

프로그래밍 역량 강화 전문기관, 민코딩

---

# 클린코드를 위한 OOP 개요



# 목차

1. 객체지향 vs 절차지향
2. 의존도
3. Attribute 와 Behavior
4. OOP개요1
5. OOP개요2
6. 클래스 다루기
7. 캡슐화
8. 상속
9. Interface 개요
10. Interface 구현
11. 다형성 구현
12. 확장성 고려한 Class 제작 Mission

객체지향 vs 절차지향

# 절차지향

무한 Loop 속 순서를 정하여 구현한다.

머릿속으로 전체를 시뮬레이션 해보면서  
어디에 넣어야 정상 동작하는지  
고민하면서 코딩한다.

```
while(1)
{
    //1. 버튼인식

    //2. 만약 stop 일 때
    모터stop( )

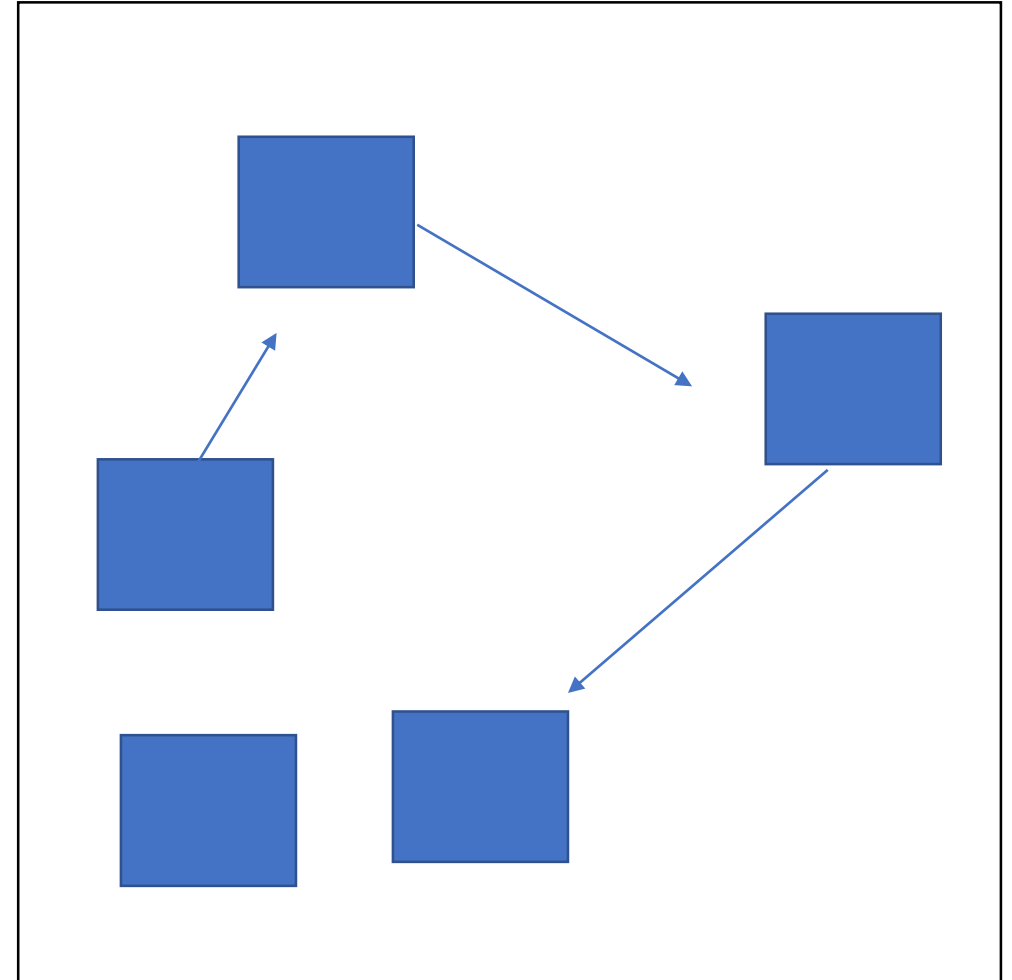
    //3. 만약 1 단계
    모터ON( )

    //4. 만약 2 단계
    모터 ON( )
    전력 UP( )
}
```

# 객체지향

하나의 시스템은 여러 모듈들로 구성되어있고  
모듈들이 메시지를 주고 받으면서  
전체 모듈이 하나의 시스템을 구성한다.

각 모듈은 본인의 역할에만 충실한다.



# 객체지향 : 실제와 비슷하게 설계

## 컨트롤러

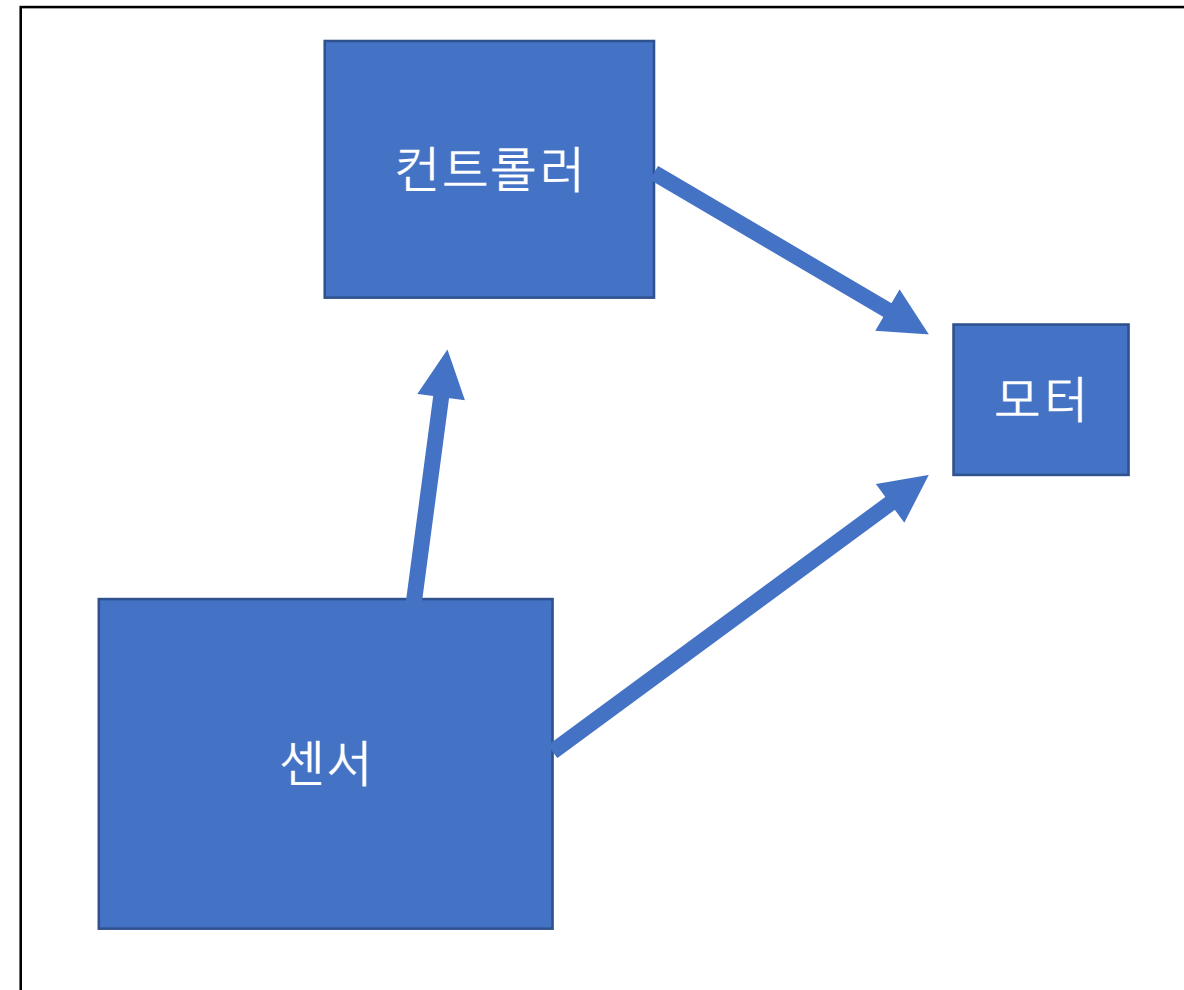
- 버튼이 눌리면 모터에게 메세지 보낸다.

## 모터

- 누군가에게 메세지 받으면 동작한다.

## 센서

- 모터가 과부하되면 컨트롤러에게 Stop메세지를 보낸다.



# 시스템 개발 시 중요한 것 - 모듈화

## ✓모듈화 (용어 암기!)

- 한 시스템을 어떤 구성요소로 이뤄지도록 설계할 것인가

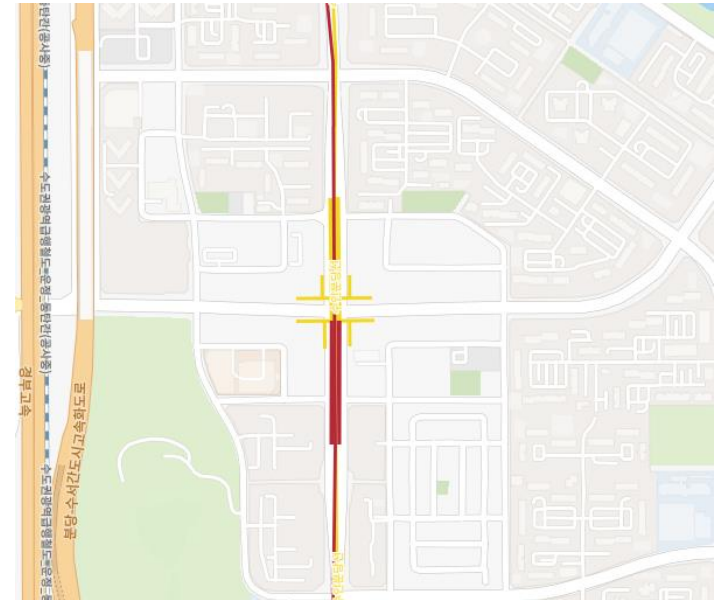
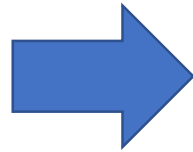
## ✓모듈화 설계 예시 (선풍기)

- 컨트롤러 + 모터 + 센서
- 전원부 / 버튼부 / 리모컨인식부 / 모터부 / 센서부
- 컨트롤러 + 모터 + 센서 (1. 모션센서 / 2. 온도센서 / 3. 진동센서)

# 모듈화를 하려면!?

설계에는 **추상화**는 필수!!

- 중요한 정보만 취급한다.





# [도전] 절차지향 vs 모듈화 (3분)

## 범퍼카 설계하기

1. 절차지향으로 디자인
2. 객체지향으로 디자인

모듈화 : 그림판에 간단히 설계해본다.



의존도

# 변경의 의미

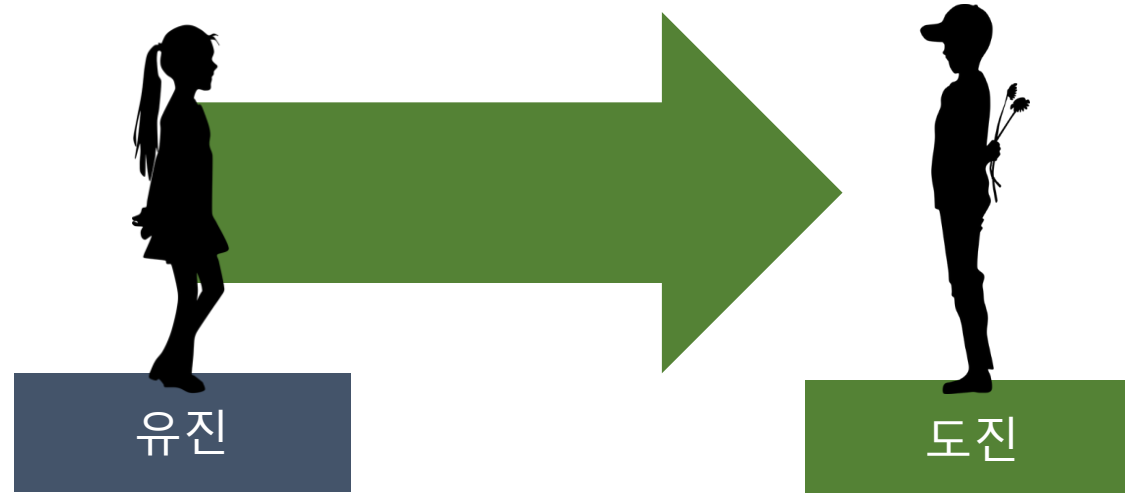
소스코드가 **요구사항에 의해** 수정되는 것

1. 메서드 / 필드 수정 / 제거
2. 메서드 / 필드 이름 변경
3. Parameter / Return 값 변경 등등

# 의존도 = 의존성 정도

유진이가 도진에게 의존성을 갖는다.

