같은 과정에 의해 수행됨.

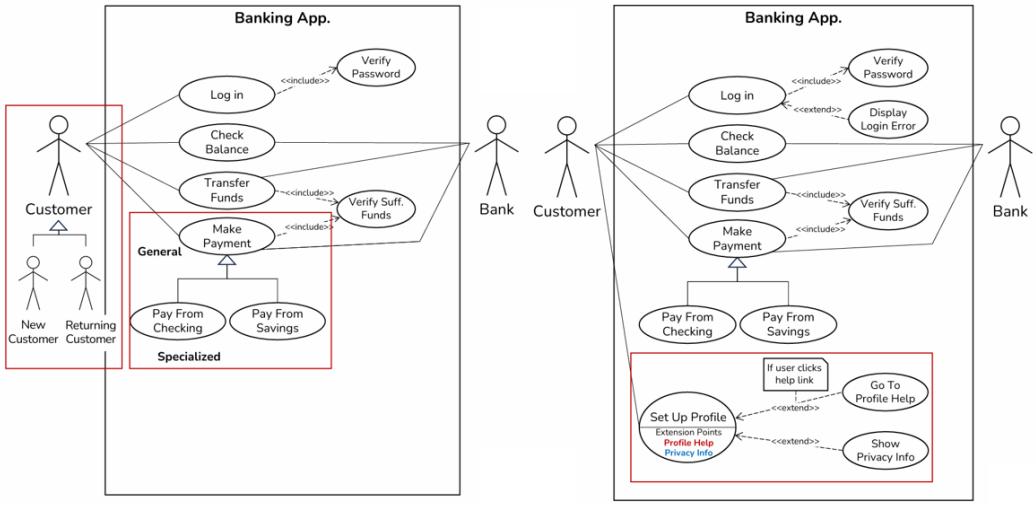
1) 시스템 requirment anaylsis 수행.
2) 시스템 및 actor 정의.
3) use-case 정의.
4) use-case diagram 그리기.
5) use-case specification 작성.

1번부터 3번까지의 과정은 다음 그림과 같이 간단히 수행이 가능함. 이때 actor는 시스템을 사용하려고 하는 대상(사용자, 다른 시스템 등)임.



2. Use-case Diagram

Use-case Diagram은 use-case를 modeling한 것으로, UML에 의하면 다음과 같이 그릴 수 있음.



- *시스템: 만들려고 하는 시스템은 사각형으로 그림. 사각형 내부는 시스템 내부, 외부는 시스템 외부임.*
- *Actor: actor는 stick figure로 시스템 외부에 그림. 이때 시스템 사용을 시작하는 Primary Actor는 시스템 왼쪽에, primary actor의 동작에 반응하는 Secondary Actor는 시스템 오른쪽에 그림.*
- *Use-case: use-case(시스템이 제공하는 기능)는 타원형으로 그림. use-case 이름은 동사로 시작하며 간결히 표기함. use-case는 논리적 순서로 배치함.*
- *Association: actor와 use-case 사이의 상호작용이 가능한 경우 실선을 그림.*
- *Inclusion(포함): base use-case(A)가 수행되면 included use case(B)는 반드시 수행되어야 하는 경우 «include»와 함께 점선으로 $A \rightarrow B$ 화살표를 그림.*
- *Extension(확장): base use-case(A)가 수행되고 특정 조건을 만족시키는 경우 extension use case(B)가 수행되는 경우 «extend»와 함께 점선으로 $A \leftarrow B$ 화살표를 그림. Extension Points(확장 지점)를 정의해 extension 사항을 작성해두거나, 화살표에 조건을 붙여 명시할 수도 있음.*
- *Generalization(일반화): 유사한 actor나 use-case는 공통된 것들끼리 묶어 일반화 관계를 나타낼 수 있음.*

2.2.2. Use-case Specification

Use-case Specification은 use-case에 대한 더 상세한 설명을 담고 있는 문서임. 다음과 같은 사항들을 작성함.

1) Use-case 이름
2) Actor
3) 목적: actor가 해당 use-case를 실행하는 목적.
4) 시작 조건: actor가 해당 use-case를 실행하기 위한 선행 조건.
5) 관련 use-case: 다른 use-case와의 연관성.
6) 사건의 흐름: actor 및 시스템의 동작을 시간 순으로 나열.
7) 종료 조건: use-case 종료 후 시스템의 상태.

사용 사례	Transfer Funds (송금)
액터	Customer, Bank
목적	송금이 허용된 고객의 계좌로부터 요청받은 금액을 요청받은 계좌로 이체 하고 이에 대한 기록을 남김
시작 조건	고객이 로그인 된 상태여야 하고 송금 가능한 계좌를 가지고 있어야 함
관련 사용 사례	(Verify Sufficient Funds) 사용 사례를 포함
사건 흐름	**맥터 측 액션** / **시스템 측 반응** 1. [고객] 송금 명령 선택 / 2. 송금 화면 디스플레이 3. [고객] 이체할 계좌 및 금액 입력 4. [은행] 잔고 확인/이체 계좌 정보 확인 / 5. 이체할 계좌 정보 디스플레이 6. [고객] 이체 확인/실행 버튼 클릭 / 7. 사용자 인증 화면 디스플레이 8. [고객] 사용자 인증 9. [은행] 송금 실행 및 이력 남기기 / 10.a 송금 성공 시 성공 화면 출력 10.b 송금 실패 시 실패 화면 출력
종료 조건	계좌 이체 (송금) 결과를 고객에게 보여줌

2.3. Domain Model

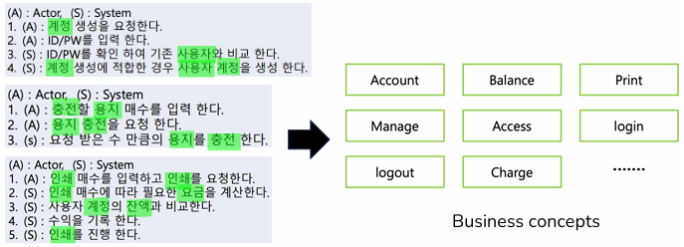
2.3.1. Domain Model

Domain Model은 domain에서 개념 또는 실세계의 객체에 대한 modeling임. domain은 sw로 다루고자 하는 영역이고, domain object는 domain 내의 conceptual class임. domain model은 주로 use-case analysis 이후에 그림.

domain model을 그리는 것은 domain 내부의 domain object, 속성, 관계, 동작 등을 대략적으로 파악하고, 문제 이해와 의사소통을 원활히 하며, design 단계에서의 class diagram과의 표현적인 차이를 줄이는 데(lower representation gap)에 도움을 줌. 이에 따라 domain model을 conceptual class diagram이라고도 함. 즉, domain object와 그 추상적인 관계를 파악하는 과정으로, 이때의 domain object는 software object와는 구별됨.

domain modeling은 다음과 같은 과정을 거쳐 수행됨. 이때 domain object가 가지는 operation은 고려하지 않고, domain object와 domain object 간의 association(연관 관계), domain object가 가지는 대표적인 attribute만 그림.

1. *목표 domain의 business concept을 나열함. use-case analysis specification에서 자주 등장하는 용어(명사)를 활용하는 것이 편리함.*



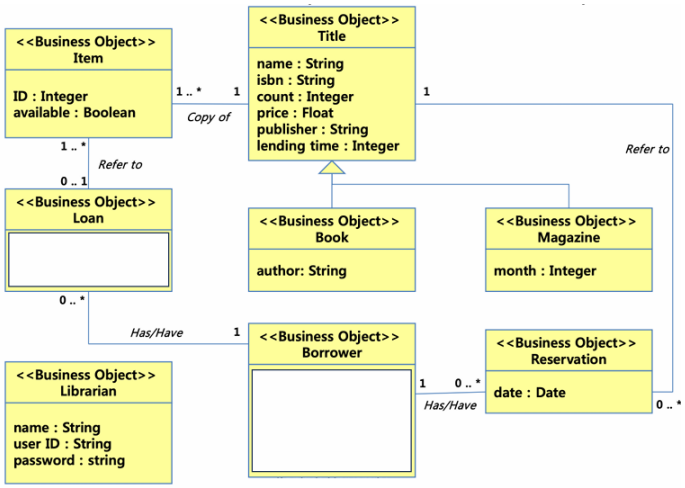
2. *concept에 대응되는 domain object를 정의함. 이때 공통된 특성을 가지는 concept들은 generalization해서 추후에 특성을 상위 domain object에만 작성할 수 있음.*

3. *domain object 간의 association을 정의함. association은 obejct-동사-object 꼴이고, 어떤 동사인지에 따라 association을 몇 가지 association category로 분류할 수 있음.*

Association Category	Associations
A has B	ManagementSystem - Printer ManagementSystem - Account Account – User Account – Manager

4. *role/multiplicity를 정의함.*

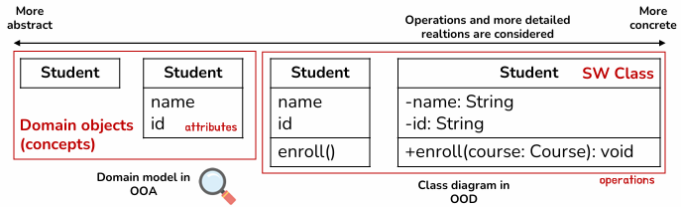
5. *attribute를 추가함. 간단하거나 pure data value(ex. boolean, date, string, time 등)인 것만 추가함. 만약 pure data가 아닌 경우 다른 domain object로 빼서 그리는 것이 좋음.*



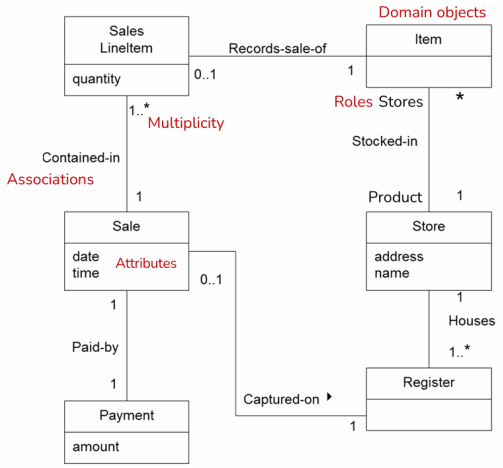
domain model을 정의해 두면 design 단계에서 class 이름을 정의된 그대로 가져다 쓴다거나 하는 편리함이 존재함.

2.3.2. Conceptual Class Diagram

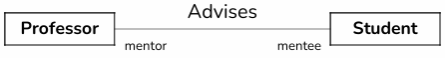
UML의 Class Diagram은 시스템의 class와 class 간의 관계를 표현하는 diagram임. OOA에서는 domain model을 그릴 때, OOD에서는 class diagram을 그릴 때 사용됨.

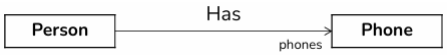


domain model(conceptual class diagram)을 UML의 class diagram으로 그리면 다음과 같음.



- *Association: 두 class 사이에 association이 존재하면 선으로 이음. 이때 양쪽 class가 서로를 인식할 수 있는 양방향 association의 경우 화살표를 그리지 않고, 한쪽 class만 반대편을 인식할 수 있는 단방향 association의 경우 인식할 수 있는 쪽에서 반대쪽으로 화살표를 그림.*





- *Role: association을 가지면 각 class는 해당 관계에서 특정 역할을 가지는데, 이를 선 위에 작성함. role 이름은 실제 코드 작성 시에 해당 이름의 변수명을 사용하는 것으로 이해할 수 있음.*
- *Multiplicity(다중성): association에서 관련된 object의 개수를 작성함. 1인 경우는 생략하기도 함. 다음과 같은 방식으로 작성함.*

다중성 표기	의미
1	엄밀하게 1 (생략)
*	0 또는 그 이상 (0 생략)
0..*	0 또는 그 이상
1..*	1 또는 그 이상
0..1	0 또는 1
2..5	2에서 5사이의 값
1, 2, 6	1 또는 2 또는 6
1, 3..5	1 또는 3에서 5사이의 값

2.4. SSD

2.4.1. SSD

1. SSD

SSD(System Sequence Diagram)는 하나의 use-case에 대해 actor와 시스템 사이의 system event 및 상호작용을 순차적으로 modeling한 것으로, 특정 use-case에 있어서 actor와 시스템 사이에 어떤 이벤트가 발생하는지 파악하기 위한 그림임. 이때 일반적으로 SSD는 해당 use-case에 대해 main success 시나리오를 나타냄. SSD는 주로 use-case analysis 이후에 그림.

