Chapter 01. 객체지향

객체지향이란?

1. 객체지향의 등장

객체지향의 개념은 1970년대에 들어서 용어가 나왔으며, 이는 벌써 50년이 넘어가는 역사를 가지고 있습니다.

하지만 이전에는 C언어 처럼 실행하고자 하는 순서대로 명령어를 입력해서 실행되는 **"절차 지향"** 이 주를 이뤘으며, 이러한 방법으로 코딩 하는 언어들을 **"절차지향 언어"** 라고 합니다.

이 때에는 프로그램의 단위가 크지 않았으며, 대체적으로 **간단한 Logic을 순차적으로 처리하여 결과**를 얻는데 그쳤지만, 점점 컴퓨터의 발전과 이로 인하여 프로그램의 복잡도가 증가하면서 이에 들어가는 유지보수, 개발기간 등 다양한 부분에서 **비 효율이 발생** 하였습니다.

이런 어려움을 해결하기 위해 선택한 방법이 **"효과적인 개발방식"** 을 채택하게 되었고, 이는 이전에 사용하던 흐름에 따른 개발 방식에서 벗어나,

객체지향의 특성인
추상화, 상속, 은닉, 재사용, 인터페이스 등 여러 곳에서 객체지향으로 개발을 시작.

1. 객체지향의 등장

함수의 활용으로도 충분히 좋은 프로그램을 개발 할 수 있었으나, 새로운 시각으로 바라보기 시작 하였습니다.

객체지향이란 현실에 존재하는 사물을 있는 그대로 모델링하여, 이들의 행위와 속성을 정의 하고, 절차적이 아닌 객체가 중심이 되어 실제 사물이 동작하는 방식으로 설계하기 시작하였습니다.

이는 사물에 대해서는 객체 Object라고 부르며, 해당 사물이 하는 행위를 Method로 정의 하고 해당 사물이 가지는 속성을 변수 Variable라고 정의 합니다.

실제 사물을 중심으로 설계를 하기 때문에 기존의 절차지향 보다는 조금 더 편리하게 설계가 가능 해졌습니다.

1. 객체지향의 등장

Java 는 1995년 Sun Microsystems 작은 언어를 지향하는 객체지향 언어로 Java를 소개 하였습니다. 이는 당시에 C++ 과 유사한 언어의 구문을 채택 하였으나, C++ 이 가지고 있는, 시스템 레벨 접근, 메모리 직접 할당 및 해제, 포인터 등 복잡한 개발 방식을 사용하지 않았습니다.

또한 어떠한 운영체제에서도 자바 가상 머신 만 있으면, 독립적으로 실행 될 수 있도록 설계가 되어 있어서, 여러 플랫폼에서 호환성을 제공하는 장점을 가지고 있습니다.

2. 객체 설계하기

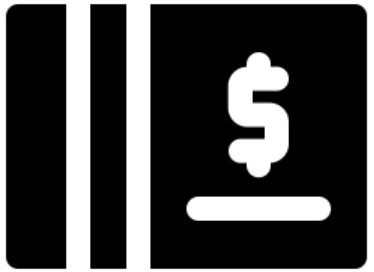
객체 == 사물 == Object → Class의 정의

속성

자동차 이름
자동차 번호
등록년월
모델명
…
…

속성

아이디
패스워드
이메일
전화번호
…
…



속성

계좌번호
잔고
예치금
이율
…
…

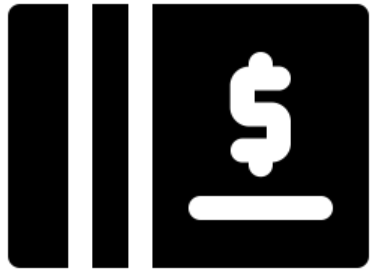
2. 객체 설계하기 method

행위

주행거리
연비계산
번호교체
등록증갱신
…
…

행위

계정등록
계정삭제
비밀번호변경
이메일변경
…
…



행위

잔고계산
이율계산
입금기록
출금기록
…
…

2. 객체 설계하기

객체의 3가지 요소

1 - 상태 유지 (객체의 상태)

객체는 상태 정보를 저장하고, 유지되어져야 하며 이러한 속성(Variable)은 변수로 정의 되어져야 한다. 이러한 속성값이 바뀜으로 인하여, 객체의 상태가 변경 될 수 있어야 한다.

2 - 기능 제공 (객체의 책임)

객체는 기능을 제공해야 한다. 이 부분은 Method의 제공으로 이루어진다.
이 부분은 캡슐화와 연관이 있으며, 외부로 부터 직접 속성에 접근하여 변경 하는 것이 아닌 객체가 제공하는 Method로 기능이 제공 되어져야 한다.

3 - 고유 식별자 제공 (객체의 유일성)

각각의 객체는 고유한 식별자를 가져야 한다.
예를 들면 카드번호, 계좌번호, 자동차 번호와 같은 속성을 통해서 각각 고유한 값을 줄 수 있으며,
이는 이후 Db에서 Unique Key, 또는 Primary key 로도 작성이 가능하다.

3. 물리 객체와 개념 객체

물리객체

물리적 객체는 실제로 사물이 존재하며, 이를 클래스로 정의한 객체를 의미한다.

Ex)
자동차 렌탈 시스템 : 자동차, 고객, 직원, 사업장, 정비소 등등

급여 관리 시스템 : 직원, 월급통장 등등

PC방 관리 시스템 : PC, 직원, 공간, 책상, 의자

3. 물리 객체와 개념 객체

개념객체

이후 우리가 개발할 웹 시스템에서 Service에 해당되며, 이는 business logic을 처리하는 부분을 의미합니다.

Business logic에서는 여러 객체를 서로 상호작용 하도록 하며, 객체가 제공하는 오퍼레이션 method를 통하여 객체의 속성을 변경 시킵니다.

EX)
사용자 관리 시스템
사용자 객체의 마지막 접속일자를 이용하여, 계정만료, 비밀번호 초기화, 재등록 처리 등등

ATM 시스템
사용자(Object)의 Action에 따라, 계좌(Object) 의 잔고의 속성을 변경 하는 , 입금/출금 Logic 처리

3. 물리 객체와 개념 객체

객체지향에서의 대부분의 코딩은 각 객체에 기능을 정의하고 이를 business logic을 처리 하는 Service 에서 객체의 Method를 활용하여, 여러 가지 조건을 확인하여, 객체의 속성을 변경하는 작업이 주된 코딩 이 됩니다.

이러한 작업을 하기 위해서는 각 객체의 속성(Variable) 이러한 속성을 변경하거나 상태를 변경 할 수 있는 오퍼레이션(Method)을 잘 정의 해야 합니다.

Chapter 01. 객체지향

객체지향의 4대 특성

1. 캡슐화

캡슐화는 객체의 속성(Variable) 을 보호하기 위해서 사용 합니다.

