# SOLID

CONTENTS

# 목차

# OOP 주요 내용

SOLID 이해를 위한 OOP 주요 내용

# 객체지향프로그래밍

## Object-Oriented Programming이란?

객체들로 복잡한 문제 -> 작은 문제로 분해

- 시스템의 복잡도를 줄이기 위한 분해(Decomposition)

작은 문제를 해결하기 위한 객체를 구성한다.
각 객체가 고유한 기능을 제공하고, 이러한 객체와 객체의 관계를 이용하여 프로그램을 완성한다.
이를 이용하여 객체의 재사용, 프로그램의 관리와 확장이 편리하게 된다.

## 좋은 소프트웨어 설계의 시작

모듈화

- 소프트웨어를 각 기능별로 분할, 설계 및 구현하는 기법
- 모듈화를 수행하면 복잡도가 감소되고, 변경과 구현이 용이하며 성능을 향상시킨다.
- 모듈간의 기능적 독립성을 보장한다.

결합도

- 모듈간의 상호 의존하는 정도, 연관관계.

응집도

- 하나의 모듈 안의 요소들이 서로 관련된 정도.

# 객체지향 4가지 기본원리

- **추상화(Abstraction)**

  중요하지 않는 자세한 사항은 감추고, 중요하고 필수적인 사항만 다룸으로써 복잡함을 관리할 수 있게 하는 개념

  중요여부의 판단은 업무나 관심사항에 따라 다르게 나타난다.

- **캡슐화(Encapsulation)**

  구현방법에 대한 자세한 사항은 블랙박스화 하여 드러내지 않고 외부로 노출된 인터페이스를 통해서만 사용할 수 있게 하는 개념

  인터페이스를 변경시키지 않는 한 사용자는 구현의 변경에 영향을 받지 않고 사용할 수 있고, 개발자는 내부 구조나 구현방법을 자유롭게 변경할 수 있다.

- **상속**

  클래스간의 관계를 계층구조화 하여 구체화와 일반화함.

  구체화 될수록 고유특징이 늘어나고, 일반화 될수록 더 많은 객체에 영향을 준다.

- **다형성**

  하나의 속성이나 행위가 여러 형태로 존재하는 것.

  Overriding 과 Overloading

# 객체지향 다섯가지 설계원칙 : SOLID 란?

- SOLID는 로버트 C. 마틴이 객체 지향 프로그래밍 및 설계의 다섯가지 기본원칙으로 제시한 것을 마이클 패더스가 알파벳 첫글자를 따서 소개한 것이다.

| | |
|---|---|
| **S**ingle Responsibility Principle (SRP) | 하나의 클래스는 하나의 책임만 가져야 한다. |
| **O**pen/Closed Principle (OCP) | 클래스는 확장에 대하여 열려 있어야 하고, 변경에 대해서는 닫혀 있어야 한다. |
| **L**iskov Substitution Principle (LSP) | 기반 클래스의 메소드는 파생 클래스 객체의 상세를 알지 않고서도 사용될 수 있어야 한다. |
| **I**nterface Segregation Principle (ISP) | 클라이언트가 사용하지 않는 메소드에 의존하지 않아야 한다. |
| **D**ependency Inversion Principle (DIP) | 추상화된 것은 구체적인 것에 의존하면 안 된다. (자주 변경되는 구체적인 것에 의존하지 말고 추상화된 것을 참조) |

# Github 에서 실습 소스코드 준비

## 소스코드 링크

https://github.com/mincoding1/SOLID

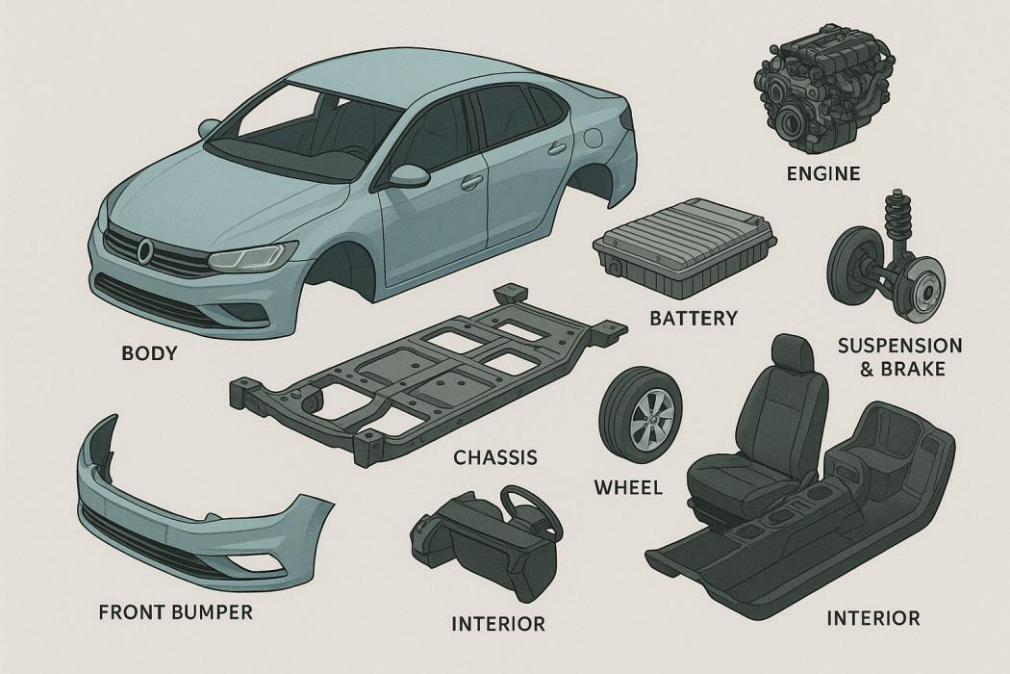
- Kata 출처 1 (Vehicle 컨셉) : https://github.com/bsferreira/solid
- Kata 출처 2 : https://github.com/mikeknep/SOLID

# SRP

Single Responsibility Principle, 단일 책임 원칙

# 시스템에서 모듈의 책임

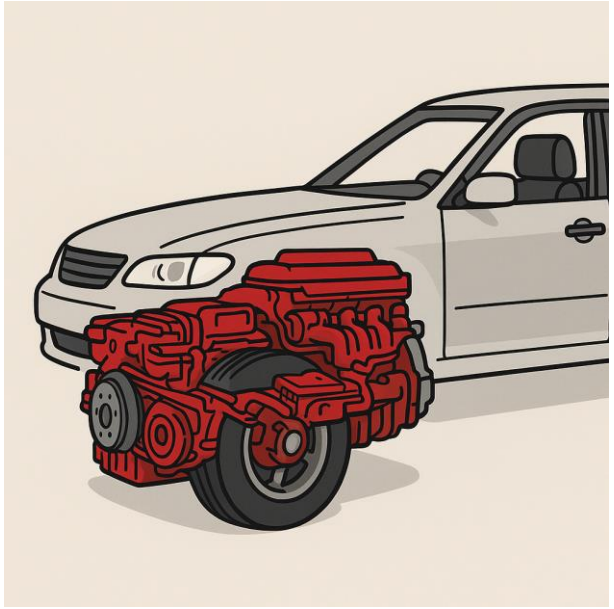
시스템은 **각자의 책임을 수행하는 모듈**로 구성되어 있다.




엔진은 동력을 생산하는 책임을 지닌다.
바퀴는 자동차의 실제 움직임을 발생시키는 책임을 지닌다.