System Event는 시스템과의 상호작용을 위해 actor에 의해 발생하는 event임. system event 이름은 동사+객체명으로 작성하는 것이 관례임(ex. enterInfo).

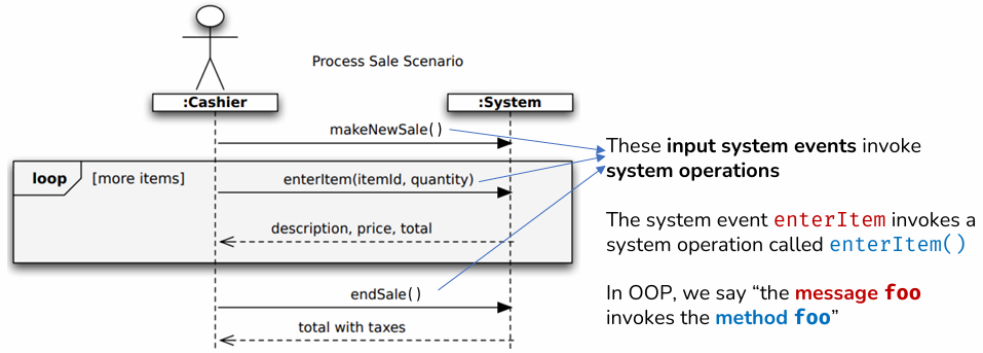
Use Case	Login
Actor	User, Manager
Purpose	User와 manager가 시스템에 접속하기 위해 로그인 할 수 잇도록 한다.
Overview	ID/PW 를 입력 받아 계정과 비밀번호가 일치하는 경우 user or manager로 로그인 한다. 일치하는 계정이 없는 경우 로그인 되지 않는다.
Type	Primary
Cross Reference	Functions: R 1.1, Use Cases:
Pre-Requisites	N/A
Typical Courses of Events	(A) : Actor, (S) : System 1. (A) : ID/PW를 입력 한다. 2. (A) : 로그인을 요청한다. 3. (S) : 일치하는 계정이 있는지 검사 후 존재 하는 경우 user or manage로 접속 승인 한다.
Alternative Courses of Events	...
Exceptional Courses of Events	E1. 일치하는 계정이 없는 경우 접속되지 않는다.

Namining convention for system events

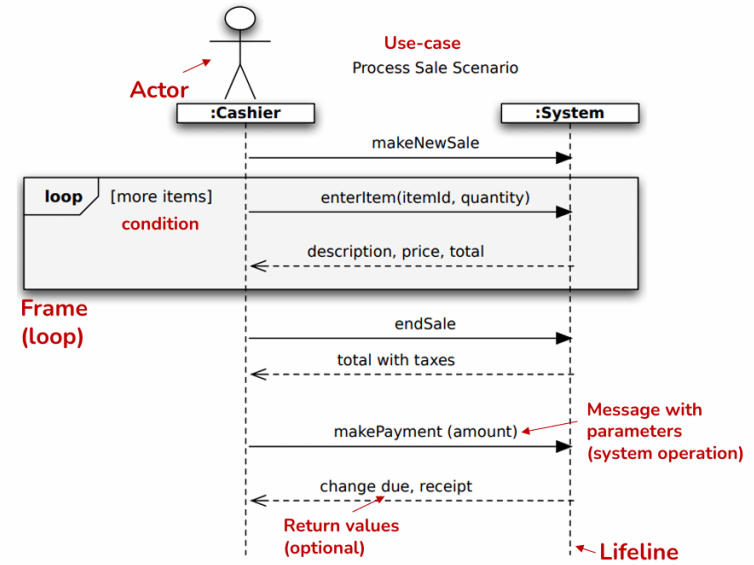
- Should be expressed at the level of intent
- With a verb + an object like "enterInfo"

Use Case	Name of Actor-Activated Event
1. Login	enterInfo
	reqLogin
2. Logout	reqLogout
3. Make Account	reqMakeAcc
	enterAccInfo
	reqAccount
4. Identify Balance	reqBalance
5. Recharge Balance	enterFee
	reqRecharge
6. Request Print	enterSheet
	reqPrint
8 Identify Paper	req Identify Paper
9. Recharge Paper	enterPaperNum
	reqCharge
10. Identify User	reqUserInfo
11. Identify Money	reqMoneyInfo

System Operation은 시스템이 public interface를 통해 actor에게 제공하는 operation이고, system operation들의 집합을 System Interface라고 함. SSD에서 시스템은 black box로 취급되고, actor에 의해 발생하는 system event가 system operator를 호출하여 동작함.



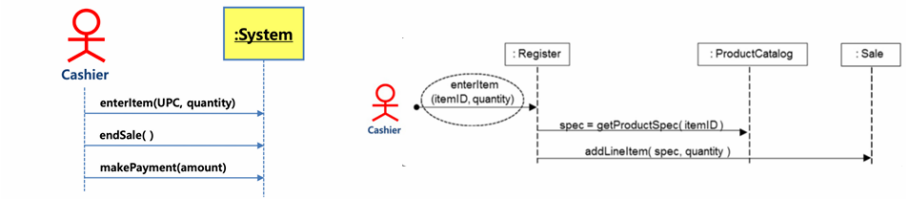
SSD는 시스템, actor, 시스템 이벤트로 구성됨. UML의 SSD는 다음과 같이 그림.



- *actor는 stick figure로, actor에서 시스템 쪽으로의 system operation(event)는 실선 화살표로, 시스템에서 actor로의 상호작용은 점선 화살표로 그림. lifeline은 점선으로 그림.*
- *system operation(event)과 return value는 화살표 위에 작성함. system operation에 parameter가 있으면 함께 작성함.*
- *반복문은 loop 사각형을 그리고 condition을 작성함.*
- *조건문은 alt(alternative) 사각형을 그리고 condition을 작성함.*

2. SSD vs. Sequence Diagram

analysis에서의 modeling인 SSD는 actor와 시스템 간의 상호작용을 순차적으로 표현하며, 시스템의 외부 동작에 집중함. 반면 design에서의 modeling인 Sequence Diagram은 actor와 object 간의 메시지 흐름을 표현하며, 시스템의 내부 동작에 집중함.



System Sequence Diagram in OOA　　Sequence Diagram in OOD

2.4.2. Operation Contract

Operation Contract(작업 규약)는 system operation의 동작 및 제약 조건을 작성한 것임. 시스템 동작에 대한 더 자세한 설명을 위해 작성하고, 이후 design에서 object 메소드 설계 시에 활용할 수 있음.

SSD를 작성함으로써 domain model의 내용을 보강하거나, operation 또는 use-case의 동작을 명확히 할 수 있음.

operation contract는 특정 system operation에 대해 다음과 같은 사항들을 작성함.

- *Name: operation의 이름과 parameter.*
- *Responsibility: 해당 operation이 가지는 책임.*
- *(Optional) Type: operation의 type(sw class. interface 등).*
- *Cross Reference: 해당 operation에 대한 reference 번호나, 포함되는 use-case 등.*
- *(Optional) Note: 해당 operation에 대한 메모.*
- *(Optional) Exception: exception case들.*
- *(Optional) Output: 해당 operation에 의해 시스템에서 외부로 전달되는 출력.*
- ***Pre-condition**: 해당 operation이 실행되기 위해 필요한 조건 또는 제약사항.*
- ***Post-condition**: 해당 operation이 완료된 후 시스템 또는 객체 상태의 변화(ex. obejct/association/attribute 생성/삭제 등). 주로 operation 종료 후 해당 과정에서 일어난 상태의 변화를 과거형으로 작성함(ex. A SalesLineItem was created).*

Use Case	Name of Actor-Activated Event	System Operations
1. Login	1:enterInfo()	1:enterInfo()
	2:reqLogin()	2:reqLogin()
2. Logout	3:reqLogout()	3:reqLogout()
3. Make Account	4:reqMakeAcc()	4:reqMakeAcc()
	5:enterAccInfo()	5:enterAccInfo()
	6:reqAccount()	6:reqAccount()
4. Identify Balance	7:reqBalance()	7:reqBalance()
5. Recharge Balance	8:enterFee()	8:enterFee()
	9:reqRecharge()	9:reqRecharge()
6. Request Print	10:enterSheet()	10:enterSheet()
	11:reqPrint()	11:reqPrint()
8 Identify Paper	12:req Identify Paper()	12:reqPaperIdentify()
9. Recharge Paper	13:enterPaperNum()	13:enterPaperNum()
	14:reqCharge()	14:reqCharge()
10. Identify User	15:reqUserInfo()	15:reqUserInfo()
11. Identify Money	16:reqMoneyInfo()	16:reqMoneyInfo()

Ref. #	Function	Category
R 1.1	System Access	Event
R 1.2	Make Account	Event
R 1.3	Identify Balance	Event
R 1.4	Recharge Balance	Event
R 2.1	Request Print	Event
R 2.2	Check Balance	Hidden
R 3.1	Identify Paper	Event
R 3.2	Recharge Paper	Event
R 3.3	Identify User	Event
R 3.4	Identify Money	Event

System functions (requirements)

Name	reqPrint
Resposibilities	인쇄를 요청 한다.
Type	System
Cross References	R 2.1, R 2.2
Notes	
Exceptions	잔액이 매수 * 요금보다 작으면 인쇄가 진행되지 않는다.
Output	인쇄 결과, User.balance 변경
Pre-Conditions	입력된 인쇄 매수가 있어야 한다. 사용자 로그인 상태 이어야 한다.
Post-Conditions	사용한 금액만큼 사용자의 잔액이 감소 된다. 입력된 인쇄 매수를 초기화 한다.

3. OOD - Structure Diagram

Static Modeling에 의한 diagram인 Structure Diagram에 대해 알아보자.

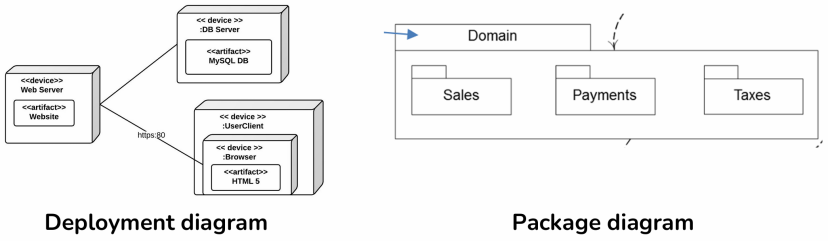
3.1. Package Diagram

3.1.1. Logical Architecture

1. Logical Architecture

Logical Architecture는 software class들을 package, subsystem, layer 등과 같이 더 큰 논리적 단위로 구성한 것임. 이때 logical architecture는 물리적인 위치나 배치는 고려하지 않고, 논리적인 구조만 나타낸 것으로, 물리적인 부분을 고려한 것은 Deployment Architecture라고 함.

logical architecture에 대한 modeling 시에는 물리적인 model을 섞어 그리지 않는 것이 좋음.