객체의 캡슐화는 현실 세계에서도 볼 수 있습니다. 컴퓨터 본체 안에 수 많은 부품이 있지만, 전원을 켜기 위해서는 메인보드에 전기 신호를 직접 주는 것이 아닌, 외부 케이스에 있는 전원 버튼을 통해서 상태 속성을 On/Off 하도록 변경 합니다.

1. 캡슐화

Method 설계

- 속성이 선언되었으나, 이의 상태를 변경하는 method가 없다면, 잘못 선언된 속성이다.
  즉, 자신이 가지고 있는 속성에 대해서는 해당 상태를 변경하는 기능을 제공해야 한다.

- 실물 객체가 가진 기능을 모두 제공 해야 한다.
  예를 들면, 자동차의 렌탈, 반납, 주행거리 계산 등등

- 각각의 Method는 서로 관련성이 있어야 한다.
  차량의 렌탈/반납, 자동차 등록증 등록/해지 등 각 속성의 상대 되는 기능을 제공해야 한다.

- 객체 안의 Method는 객체 안의 속성을 처리해야 하며, 다른 객체를 전달받아 해당 다른 객체에 정의 된 속성을 직접 처리 하면 안 된다.

  단, Method에 실행에 필요한 값들은 객체의 형태가 아닌 매개변수의 형태로 전달되어져야 한다.

1. 캡슐화

- Getter / Setter Method
  외부에서 내부 속성(Variable)에 직접 접근 하는 것이 아닌 Getter/Setter Method를 통해서 접근 하도록 적용

- CRUD Method
  데이터 처리를 위한 기본적인 CRUD Method를 제공

- Business Logic Method
  비즈니스 로직 처리를 위한 Method를 제공

- 객체의 생명 주기 처리 Method
  흔히 destroy(), disconnect() 등 quit() 등 소멸에 대한 method

- 객체의 영구성 관리 Method
  영구성(유효성) 속성에 대한 변경이 필요한 경우 외부에서는 접근이 불가능 하도록 private로 선언하며, 내부의 다른 Method를 통해서 사용 되도록 한다.

Method의 속성은 반드시 1개에 속할 필요는 없으며, 여러 속성에 해당 될 수 있다.

1. 캡슐화

장점

- 객체지향의 패러다임 중 하나인 추상화를 제공한다.
  실제로 Method가 어떻게 동작하는지는 외부에서는 이해할 필요가 없으며, 이를 단순 호출만으로 해당 기능을 실행 할 수 있고, 이를 통해서 객체 단위로 프로그램 설계가 가능하다.

- 재 사용성 향상
  한 객체에 관련된 속성 및 Method는 모두 캡슐화의 형태로 제공됨으로, 객체의 모듈성과 응집도가 높아진다. 이를 통하여 재 사용성이 높아진다.

  만일 절차적 프로그래밍에서, Method 를 재사용한다면, 함수가 참조하고 있는 전역변수 및 내부에서 호출하는 Method가 미치는 영향을 모두 체크 해야 하나, 객체의 경우는 단일 객체에만 영향을 주기에 재 사용성이 높다.

- 앞선 이유로 인하여, 유지보수의 효율성이 향상 된다.

1. 캡슐화

무결성

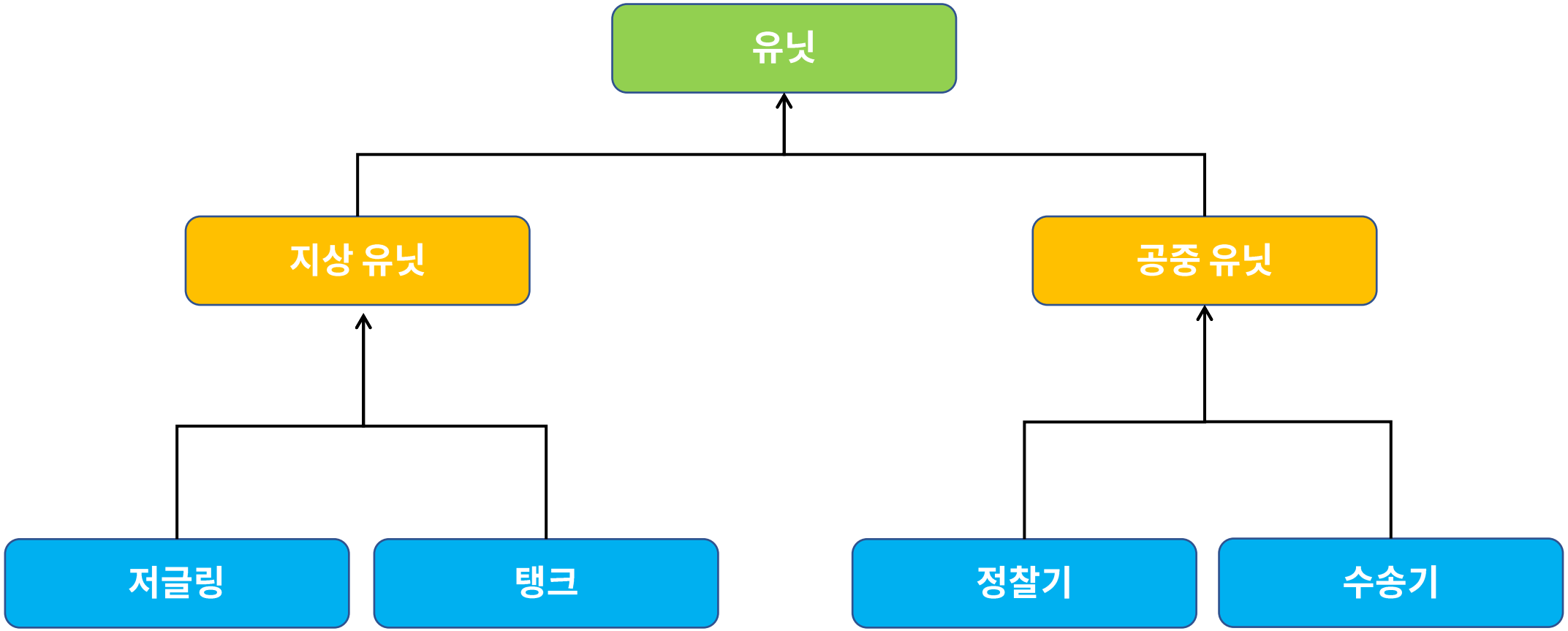
보통의 캡슐화 코딩이라고 한다면, 주로 변수는 private 로 선언하고, Method를 public으로 선언하는 형태를 많이 가지게 됩니다.

이는 객체의 무결성을 위함 입니다. Getter/Setter를 제외 하고는 public method는 입력된 매개변수를 Validation을 한 후에 실행 하는 것을 기본으로 합니다.