- 의존성이 크면, 변경에 큰 영향을 미침  
ex) 도진이가 힘들면, 유진이 삶에 영향을 끼친다.
- 의존성이 낮으면, 변경에 큰 영향을 끼치지 않음



# 클래스에서 의존

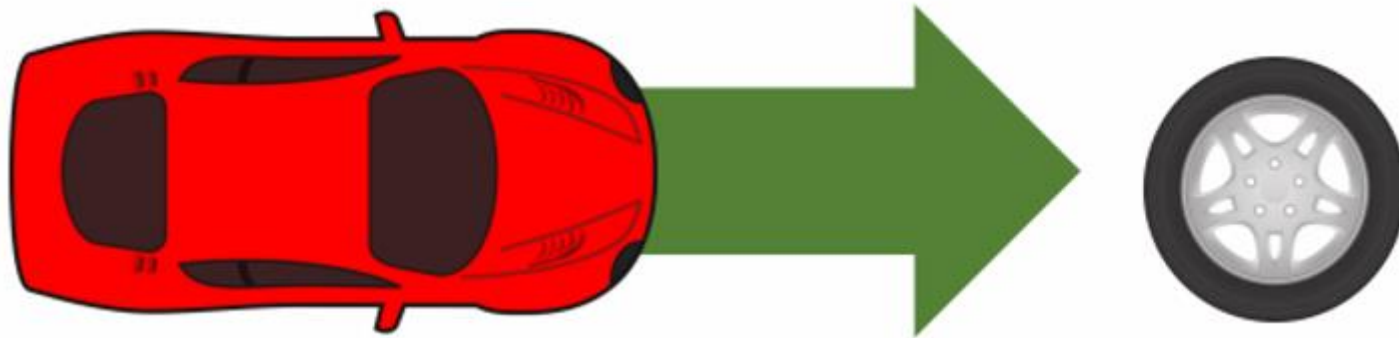
A 클래스가 B 클래스에 도움을 받으며 동작될 때,  
A 클래스는 B 클래스가 없으면 안 되는 상황이 된다.

이때 A 클래스는 B 클래스를 의존한다. 라고 표현한다.

# 다른 클래스에 의존하면?

A가 B Class에 의존할 때, 발생할 수 있는 일

- 만약 B 클래스의 변경이 일어나면, A 클래스도 변경을 해야 할 수 도 있음

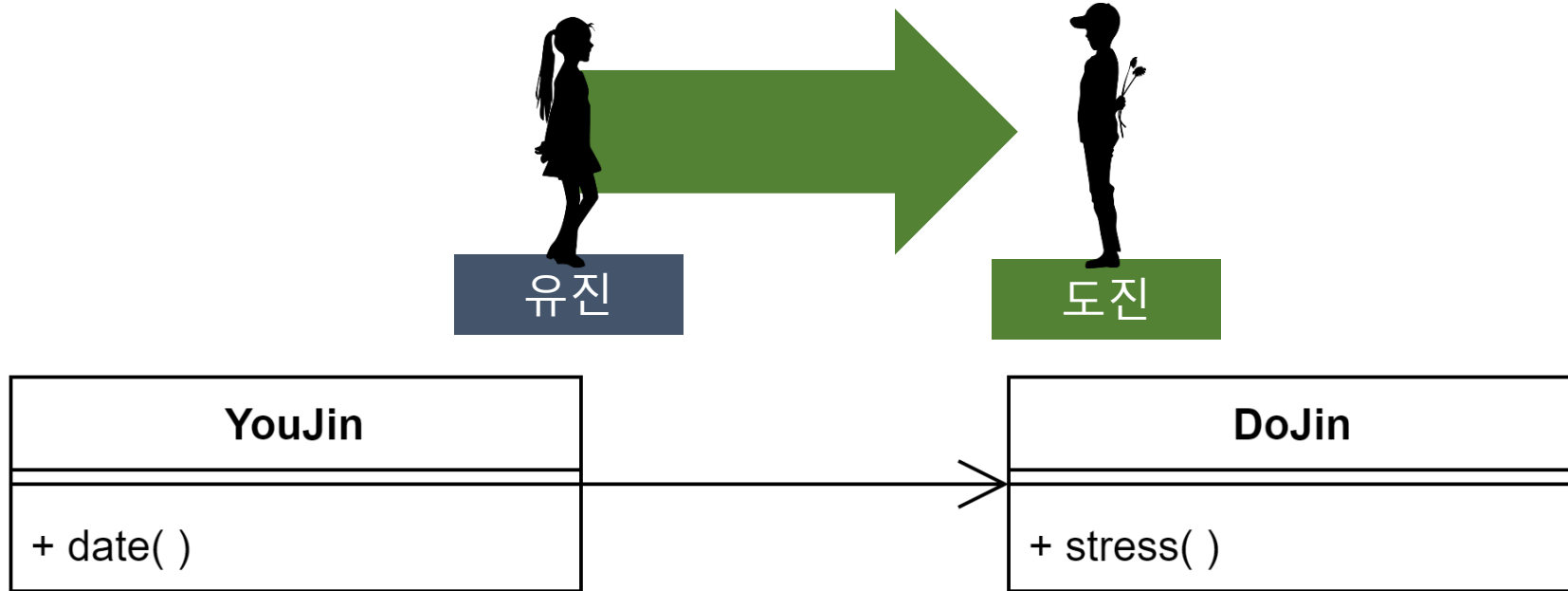


자동차가 바퀴에 의존할 때,  
바퀴 사이즈가 변경되면 자동차 설계를 변경해야 할 수도 있음.

# 의존도

YouJin 이 DoJin 에 대한 의존도가 큰 경우,

DoJin가 변경될 때  
YouJin도 함께 변경을 해야 하는 경우가 잦다.



# 의존도 = 결합도 (Coupling)

의존하는 정도를 결합도 (Coupling) 이라고 부른다.



Loose Coupling  
= 낮은 결합도



Tight Coupling  
= 높은 결합도



# [도전] 객체지향 설계 + 의존관계 방향

냉장고를 추상화 + 모듈화 진행  
그리고 각 모듈간 의존관계 방향을 설정



Attribute 와 Behavior

# 각 모듈은 세부 내용을 갖는다.

## Attribute

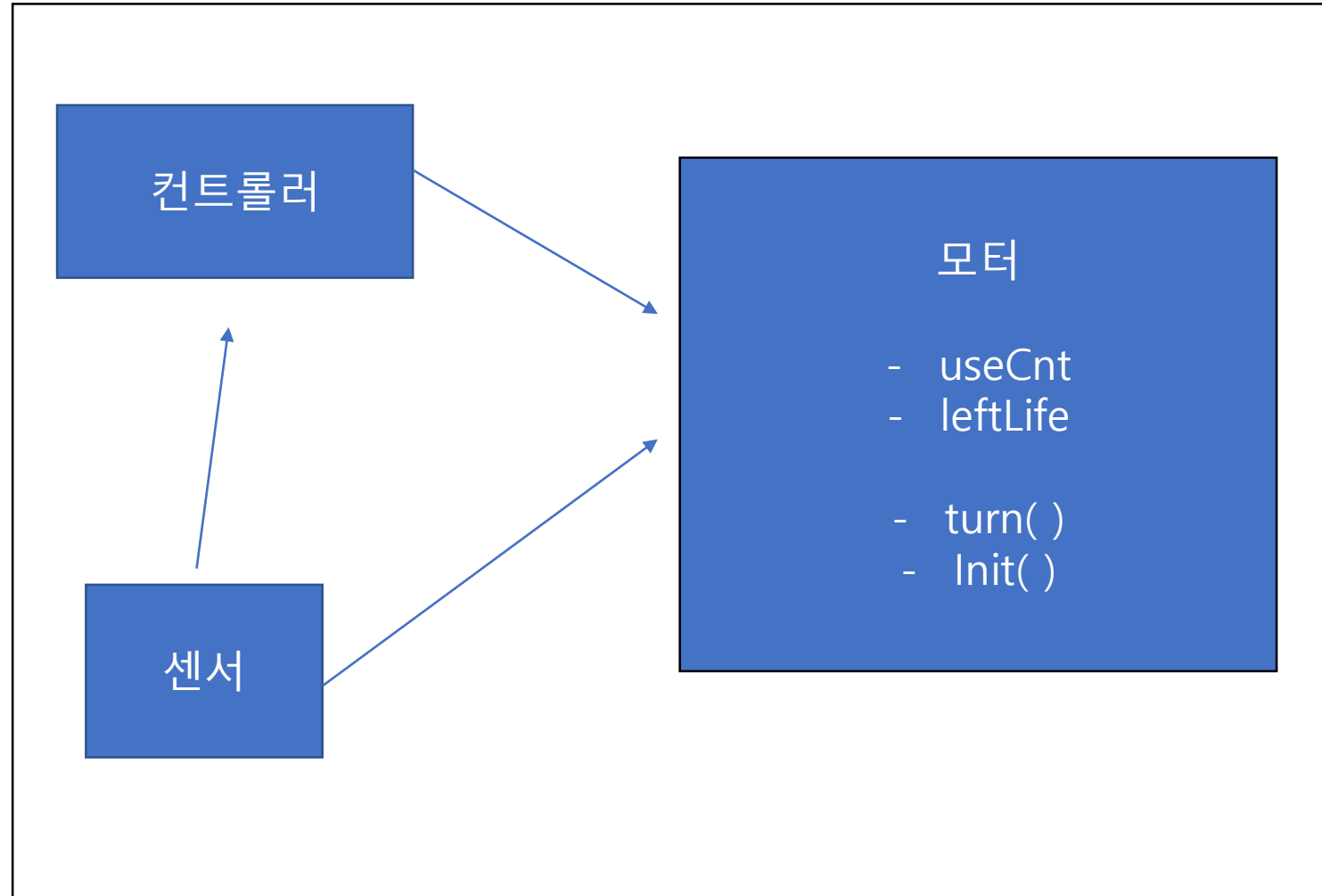
- 데이터 값
- OOP 언어에서 field 로 표현

## Behavior

- 동작
- OOP 언어에서  
메서드로 표현

추상화 필수!!

- 모든 내용을 넣지 말자



# 이상한 Attribute

이 모듈에 어울리지 않는 Attribute (??)  
이건 설계가 이상하다.

모듈과 관련있는 Attri. / Behavior 만  
모여있는 정도를 표현하는 단어는? → Co..

모터

- **buttonCount** (?)
- **PersonCount** (?)
- turn( )
- init( )

# Cohesion (응집도)

✓모듈이 한 가지 책임을 갖도록 Attri. / Behavior가 구성된 정도

마틴파울러는  
모듈이 담당해야하는 임무를  
모듈의 “책임(Responsibility)” 이라고 표현했다.

# 다음 멘트 이해하기

✓객체지향 설계는...

- 적절한 추상화가 되어있고,
- 적절한 작은 사이즈의 모듈화가 되어야 하고,
- 각 모듈끼리는 커플링이 루즈할수록 좋으며,
- 하나의 모듈은 Cohesion이 높도록 설계가 이뤄져야 한다.

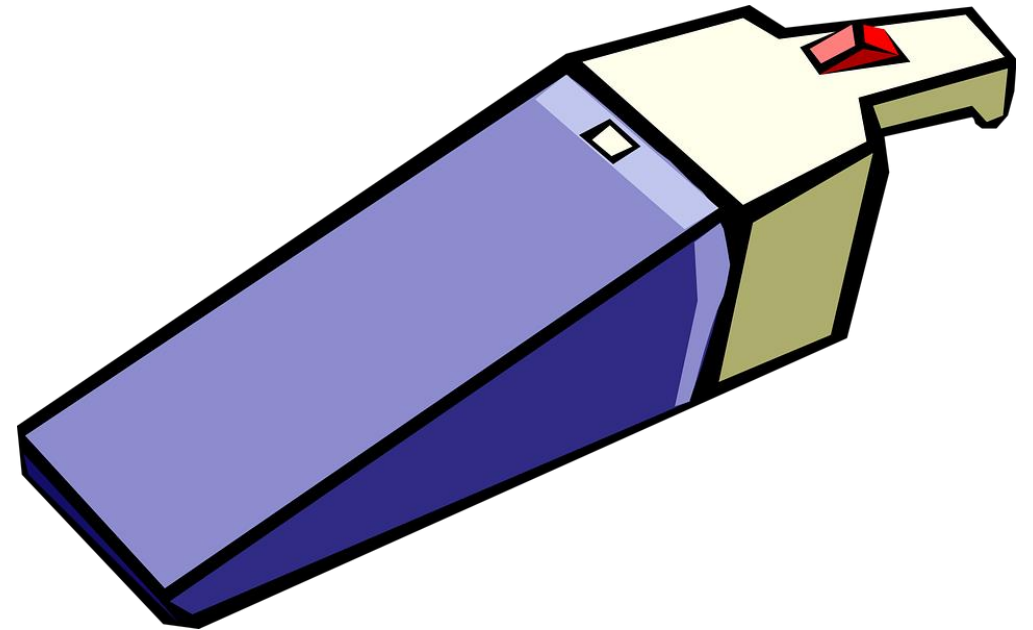
# [도전] OOP 설계하기

## 청소기 설계하기

- 어떤 모듈이 필요하고, 어떤 의존관계 방향인지, 그리고 그 내부 Attribute / Behavior 기입한다.

