LSP

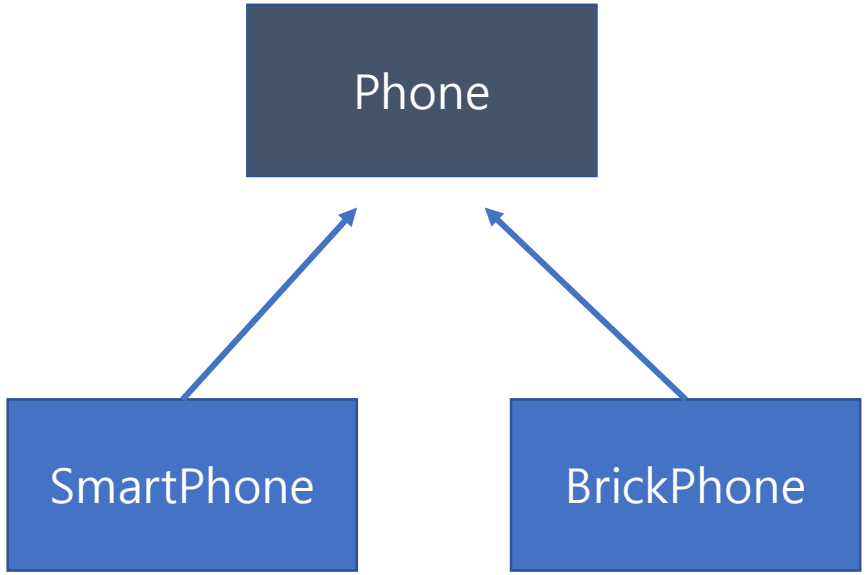
Liskov Substitution Principle

리스코프 님의 원칙

LSP (리스코프 Principle)