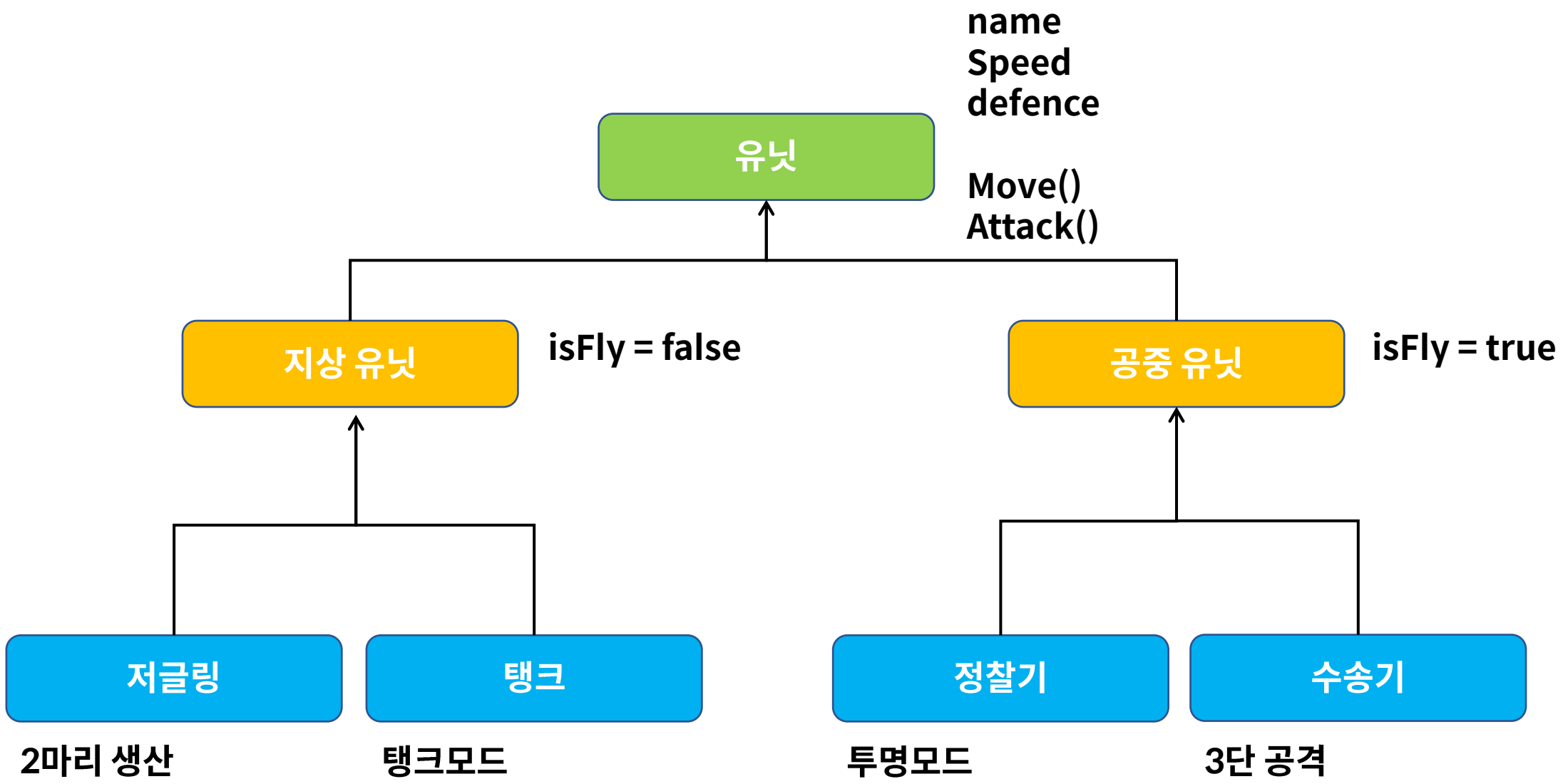
Validation을 통하여, 객체의 값을 바꾸거나, 값의 대한 유효성을 가질 수 있습니다.

2. 상속

- 객체지향에서의 상속은, 속성의 상속이 아닌, 하위로 내려갈 수록 구체화 되는 것이다.



2. 상속



2. 상속

상속의 효과

- 프로그램 구조에 대한 이해도 향상
  최상위 클래스의 구조를 보고, 하위 클래스의 동작을 이해 할 수 있다.

- 재사용성 향상
  상속을 이용하여, 해당 클래스에 필요한 속성 및 메소드를 모두 정의 하지 않고, 상속을 받아서 사용 할 수 있다.

- 확장성 향상
  일관된 형태의 클래스 객체를 추가 할 수 있어, 간단하게 프로그램 확장이 가능하다.
  Ex) 신규 유닛

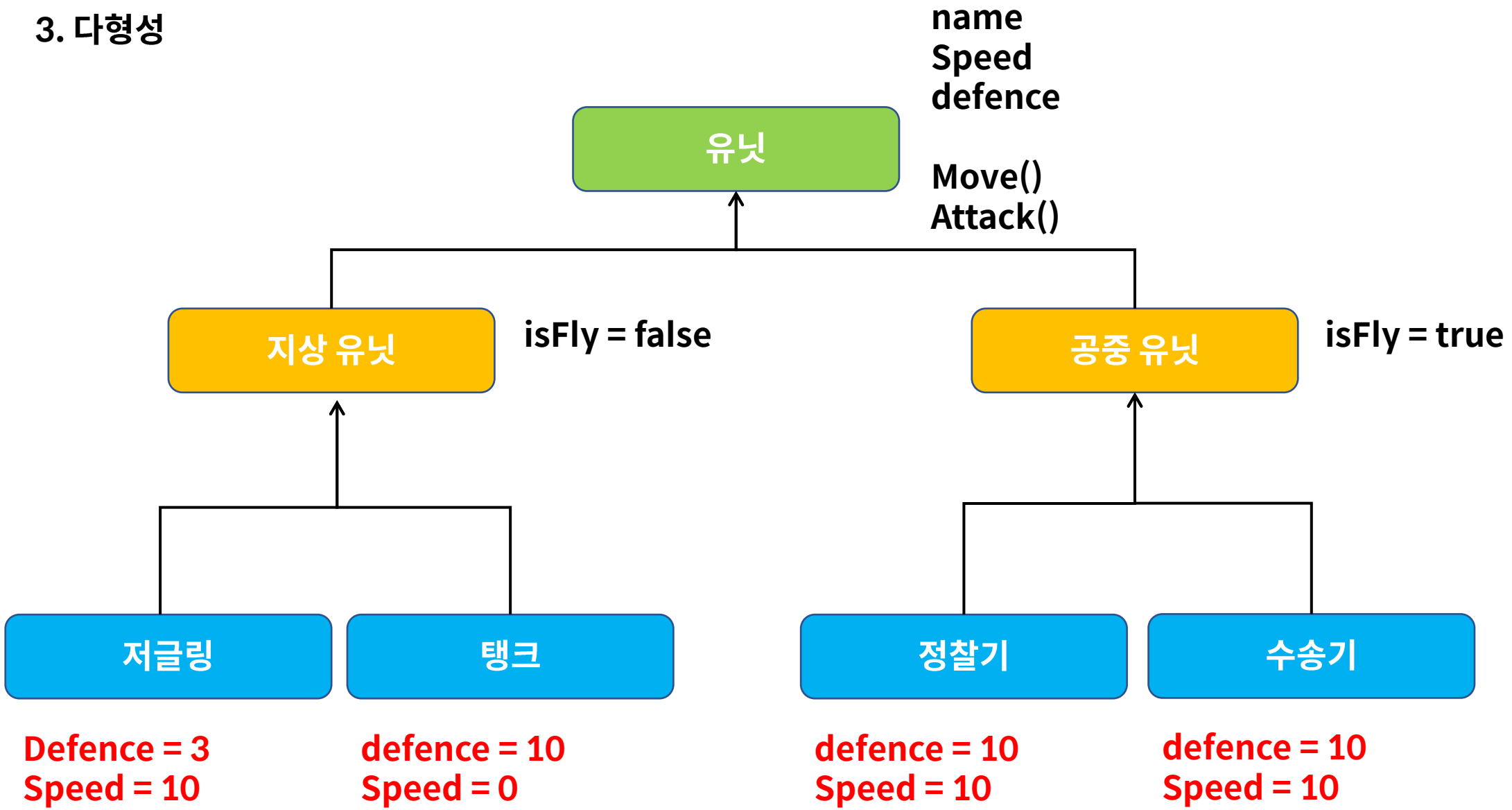
- 유지보수성 향상
  각 객체마다, 자신의 메소드를 정의 하고 있다면, 코드 수정에서 많은 작업이 필요 하지만, 상속을 사용한 경우 일관된 형태로 작성이 가능하다.