Deployment diagram　　Package diagram

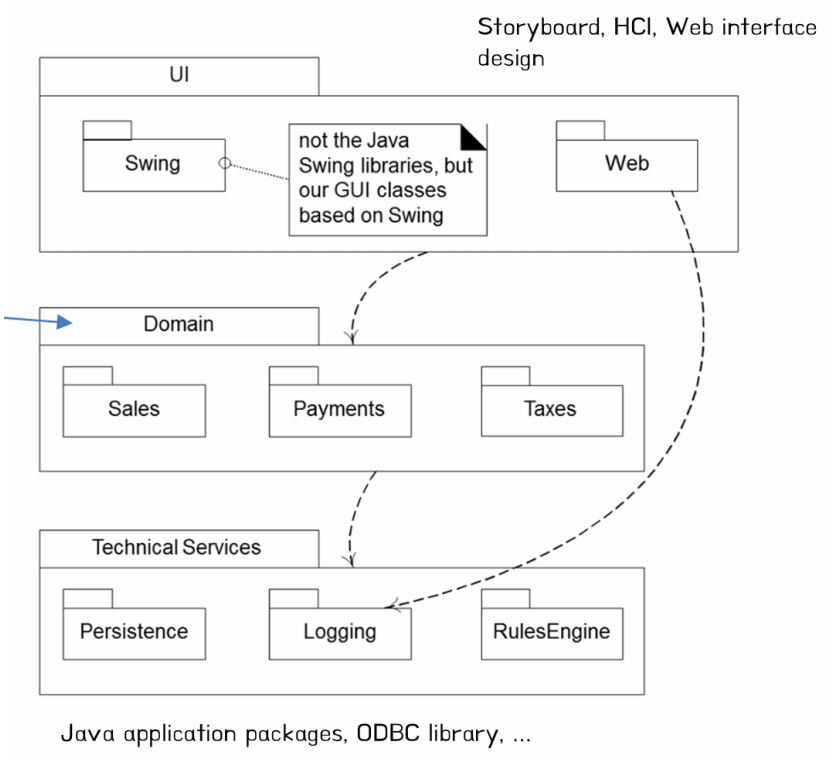
2. Layer

Layer(계층)는 응집된 책임을 가지는 class, package, subsystem 등을 묶은 것임.

큰 규모의 시스템을 각각의 책임을 가지는 여러 개의 layer로 나누고, 상위 layer가 하위 layer의 서비스를 호출할 수 있도록 구성할 수 있음(ex, OSI 7계층). 이에 따라 모듈성, 재사용성을 확보할 수 있고, 더 쉽고 직관적으로 시스템을 개발할 수 있음.

OO system은 일반적으로 다음과 같은 계층(object) 구조를 가짐. 특히 UI layer와 technical service layer는 이미 잘 정의되어 있는 경우가 많아 필요에 따라 활용하면 되지만, domain layer는 응용에 따라 다른 동작이 필요하기 때문에 적절한 analysis 및 design이 필요함.

- *UI(Presentation) Layer: actor와의 입출력(user interface)을 담당하는 layer.*
- *Domain Layer: requirment에 대한 domain과 응용 로직을 담당하는 layer.*
- *Technical Service Layer: DB 연결, 로깅과 같이 기술적 서비스를 제공하는 layer.*

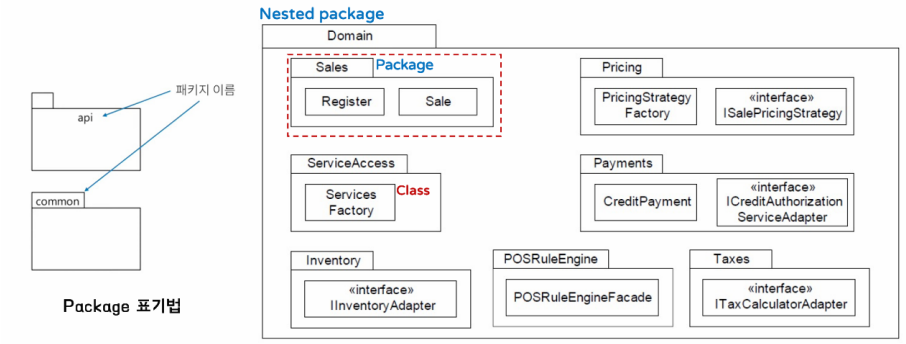


3.1.2. Package Diagram

1. Package Diagram

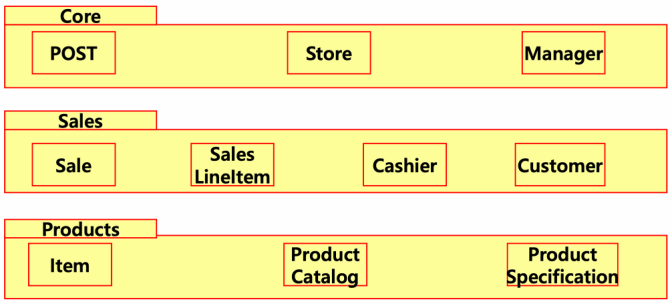
Package Diagram은 시스템을 package의 집합으로 나타내는 modeling 방식임. 즉, package diagram으로 logical architecture를 modeling할 수 있음.

package는 다음과 같이 package 이름을 작성하며, nested package가 존재할 수 있음.



package diagram을 그리는 과정은 다음과 같음.

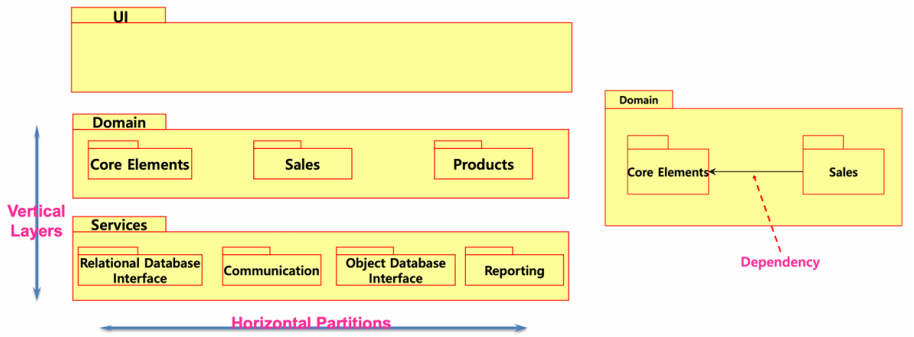
1. *class를 package로 묶음. 이때 class는 동일한 주제나 개념, use-case에 속하는 것끼리 묶을 수 있고, class 이름은 OOA로 도출된 domain을 활용할 수 있음.*



2. *vertical layer(UI, domain, service layer)를 package로 그리고, 한 layer 내에서 병렬적인 package들은 horizontal partition으로 구분해 묶은 package를 그러넣음.*

   package A가 package B에 대한 dependency를 가진다면(A가 B 사용하는 경우), A → B로 점선 화살표를 그림.

   UI layer에서 domain layer로 요청을 전달하는데, 이때 UI layer에서 보내는 요청은 SSD에서 actor가 시스템에게 보내는 것과 동일하게 할 수 있음.



2. Model-view Separation

Model-view Separation(모델-뷰 분리 원칙)은 모델(domain layer의 object)이 뷰(UI layer의 object)에 대한 직접적인 지식을 가져서는 안 된다는 원칙임. 이는 유지보수, 관심 분리(model은 데이터 처리, view는 사용자 편의성) 등의 측면에서 효과적임.

특히, 하나의 model을 가지고 여러 view를 만들어야 하는 상황(ex. pc와 모바일)이 존재하는 경우가 많은데, 이런 경우를 위해서도 model-view separation이 유효함.

3.2. Class Diagram

3.2.1. Class Diagram

Class Diagram은 시스템의 class와 class 간의 관계를 modeling한 것임. domain model은 conceptual한 관점에서 domain object와 각각의 관계를 다뤘다면, class diagram은 software 관점에서 class와 각각의 관계를 다룸.

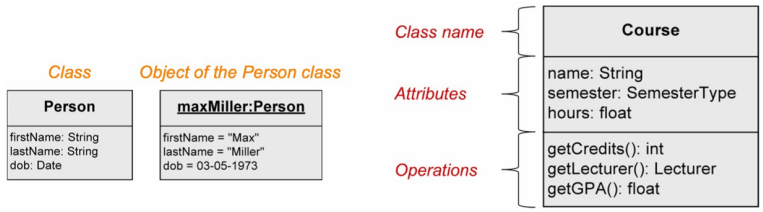
dynamic modeling인 sequence diagram을 보면 class, method 등이 등장함. 즉, class diagram과 sequence diagram을 동시에 반복적으로 그리며 software의 동작을 잘 design하고 이해할 수 있음.

3.2.2. Class Diagram Notation

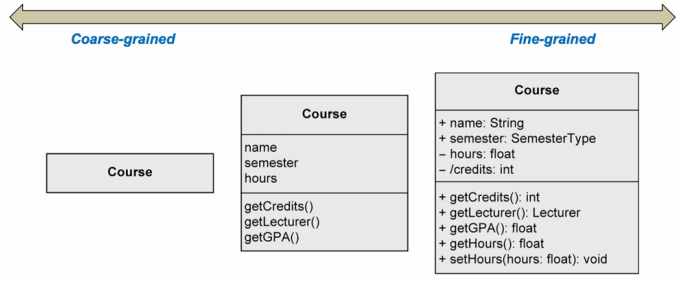
1. Class

Class는 attribute와 operation의 집합인 data type임. class의 intance를 Object라고 함. Attribute는 class의 구조적 특징을 설명하는 데이터로 object마다 다른 값을 가질 수 있고, Operation은 class의 행위를 나타내는 것으로 모든 object에 대해 동일함.

UML의 class diagram에서 class는 다음과 같이 class name, attribute, operation 순서로 작성함.



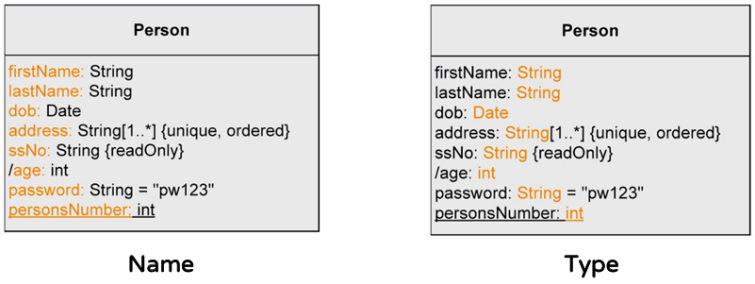
물론 다음과 같이 대략적으로 나타낼 수도 있고, 자세히 나타낼 수도 있는 등 다양한 표기법이 존재함. 필요에 따라 적절한 표기를 선택하는 것이 좋음.



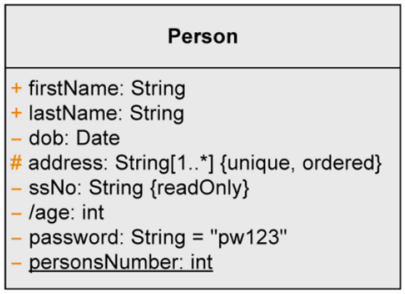
2. Attribute Syntax

UML의 class diagram에서 attribute에 대한 syntax는 다음과 같음.