# 책임이 섞여 있는 모듈

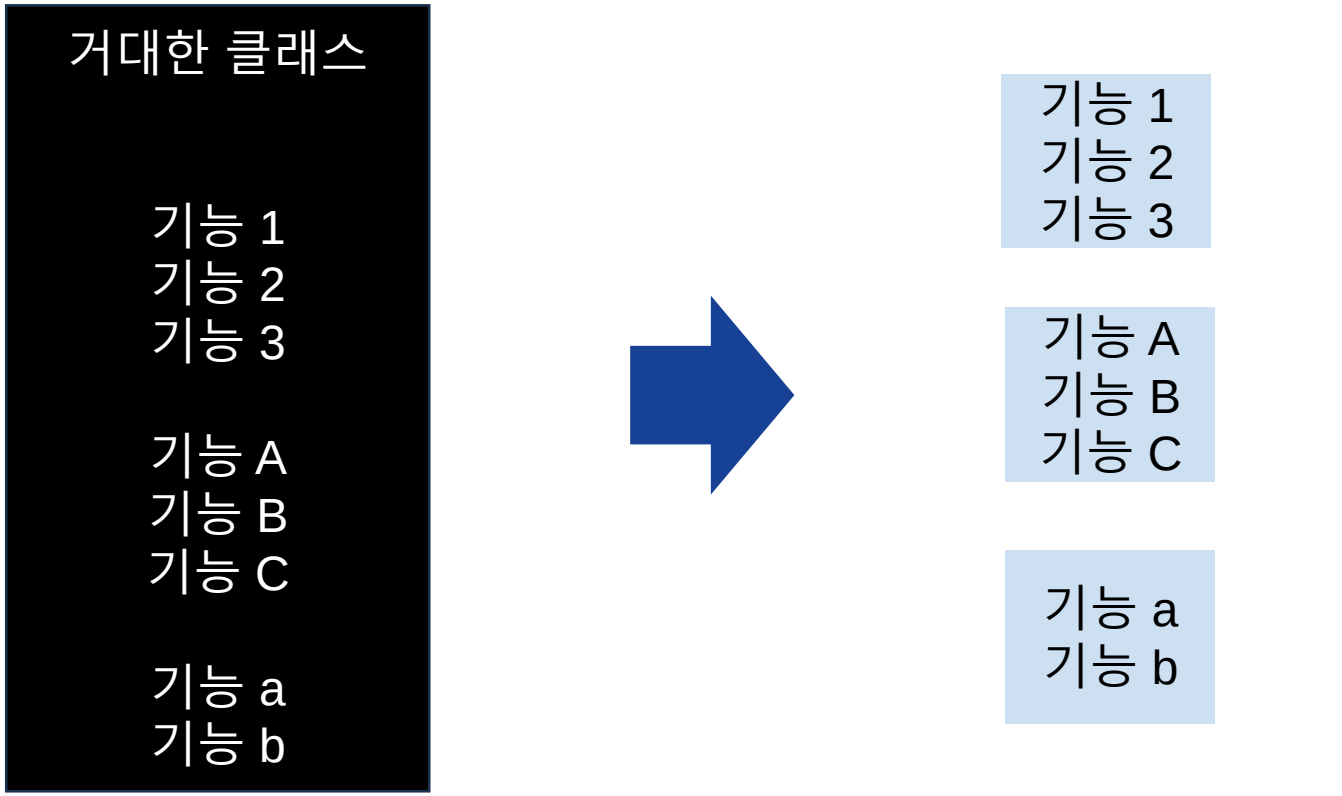
- 서로 다른 책임이 하나의 모듈에 섞여 있다면 유지보수에 문제점이 발생할 수 있다.
- 유지보수를 위해 **하나의 책임을 수행하는 모듈**로 분리를 해줘야 한다.



만약 엔진과 바퀴가 **하나의 모듈**로 되어있다면 어떤 문제가 생길 수 있을까?

# SRP

## 거대한 클래스, 많은 일을 하고 있는 클래스를 분리한다




많은 기능들이 뒤엉켜서 한 곳에서 유지보수

독립적으로 유지보수

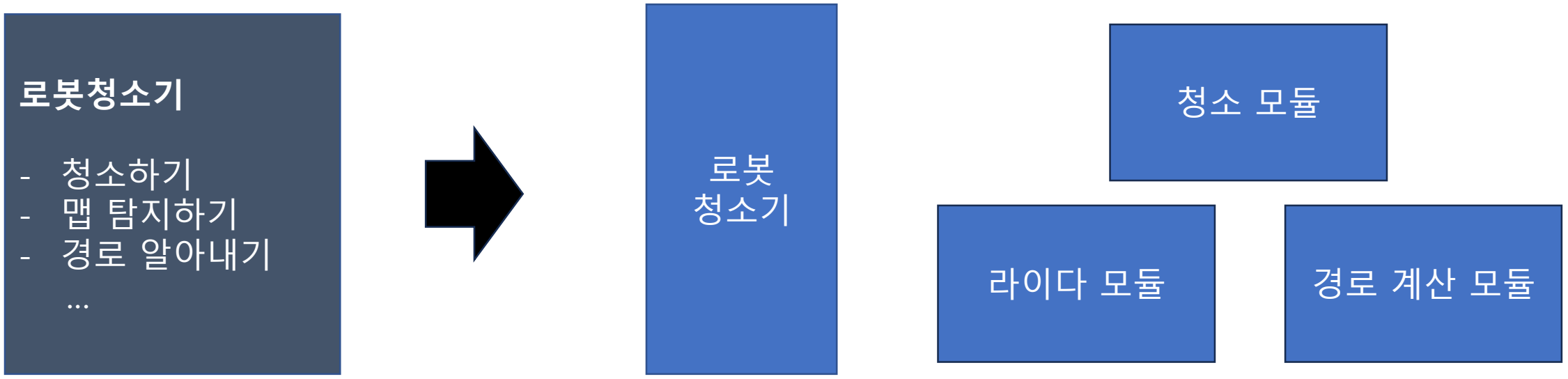
# 각자 변경되는 독립된 위치로 옮기기

로봇 청소기의 책임을 "청소 / 맵 탐지 / 경로 계산" 으로 구분 할 수 있다.

책임에 따라 변경을 다음과 같이 분류할 수 있게 된다.

- 청소에 관련된 요구사항 변경
- 맵 탐지에 관련된 요구사항 변경
- 경로 계산에 관련된 요구사항 변경

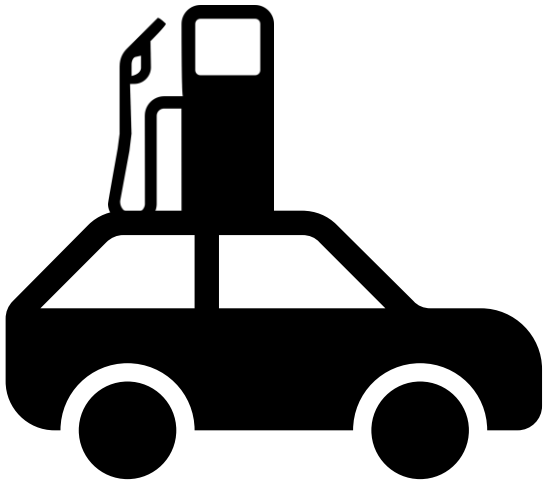
다른 책임과 관련된 변경에 영향 받지 않도록 코드를 분리해준다.



# [도전] Vehicle 분리하기

## Vehicle 클래스에서 Vehicle 의 책임이 아닌 부분을 분리한다.

현재 Vehicle 클래스에 refuel 의 구체적인 로직이 들어가 있다 (주석으로 표기 되어 있음)
어떤 변경이 발생할 수 있을지 생각해보고 이에 맞게 SRP 를 적용해보자

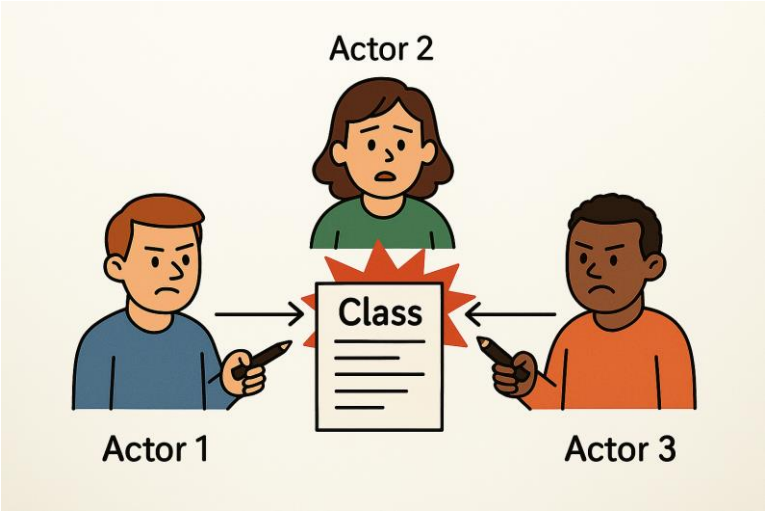


- 연료리필( )
- MAX량 확인( )
- 남은 연료 확인( )
- 연료 채우기( )
- 가속( )

# [정리] SRP 정리

- **Single Responsibility Principle (SRP)**
  모든 클래스는 하나의 책임만 가지며, 클래스는 그 책임을 완전히 캡슐화해야 한다.
  클래스가 제공하는 모든 기능은 이 책임과 부합해야 한다.

- **로버트 C 마틴의 책임**
  로버트 C 마틴은 **하나의 책임**을 **하나의 변경하는 이유**라고 언급했다.




변경 방향이 서로 다르면 충돌이 일어날 수 있다.
변경하는 이유에 따라 클래스를 분리해야 한다.

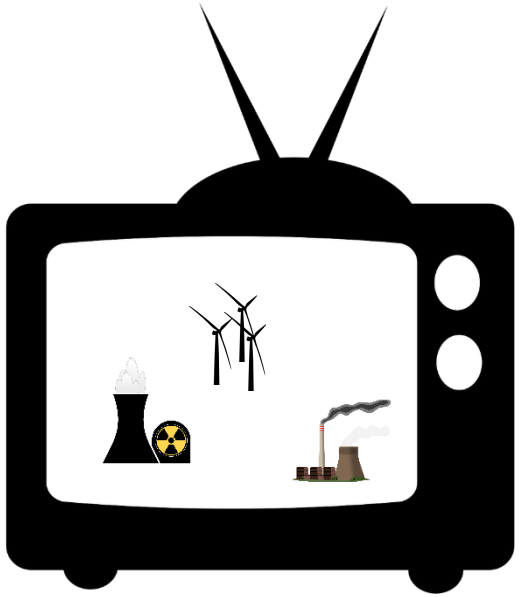
# OCP

## Open Closed Principle , 개방-폐쇄 원칙


열린마음 닫힌마음

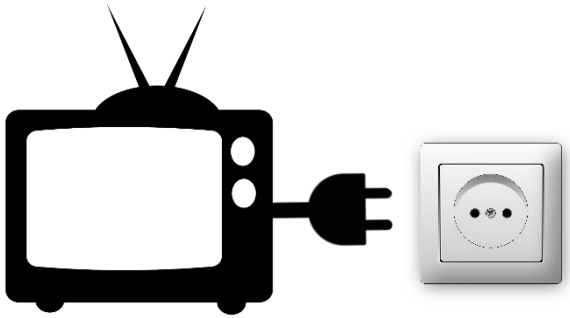
# 변경이 많은 곳을 분리

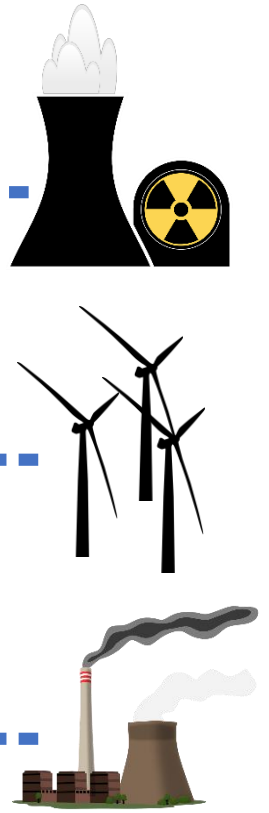
OCP는 변경이 많은 곳을 캡슐화를 통해 분리하여 새로운 기능을 추가하더라도 변경이 없는 구조로 만든다.