3. 다형성

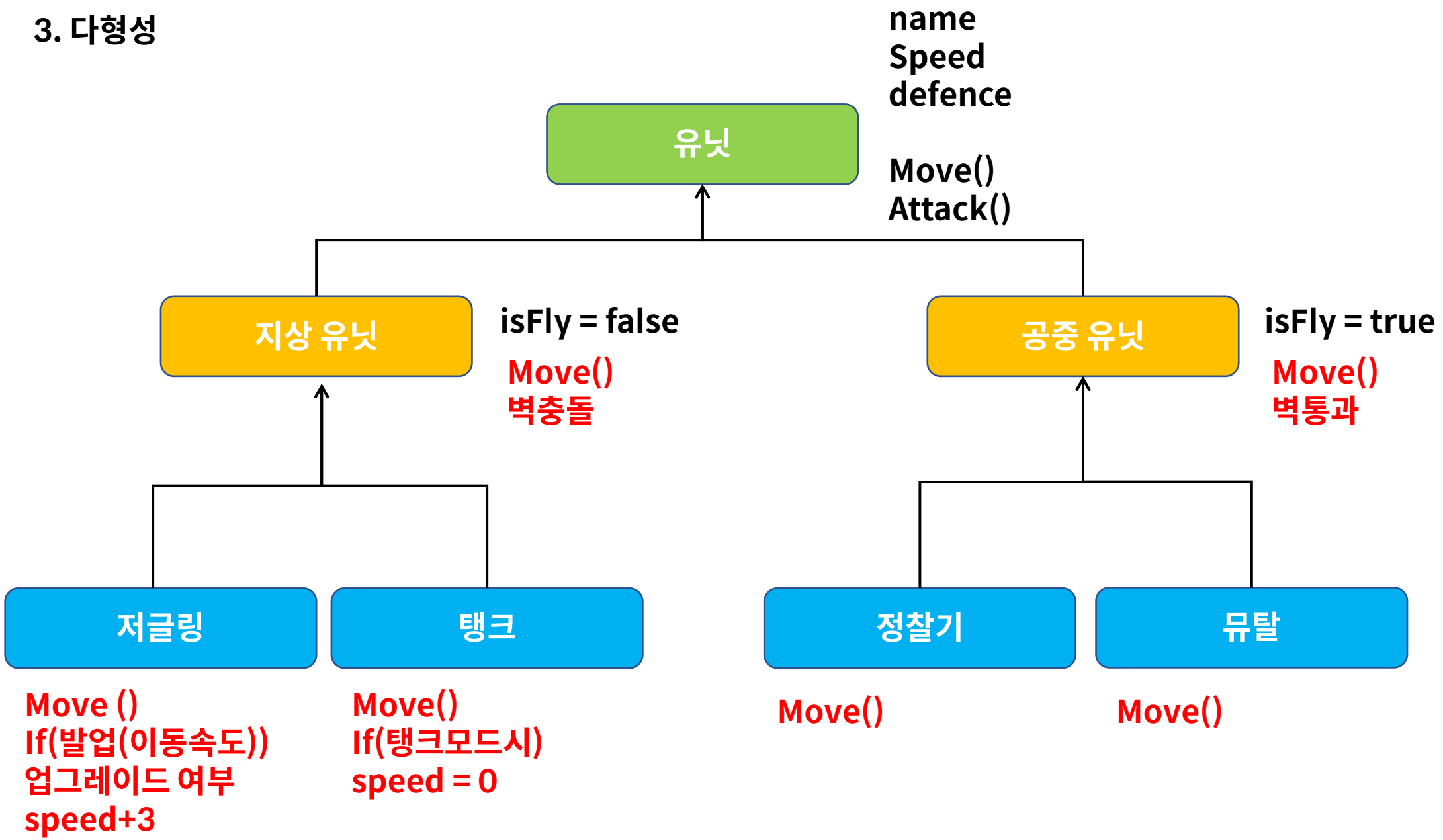
- 다형성은 하나의 개체가 여러 개의 형태로 변화 하는것을 말하며, 이를 객체지향 에서도 유사하게 사용을 하고 있습니다.

- 다형성을 하기 위해서는 오버라이딩을 통해서 가능 합니다.