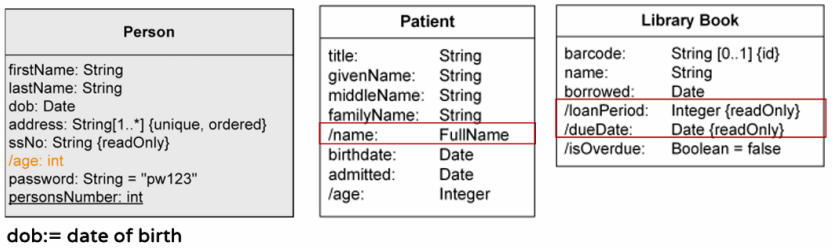
- *Name and Type: attribute는 다음과 같이 name:type 형태로 작성함. 이때 data type은 primitive 일 수도 있고, user-defined일 수도 있음.*



- *Visibility: 접근 지정자에 의한 visibility는 다음과 같이 attribute 앞에 +로 public, -로 private, # 으로 protected를 명시함.*

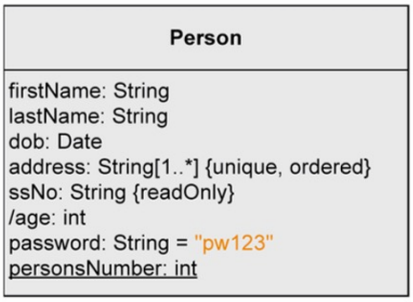


- *Derived Attribute: Derived Attribute는 해당 class의 다른 attribute에 의해 값이 결정되는 attribute로, attribute name 앞에 /를 붙여 표기함. 또한 관례적으로 {readOnly}를 붙이기도 함.*



- *Multiplicity: 하나의 attribute가 가지는 값의 개수는 data type 바로 뒤에 [min...max] 꼴로 명시하고, 제한이 없음은 *로 나타냄. default multiplicity 값은 1임. 이에 따라 list 등의 자료구조를 표현할 수 있음.*

- *Default Value: attribute가 가지는 default 값은 data type 바로 뒤에 =값 꼴로 명시함.*

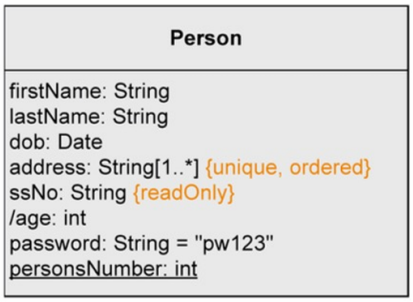


- *Property: attribute에 대한 Property는 추가적인 특성이나 제약 조건을 말하고, attribute의 가장 끝에 명시함. 여러 개의 property를 가지는 경우 여러 개를 함께 작성함. property로는 다음과 같은 것들이 있음.*

  {readOnly}: attribute 값 수정 불가(final).
  {unique}: 해당 attribute의 여러 값 간 중복을 허용하지 않음.
  {non-unique}: 해당 attribute의 여러 값 간 중복을 허용함.
  {ordered}: 해당 attribute의 여러 값 간 순서가 존재함.
  {unordered}: 해당 attribute의 여러 값 간 순서가 존재하지 않음.

  또한 이를 활용하면 각 자료구조는 다음과 같이 나타낼 수 있음.

  {unordered, unique}: set.
  {unordered, non-unique}: multi-set.
  {ordered, unique}: ordered set.
  {ordered, non-unique}: list.

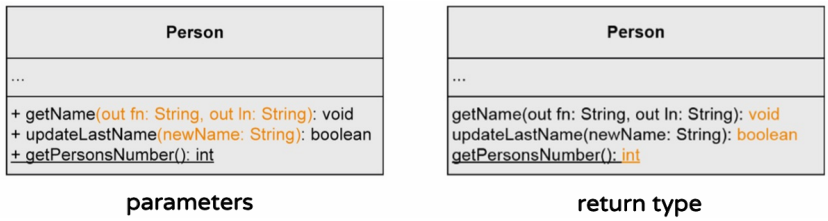


3. Operation Syntax

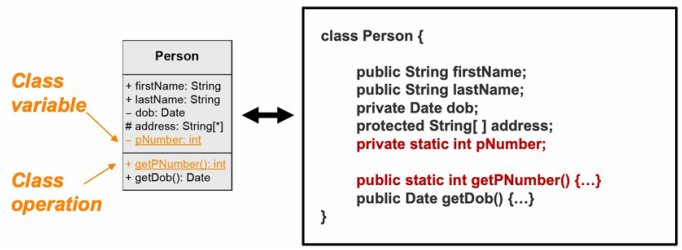
UML의 class diagram에서 operation에 대한 syntax는 다음과 같음.

- *Name and Type: operation은 다음과 같이 visibility name(parameter-list) : return type {property} 꼴로 작성함. 이때 visibility가 생략되었으면 public으로 취급함.*

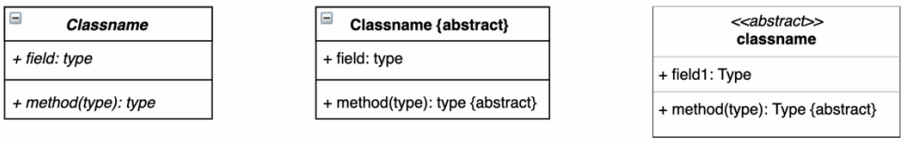
  parameter는 name:type 꼴로 작성하고, in은 input parameter(생략하기도 함), out은 output parameter(parameter에 결과 저장), inout은 input 및 output 모두로 사용되는 parameter임.



- *Class Variable/Operation: class variable/operation, 즉 static variable/operation은 밑줄로 표기함.*



- *Getter and Setter: getter와 setter는 직접 대응되는 식별자로 작성해야 하는데, 모든 attribute에 대해 작성하는 건 너무 많기 때문에 생략하는 경우도 있음. 일반적으로 UML tool에는 getter/setter를 켜고 끄는 토글 스위치가 있다고 함.*
- *Abstract Operation/Class: abstract operation/class는 italic으로 작성하거나, abstract property를 작성하거나, class의 경우 «abstract»를 붙여 표기함.*



상기하자면 protected는 해당 class 내부와, 해당 class를 상속받는 subclass에서만 접근이 가능함.

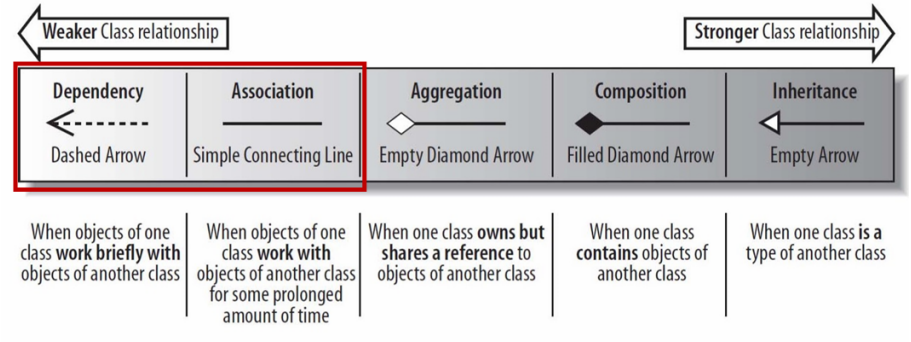
참고로 default 값은 값을 따로 지정하지 않았을 때의 값을 말함.

참고로 attribute name이 pNumber이면 getter와 setter의 식별자는 getPNumber(), setPNumber()임.

3.3. Class Diagram Relationship

3.3.1. Class Diagram Relationship

class diagram에서 class/object 간의 다양한 관계를 표현할 때는 다음과 같이 5가지 분류를 사용함.



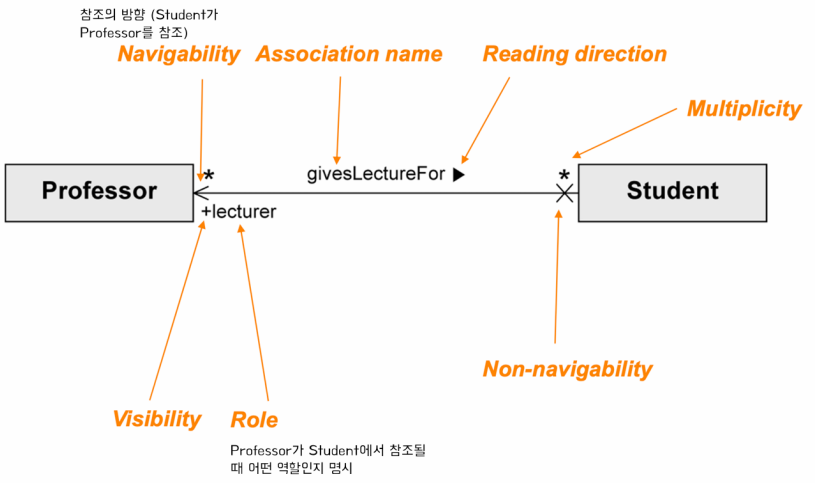
3.3.2. Dependency

Dependency(의존 관계)는 한 class의 object에서 다른 class의 object를 일시적으로(temporarily) 사용하는 관계로, 점선 화살표로 표기함.

예를 들어, method/parameter 값의 data type으로 활용하거나, 지역 변수로 사용하는 경우 등이 있음.

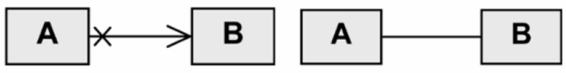
3.3.3. Association

Association(연관 관계)은 한 class의 object가 다른 class의 object와 장기간 연관되어 동작하는 관계로, 실선으로 표기함. 또한 두 class에서의 association을 Binary Association이라고 함. 더 구체적으로는 다음과 같이 association을 나타냄.



1. Navigatability

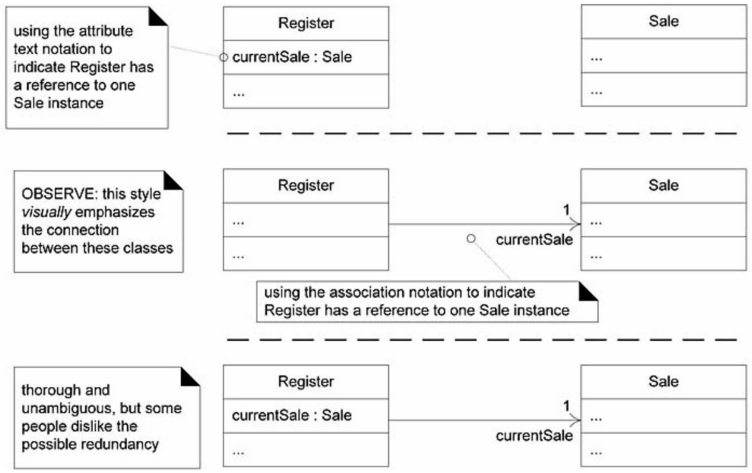
Navigatability(가항성)은 한쪽이 다른 쪽에 대한 정보에 접근할 수 있는지를 말함. 단방향으로만 navigability가 존재해 A만 B를 알 수 있으면 A 쪽은 X, B 쪽은 화살표로 나타내고, 양방향으로 navigability가 존재하면 단순히 실선으로 그림.



2. Attribute 설정

navigability가 존재하면 참조하는 class의 object에서 참조 대상 class의 object를 attribute로 가짐. 이때 role 이름이 attribute name이 됨.

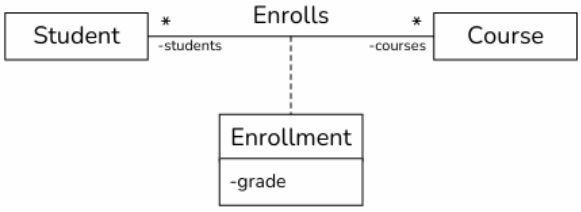
이를 attribute만으로 나타낼 수도 있고, assocation line에서 화살표, multiplicty, role로 나타낼 수도 있고, 둘 다를 사용해 나타낼 수도 있음. data에 해당하는 것은 attribute로, association 관계임을 나타내기 위한 것은 association line으로 나타내는 것이 좋음.



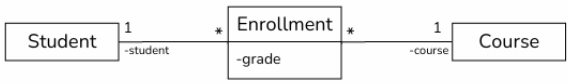
3. Association Class

Association Class는 association 자체에 attribute나 operation을 추가할 필요가 있을 때, association을 class로 표현한 것임.

예를 들어, Student class와 Course Class가 있을 때 어떤 학생이 특정 과목에 대해 가지는 grade는 이 두 class 중 하나가 아니라, 이 관계 자체에 정의되어야 함. 어느 한쪽에 정의되는 경우 의미 상 적절하지 않음.



이런 association class는 다음과 같이 별도의 class로 표현할 수도 있음. 이때 새로 생성된 class는 다른 두 class에 대한 object를 attribute로 가짐.