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

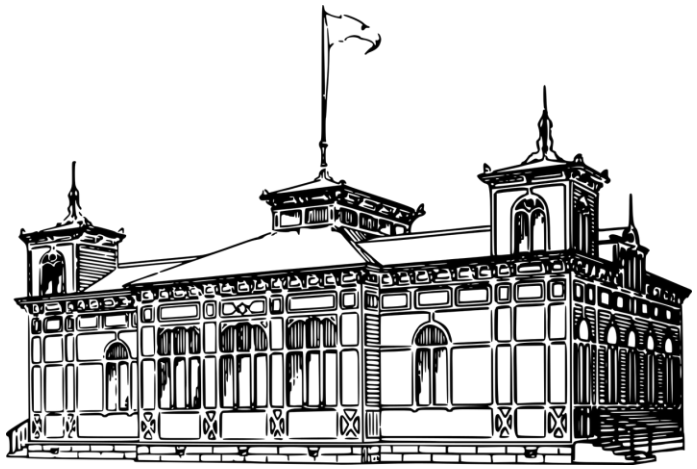
어떤 sub 클래스가 매개변수로 들어오던지, tryDate 메서드는 정상 동작해야한다.




```python
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.

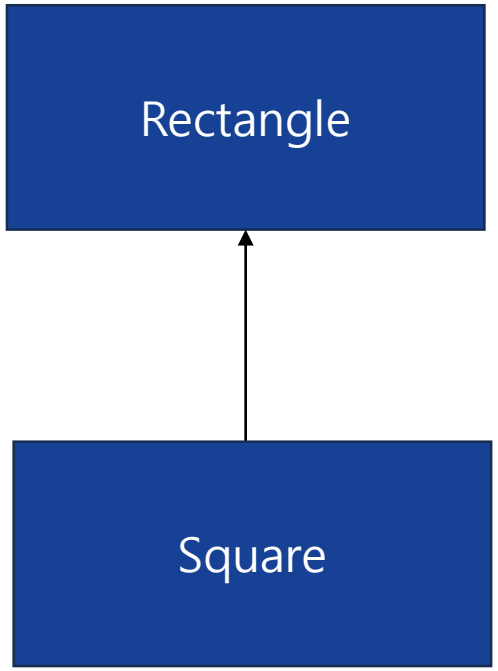
Client 코드에서 특정 Sub 클래스에 대한 예외적인 코드가 없도록 상속이 이뤄져야야 한다.

[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

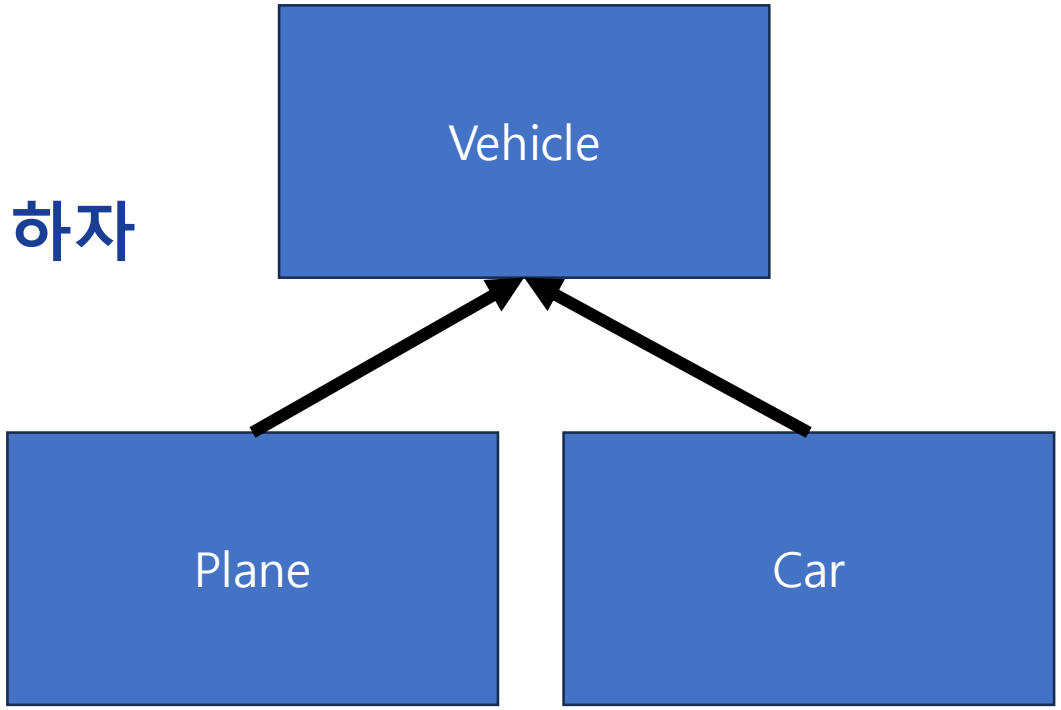
LSP 위반이 되지 않도록 코드를 변경한다

가정

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

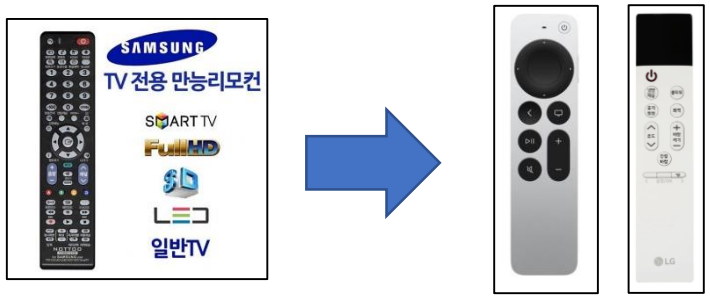
일부 Client 코드 측에서

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?

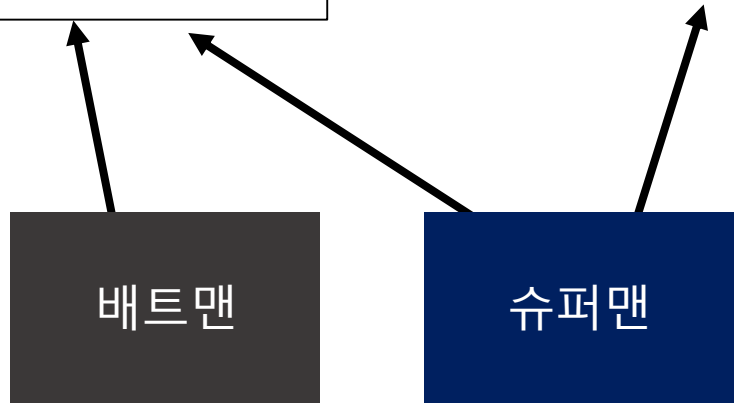
배트맨 : 걷거나 뛰어다님

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
```