3. 다형성



3. 다형성

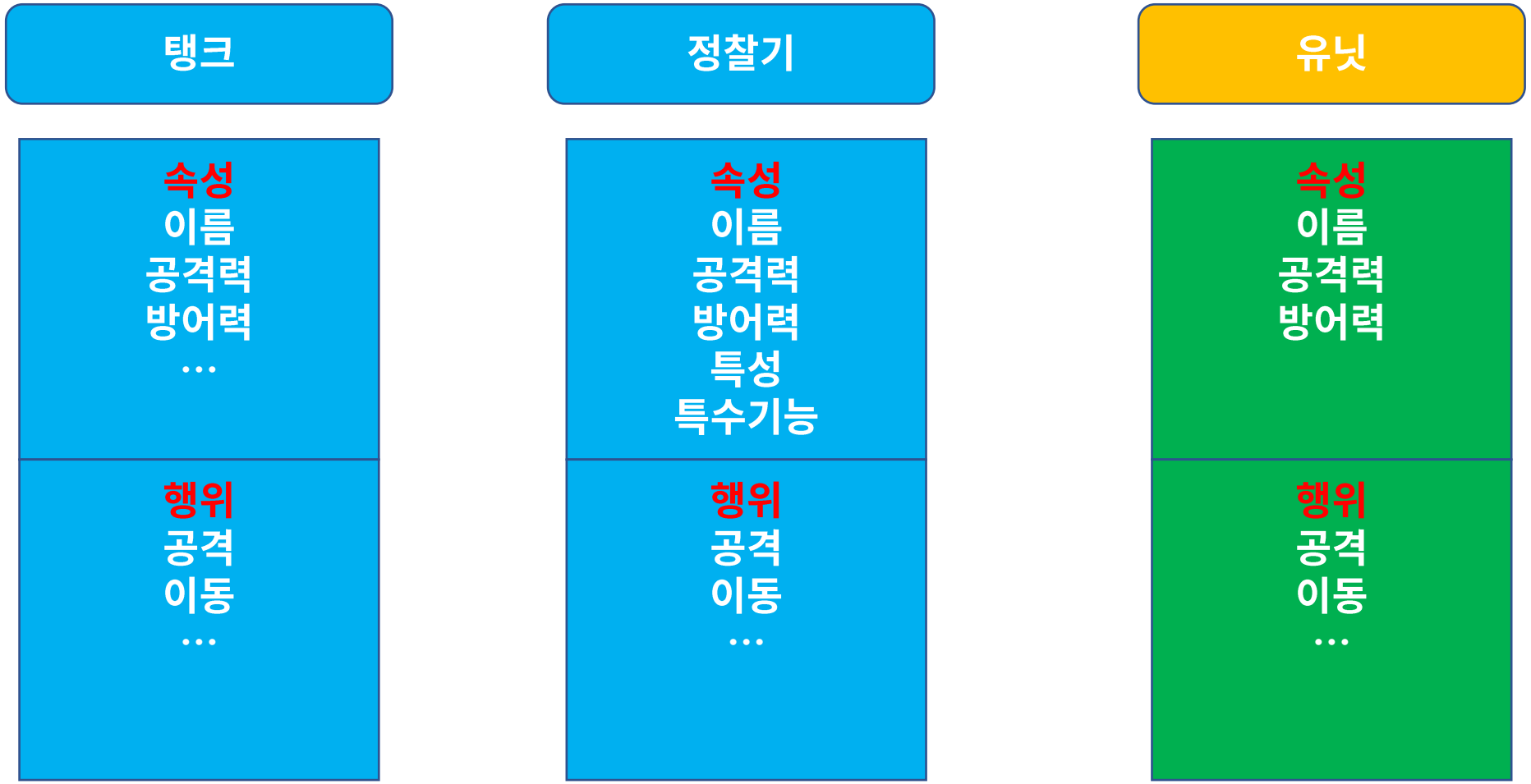


3. 다형성

```
Unit 저글링 = new 저글링();

Unit 시즈탱크 = new 시즈탱크();

Unit 레이스 = new 레이스();

Unit 뮤탈 = new 뮤탈();

unitMove(저글링);
unitMove(시즈탱크 );
unitMove(레이스 );
unitMove(뮤탈 );


Private void unitMove(Unit unit){

    unit.move()
}
```

4. 추상화

- 객체지향에서의 추상화는 모델링 이다.
- 구체적으로 공통적인 부분, 또는 특정 특성을 분리해서 재조합 하는 부분이 추상화 입니다.
- 앞에서 배운 다형성, 상속 모두 추상화에 속한다.



Chapter 01. 객체지향

객체지향 설계 5원칙 SOLID

응집도와 결합도

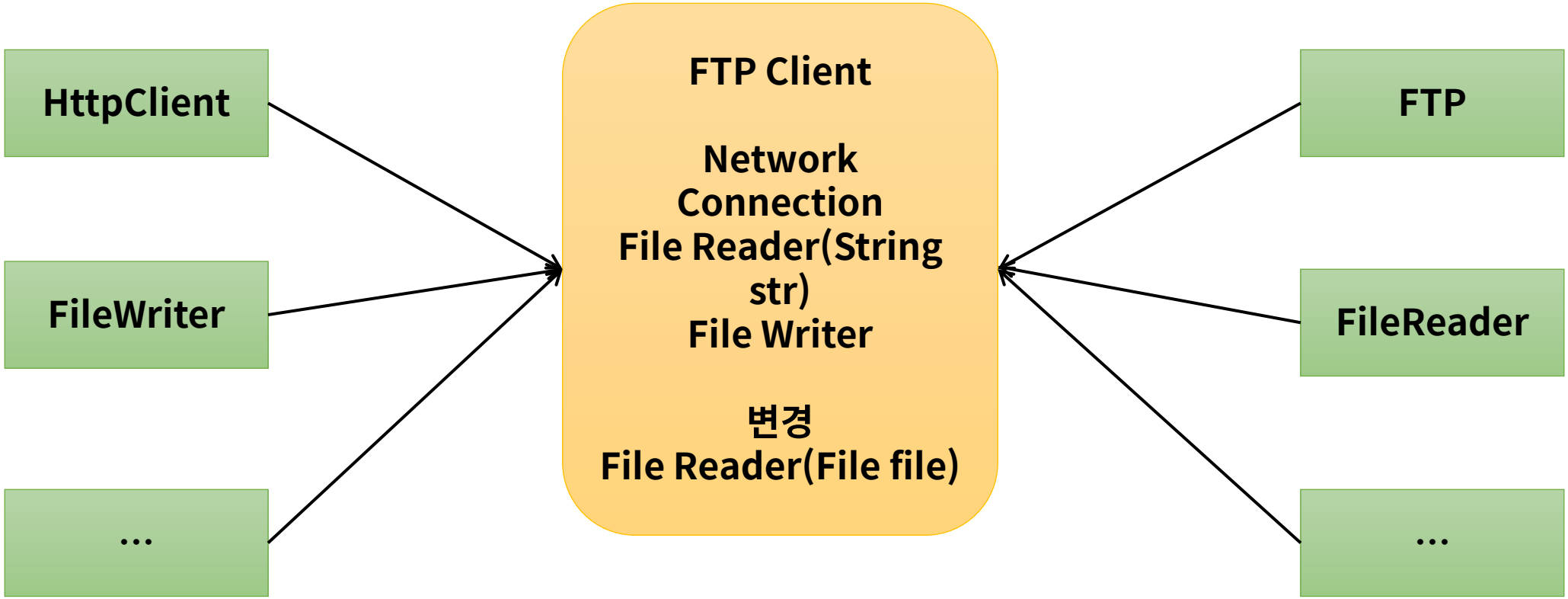
좋은 소프트웨어 설계를 위해서는 결합도(coupling)는 낮추고 응집도 (cohesion) 는 높여야 한다.