3.3.4. Aggregation&Composition

1. Whole-part Relationship

Whole-part Relationship은 전체 개념에 해당하는 class(whole)와, 부분 개념에 해당하는 class(part) 간의 관계로, 특수한 형태의 association임. 특히 1) 어떤 class가 다른 class의 부분일 때(is part of)와, 2) 어떤 class가 다른 class들을 소유할 때(own) 같은 상황에 적용이 가능함.

whole-part relationship에 대해서는 다음과 같은 당연한 속성이 존재함.

- *Transitive Property: A is part of B이고, B is part of C이면, A is part of C임.*
- *Asymetric Property: A is part of B와 B is part of A는 동시에 참일 수 없음.*

UML에서는 whole-part relationship을 전체와 부분에 대한 생명주기에 따라 aggregation과 composition

으로 구분함.

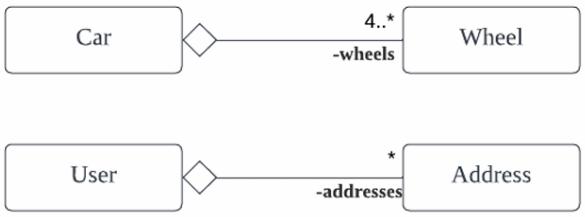
Aggregation	Composition
느슨한 결합 - 전체 객체가 소멸해도 부분 객체는 소멸하지 않음	**강한 결합** - 전체 객체가 소멸하면 부분 객체도 소멸
Weaker ownership - 부분 객체는 전체 객체에 의해 온전히 소유되는 것은 아니며, 공유되거나 재사용될 수 있음	**Strong ownership** - 전체 객체가 부분 객체의 직접적인 책임(생성/관리/소멸)을 가질 때
Shared resources - 한 번 만들어둔 객체가 부분으로서 다른 곳에서 여러번 사용되어야 할 때 유용	**Encapsulation** - 부분이 전체 객체의 맥락 내에서만 사용되어야 할 때 유용

2. Aggregation

Aggregation(집합 관계)은 부분이 전체에 약하게(use) 속하는 관계임. hollow diamond(빈 다이아몬드) 화살표로 표기하는데, 전체에 해당하는 class에 hollow diamond를 그림. 이때 부분은 전체와 상관없이 독립적으로 존재할 수 있음. 즉, 전체 object가 사라져도 부분 object는 존재할 수 있는 관계로, 전체와 부분이 생명주기가 다름.

aggregation은 association의 일종이지만, whole-part relationship이므로 양쪽에 hollow diamond를 그리거나 서로 다른 방향의 두 hollow diamond 화살표를 그려서는 안 됨.

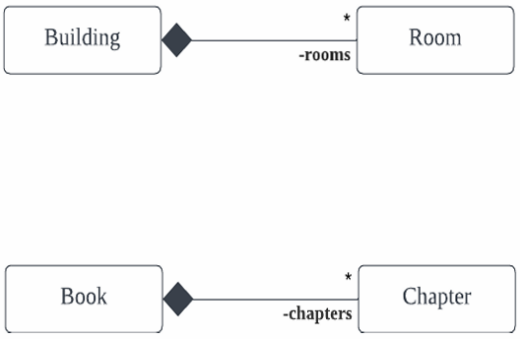
예를 들어, car와 wheel의 관계로 이해할 수 있음.



3. Composition

Composition(합성 관계)은 부분이 전체에 강하게(own) 속하는 관계임. filled diamond(차 있는 다이아몬드) 화살표로 표기하는데, 전체에 해당하는 class에 filled diamond를 그림. 이때 부분은 전체에 종속되어 독립적으로 존재할 수 없음. 즉, 전체 object가 사라지면 부분 object로 사라지고, 전체와 부분의 생명주기가 같음.

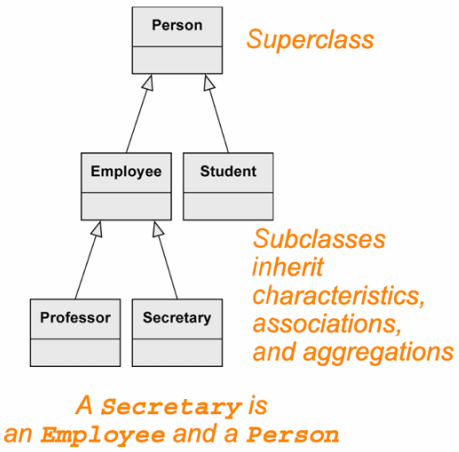
예를 들어, building과 room의 관계로 이해할 수 있음.



3.3.5. Inheritance

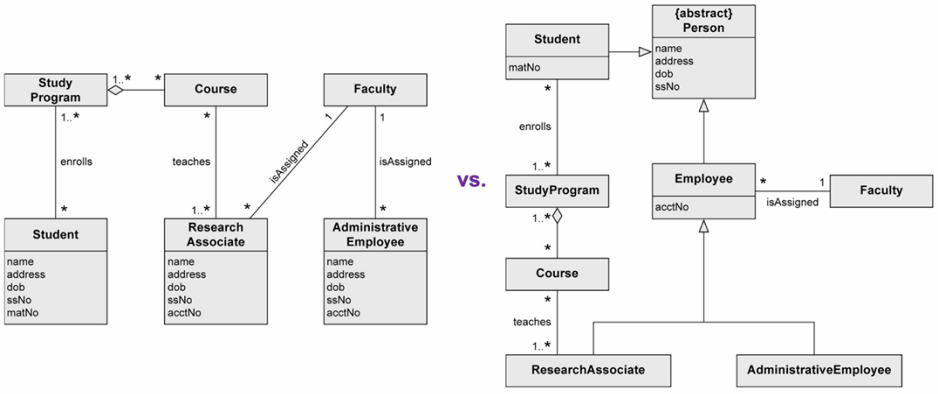
Inheritance(상속 관계, 일반화 관계)는 subclass들의 공통된 특성을 super class로 일반화한 관계로, 화살표로 표기함. subclass는 private인 것들을 제외하고 super class의 모든 것을 상속받으며, subclass

별로 추가적인 속성을 가질 수 있음. 이에 따라 subclass의 instance는 super class에 대한 간접적인 instance임.



abstract class는 그 자체로 instance화되지는 못하지만, subclass의 공통 특성을 강조함. 이때 subclass는 상속받은 abstract method를 모두 구현해야 instance화될 수 있음.

다음과 같이 inheritance를 활용하면 class diagram을 더 직관적이고 이해하기 쉽게 그릴 수 있음.



4. OOD - Behaviour Diagram

Dynamic Modeling에 의한 diagram인 Behaviour Diagram에 대해 알아보자.

4.1. Interactive Diagram

4.1.1. Interactive Diagram

1. Interactive Diagram
Interactive Diagram은 시스템을 구성하는 object들이 어떻게 특정 시점에 message를 주고받으며 상호작용하는지를 나타내는 diagram임. interactive diagram은 특히 class의 method를 설계하고, exception 상황을 파악하는 데에 도움이 됨.

여기에서는 interactive diagram 중 sequence diagram과 communication diagram에 대해 살펴봄.

2. Communication Diagram vs. Sequence Diagram
communication diagram과 sequence diagram은 동일한 정보를 서로 다르게 표현함.

sequence diagram에서는 시간의 흐름에 초점을 맞추고 사건의 흐름을 그리므로, 메시지의 순서를 순

차적으로 확인하고 use-case 기반으로 표현하기에 좋음. communication diagram에서는 object 간의 관계에 초점을 두고 흐름을 network 형태로 그리므로, object 간 관계를 파악하기에 좋음.

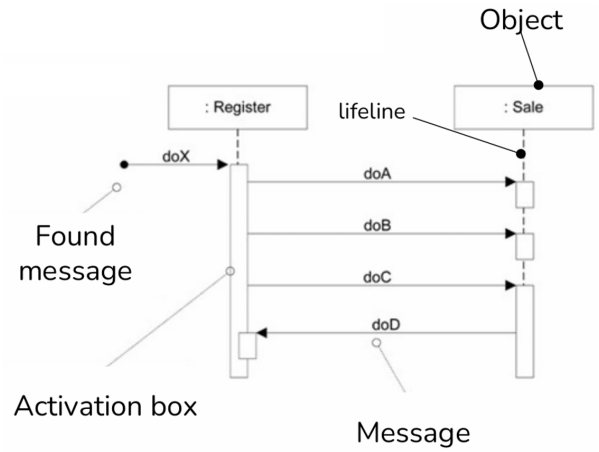
	장점	단점
Sequence	- 메시지 순서를 시간순으로 명확하게 표현 - 메시지 호출 순서/반환 값을 보여주므로 객체 간의 동작을 이해하기 쉬움	- 공간 제약: 새로운 객체가 추가되면 오른쪽 으로 확장하는 것이 강제됨 -
Communication	- 객체의 연관 관계를 쉽게 파악할 수 있음 - 새로운 객체가 추가되었을 때 공간 압축적으로 효율적 표현 가능	- 메시지의 순서나 흐름을 한눈에 파악하는게 어려워짐

4.1.2. Sequence Diagram

1.Sequence Diagram

Sequence Diagram은 특정 use-case나 operation을 대상으로 하여, 시간의 흐름에 따라 object 간의 상호작용을 순차적으로 나타낸 diagram임.

UML에서 sequence diagram은 다음과 같은 notation을 가짐.



- *Message: 보내진 방향에 대해 화살표로 표기함. 순서에 따라 번호를 붙여 표기하기도 함.*
- *Lifeline: object가 시스템에 존재하는 시간으로, 점선으로 표기함.*
- *Activation Box: object가 활성화되어 실행되는 시간으로, lifeline을 따라 사각형으로 표기함.*
- *Object: box로 표기함.*

  object에 대한 box에 lifeline 정보가 추가되어 있는 것을 Lifeline Box라고 함. 다음과 같이 필요에 따라 추가 정보를 작성하기도 함.

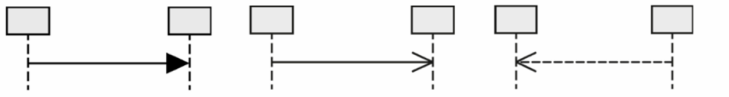
2. Message

message syntax는 다음과 같음. 이때 message를 제외하고는 생략하기도 함.

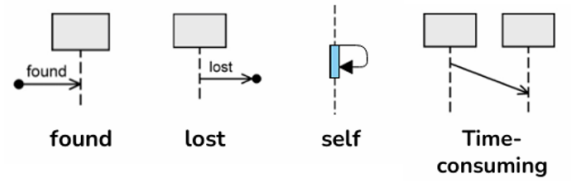
```
return = message (parameter: parameterType) : returnType
```

message는 다음과 같이 여러 종류가 있음.