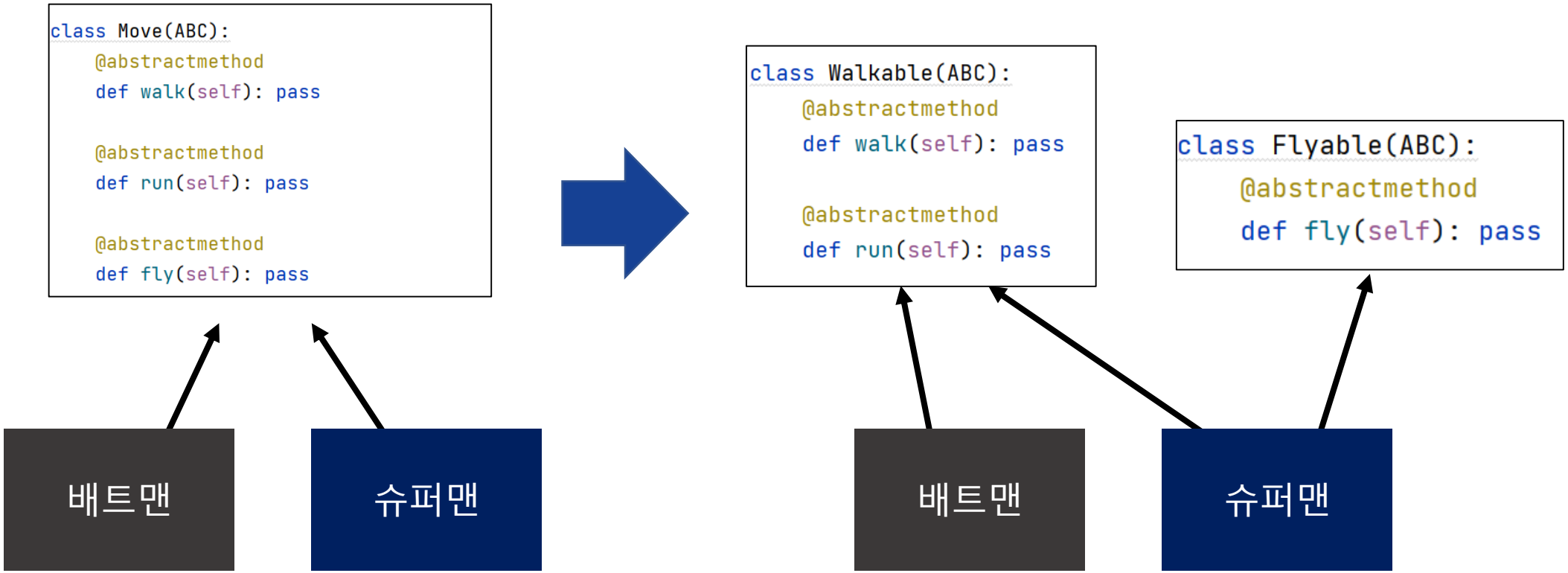
```python
class Flyable(ABC):
    @abstractmethod
    def fly(self): pass
```