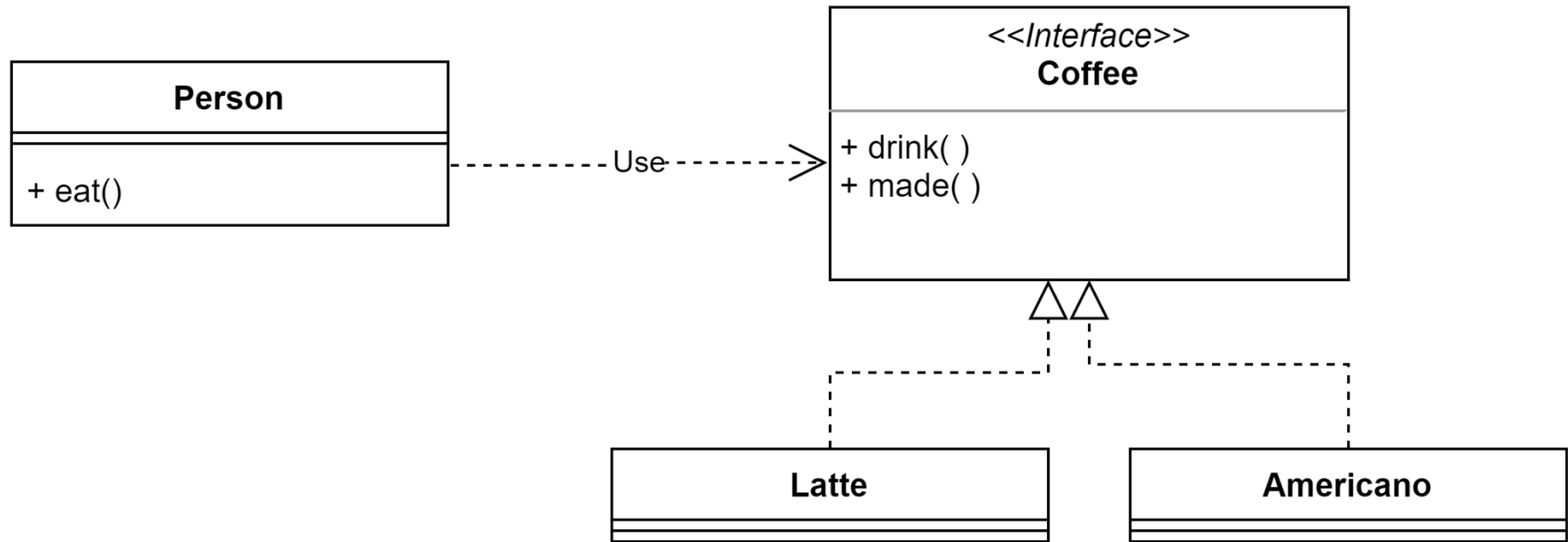
응집도가 높도록

Attribute / Behavior 기입하기



# UML – Class Diagram 예시

분석과 설계에 대한 결과물을 표현하는 공식적인 표현 방법





객체지향 개발 방법이 필요했던 이유 1

# OOP 개요 1



# 절차 지향 개발

## 커피숍 운영 시뮬레이션

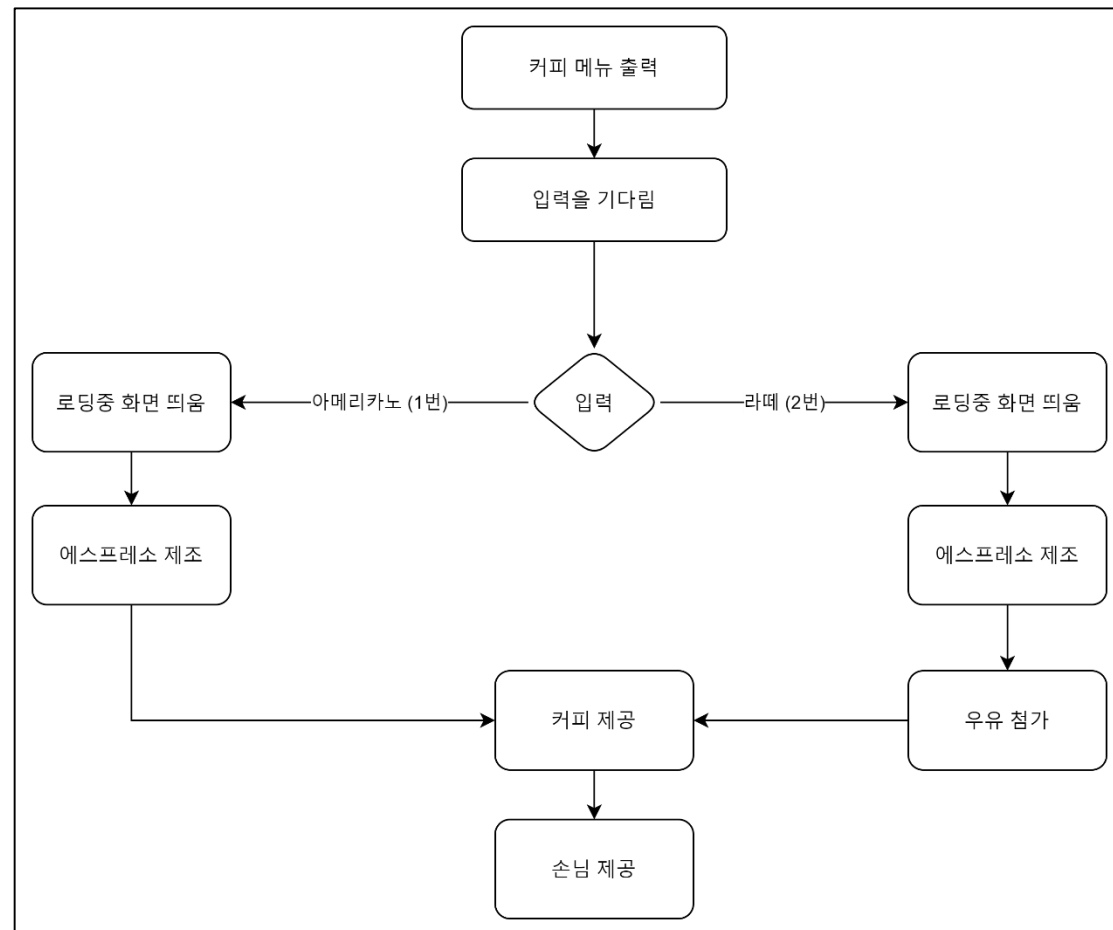
- 절차대로 개발한다.

```
int main()
{
    printMenu();

    int n;
    std::cin >> n;

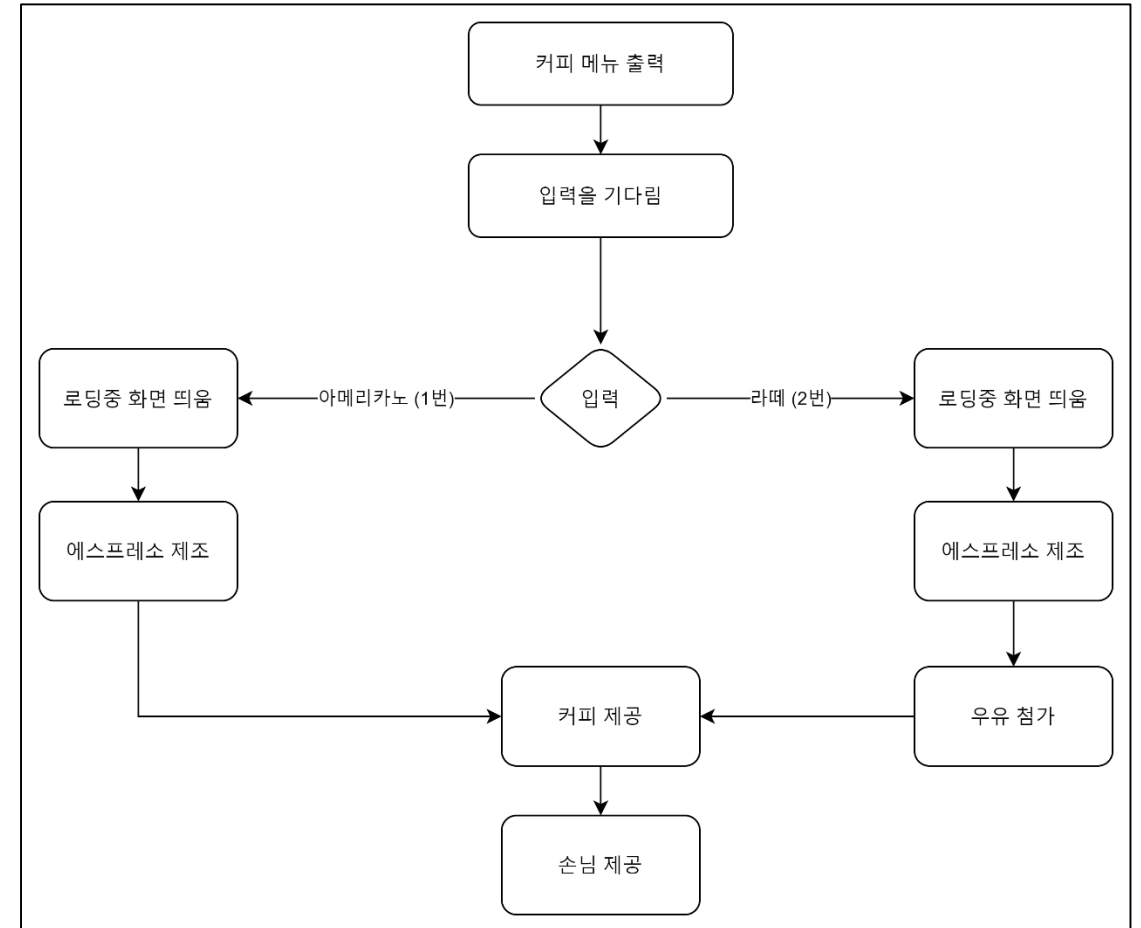
    int cup = 0;
    if (n == 1) cup = makeAmericano();
    if (n == 2) cup = makeLatte();

    std::cout << cup << std::endl;
    return 0;
}
```



# 작은 규모의 개발

혼자 개발하는데 불편함 없음



# 큰 규모의 개발 시

큰 규모의 커피숍을 운영하기 위해서는  
세세하게 신경 써야 할 내용들 매우 많아진다.

→ 독립적으로 수행하는 역할을 나누어  
운영하는 시스템화가 필요하다.