다양한 전기공급 이용
(TV 내부에 구현)

캡슐화





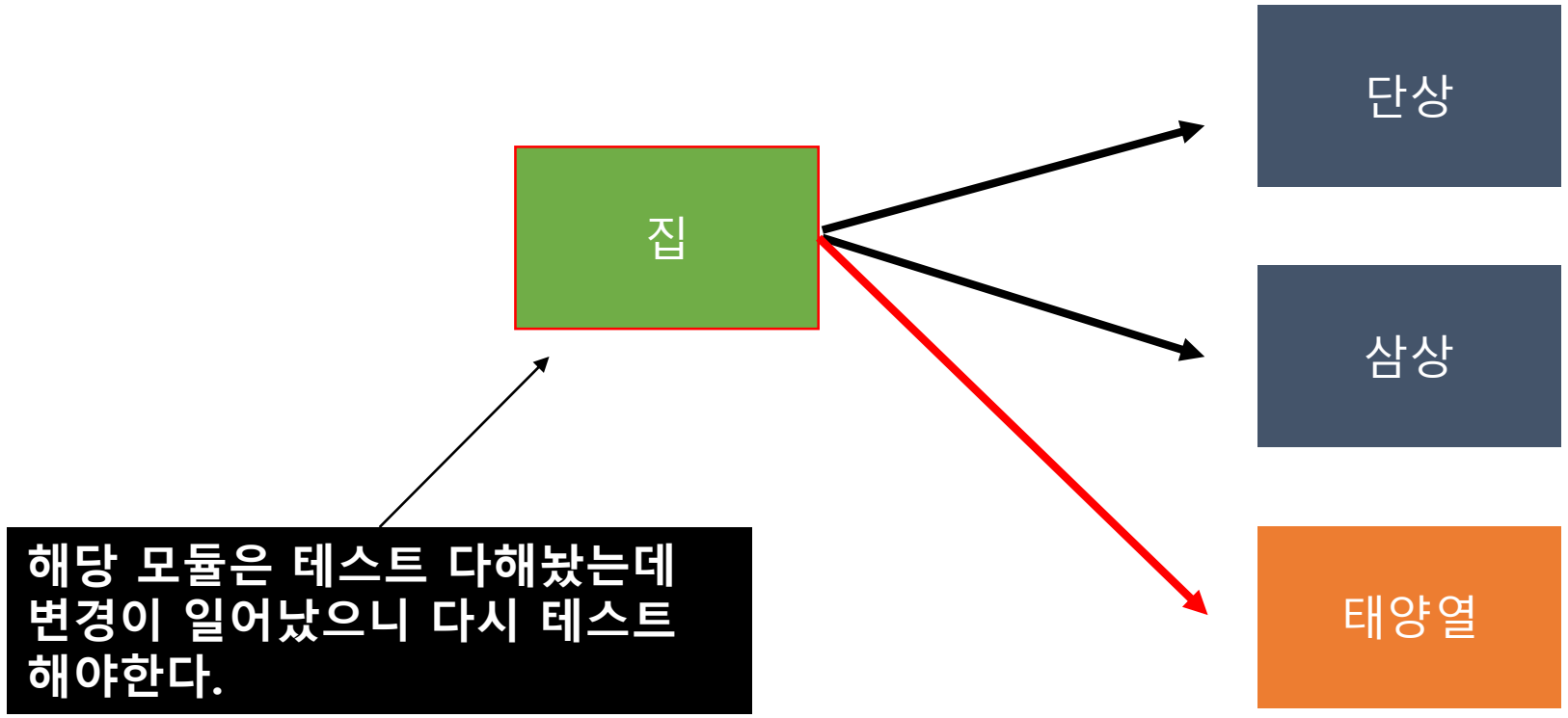
다른 클래스로 캡슐화

추가 확장

# 한 클래스가 다른 여러 클래스 의존하는 경우

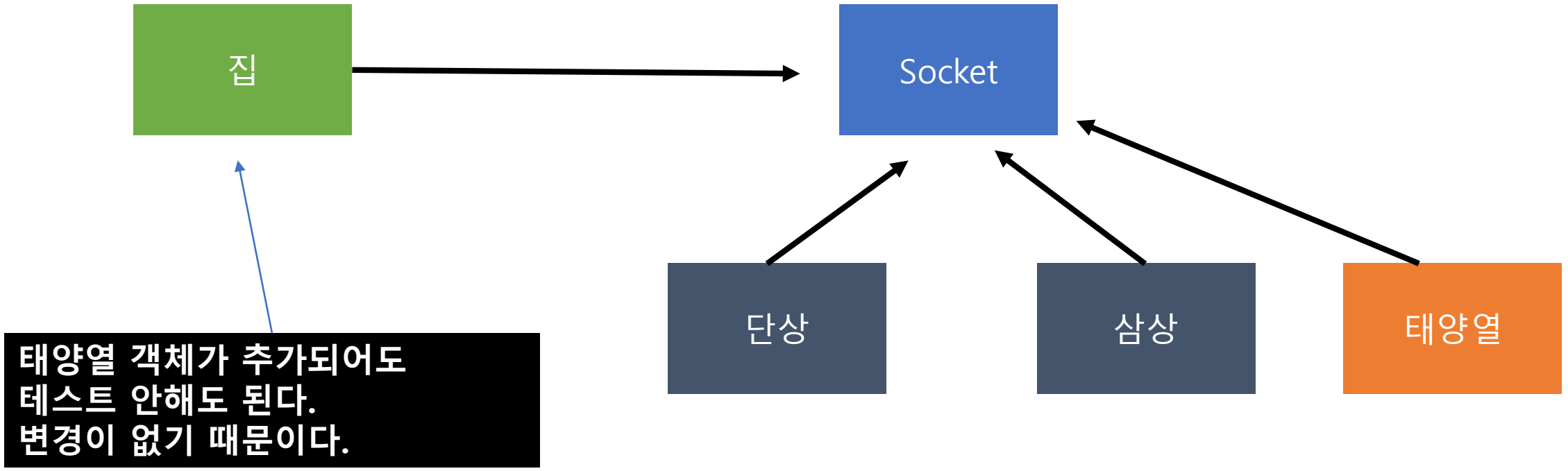
기능이 하나 더 추가되는 경우, 집 Class에 변경이 일어난다.

태양열 클래스 하나 더 추가시, '집' 객체는 변경이 일어남



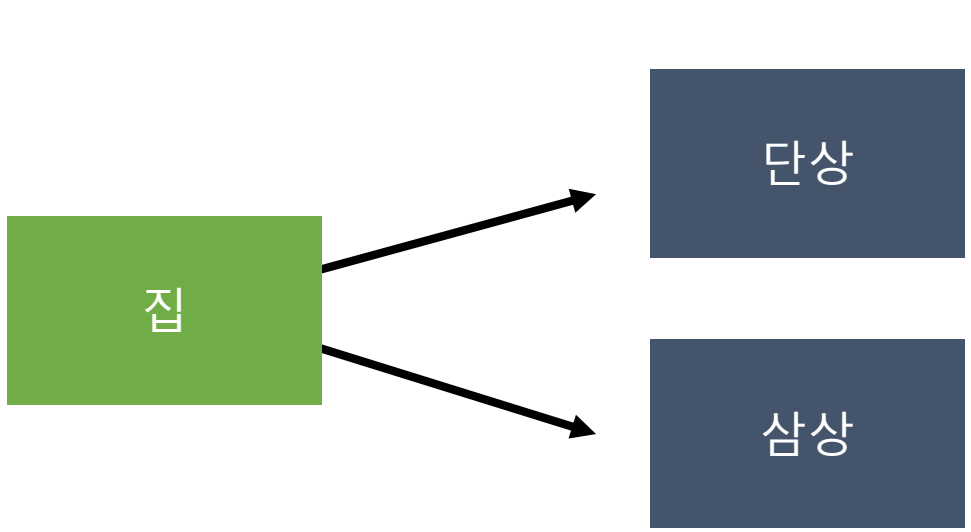
# Interface를 추가한다.