배트맨

슈퍼맨

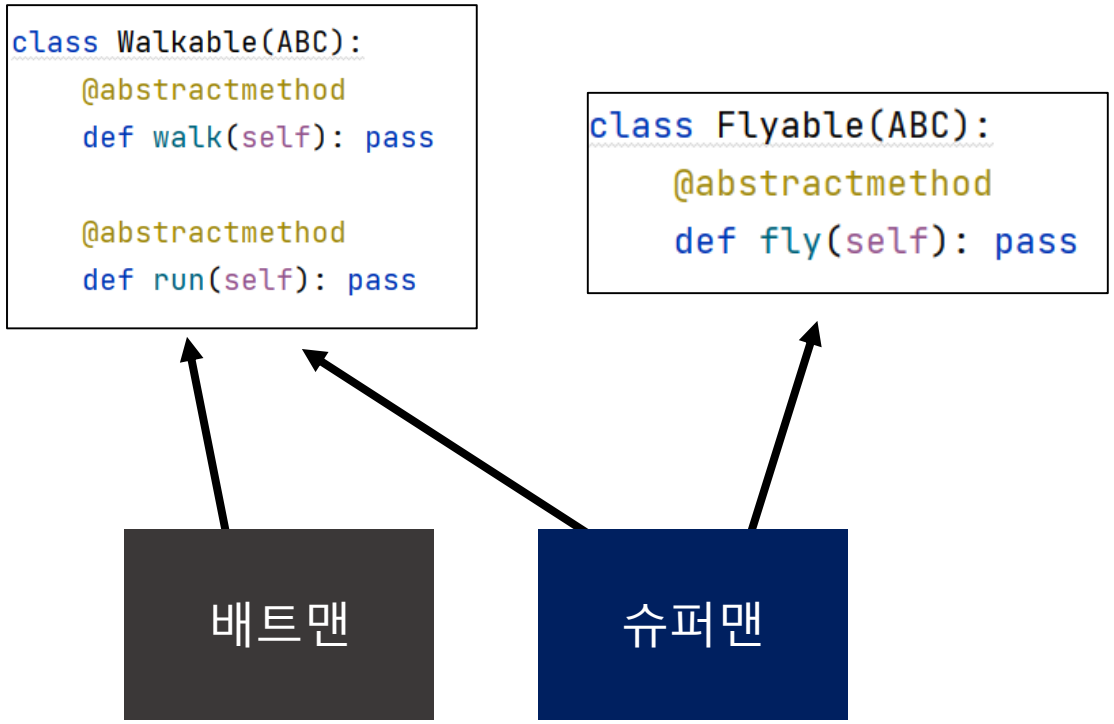
ISP 의 이점 -1

ISP 적용시 인터페이스를 구현하는 구체 클래스도 구현이 덜 복잡해진다



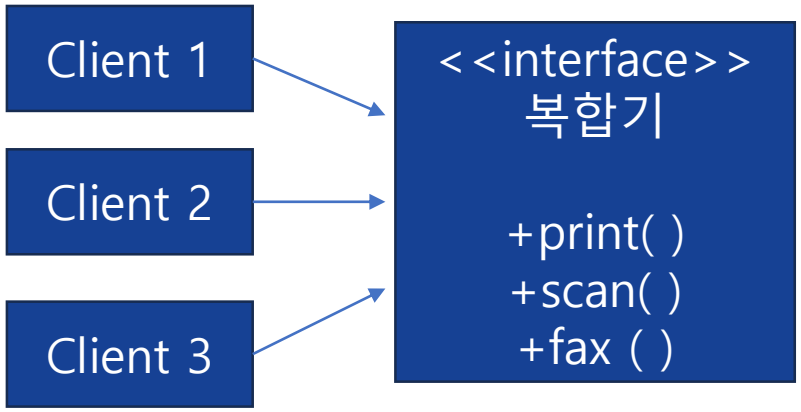
[도전] 직접 구현해보기

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.