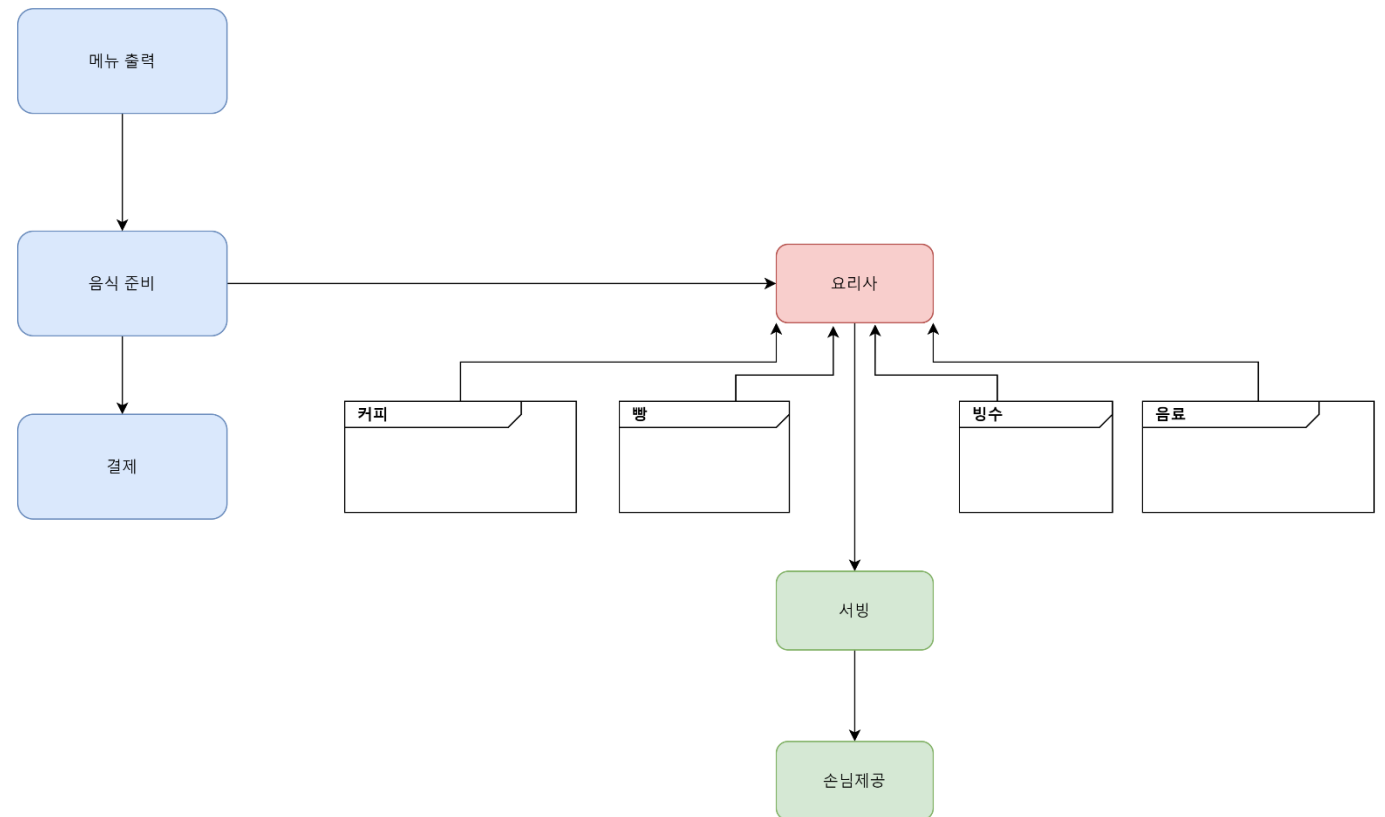
# 3명의 객체가 독립적인 역할을 수행

세 명의 독립적인 역할을 수행

- 담당 내용의 일부가 바뀌더라도,  
다른 사람에게 영향을 받지 않음

음식이 추가되더라도,  
다른 사람에게 영향을 안 끼침

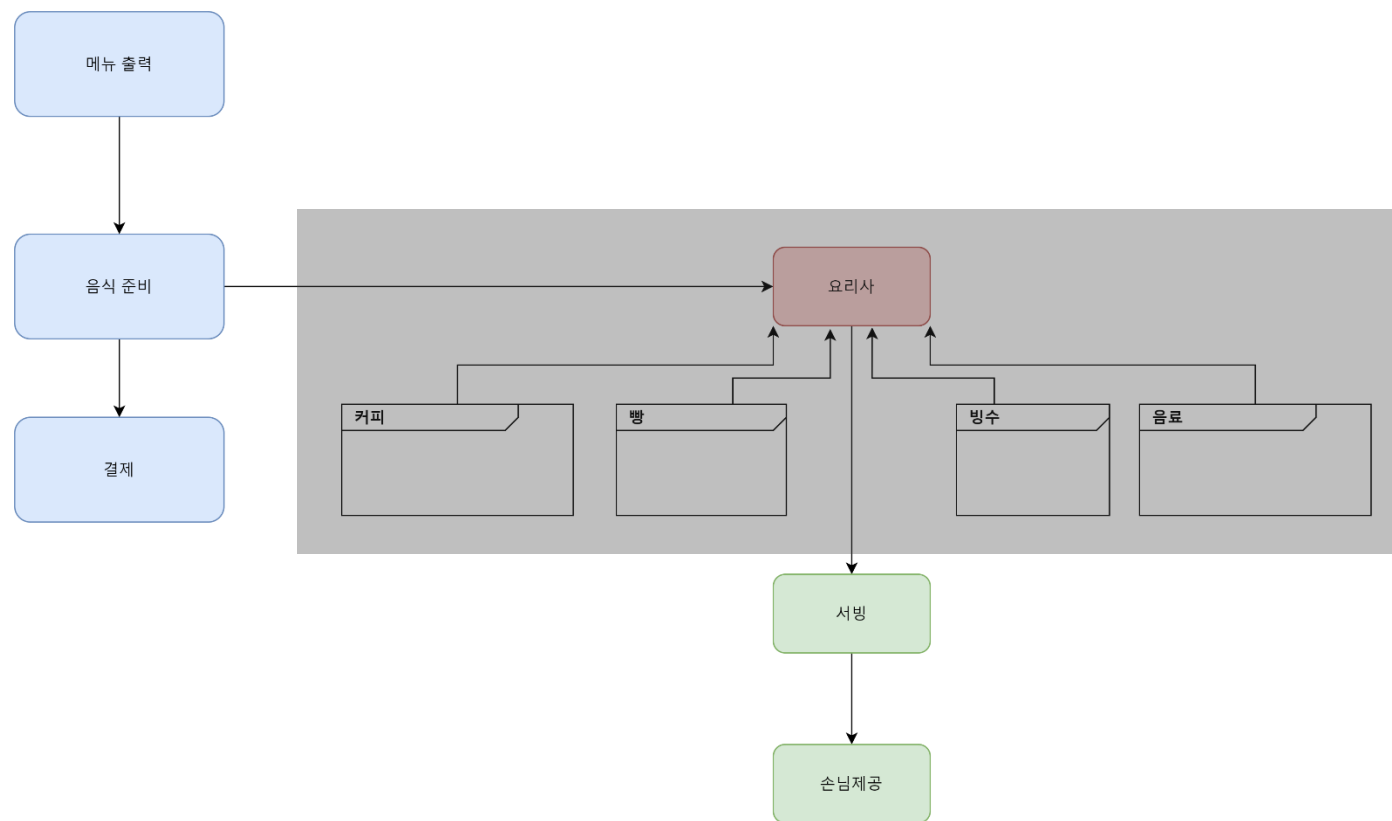
서빙 방법을 바꾸더라도,  
다른 사람에게 영향을 안 끼침



# 재사용성이 좋다.

커피숍이 아닌  
새로운 한정식집을 오픈해도

요리사만 교체하고,  
다른 모듈은 그대로 재사용 가능



# 절차지향개발

## 함수 기반으로 절차적 프로그래밍

- 여러 용도로 쓰이는 함수들이 많아짐으로써,  
소스코드 변경시, 여러 곳에 영향을 끼쳐 유지보수가 어렵다.
- 협업이 어렵다.  
역할 분배 / 기능추가로 인해 Side Effect 논의 필요

객체지향 개발 방법이 필요했던 이유 2

## OOP 개요 2

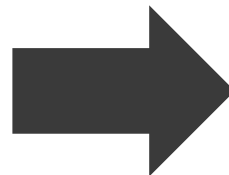




# 재사용성에 대한 이해

스타1 → 스타 2

- 기존 함수를 그대로 사용 불가
- 소스코드 분석 후 수정 필요.



# 클래스의 등장

객체 단위로 구현하기 시작

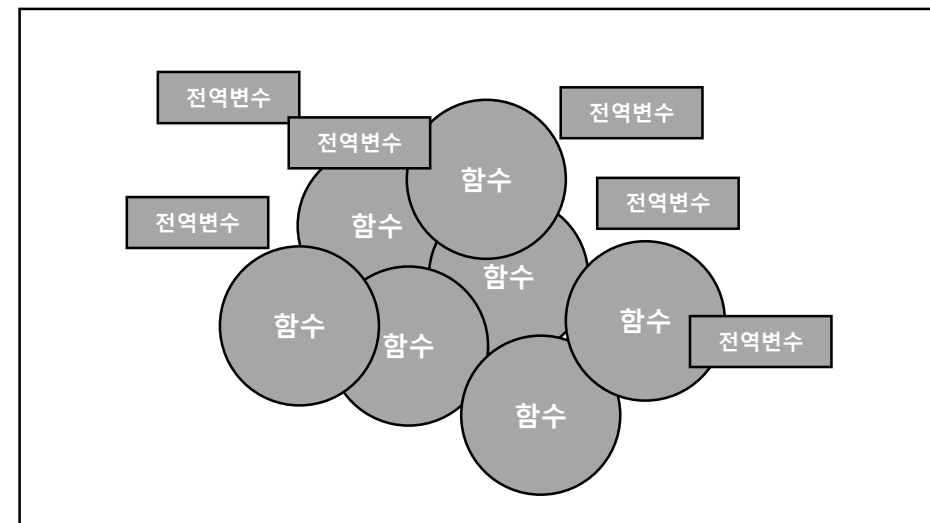
- 클래스로 객체를 구현한다.

C언어

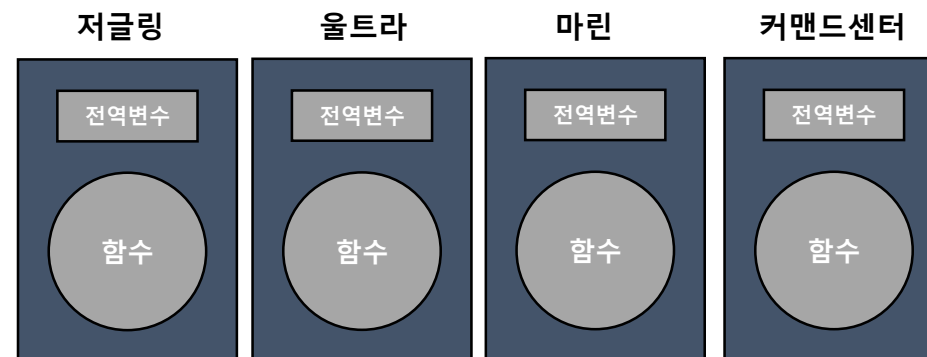
- 전역변수 / 함수  
중심의 개발

C++ / java

- 클래스 (전역변수 + 함수)  
중심의 개발



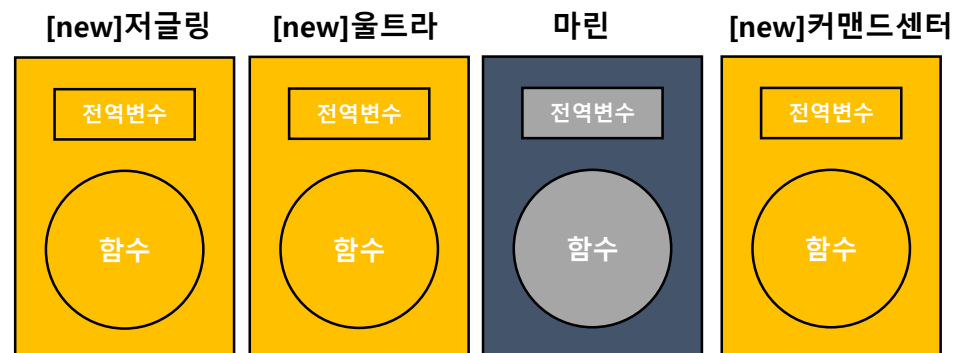
C언어 스타일의 코드, 가시화



객체지향 스타일의 코드, 가시화

# 장점

부속품과 같이 내부에 대한 학습 없이, 그대로 가져다 쓸 수 있음  
→ 재사용성이 좋다.



마린 객체는 기존 코드를 재사용한다.

# 정리. 객체지향의 장점

## 유지보수성이 좋다.

- 연관된 클래스만 코드를 변경하면 된다.  
절차지향에 비해, 유지보수하기 편리하다.

## 재사용성이 좋음

- 만들어 둔 객체를 다른 곳에서도 쓸 수 있음
- 외부에서 만든 객체를 가져다 쓰기 편리하다.

## 협업이 가능하다.

- 절차지향에 비해, 담당 파트를 정하기 편리하다.

클래스를 간단히 구현해본다.