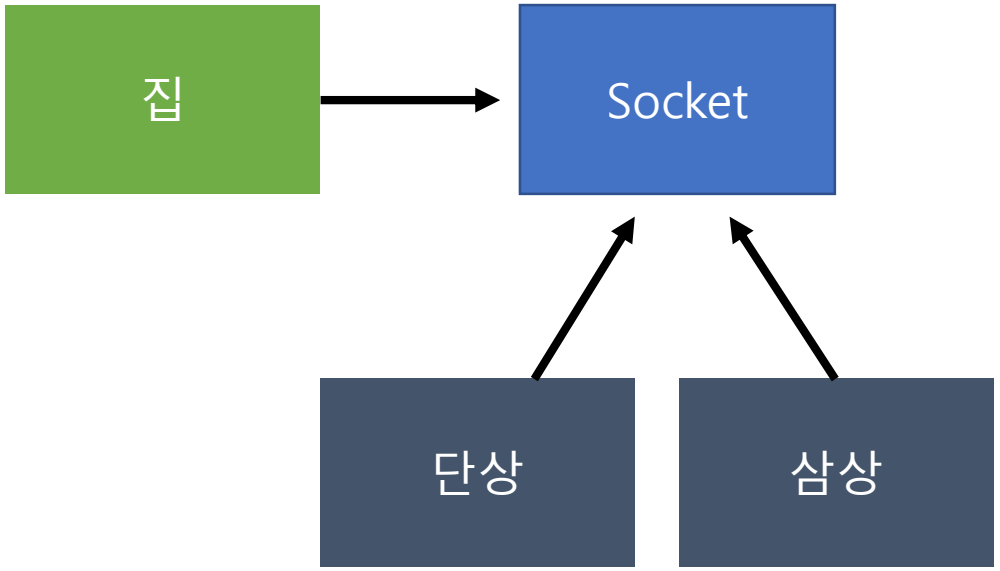
- Interface를 추가하면, 집 Class는 변경이 없다.
  태양열이 하나 더 추가되더라도, 집 Class는 변경이 없다.



# [도전] 직접 구현해보기

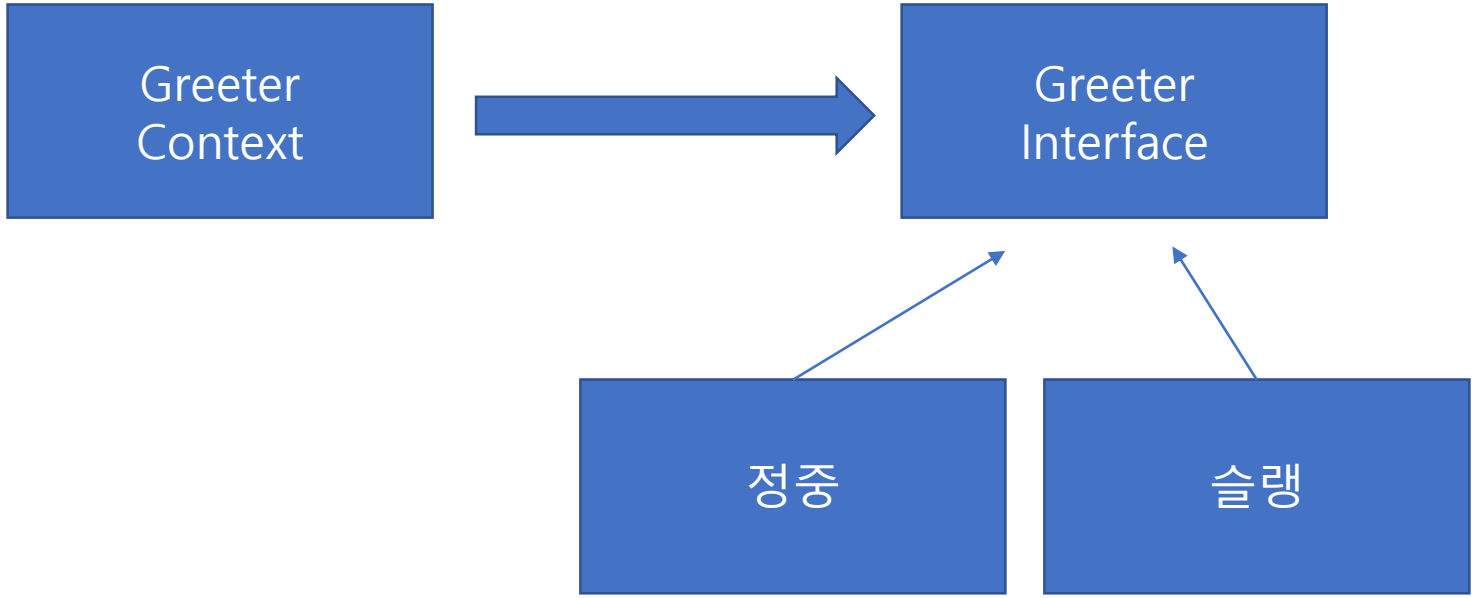
다음과 같이 Server Code를 구현하고,
이를 Test할 수 있는 Client Code도 구현해본다.



# [도전] step1 : Greeter

새로운 인사 방식이 확장된다고 가정하고, 이를 OCP 설계에 맞도록 개선한다.



# [도전] step2 : EventHandler

## 직접 해결해보자.

자동차는 Sport 모드 / Comport 모드 등이 존재

- 서스펜션 높이와 Power가 모드마다 달라짐

EventHandler 모듈의 변경을 최소화해보자.
그리고 Economy Driving Mode를 추가한다.

# LSP

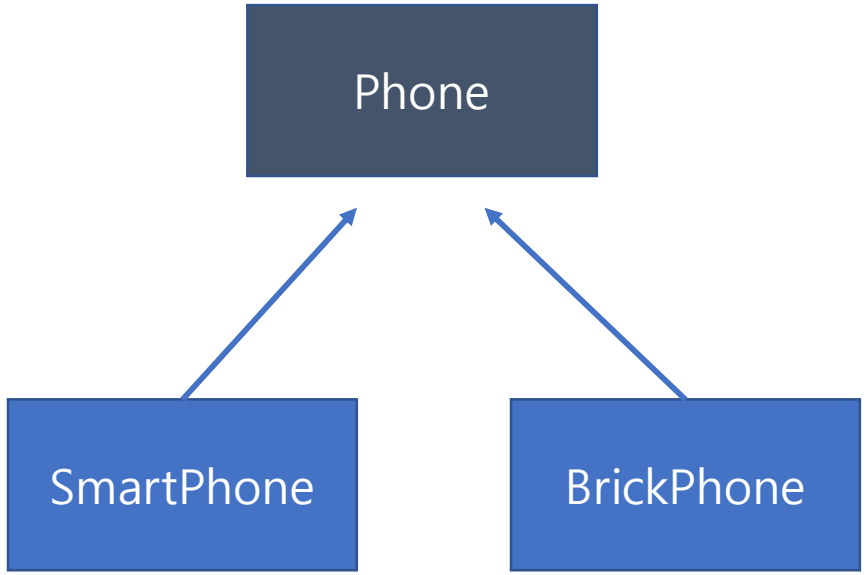
## Liskov Substitution Principle

리스코프 님의 원칙

# LSP (리스코프 Principle)

Client는 동작을 바꾸지 않고, base 대신 sub 클래스를 사용할 수 있어야한다.

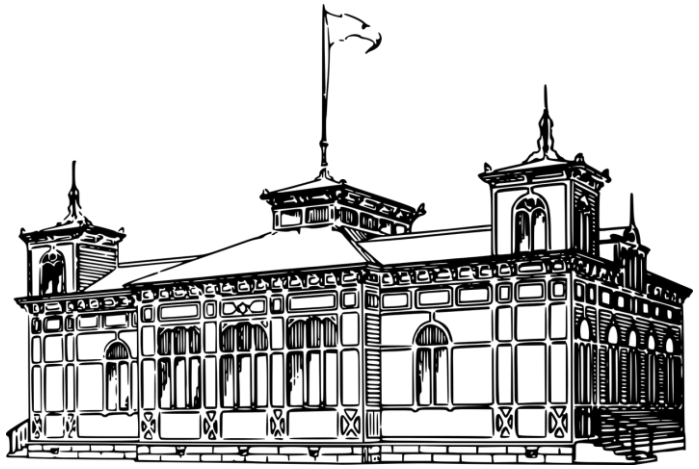
어떤 sub 클래스가 매개변수로 들어오던지, tryDate 메서드는 정상 동작해야한다.




```
def go(p: Phone):
    p.call()


if __name__ == '__main__':
    go(SmartPhone())
    go(BrickPhone())
```

# 부모의 역할을 자식이 대체할 수 있어야 한다

상속관계에 문제가 있는 경우, 부모로 정의된 프로그램에 문제가 생기거나 예외적인 부분이 발생할 수 있다.



도서관에서는 **사람**이 책을 읽을 수 있다

도서관에서는 학생이 책을 읽을 수 있다 (O)
**도서관에서는 아기가 책을 읽을 수 있다 (X)**


학생 is a 사람

아기 is a 사람

# 치환이 안된다는 것은?

## 명목적인 상속만으로는 안된다!

명목적으로 상속을 이용해 Type 끼리 상속관계를 갖는다고 해서 치환이 되는 것은 아니다.