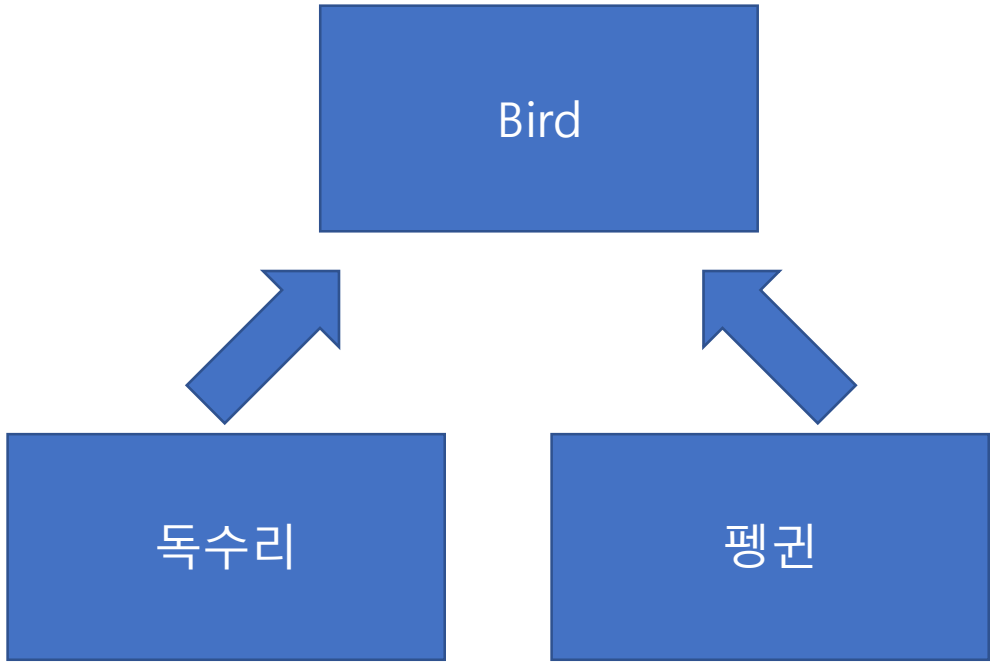
```
class 프린터(ABC):
    @abstractmethod
    def print(self, doc): ...

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

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