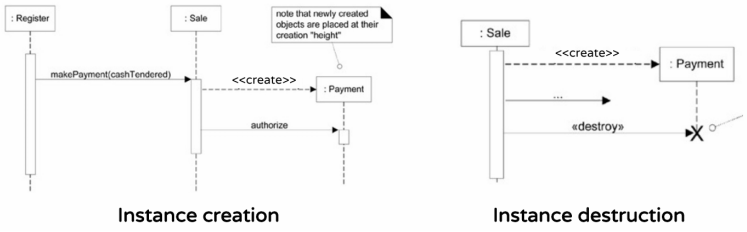
- *Synchronous Message: sender가 response message를 받을 때까지 기다려야 하는 message로, 차 있는 화살표로 표기함.*
- *Asynchronous Message: sender가 response message를 받지 않아도 다른 작업을 할 수 있는 message로, 단순 화살표로 표기함.*
- *Response Message: 응답에 해당하는 message로, 점선 화살표로 표기함.*



- *Found Message: 보낸 사람이 누군지 모르는 message로, 점에서 시작하는 화살표로 표기함. 주로 시작점임.*
- *Lost Message: 받는 사람이 누군지 모르는 message로, 점에서 끝나는 화살표로 표기함.*
- *Self Message: 스스로에게 보내내는 message로, 화살표의 끝을 자기 자신에게 보냄.*
- *Time-consuming Message: 처리에 시간이 걸리는 message로, 화살표의 끝을 단순히 끝나는 시점에 보냄. 이때 message를 보내서 받는 데에 걸리는 시간을 Message With Duration이라고 함.*



- *«create»: 해당 object를 생성하는 message는 관용적으로 점선 화살표에 «create»를 표기함.*
- *«destory»: 해당 object를 삭제하는 message는 관용적으로 점선 화살표에 «destroy»와 X를 표기함.*

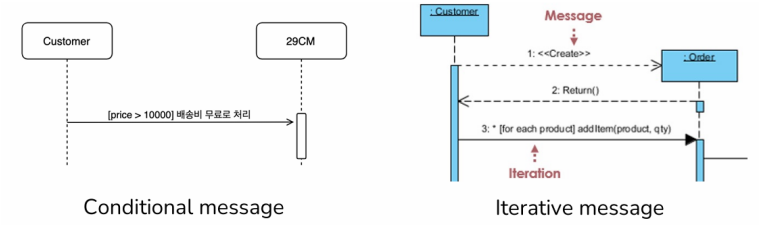


Instance creation　　Instance destruction

3. Gaurd Notation

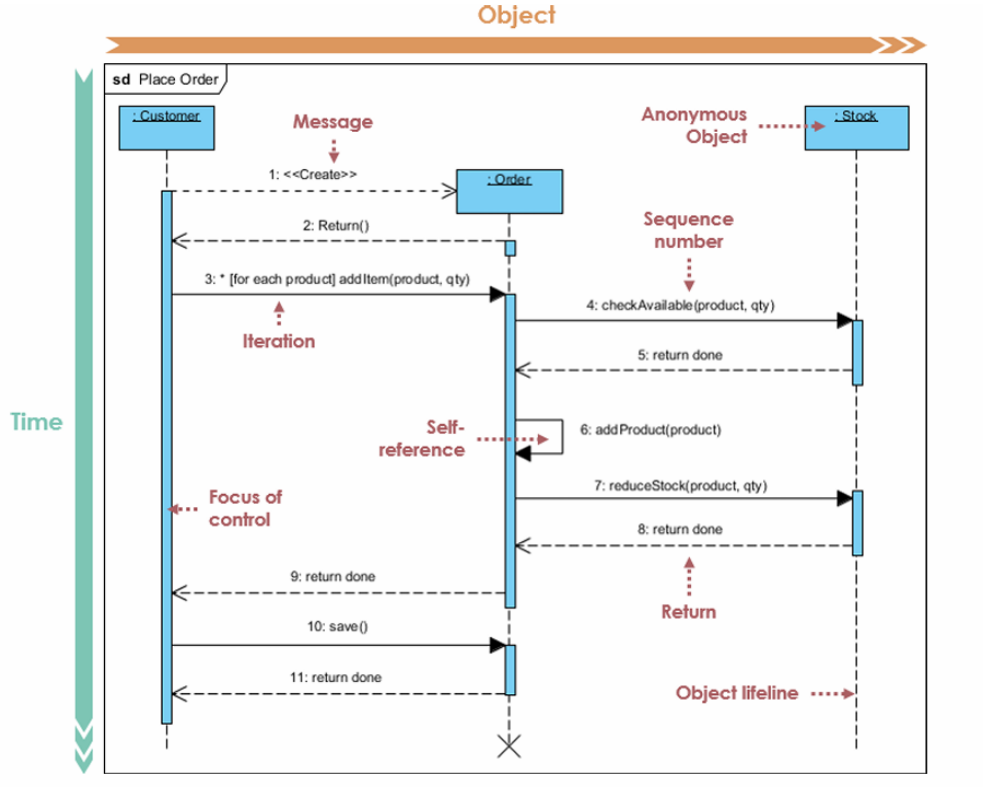
Gaurd Notation으로 message에 대한 조건과 반복을 지정할 수 있음. 조건의 경우 해당 condition을 만족하면 message를 전송하는 것이고, 반복의 경우 해당 condition에 따라 message를 여러 번 전송하는 것임.

*다음과 같이 조건문은 [condition] message와 같이 표기하고, 반복문은 *[for loop condition] message와 같이 표기함.*



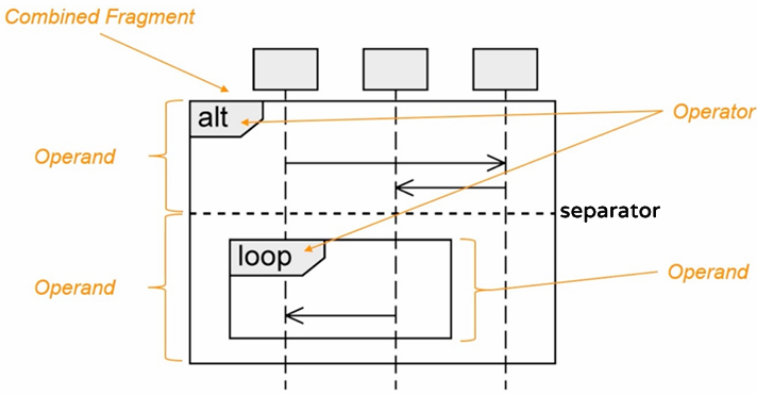
Conditional message　　Iterative message

다음은 sequence diagram에 대한 예시로, 여러 product에 대해 재고를 확인하고 주문하는 use-case임.



4.1.3. Combined Fragments in Sequence Diagram

sequence diagram의 Combinded Fragment는 각 요소를 그룹화하고 실행 흐름을 효과적으로 modeling하기 위해 사용하는 표기임. 다음과 같이 frame으로 묶고 operator, condition 등을 그림.



- *Operator: 해당 fragment의 종류를 나타냄.*
- *Operand: 작성한 condition이 만족할 때 수행되는 공간으로, condition이 없다면 항상 수행됨. frame 안에서 점선으로 구분함.*
- *Guard Condition: 수행 조건을 명시함 것임.*

1. Branches and Loops

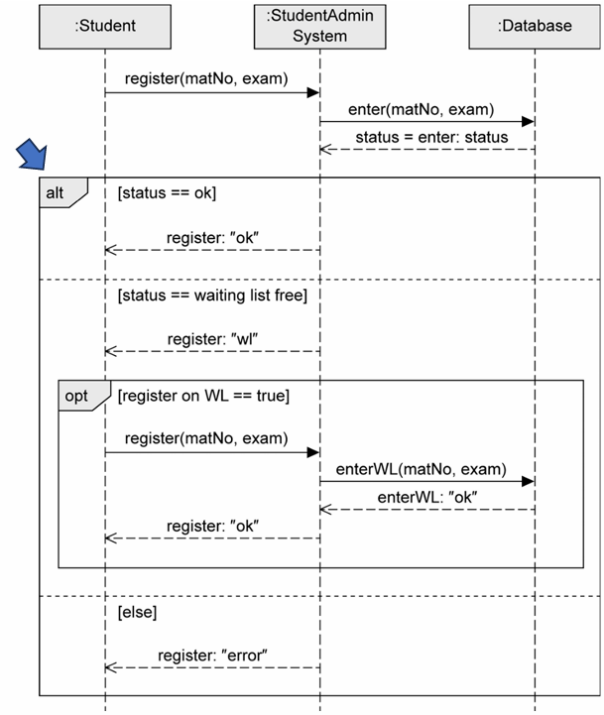
분기와 반복에 대한 fragment로는 다음과 같은 것들이 있음.

- *alt: 조건에 따라 다른 작업을 수행할 때 사용하는 fragment. operand별로 guard condition을 작성하고 참인 것만 수행함. 조건이 겹치지 않아야 하고, else인 경우에 대해서는 [else]로 작성함.*

  switch문과 유사함.

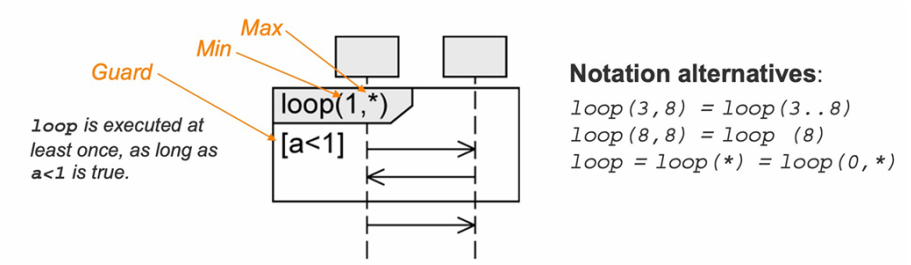
- *opt: 조건이 참일 때만 수행하는 fragment. 하나의 operand만 존재함.*

  else가 없는 if문과 유사함.



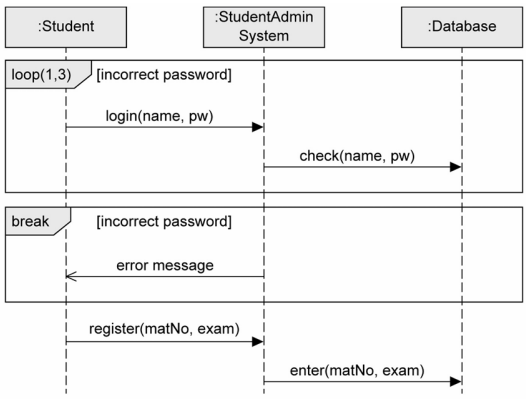
- *loop: 반복을 수행하며 조건이 거짓인 경우 반복을 중단하는 fragment. operator에 loop(min, max) 또는 loop(min..max) 꼴로 반복 횟수를 지정함(최소 min만큼, 최대 max만큼 반복 수행.). *도 사용할 수 있으며, 반복 횟수를 작성하지 않은 경우 default는 *임. guard condition으로 반복 조건을 지정할 수 있음.*

for문과 유사함.



- *break: 조건이 참이면 해당 operand를 수행한 뒤, 자신이 속한 fragment는 더 이상 수행하지 않고 다음 higher level fragment에서 이어서 수행하는 fragment.*

  break문과 유사함. loop 등의 내부에서 사용해 반복을 중단할 수도 있지만, 다음과 같이 예외 처리에도 사용될 수 있음.

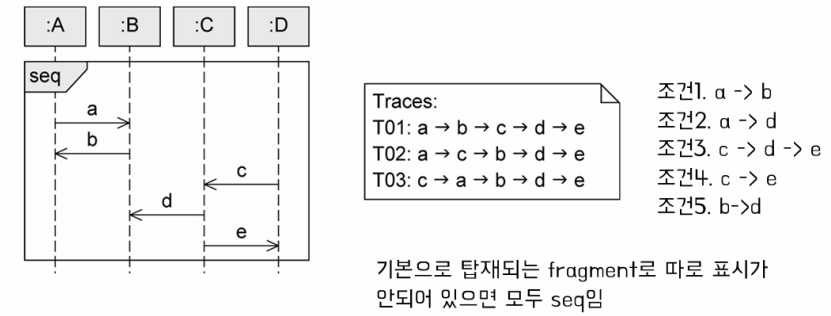


2. Concurrency and Order

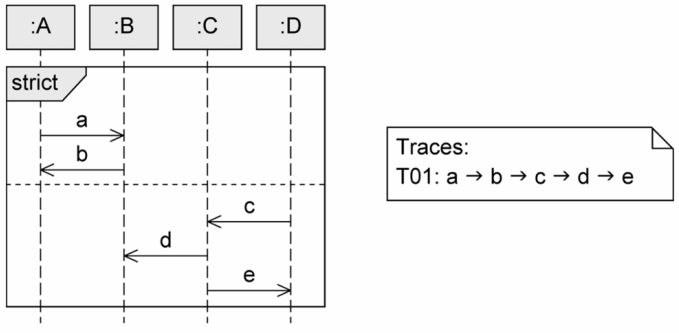
순서 등에 대한 fragment로는 다음과 같은 것들이 있음.