Client 코드가 Sub 클래스들간의 차이점을 모르고도,
Base 클래스의 인터페이스를 통해 Sub 클래스들을 사용할 수 있어야 한다.

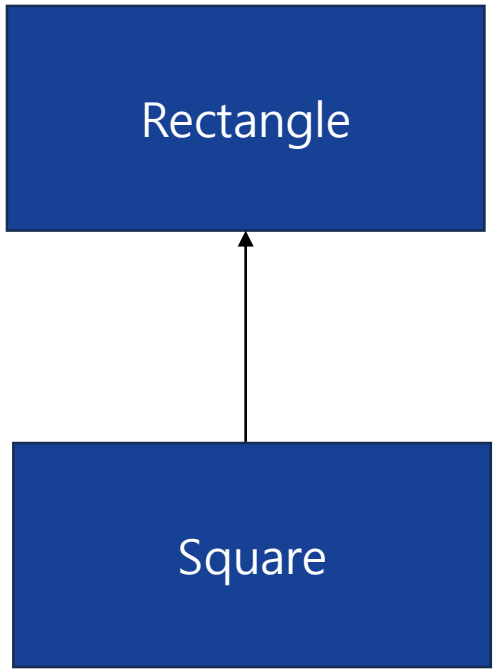
Client 코드에서 특정 Sub 클래스에 대한 예외적인 코드가 없도록 상속이 이뤄져야야 한다.

# [도전] Rectangle 을 상속한 Square

**[LSP 위반 사례 작성해보기]**

직사각형과 정사각형 클래스를 상속관계로 나타냈다.

Rectangle 로 정의된 Client 코드를 작성하되, **Square 인스턴스를 이용하면 깨지는 Client 코드**를 작성 해본다




https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/solid/lsp%EC%9C%84%EB%B0%98%EC%82%AC%EB%A1%80.py

# [도전] step1 : Car vs Plane

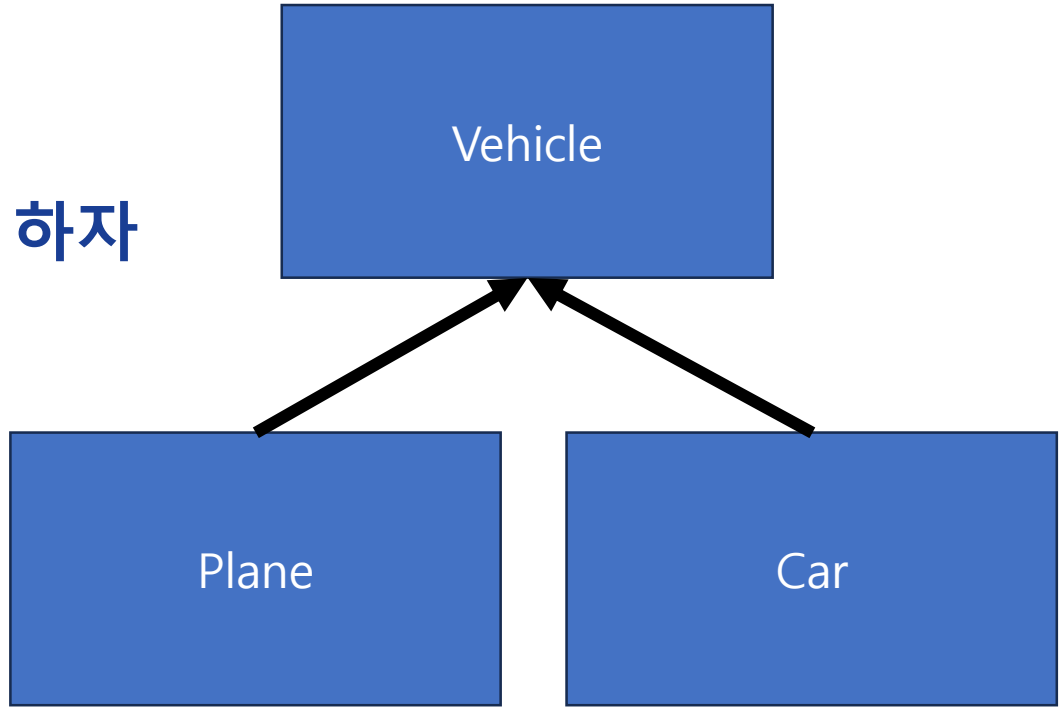
## LSP 위반이 되지 않도록 코드를 변경한다

## 가정

- 자동차는 Drive 모드에서 멈추지 않고 즉시 후진은 불가하다.
- 그리고 후진 도중에 즉시 Drive 모드로 전환 불가능하다.
- 그런데 Plane은 Drive 도중 바로 후진이 가능하다.

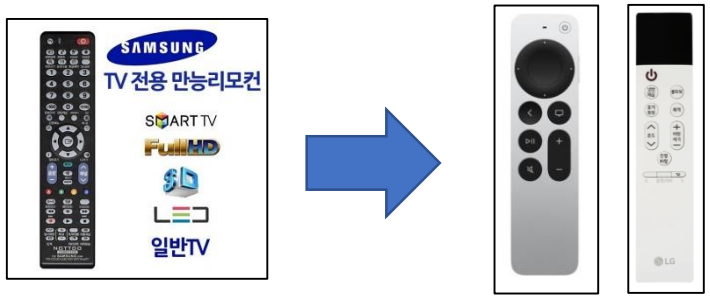
## 일부 Client 코드 측에서

## Gear 를 D에서 R로 변경하는 코드가 있다고 하자



# ISP

Interface Segregation Principle




큰 Interface 보다는
전용 Interface를 선호한다.

# Large Interface vs Small Interface 사용

## 어떤 것이 더 좋은 것일까?

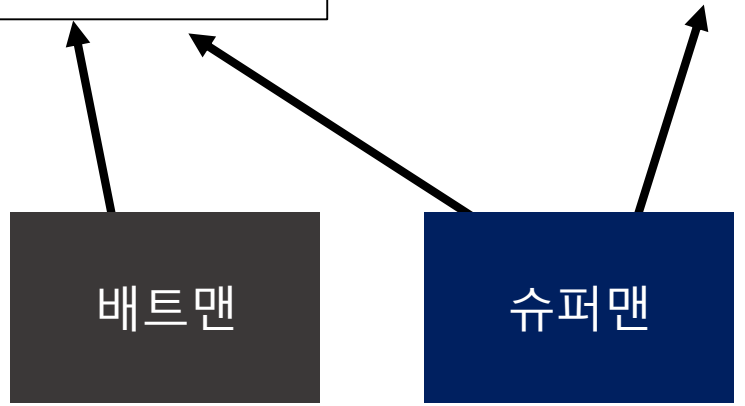
배트맨 : 걷거나 뛰어다님

슈퍼맨 : 걷거나 뛰어다니거나 날아다님

```python
class Move(ABC):
    @abstractmethod
    def walk(self): pass

    @abstractmethod
    def run(self): pass

    @abstractmethod
    def fly(self): pass
```


```python
class Walkable(ABC):
    @abstractmethod
    def walk(self): pass

    @abstractmethod
    def run(self): pass
```

```python
class Flyable(ABC):
    @abstractmethod
    def fly(self): pass
```



# 구현 클래스의 복잡함

인터페이스가 거대한 경우 구현 클래스 또한 **불필요한** 메서드를 구현해야 한다

```python
class Move(ABC):
    @abstractmethod
    def walk(self): pass

    @abstractmethod
    def run(self): pass

    @abstractmethod
    def fly(self): pass
```

배트맨

슈퍼맨

```python
class Walkable(ABC):
    @abstractmethod
    def walk(self): pass

    @abstractmethod
    def run(self): pass
```

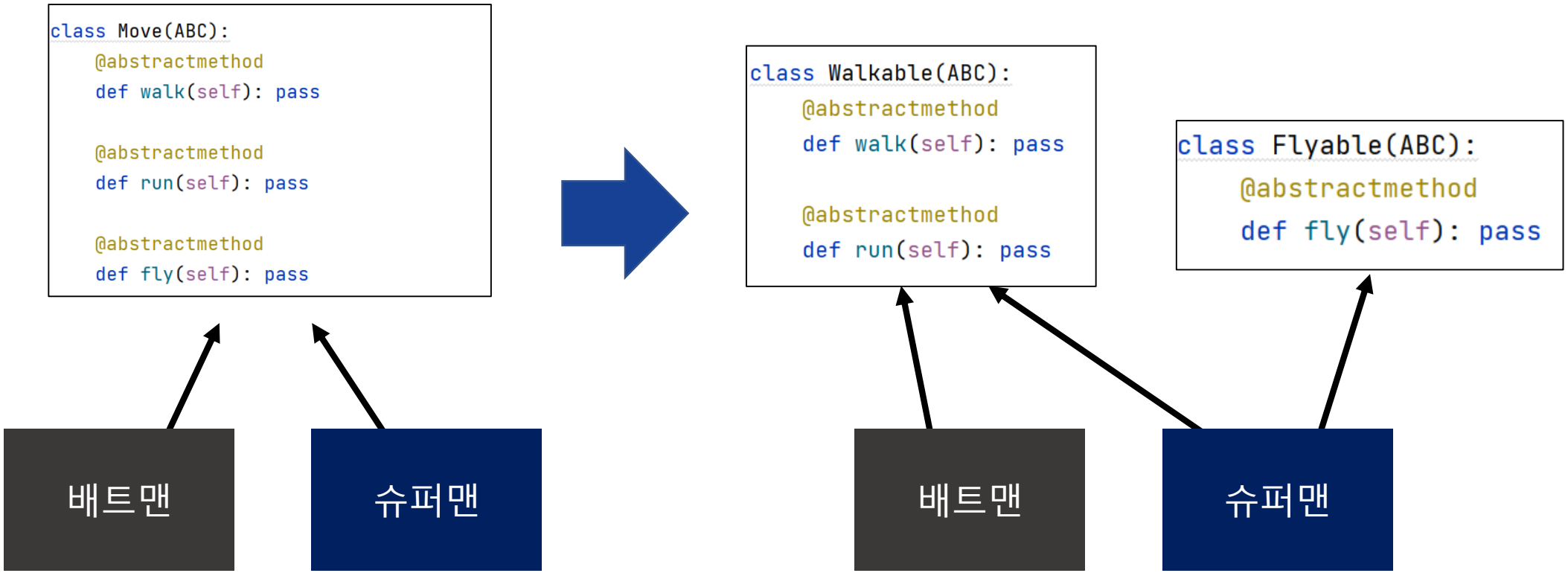
```python
class Flyable(ABC):
    @abstractmethod
    def fly(self): pass
```