```
class PrintClient:
    my_printer: 프린터

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

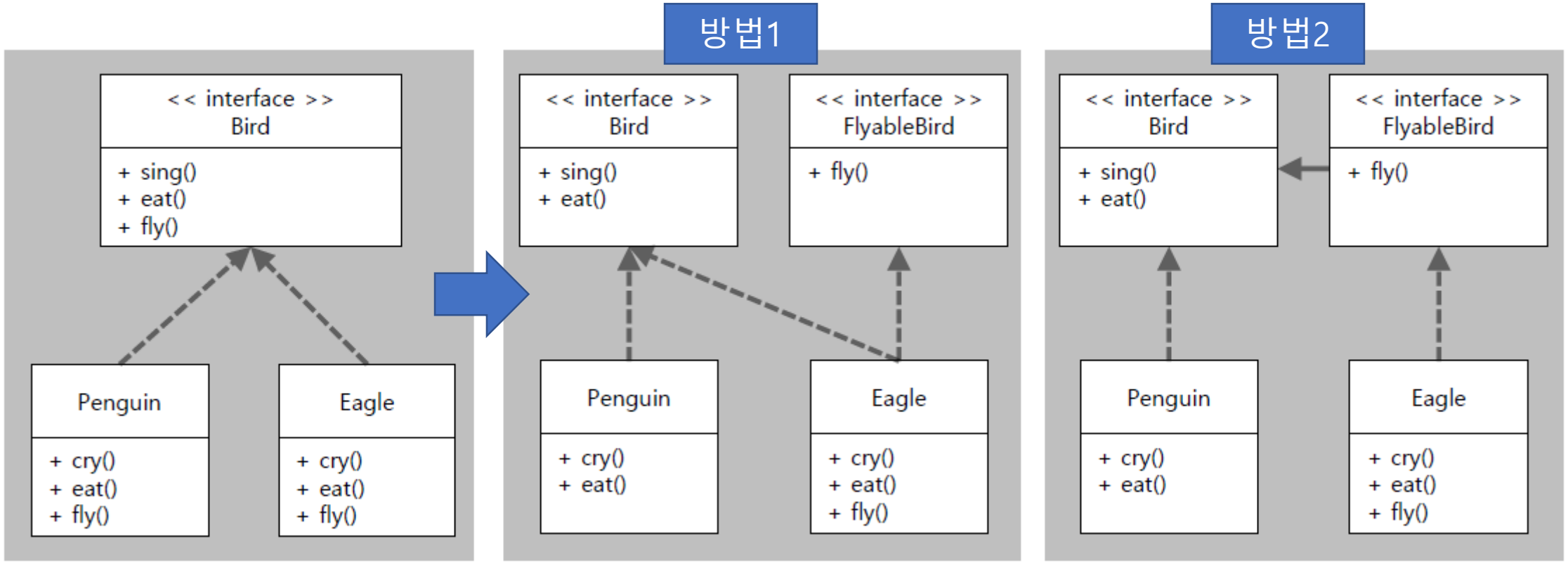
Client 안성맞춤 인터페이스

[도전] step1 : 팽귄과 독수리

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



두 가지 해결방법


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

[도전] step2 : 자동차와 드론

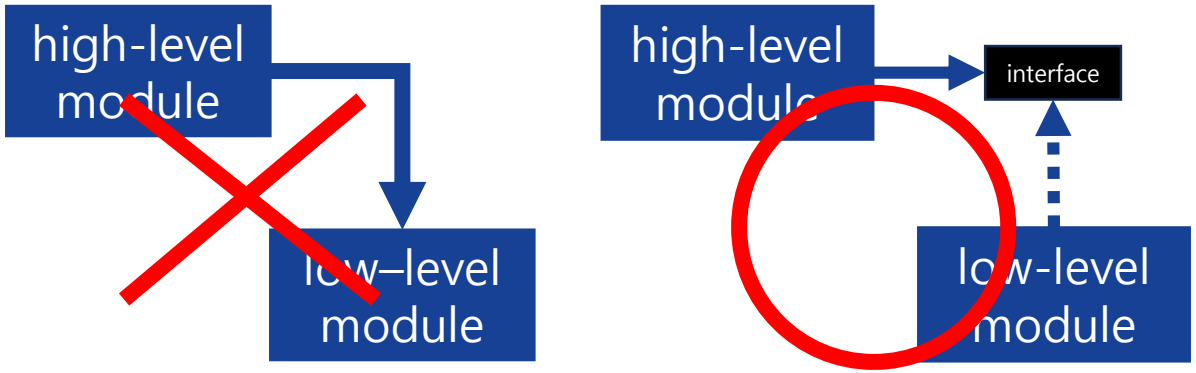
- 자동차 (Vehicle 상속받음)
  라디오 ON/OFF 기능 있음
  카메라 ON/OFF 기능 없음

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

거대한 Vehicle Interface를
분할해보자.