클래스 다루기

# Marin Class 제작

## 용어 이해하기

- 클래스 / 인스턴스 / 필드 / 메서드

```
int main()
{
    Marin* m1 = new Marin();

    std::cout << m1->hp << std::endl;
    m1->run();
    m1->run();
    m1->run();
    m1->run();
    m1->run();

    return 0;
}
```

Main.cpp

```
class Marin {
public :
    int hp = 100;

    void run()
    {
        hp -= 10;
        std::cout << "RUN " << hp << std::endl;
    }
};
```

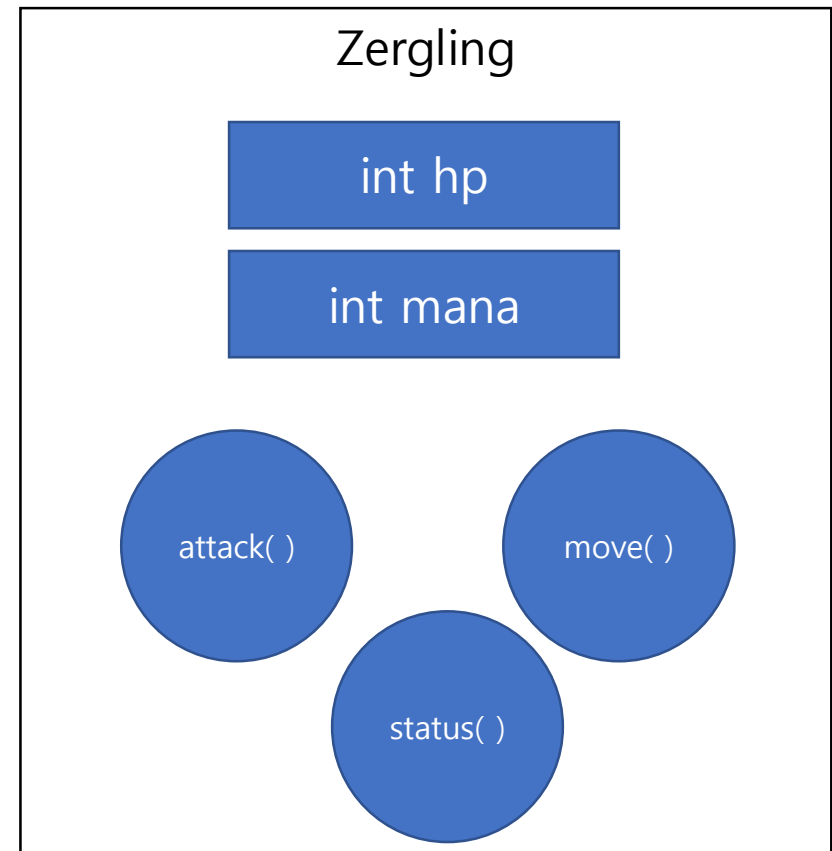
Marin.cpp

# [도전] Class / Instance 생성

## Zergling Class 제작

- hp, mana 기본 값 : 80, 200
- attack 메서드
  - hp가 1 증가
  - mana가 10 감소
- move 메서드
  - hp가 10 감소
  - mana 5 증가
- status 메서드
  - 현재 hp와 mana 값 화면 출력

Instance 2개 생성 후, 동작 테스트 하기



# Server Code와 Client Code

## Server Code

- Client 요청을 받으면, 처리해주는 코드
- Library == Server Code

## Client Code

- Server Code에게 일을 요청하는 코드
- Library 사용자 == Client Code



# [도전] Calculator 제작

## Server Code 구현

- Calculator Class 제작
- plus / minus / divide / multiple 메서드
  - 두 수를 받고, 계산 결과를 result 필드에 저장
- printResult 메서드
  - result 결과를 화면에 출력

## Client Code 구현

- 인스턴스 생성 후, 메서드 호출
- printResult 메서드 호출하여 결과 확인

Calculator
+ result: int
+ plus(int, int): void + minus(int, int): void + divide(int, int): void + multiple(int, int): void + printResult( ): void

Server와 Client Code 관점에서 캡슐화를 이해한다.

캡슐화



# Client들은 Readme를 읽지 않는다.

## 읽지 않는 Readme 내용

- 해당 클래스를 사용할 때, divide에 / 0 이 되지 않도록 한다.
- result 변수를 사용하지 않는다.

Server Code Level 에서 이를 제한해야 한다.

# Calculator Class 개선 1

private로 접근하지 못하도록 막아줌

Calculator
- result: int
+ plus(int, int): void + minus(int, int): void + divide(int, int): void + multiple(int, int): void + printResult( ): void

```
int main()
{
    Calculator* c = new Calculator();

    c->plus(3, 5);
    c->printResult();
    return 0;
}
```

```
class Calculator {
public :
    void plus(int a, int b) {result = a + b;}
    void minus(int a, int b) {result = a - b;}
    void divide(int a, int b) {result = a / b;}
    void multiple(int a, int b) {result = a * b;}
    void printResult() {std::cout << "결과 :" << result
                                << std::endl;}
private :
    int result;
};
```

# Calculator Class 개선 2

Client 가 어떻게 사용하더라도, 버그가 발생하지 않도록, 처리

```
void divide(int a, int b) {  
    result = a / b;  
}
```

```
void divide(int a, int b) {  
    if (b == 0) {  
        std::cout << "ERROR" << std::endl;  
        return;  
    }  
    result = a / b;  
}
```

# 캡슐화

## 캡슐화

- 데이터와 필드를 넣는다.
- 허용하는 데이터 / 필드로만, 데이터 제어 가능
- 허용하지 않는 데이터 / 필드 접근 막음. 은닉한다.



# 캡슐화 장점

- ✓ Server Code가 허용한 방법대로 Client Code를 작성하도록 유도한다.

```
class Robot {  
public :  
    int hp;  
    int x;  
  
    void run() {  
        hp--;  
        x += 3;  
        show();  
    }  
  
private :  
    void show() {  
        std::cout << hp << std::endl;  
    }  
};
```

```
Robot* a = new Robot();
```

```
a->show();  
a->run();  
a->run();  
a->show();
```

# [도전] GameMachine Class 제작

캡슐화

**inputCoin(집어 넣을 코인 수)**

- 코인은 최대 5 개까지 넣을 수 있음
- 입력된 코인이 10 보다 초과될 수 없음

**playGame( )**

- 1 코인 씩 감소됨

집어넣은 코인이 얼마나 되는지  
확인할 수 있어야 함

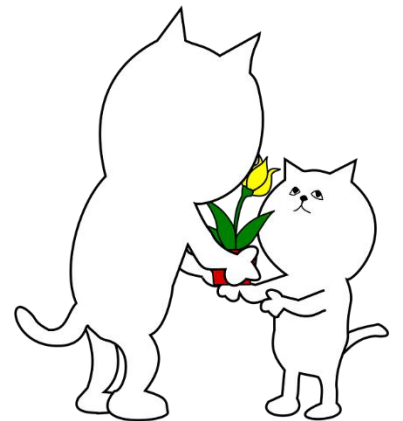
- 메서드 추가 생성 필요

GameMachine
+ totalCoin: int
+ inputCoin(int) : void + playGame( ): void



OOP에서 상속의 의미

상속



# 상속의 개념

## 상속

- 부모가 가진 요소들을  
자식들이 물려받아 사용할 수 있음

## OOP 상속

- 부모 / 자식 관계로 보기 어려움
- 코드 중복 방지를 위해 공통적인 요소를 일반화 시킴

```
class Machine {  
public:  
    std::string ver;  
};  
  
class Robot : public Machine {  
public:  
    void run()  
    {  
        std::cout << "Hi" << std::endl;  
    }  
};  
  
int main()  
{  
    Robot* a = new Robot();  
  
    a->run();  
    a->run();  
    a->ver = "1.99";  
  
    return 0;  
}
```

# [도전] 다음과 같은 코드 제작

3개의 클래스 제작

SpeedRobot
+ hp: int + modelID: int
+ move( ) + stop( ) + run( ) + walk( )

PowerRobot
+ hp: int + mana: int
+ move( ) + stop( ) + attack( ) + jump( )

SmartRobot
+ hp: int + IQ : int
+ move( ) + stop( ) + fly( ) + sitDown( ) + standUp( )

# 중복 코드의 문제점

중복된 메서드 / 필드들이 존재

- 중복 코드가 문제가 되는 이유 :  
변경시 모두 다 한꺼번에 수정 필요 / 버그 유발

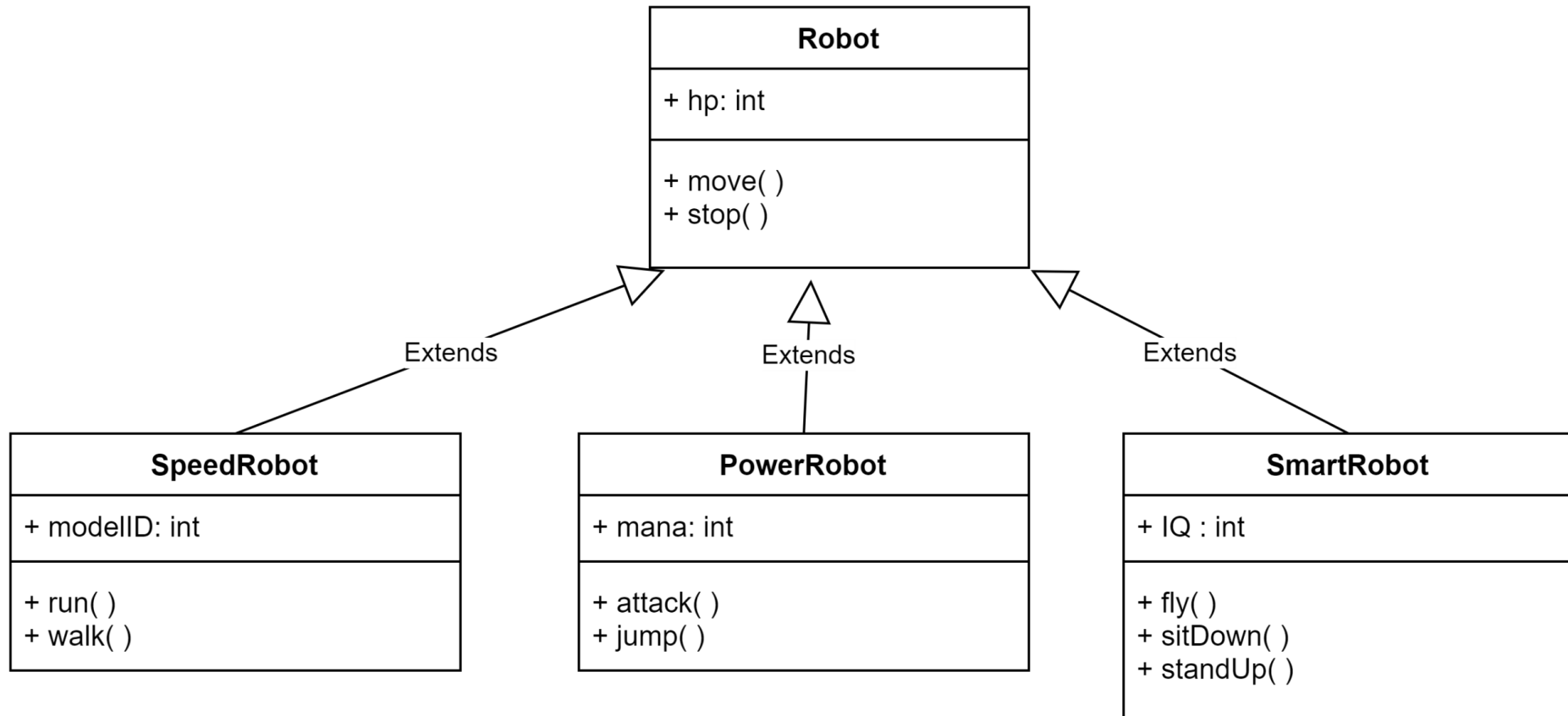
SpeedRobot
+ hp: int + modelID: int
+ move( ) + stop( ) + run( ) + walk( )

PowerRobot
+ hp: int + mana: int
+ move( ) + stop( ) + attack( ) + jump( )

SmartRobot
+ hp: int + IQ : int
+ move( ) + stop( ) + fly( ) + sitDown( ) + standUp( )