배트맨

슈퍼맨

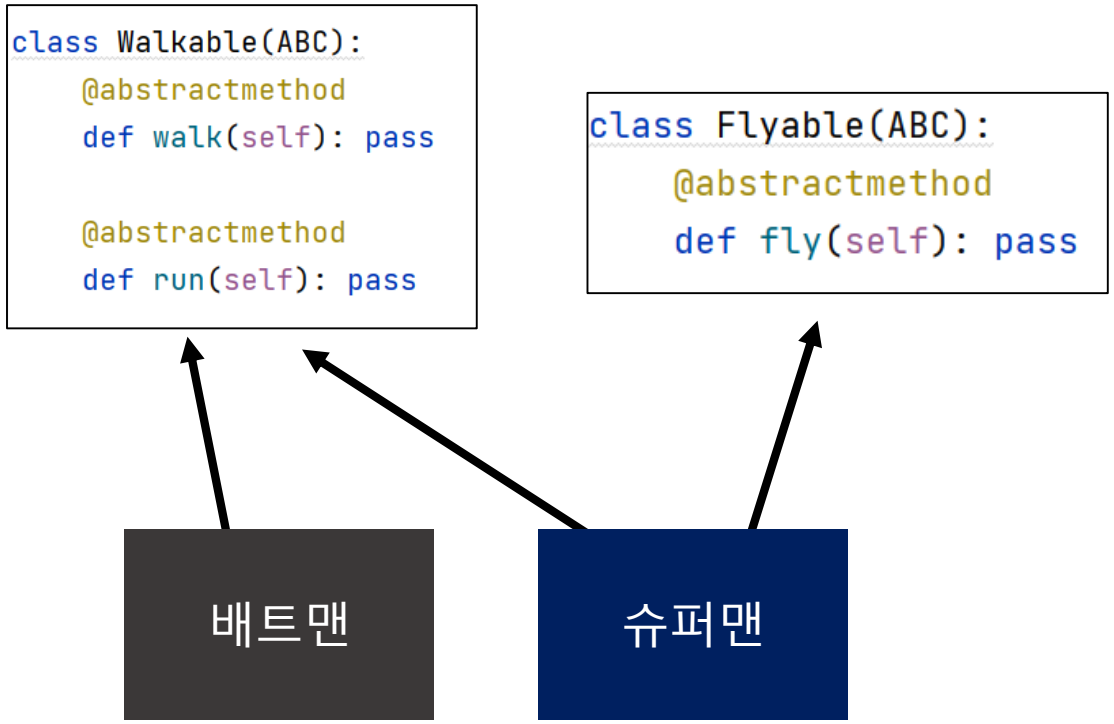
# ISP 의 이점 -1

ISP 적용시 인터페이스를 구현하는 구체 클래스도 구현이 덜 복잡해진다



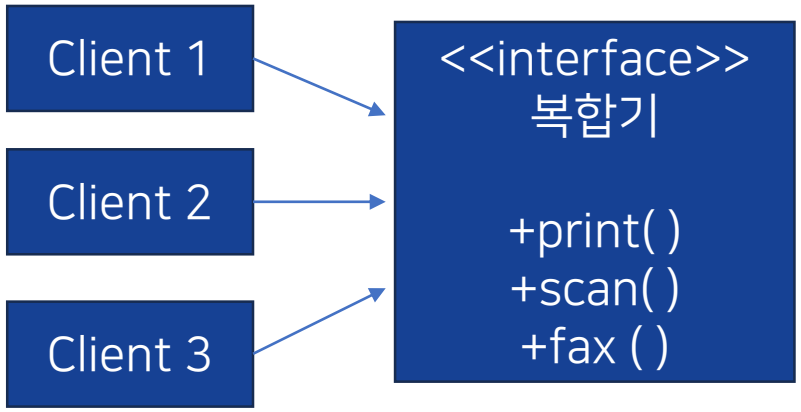
# [도전] 직접 구현해보기

Client Code 는 간단히 테스트할 수 있는 코드로 작성한다.



# 클라이언트가 사용하지 않는 인터페이스

각 클라이언트가 이용하는 인터페이스가 제 각각인 경우, 불필요한 의존성과 불명확한 인터페이스를 갖게 될 수 있다.




클라이언트1 은 print 만 이용
클라이언트2 은 scan 만 이용
클라이언트3 은 fax 만 이용

```python
class 복합기(ABC):
    @abstractmethod
    def print(self, doc): ...
    @abstractmethod
    def scan(self, doc): ...
    @abstractmethod
    def fax(self, doc): ...
```

```python
class PrintClient:
    my_printer: 복합기 # Client에게 불필요한
                       # 인터페이스를 가지고 있음

    def do_print(self, doc):
        self.my_printer.print(doc)
```

# ISP의 이점 -2

클라이언트는 자신이 사용하지 않는 메서드에 의존하지 않아야 한다.

ISP 를 적용하면 클라이언트가 명확한 추상화(인터페이스)를 가지게 된다.

```python
class 프린터(ABC):
    @abstractmethod
    def print(self, doc): ...

class 스캐너(ABC):
    @abstractmethod
    def scan(self, doc): ...

class 팩스(ABC):
    @abstractmethod
    def fax(self, doc): ...
```

```python
class PrintClient:
    my_printer: 프린터

    def do_print(self, doc):
        self.my_printer.print(doc)
```

Client 안성맞춤 인터페이스

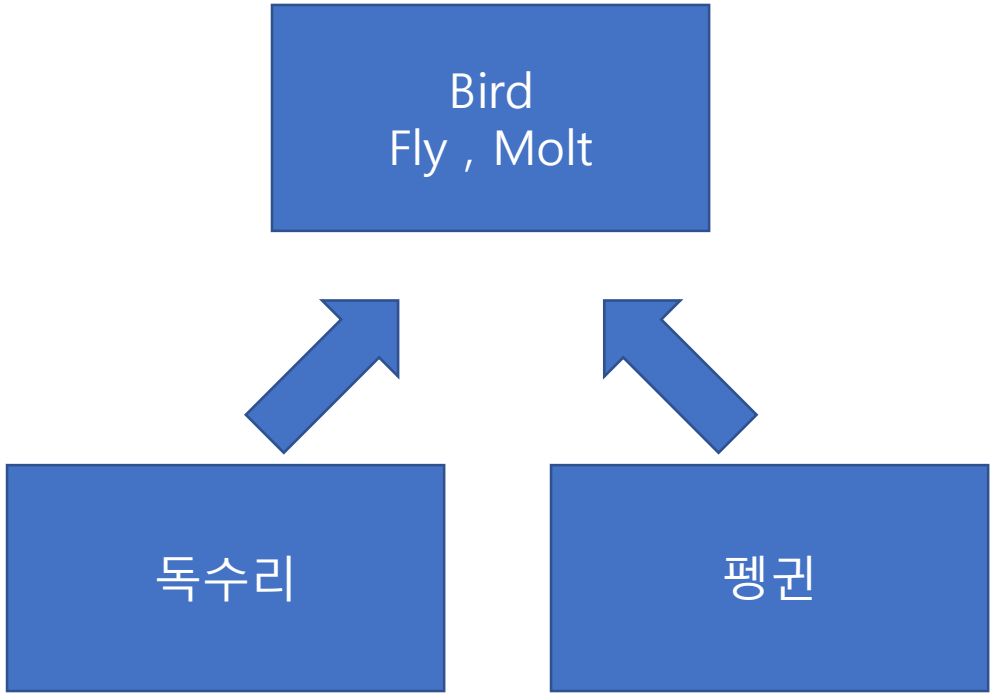
# [도전] step1 : 팽귄과 독수리

- 털갈이(molt) 는 둘 다 가능하지만, Fly는 독수리만 가능하다.