- *seq: operand 간의 weak sequence를 나타낼 때 사용하는 fragment. seq fragment를 사용하지 않아도 sequence diagram은 기본적으로 weak sequence로 동작하므로, 이는 주로 다른 fragment와 결합하여 사용함.*

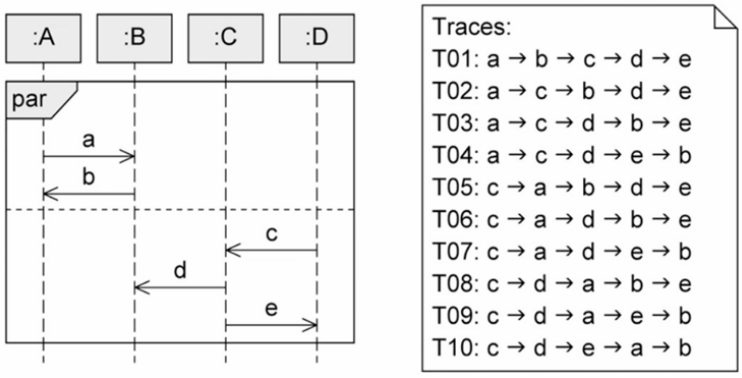
  Weak Sequence는 동일한 lifeline에서는 message의 순서가 위에서 아래로 반드시 지켜져야 하지만, 서로 다른 lifeline에 대해서는 message의 순서가 섞여도 되는 것을 말함.



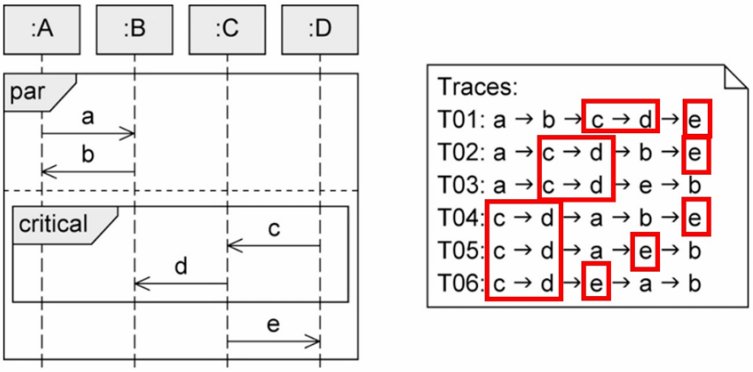
- *strict: 서로 다른 operand 간 순서를 엄격히 지키도록 할 때 사용하는 fragment. 이는 operand 간 순서에 대한 것으로, 하나의 operand 내에서는 여전히 seq임.*



- *par: 서로 다른 oprand 간의 동시 수행(concurrency)을 표현할 때 사용하는 fragment. 하나의 operand 내에서의 순서는 지켜지지만, 서로 다른 operand 간에는 순서를 고려하지 않음.*



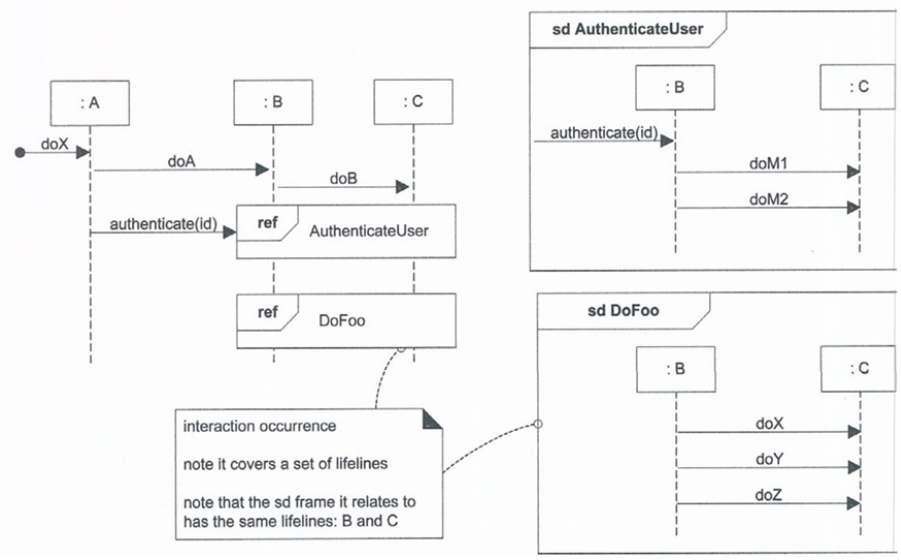
- *critical: critical region(atomic area)을 표현할 때 사용하는 fragment. 해당 frame(critical region) 내에 message들이 수행될 때 frame 외부의 다른 message가 끼어들어서 수행될 수 없음. 즉, 해당 frame이 완전히 종료되어야 다른 message가 수행될 수 있음.*



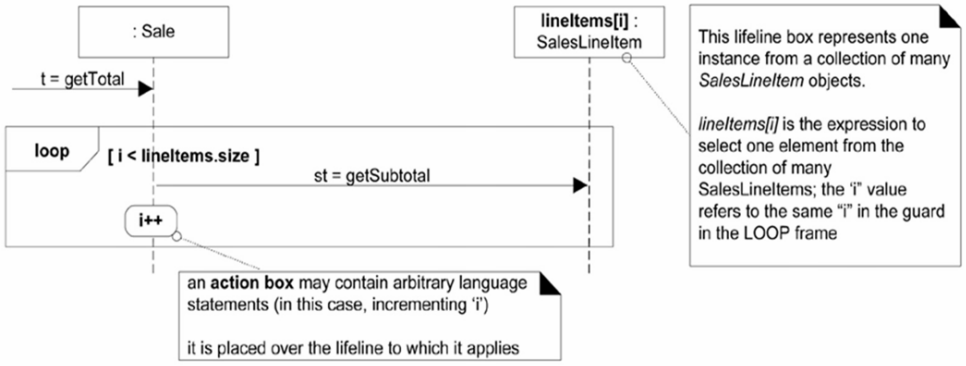
3. Others

기타 fragment로는 다음과 같은 것들이 있음.

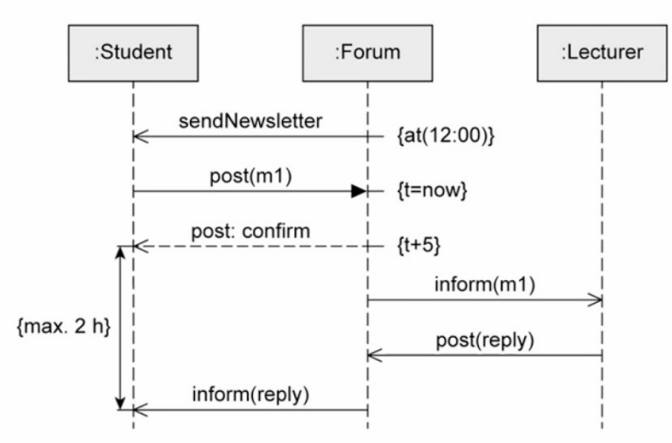
- *ref: 다른 sequence diagram을 가리키는 fragment.*



- *Iteration over a Collection: collection의 각 개별 요소에게 message를 보내는 경우, 다음과 같이 collection의 인덱스를 지정한 lifeline에 message를 보내고, i++을 하는 것으로 표현할 수 있음.*



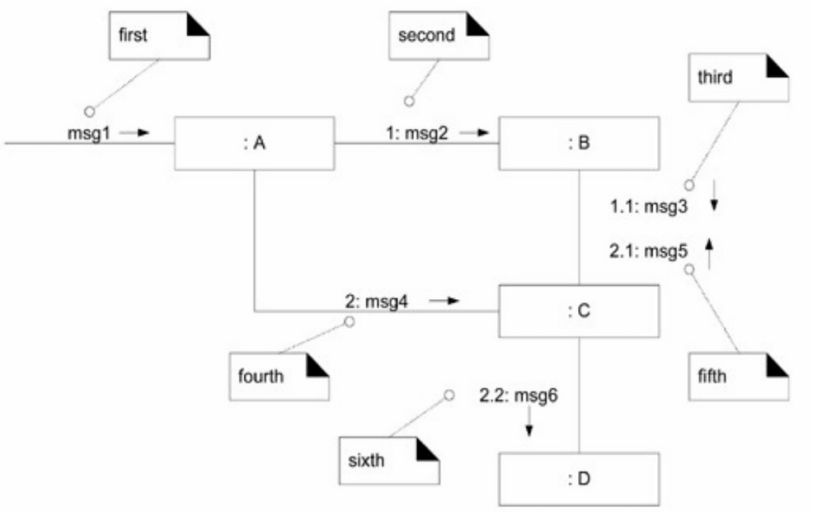
- *Time Constraint: 다음과 같이 {}를 사용해 message 송수신 시간, duration 등의 제약을 표기할 수 있음.*



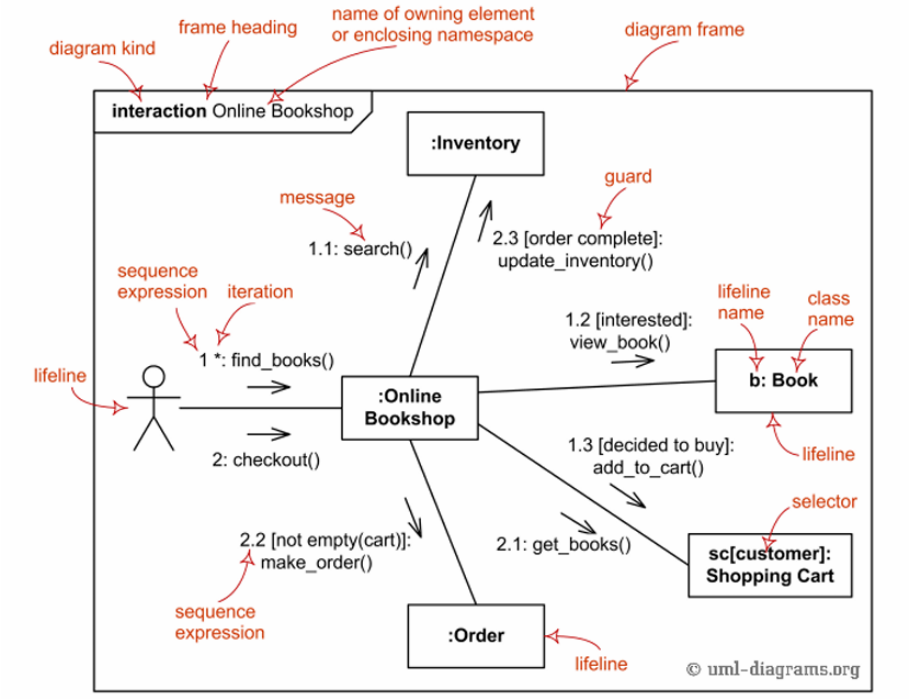
4.1.4. Communication Diagram

Communication Diagram은 상호작용하는 object 간의 관계를 표현하고, 상호작용을 메시지로 표현하는 diagram임.

communication diagram의 notation은 sequence diagram과 거의 동일함. 연관된 object는 실선으로 연결하고, message는 두 object의 실선 위에 화살표와 함께 표기함. 또한 message를 보내는 순서를 message 앞에 표기함.



다음은 communication diagram에 대한 예시임.