# [도전] Generalization (일반화) 시키기

다음과 같은 코드로 수정하기



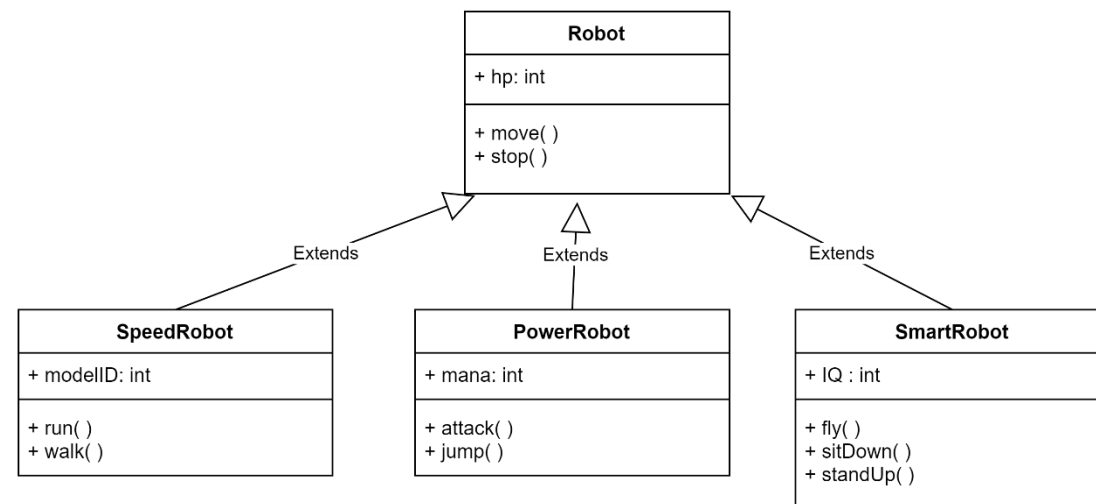
# 용어 정리

예시에 적합한 용어

- Super Class / Sub Class (상위 / 하위)

또는

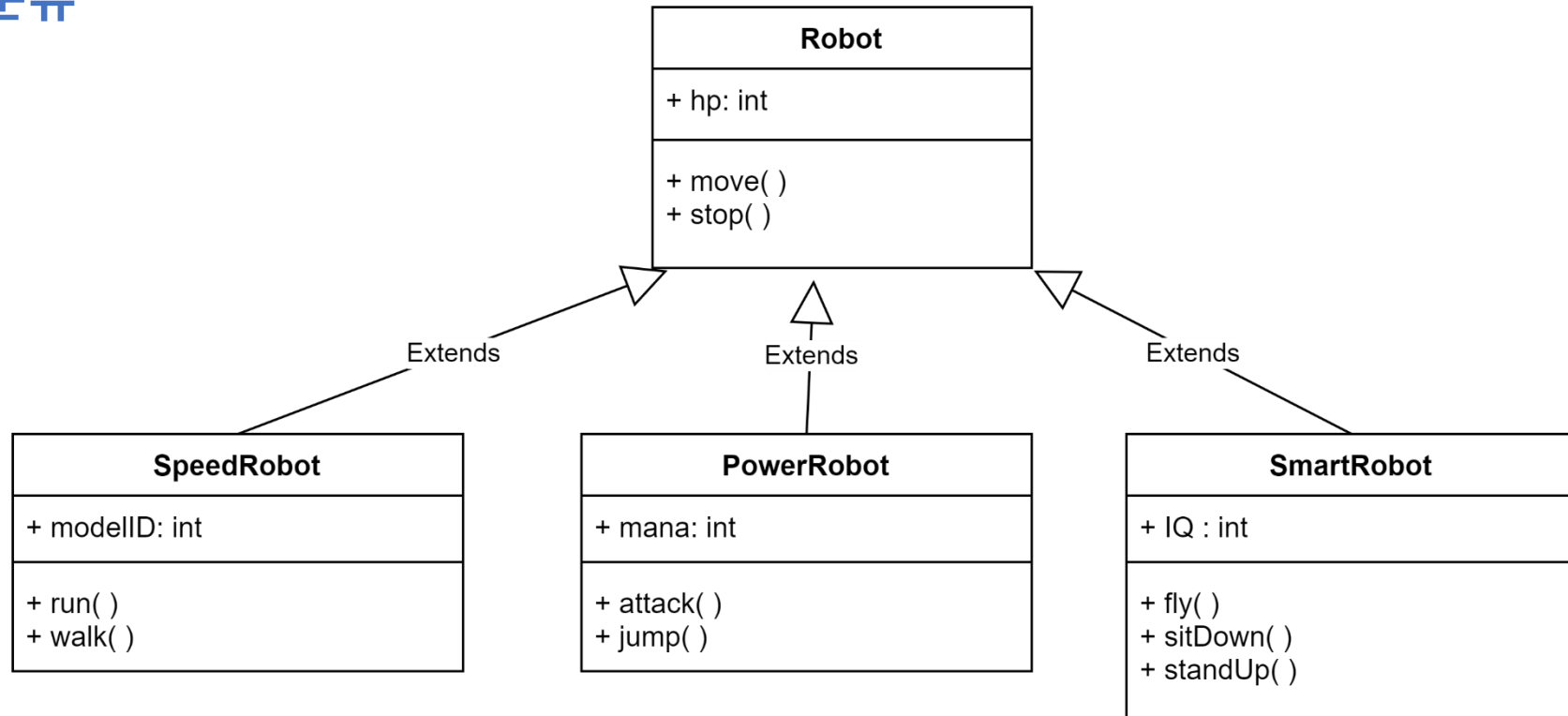
Base Class / Derived Class (기본/ 파생)



부모 클래스 / 자식 클래스라는 용어 보다 위 용어를 사용한다.

# 객체와 분류 관계

- "is a" 관계
- 국내 : "is a kind of" 관계
- 객체와 분류



# Overloading / Overriding

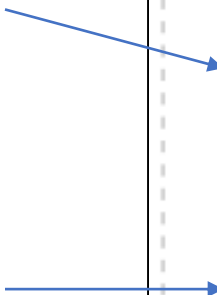
## Overriding

- Super Class 메서드 재정의

## Overloading

- 같은 이름의 메서드이지만,  
다른 Argument로 함수 구분

```
class SpeedRobot : public Robot {  
public:  
    int modelId = 114;  
    void move() {  
        std::cout << "Fast Run"  
            << std::endl;  
    }  
    void move(int i) {  
        std::cout << "MOVE: " << i  
            << std::endl;  
    }  
    void run() { }  
    void walk() { }  
};
```





# [도전] Overriding

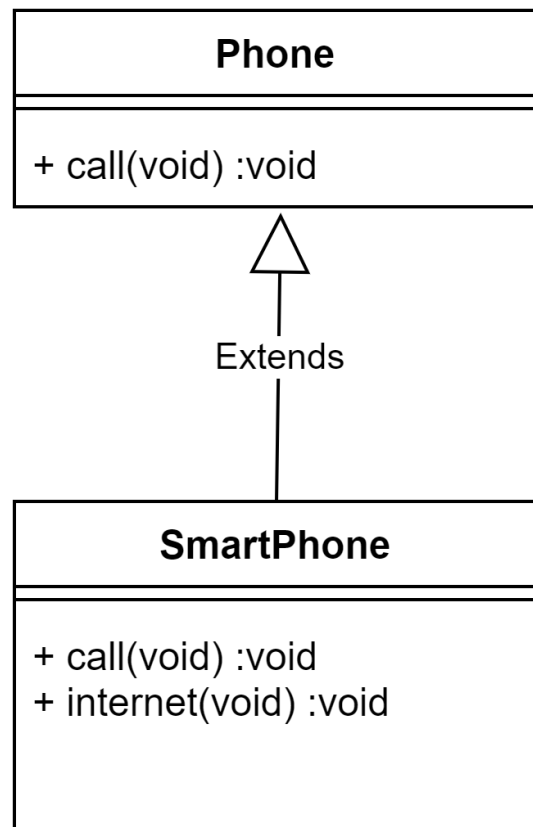
처음부터 구현해보기

**Phone Class의 call 메서드**

- "Calling..." 출력

**SmartPhone Class의 call 메서드**

- "SmartCalling..." 출력



# 어떤 값이 출력될까?

출력 결과를 예상해보자.

```
Phone* a = new Phone();  
Phone* b = new SmartPhone();  
X SmartPhone* c = new Phone();  
SmartPhone* d = new SmartPhone();  
  
a->call();  
b->call();  
c->call();  
d->call();
```