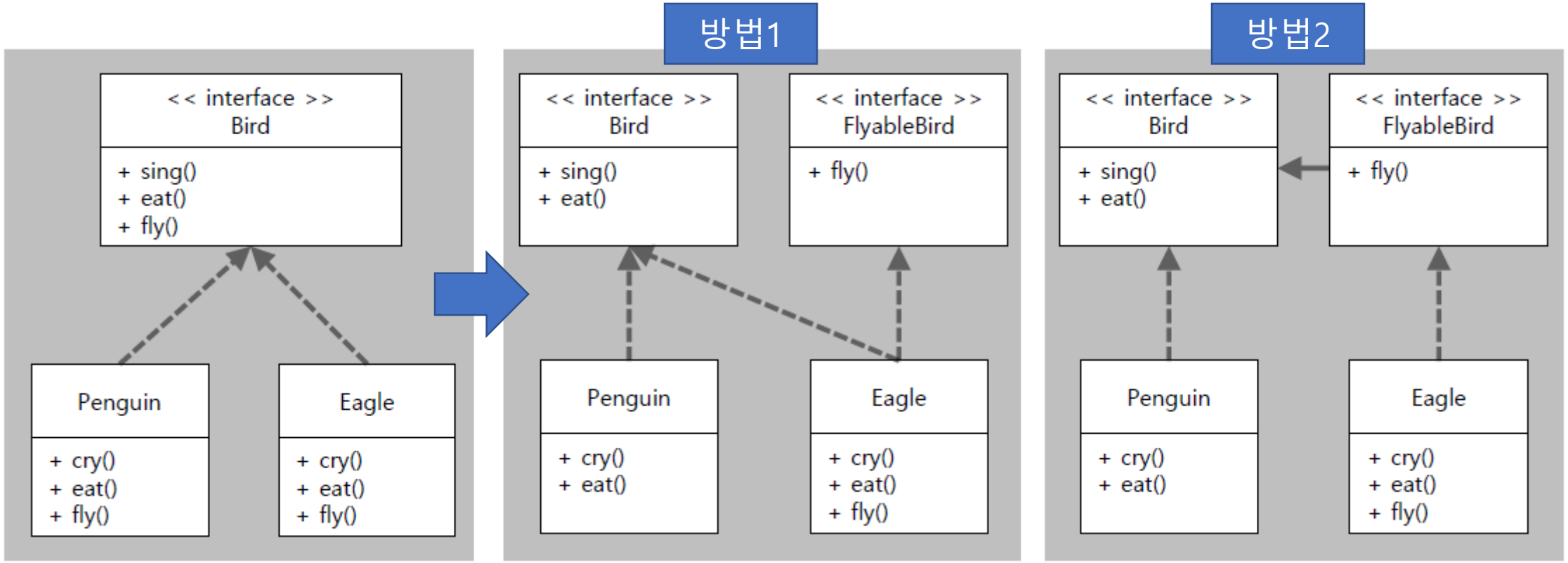
Fly 를 이용하는
Client

Molt와 Fly 를 이용하는
Client

Molt 를 이용하는
Client



# 두 가지 해결방법


방법1
방법2
<< interface >>
Bird
+ sing()
+ eat()
+ fly()
Penguin
+ cry()
+ eat()
+ fly()
Eagle
+ cry()
+ eat()
+ fly()
<< interface >>
Bird
+ sing()
+ eat()
<< interface >>
FlyableBird
+ fly()
Penguin
+ cry()
+ eat()
Eagle
+ cry()
+ eat()
+ fly()
<< interface >>
Bird
+ sing()
+ eat()
<< interface >>
FlyableBird
+ fly()
Penguin
+ cry()
+ eat()
Eagle
+ cry()
+ eat()
+ fly()

# [도전] step2 : 자동차와 드론

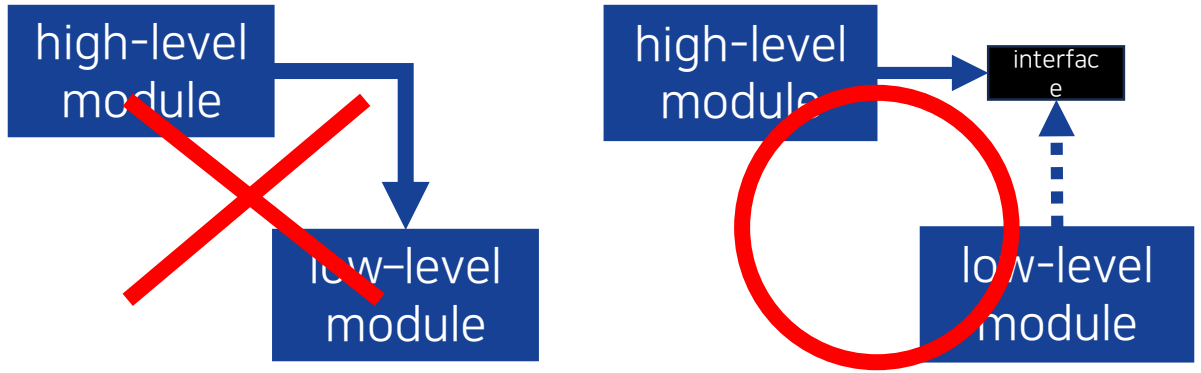
- 자동차 (Vehicle 상속받음)
  라디오 ON/OFF 기능 있음
  카메라 ON/OFF 기능 없음

- 드론 (Vehicle 상속받음)
  카메라 ON/OFF 기능 있음
  라디오 ON/OFF 기능 없음

거대한 Vehicle Interface를
분할해보자.

# DIP

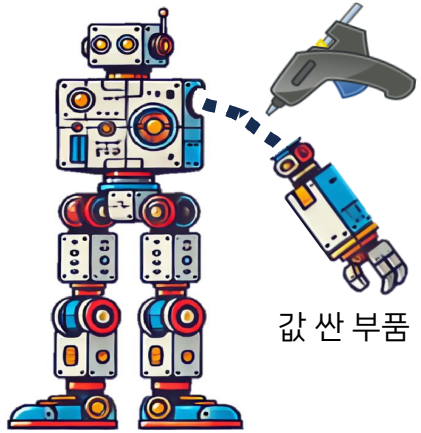
## Dependency Inversion Principle



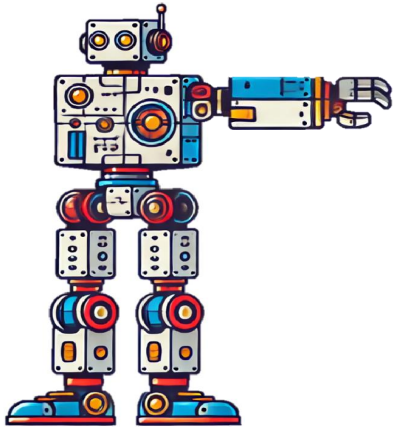
# DIP 비유적 이해 - 1

값 비싼 코어 본체가 값 싼 로봇 팔 부품의 고장, 변경에 영향을 받으면 안된다

코어 본체와 로봇 팔을 접착제로 연결 시켰다면?




값 비싼 코어 본체

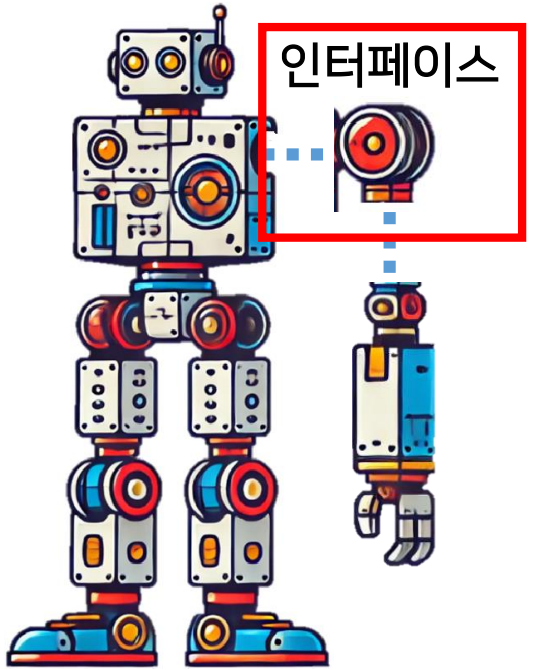


본체 + 부품의 결합도가 높다

# DIP 비유적 이해 - 2

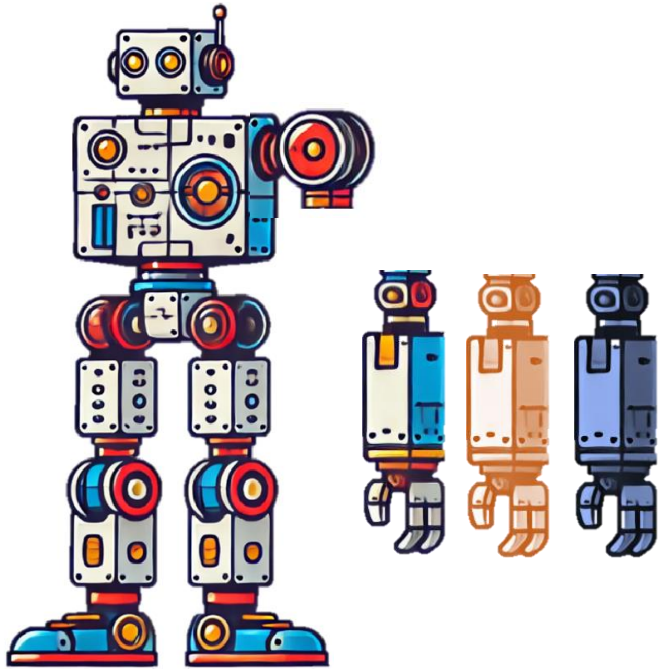
인터페이스(추상)를 이용해서 설계하면

값비싼 코어 본체가 값 싼 로봇 팔에 영향을 받지 않게 할 수 있다.