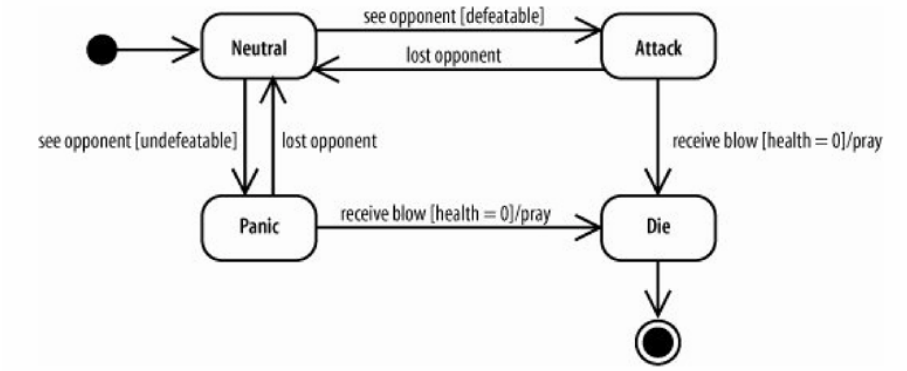
4.2. State Machine Diagram

4.2.1. State Machine Diagram

1. State Machine Diagram

State Machine Diagram(상태 전이 다이어그램)은 시스템에 대한 state가 transition에 의해 어떻게 변화하는지를 나타낸 것임. 즉, 시스템/object는 주로 어떤 상태에 있고 어떤 이벤트에 의해 그 상태가 변화하게 되는데, 이를 간결하게 modeling한 것임.

state machine diagram은 다음과 같이 state와 transition의 모음으로 그릴 수 있음.



- *State(상태): 시스템/object의 특정 상태를 rounded rectangle과 그 내부의 text로 나타냄.*

  초기 상태인 Initial State는 filled black circle로, 최종 상태인 Final State는 circled dot으로 그림.

  state는 passive quality와 active quality로 구분됨. 어떤 state가 Passive Quality이면 단순히 다음 이벤트를 기다리는 state라는 것이고, Active Quality이면 그 state에 머무는 동안 지속적인 활동을 수행하는 state라는 것임. (ex. 전구 vs. 커피머신)

  Internal Behavior는 어떤 state에 대해 내부적으로 수행되는 행위로, rounded rectangle 안에 다음과 같이 작성함.

  1) do/behavior: 해당 state에 머무는 중에 수행되는 행위(ex. do/brew coffee). 이는 active quality인 state에 존재할 수 있으며, 이 행위가 끝난 경우에는 state가 변할 수도 있고 변하지 않을 수도 있음.

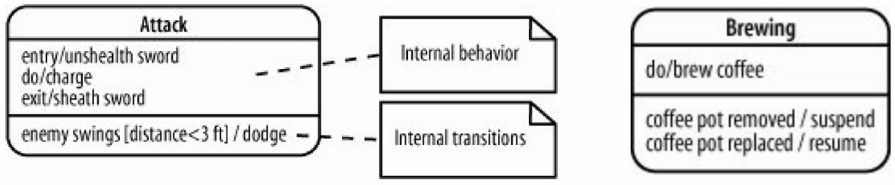
  2) entry/behavior: 해당 state가 active될 때 수행되는 행위.

  3) exit/behavior: 해당 state가 inactive될 때 수행되는 행위.

- *Transition(전이): state의 변경을 화살표와 이벤트로 나타냄.*

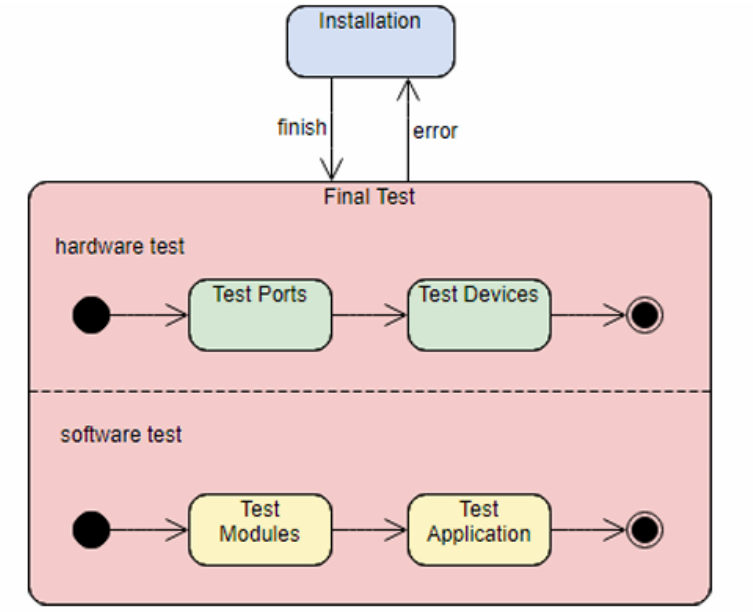
  화살표 위의 이벤트는 Trigger[Guard]/Action 꼴로 작성할 수 있고, 일부 생략할 수 있음. Trigger는 transition을 발생시키는 event, guard는 transition을 permit하거나 block하는 boolean condition, action은 transition 직후 수행되는 행위임.

  Internal Transition은 state 내에서 reaction을 유발하는 transition으로, rounded rectangle 안에 다음과 같이 transition 이벤트를 작성함.



2. Composite State

Composite State는 동시에 여러 state를 가지는 것을 말하며, 다음과 같이 rectangle 내에 나누어 그림.

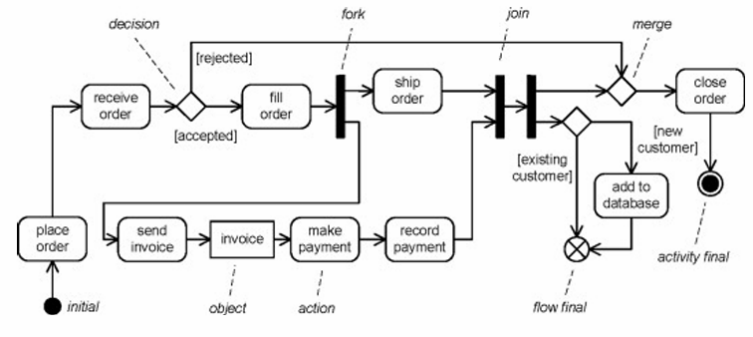


4.3. Activity Diagram

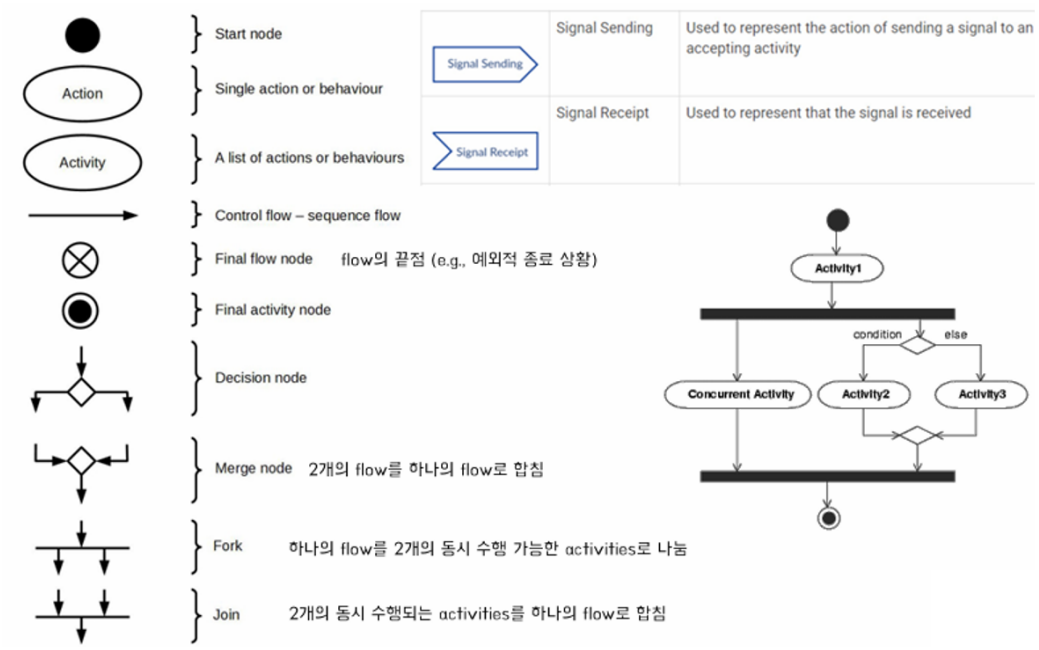
4.3.1. Activity Diagram

1. Activity Diagram

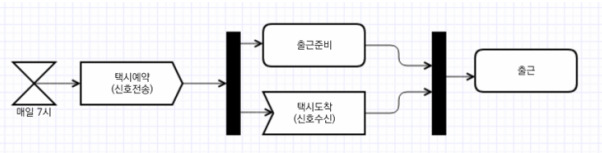
Activity Diagram(활동 다이어그램)은 시스템 내의 작업/동작의 흐름을 있게 나타낸 것임. sequential/branched/concurrent한 동작을 쉽게 나타낼 수 있음.



다음과 같이 activity를 circle로, control flow를 화살표로 그리는 등의 notation으로 표기함.

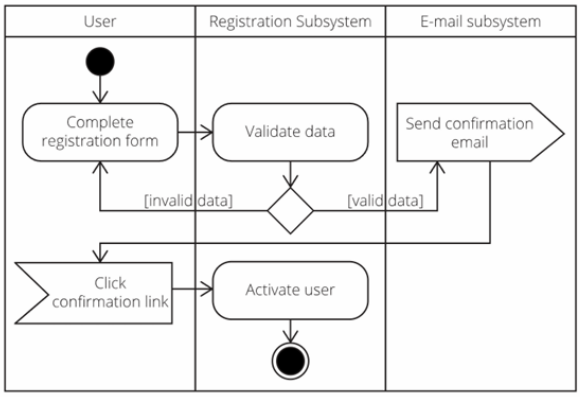


- *decision/merge node로 branch를 나타낼 수 있음. 이때 branch 조건을 화살표 위에 작성할 수 있음.*
- *fork/join으로 concurrent flow를 나타낼 수 있음.*
- *final flow node는 예외적 종료 상황 등을 나타냄.*
- *signal을 보내고 받는 상황을 나타낼 수 있음. signal을 받는 activity에서는 해당 signal이 도착할 때까지 control을 block함.*



2. Swimlane

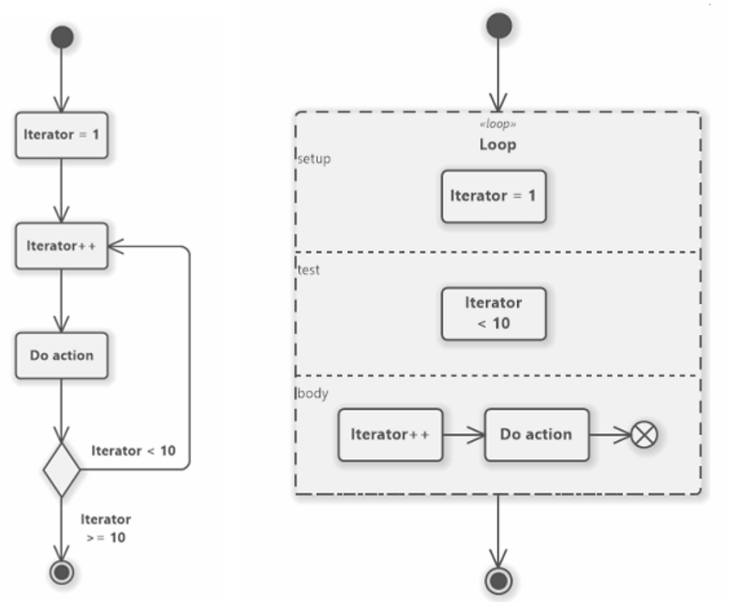
activity diagram은 여러 class에 대해 표기하게 되는 경우가 많은데, 이를 class별로 swimlane으로 그릴 수 있음.



3. Loop

activity diagram에서 loop를 나타내는 방법으로는 다음과 같은 것들이 있음.

- *Plain Notation: 기본 notation만 활용해도 loop를 쉽게 표기할 수 있음.*
- *Loop Node: loop를 하나의 node로 표기할 수 있음. setup에는 초기 조건, test에는 loop의 반복 조건(참이면 반복), body에는 test가 참일 때 수행할 활동을 작성함.*



- *Extension Region: collection의 개별 entry(원소)를 각각 처리하는 것을 표기할 수 있음. 즉, for each 구문을 처리하는 것을 나타내기 쉬움.*