# Client Code에 다형성 구현

Client Code를 다형성을 이용해서 구현

## Rifle Class

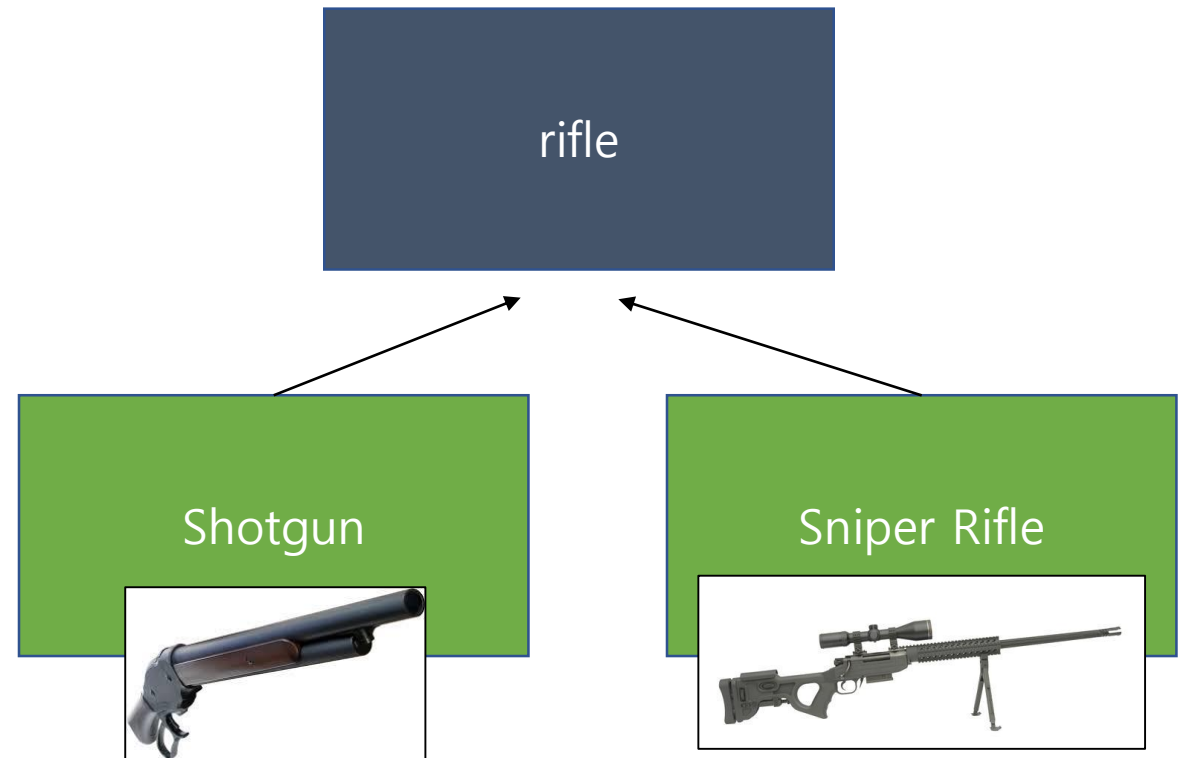
- shot() --> "RIFLE" 출력

## Shotgun Class

- shot() --> "SHOTGUN" 출력

## Sniper Class

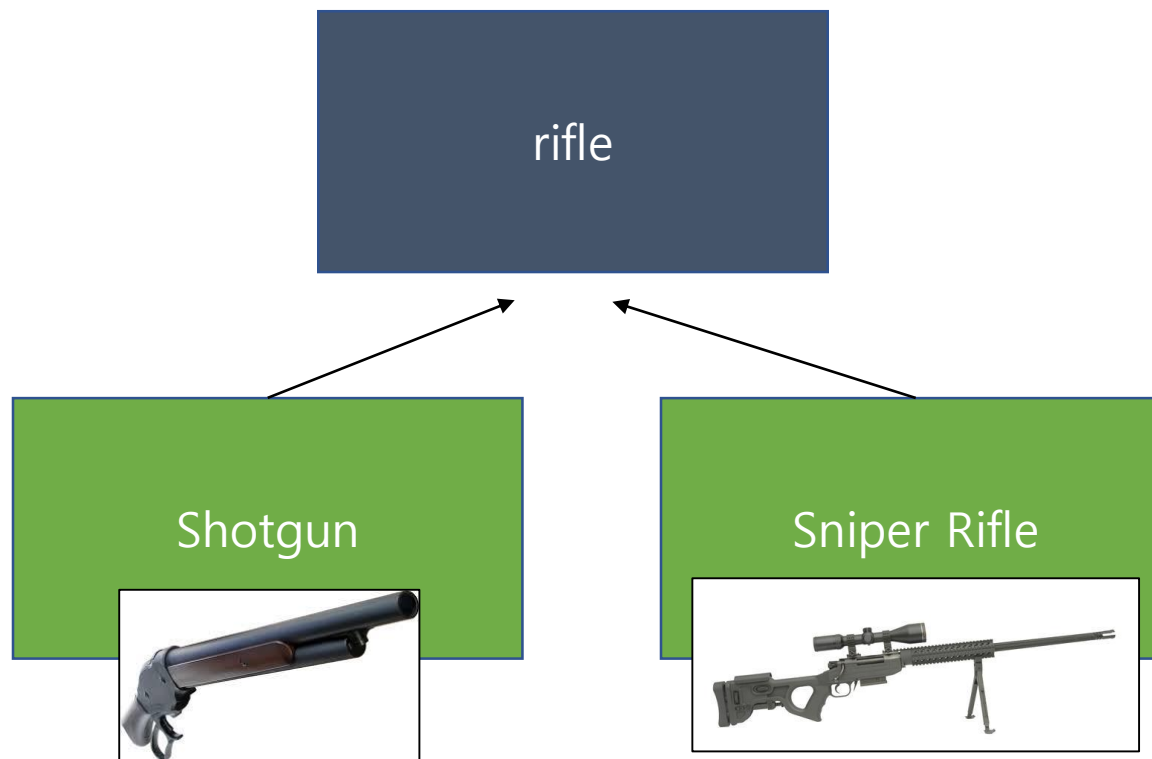
- shot() --> "SNIPER" 출력



# 다형성

한 객체가 다양한 타입을 담을 수 있는 형태

- 상속 관계 구현하여 다형성



# [참고] 구현방법 소개

✓DI

✓Factory

OOP 에서 Interface의 개념잡기

# Interface 개요

# Interface 의미

## 접점

- 두개 시스템 사이의 경계면

# Interface 예시 1

TV 내부를 알지 못해도  
리모컨 Interface만 숙지하면 여러 TV를 제어할 수 있음





# Interface 예시 2

## USB Interface

- 컴퓨터 내부와 외부 장치를 연결할 수 있는 Interface
- Interface 규격에 맞는 장치를 개발하면, 어느 PC 에서도 연결 가능



# S/W Interface

## S/W Interface

- 내부에 접근하기 위한, 공통적인 형태
- 사용자는 Interface만 알고 있으면,  
쉽게 함수 사용 가능하다.

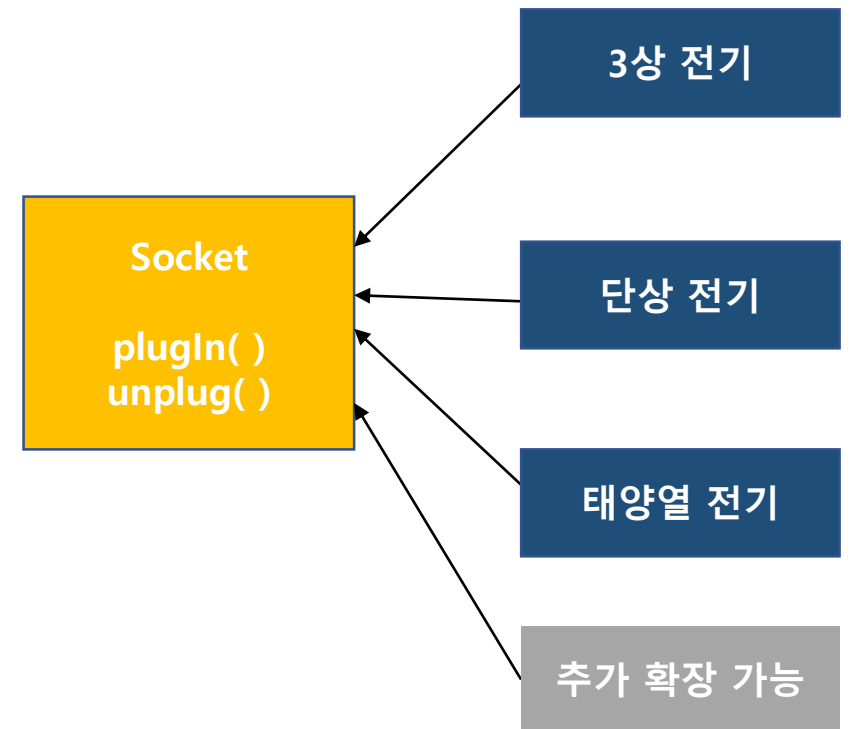


전기를 쉽게 사용하기 위한  
Interface 규격

# S/W Interface

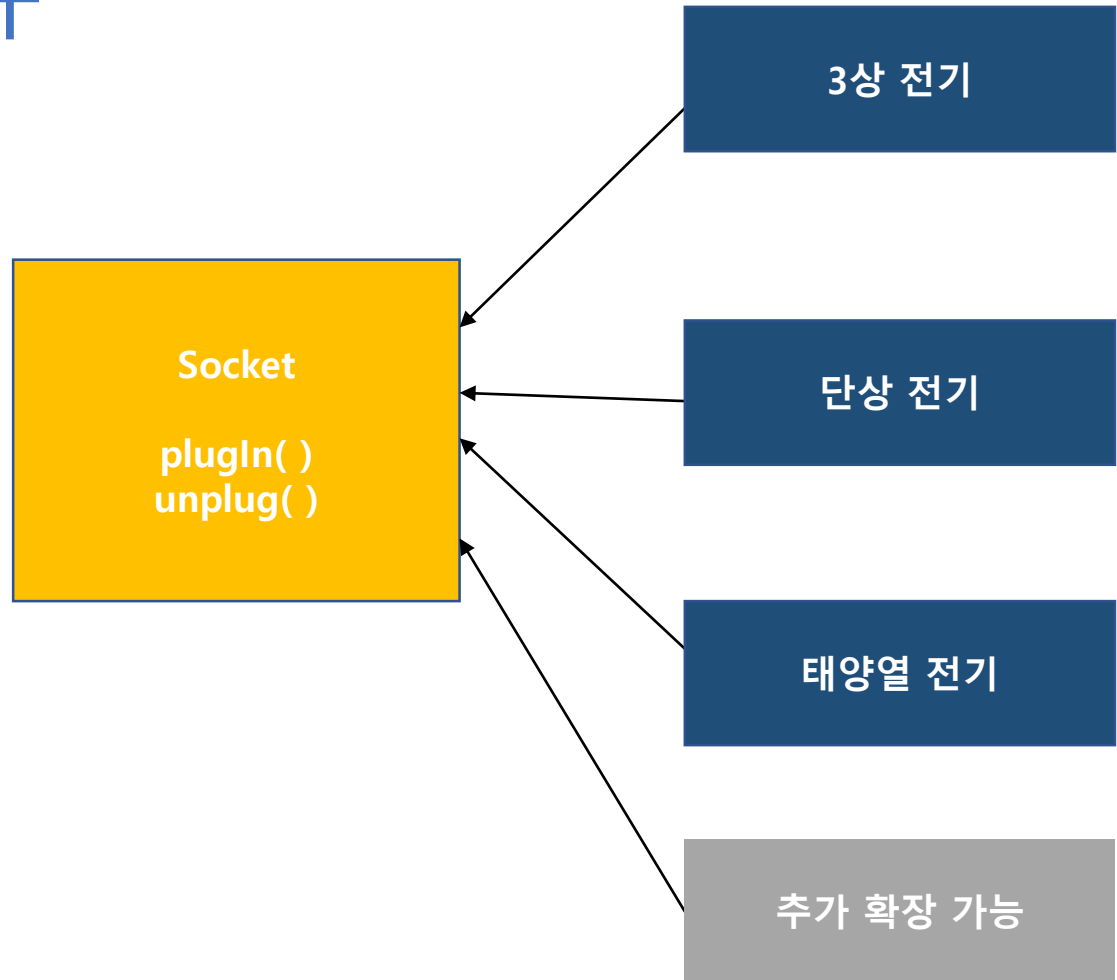
표준 규격을 만들어 놓고,  
표준 규격을 사용하는 Server 코드로 구현해 둬.

어떤 전기를 쓸지는 Client Code에서 선택



# Interface 를 쓰는 이유

1. 언젠가 추가될 유지보수를 위해  
확장 가능한 형태로 만듦
2. 객체를 쉽게 사용하고자  
표준화 시킴



# 확장 가능한 객체 형태를 구현하기 위해

학습할 내용

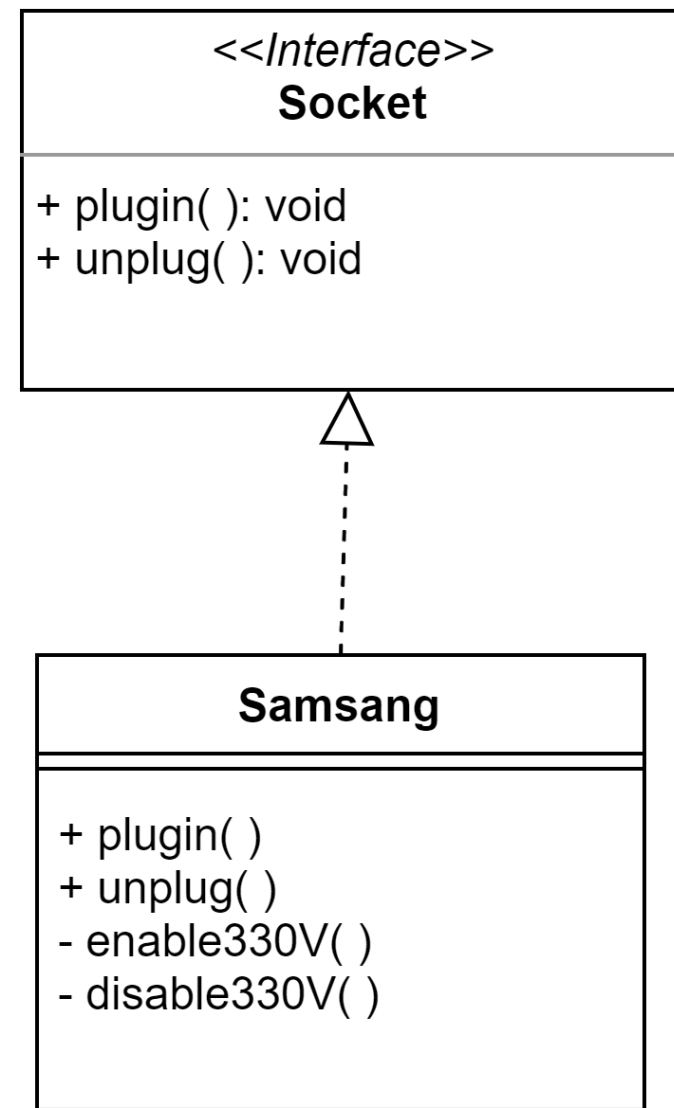
1. Interface
2. 다형성
3. Interface + 다형성 구현

S/W Interface는 Class와 비슷한 형태이다.

Interface 구현

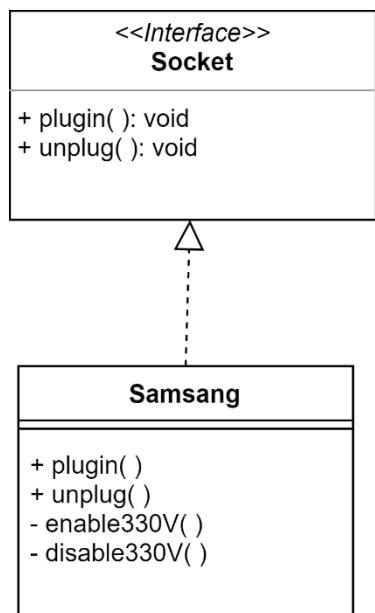
# Socket 구현하기

- ✓ Interface에 명시된 메서드는 반드시 구현해야 함
- ✓ Samsung은 Socket 이라는 Interface 규격을 따른다.
  - 이를, Realization 이라고 한다.