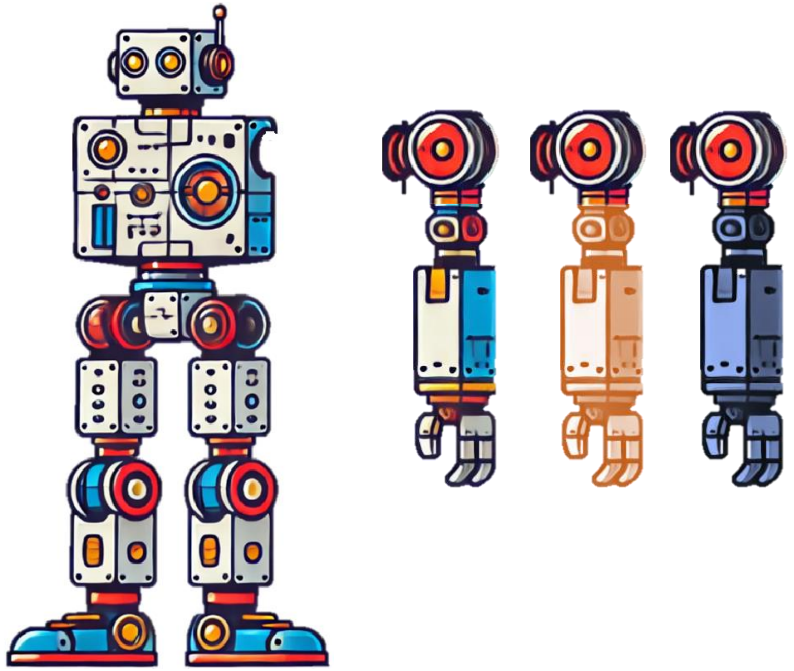


# Quiz 인터페이스는 어디에 맞춰야 할까?

## 인터페이스는 어디에 맞춰야 할까?



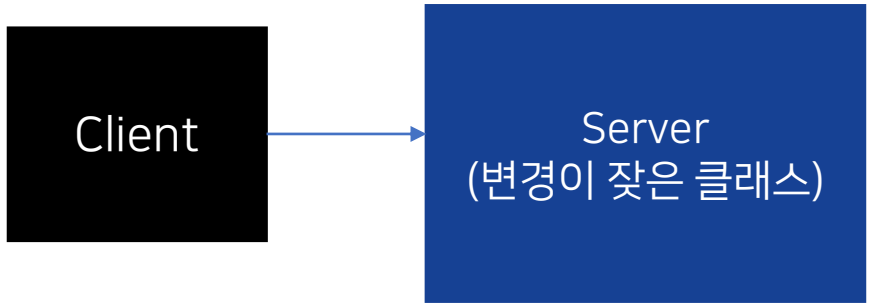
로봇 본체에 의해 인터페이스 결정
-> 인터페이스가 로봇 본체에 영향을 받는다



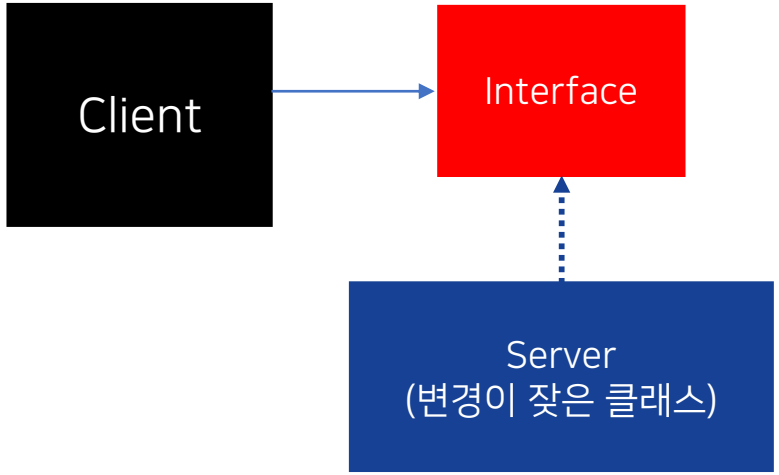
로봇 팔에 의해 인터페이스 결정
-> 인터페이스가 로봇 팔에 영향을 받는다

# DIP : 추상에 의존하자!

값 비싼 로봇 코어 본체 (상위 정책 모듈)이
값 싼 로봇 팔 (하위 정책 모듈)의 변경에 영향 받지 않게 한다




변경을 할 때, Client 도 같이 변경될 수 있다

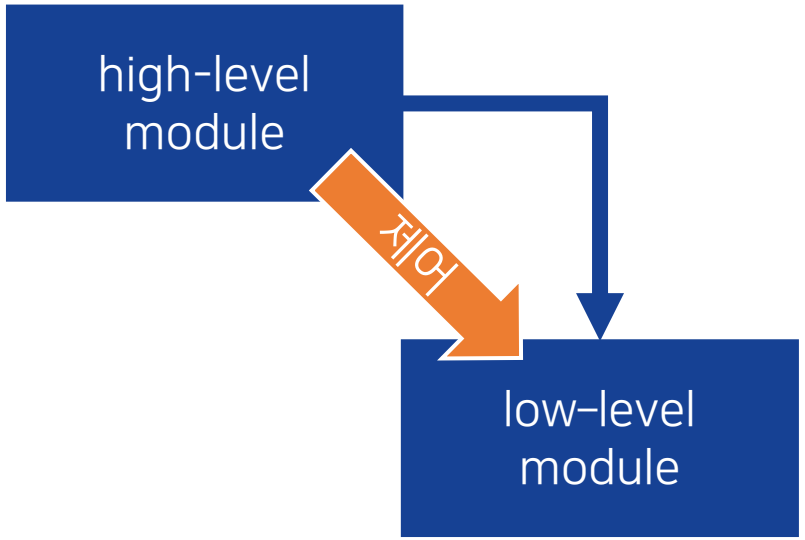



Client 는 인터페이스에 의존하여 변경이 잦은 클래스에 영향을 최소화 할 수 있다.

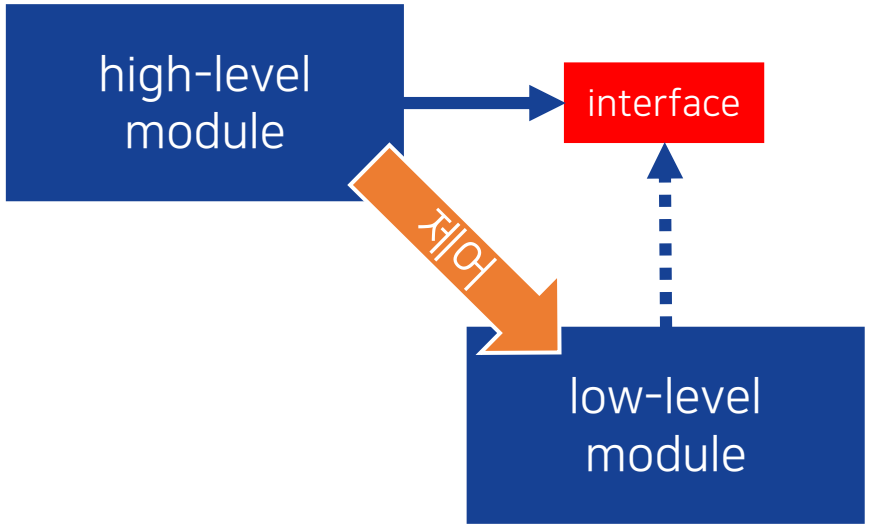
# 의존성 역전이라 하는 이유

절차지향적 프로그래밍에서는 상위 정책이 하위 정책에 의존하는 경우가 많다.

객체지향적 프로그래밍에서 추상화에 의존하면 제어의 방향과 의존의 방향이 달라진다.




high-level module이 low-level module에 의존한다. low-level module 변경이 high-level module 에 영향을 줄 수 있고 high-level module의 변경을 유발한다.




high-level module, low-level module 둘 다 interface(추상화)에 의존한다. **interface 변경에만 영향을 받는 구조이다.**

# [참고] DIP 를 적용하지 않아도 되는 경우

## str 는 매우 안정적인 타입으로 직접적인 의존을 해도 괜찮다

str 의 경우는 매우 안정적인(변경이 거의 없는) 타입으로 직접적인 의존성을 갖는 것이 좋다.
여기에 DIP를 적용한다면 오히려 불필요한 복잡성을 띄게 될 것이다

> 단, 개발중인 혹은 변경이 많은 구체적인 클래스에 직접적인 의존성을 가지지 말자!

* 구체 클래스 : 추상클래스나, 인터페이스가 아닌 구현부를 갖춘 클래스를 의미한다

# [도전] step2 : Notifier로 의존성 역전하기

인터페이스를 두어, 의존성을 낮추어 보자.