결합도
모듈(클래스)간의 상호 의존 정도를 나타내는 지표로써
결합도가 낮으면 모듈간의 상호 의존성이 줄어들어서 객체의 재사용 및 유지보수가 유리하다.

응집도
하나의 모듈 내부에 존재하는 구성 요소들의 기능적 관련성으로
응집도가 높은 모듈은 하나의 책임에 집중하고 독립성이 높아져, 재사용 및 유지보수가 용이하다.

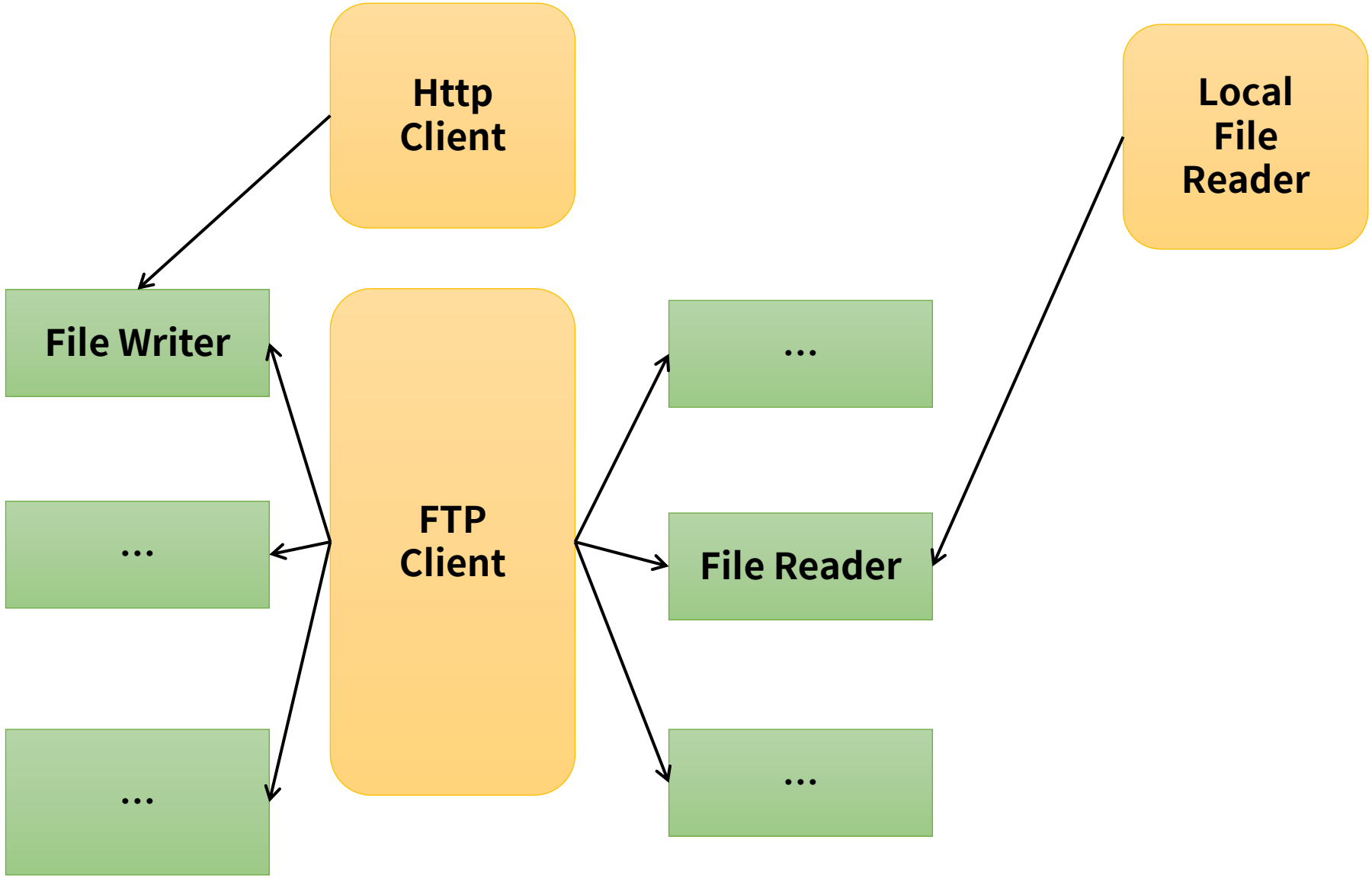
1. SRP(Single Responsibility Principle) 단일 책임 원칙

어떠한 클래스를 변경해야 하는 이유는 한가지 뿐 이여야 한다.



1. SRP(Single Responsibility Principle) 단일 책임 원칙

1. SRP(Single Responsibility Principle) 단일 책임 원칙



1. SRP(Single Responsibility Principle) 단일 책임 원칙

```
class Unit {

        private String name;
        private int speed;

        public void attack(){

        }

        public void move(){

                if(name.equals("저글링")){
                        speed += 3
                }else if(name.euqals("탱크")){

                        if("탱크모드"){
                                speed = 0
                        }else{
                                speed = 10
                        }

                }else if(name.equals("정찰기")){
                        speed = 15
                        충돌 = false
                }
        }
}
```

1. SRP(Single Responsibility Principle) 단일 책임 원칙

```
class 저글링 extends Unit {

		public void move(){
			this.speed += 3
		}
}

class 탱크 extends Unit {
		public void move(){
			if("탱크모드"){
				speed = 0
			}else{
				speed = 10
			}
		}
}

class 정찰기 extends Unit {

		public void 정찰기(){
			this.충돌 = false
		}
		public void move(){
			speed = 15
		}
}
```