# Implements

Interface를 실체화한 소스코드



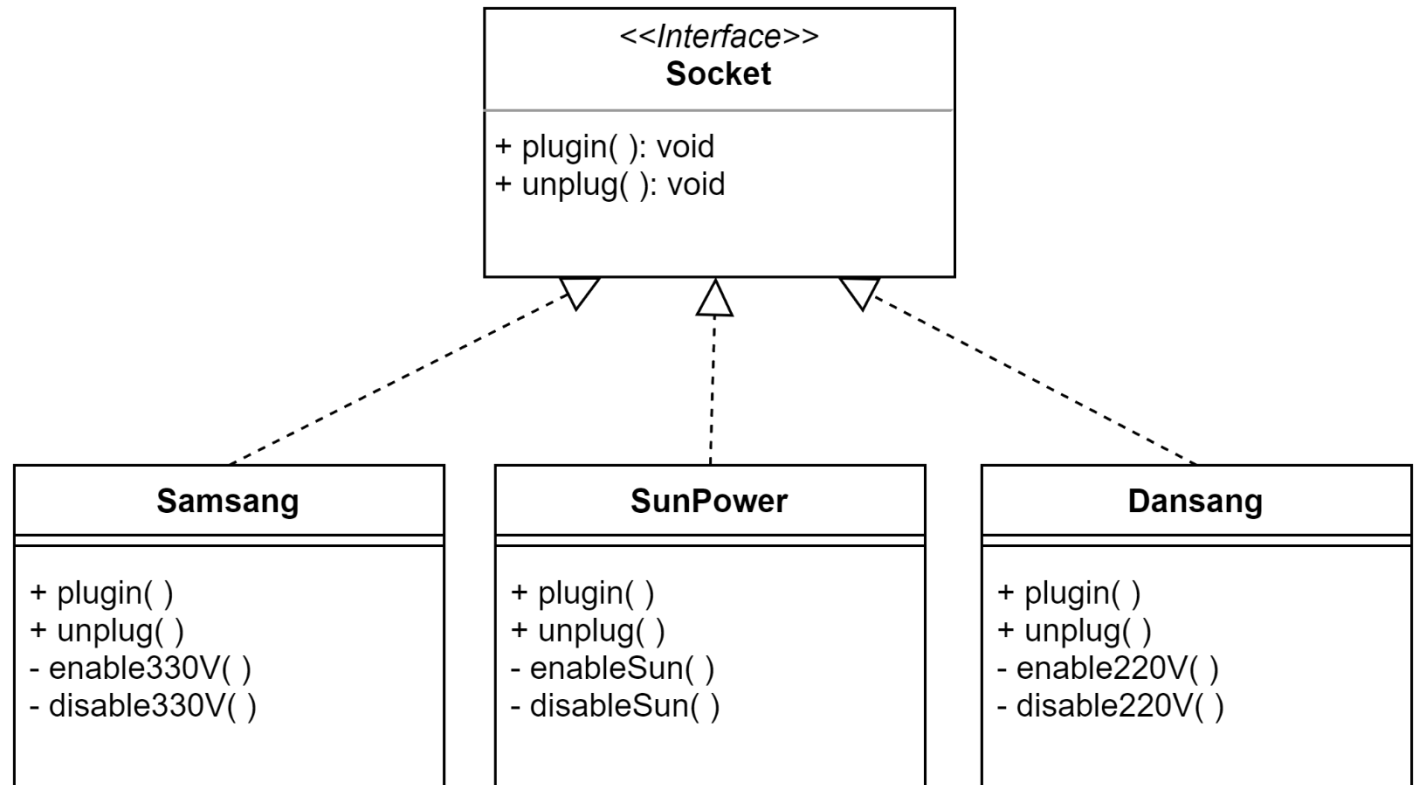
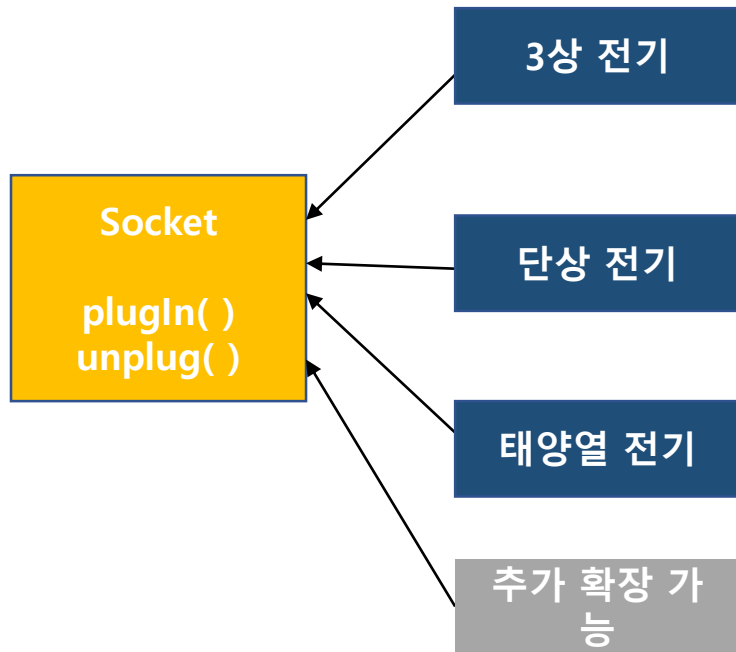
```
interface Socket {
    virtual void plugin() = 0;
    virtual void unplug() = 0;
};
```

```
class Samsang : public Socket {
public:
    // Inherited via Socket
    virtual void plugin() override {
        enable330V();
    }
    virtual void unplug() override {
        disable330V();
    }
private:
    void enable330V() {
        std::cout << "3상 전기 연결" << std::endl;
    }
    void disable330V() {
        std::cout << "3상 전기 해제" << std::endl;
    }
};
```



# [도전] Realization

SunPower / Dansang Class 모두 구현



한 타입으로 다른 타입을 나타낼 수 있다.

다형성 구현

# Client Code 이해

만든 객체들을 모두 생성하여, 메서드 호출

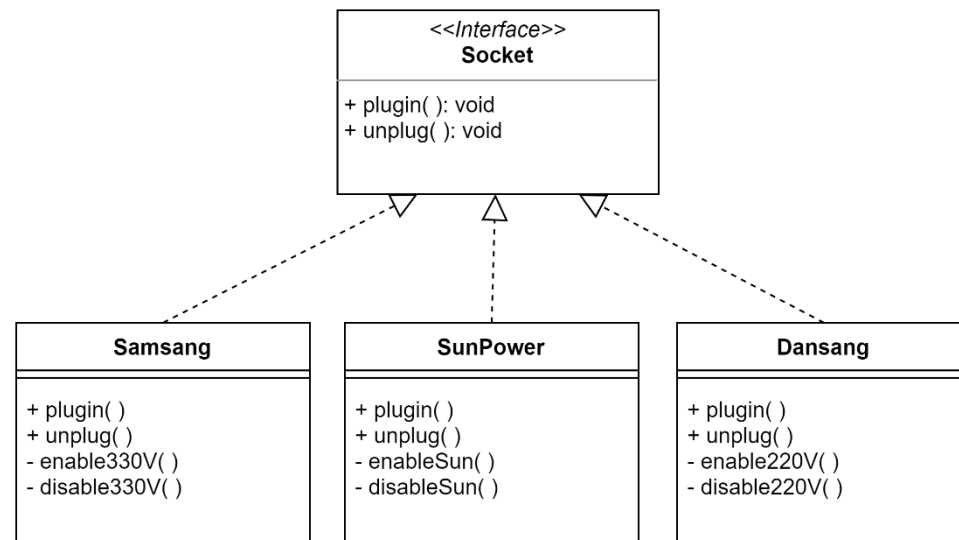
```
Samsang* sam = new Samsang();  
Dansang* dan = new Dansang();  
SunPower* sp = new SunPower();  
  
sam->plugin();  
dan->plugin();  
sp ->plugin();
```

# 다형성

한 객체가 다양한 타입을 담을 수 있는 형태

```
Socket * socket1 = new Samsang();  
Socket * socket2 = new Dansang();  
Socket * socket3 = new SunPower();
```

```
socket1->plugin();  
socket2->plugin();  
socket3->plugin();
```



# Client Server Code 예시

## plug 메서드

- Server Code
- 어떤 전기인지 상관없이, Interface에 맞게 동작 시킴

## main 메서드

- client Code
- 어떤 전기를 쓸 지 결정함

```
void dryMachine(Socket* sock) {
    std::cout << "전원 연결" << std::endl;
    sock->plugin();
    std::cout << "사용중..." << std::endl;
    std::cout << "사용완료" << std::endl;
    sock->unplug();
}

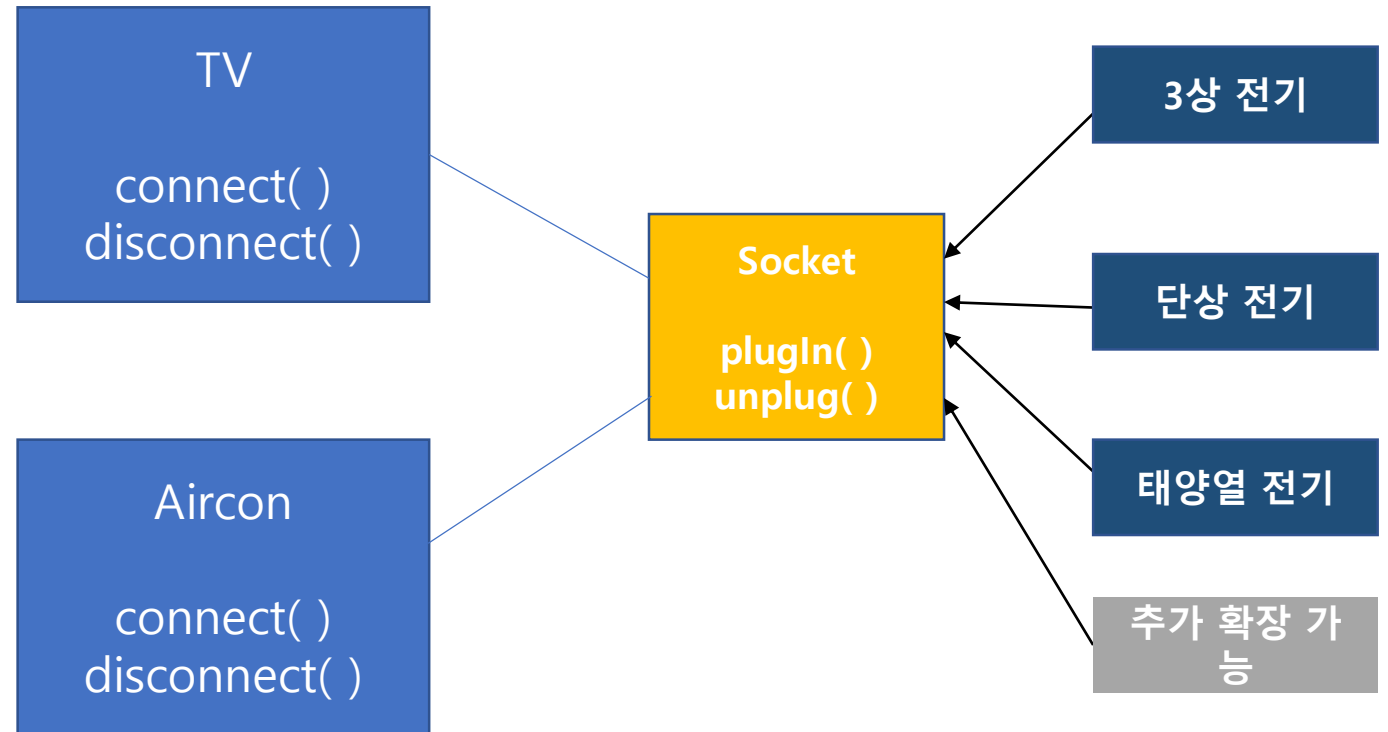
int main()
{
    dryMachine(new Samsang());
    dryMachine(new Dansang());
    dryMachine(new SunPower());

    return 0;
}
```

# [도전] 확장성 고려한 전자제품 구현하기

다양한 방법이 있음, 자유롭게 구현

- Server Code 작성
- Client Code 작성



확장성 고려한 Class 제작 Mission

# [실습] SortPrinter Machine

여러 수를 넣고, Sort를 수행해주는 머신

- 가능한 Sort
  - Bubble Sort
  - Selection Sort
- 어떤 Sort를 할지 선택을 한다.
- Run 버튼을 누르면, Sort가 진행된다.





# [도전] SortPrinter Machine 구현하기

## 확장성을 고려한 머신 구현하기

### SortPrinter 기능

- insert 기능
  - SortPrinter 에 int 값들을 넣는다.
- selection 기능
  - 어떤 sort를 할지 선택한다.
- run 기능
  - 선택된 알고리즘이 수행된다.
- show 기능
  - 정렬된 결과가 출력된다.