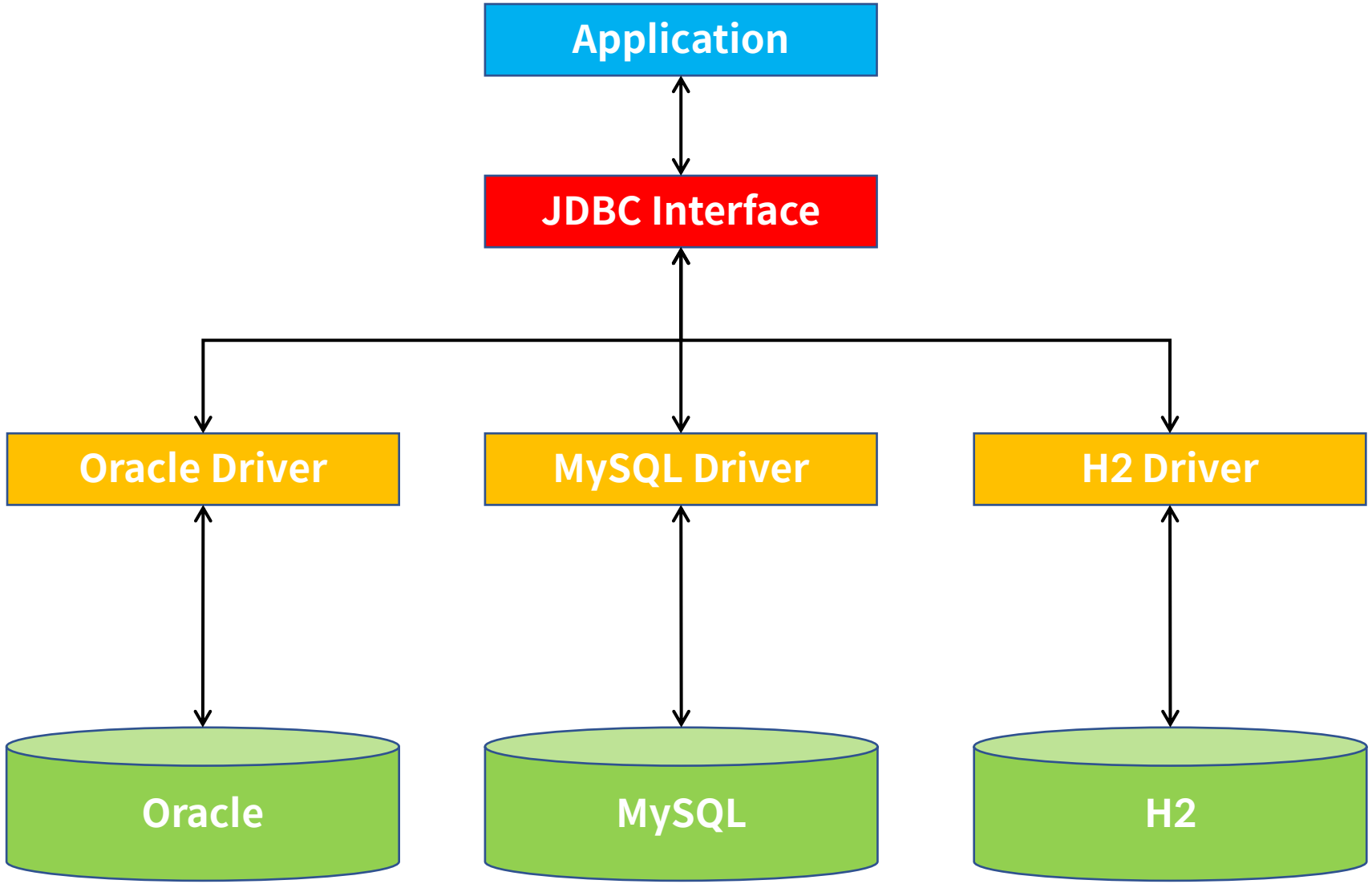
2. OCP (Open Closed Principle) 개방 폐쇄 원칙

자신의 확장에는 열려 있고, 주변의 변화에 대해서는 닫혀 있어야 한다.

상위 클래스 또는 인터페이스를 중간에 둠으로써, 자신은 변화에 대해서는 폐쇄적이지만, 인터페이스는 외부의 변화에 대해서 확장을 개방해 줄 수 있다.

이러한 부분은 JDBC 와 Mybatis, Hibernate 등 JAVA에서는 Stream(Input,Out)에서 찾아볼 수 있다.

2. OCP (Open Closed Principle) 개방 폐쇄 원칙



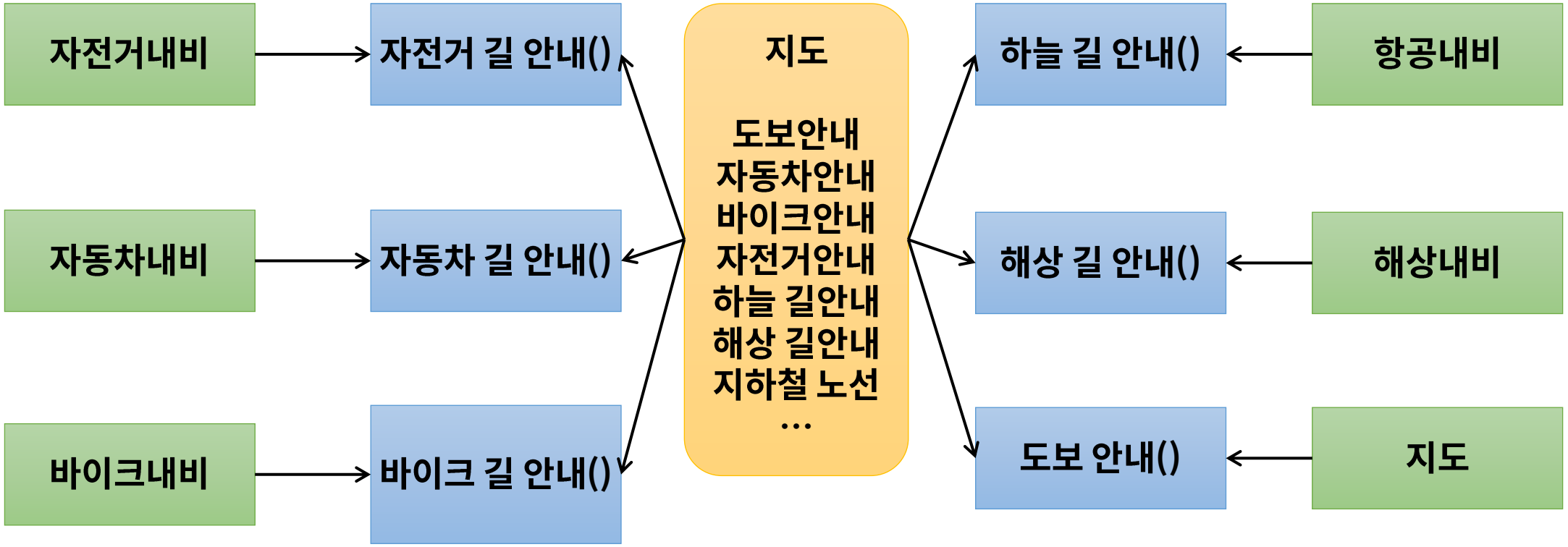
3. LSP (Liskov Substitution Principle) 리스코프 치환 원칙

서브 타입은 언제나 자신의 기반(상위) 타입으로 교체 할 수 있어야 한다.

4. ISP (Interface Segregation Principle) 인터페이스 분리 원칙

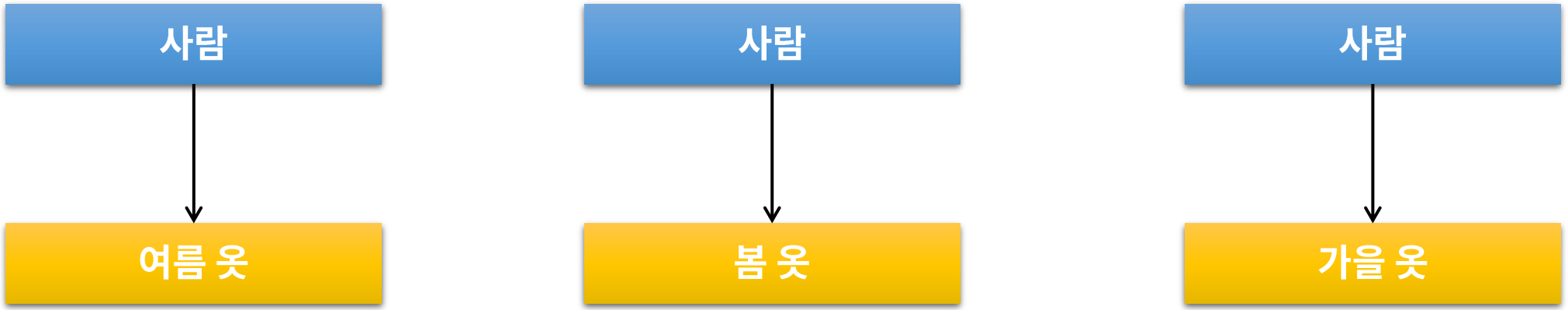
클라이언트는 자신이 사용하지 않는 메서드에 의존 관계를 맺으면 안된다.

프로젝트 요구 사항과 설계에 따라서 SRP(단일책임원칙) / ISP(인터페이스분리원칙) 를 선택 한다.



5. DIP (Dependency Inversion Principle) 의존 역전 원칙

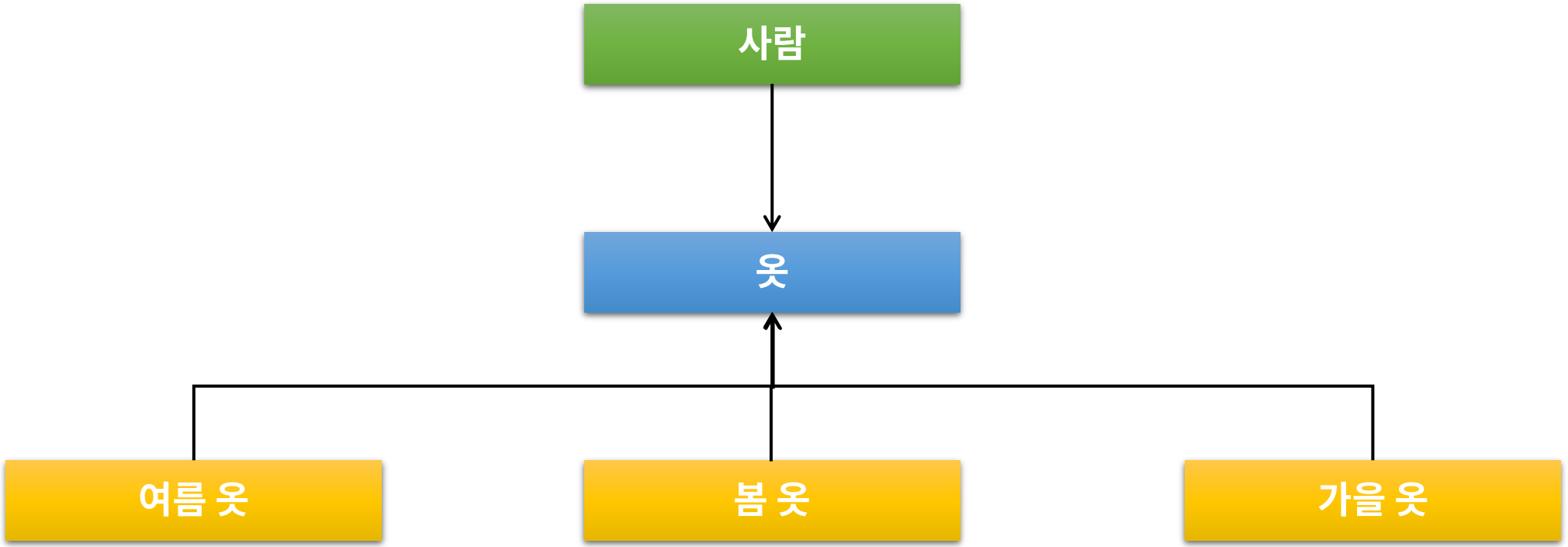
자신보다 변하기 쉬운 것 에 의존하지 말아야 한다.



5. DIP (Dependency Inversion Principle) 의존 역전 원칙

개발 폐쇄 원칙에서도 살펴본 부분이다.

SOLID는 객체 지향 4대 특성에 기반함으로써 유사한 모양을 가지고 있다.



Chapter 01. 객체지향

POJO JAVA

POJO JAVA란?

POJO (Plain Old Java Object)

순수한 자바 오브젝트를 뜻한다.

역사를 거슬러 올라가보면, 예전 EJB가 인기를 끌고, 많이 사용하던 시절에는 단순한 자바 오브젝트를 사용해서 개발하는 것이 아닌, EJB에 종속적인 부분으로 개발을 진행.

그로 인하여, Module의 교체, 시스템 업그레이드시 종속성으로 인하여 불편함 발생.

POJO 특징

1. 특정 규약에 종속 되지 않는다.

   특정 Library, Module 에서 정의된 클래스를 상속 받아서 구현하지 않아도 된다.
   POJO가 되기 위해서는 외부의 의존성을 두지 않고, 순수한 JAVA 로 구성이 가능해야 한다.

2. 특정 환경에 종속되지 않는다.

   만일 특정 비즈니스 로직을 처리 하는 부분에 외부 종속적인 http request, session 등 POJO를 위배한 것으로 간주 한다.
   또한 많이 사용하고는 있지만 @Annotation 기반으로 설정하는 부분도 엄연히는 POJO라고 볼 수는 없다.

POJO Framework

1. **Spring, Hibernate**

   하나의 서비스를 개발하기 위해서는, 시스템의 복잡함, 비즈니스 로직의 복잡함 등
   다양한 어려움을 맞이 하게 된다.

   위의 두 프레임워크는 객체지향적인 설계를 하고 있으며, 또한 POJO를 지향하고 있다.

   그러므로 개발자가 서비스 로직에 집중하고 이를 POJO로 쉽게 개발 할 수 있도록 지원하고 있다.

Chapter 01. 객체지향

마치며···

마치며…

자신의 코드에 if/else , switch 가 난무 하고 있지 않은가?

책임과 역할이 다른 코드가 하나의 클래스에 다 들어가 있지 않은가?

절차지향적으로 한 개의 파일에 모든 코드를 넣고 있지 않은가?

내가 만든 객체가 재사용이 가능한가?

앞으로는 복잡한 엔터프라이즈 로직은 Spring, Hibernate에 맡기고, 이들이 지향하는 객체지향적 프로그래밍을 이를 사용하기만 해도 배울 수 있는 장점이 있습니다.

그렇기에 스프링 프레임워크는 오랜 기간, 그리고 전 세계적으로 사랑 받는 이유이기도 합니다.