DIP

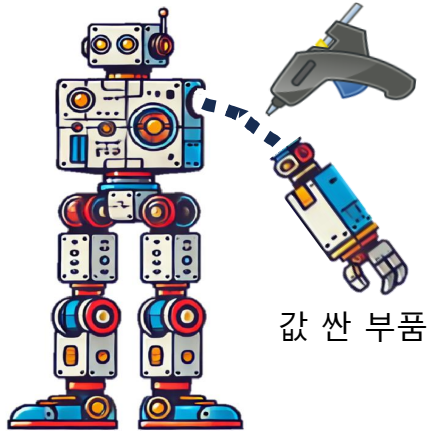
Dependency Inversion Principle



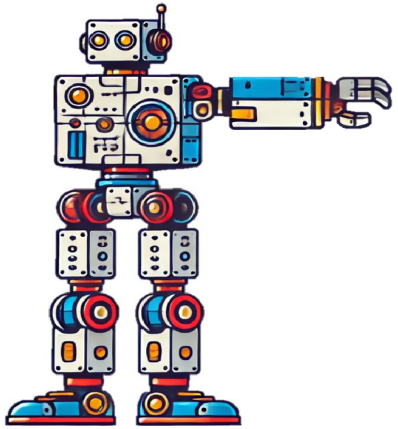
DIP 비유적 이해 - 1

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

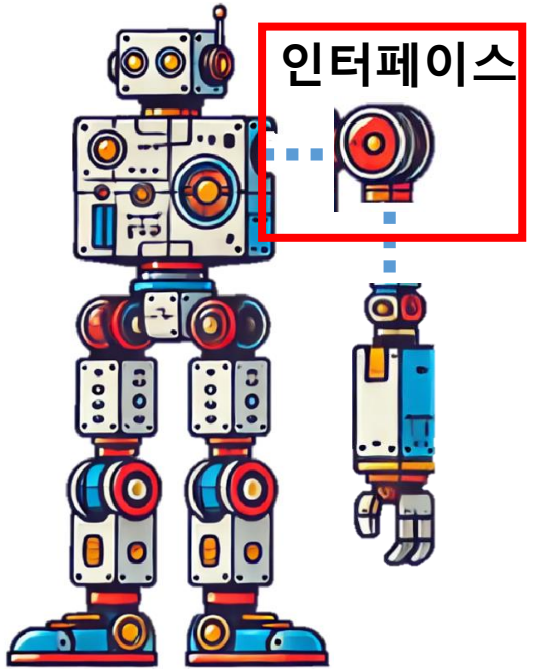


본체 + 부품의 결합도가 높다

DIP 비유적 이해 - 2

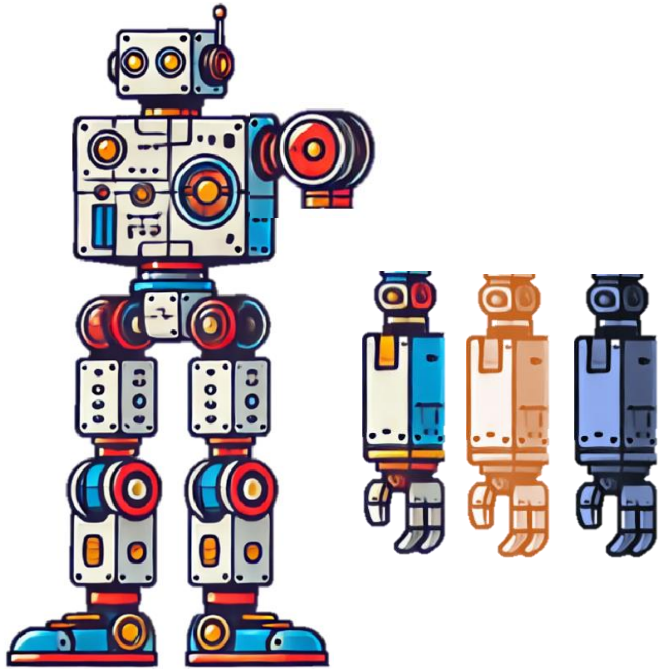
인터페이스(추상)를 이용해서 설계하면

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

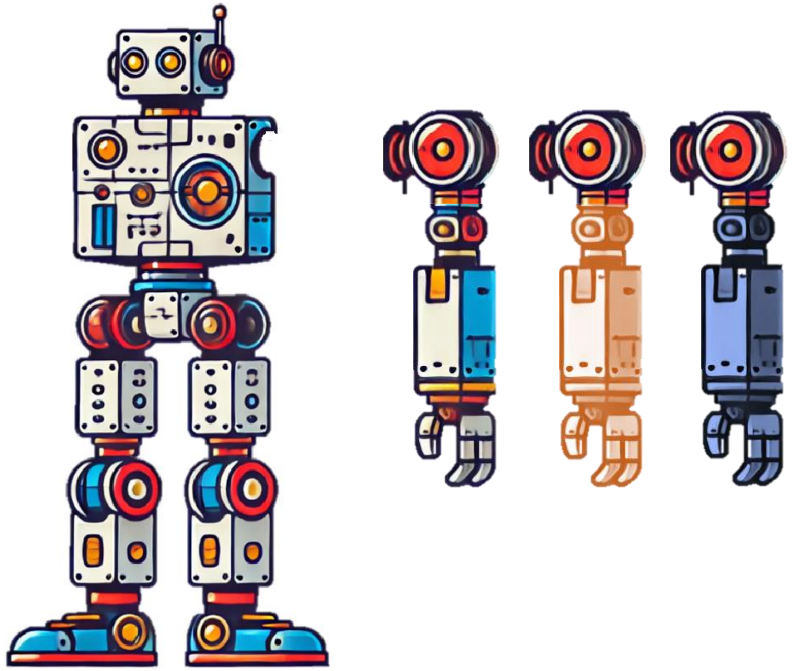


Quiz 인터페이스는 어디에 맞춰야 할까?

인터페이스는 어디에 맞춰야 할까?



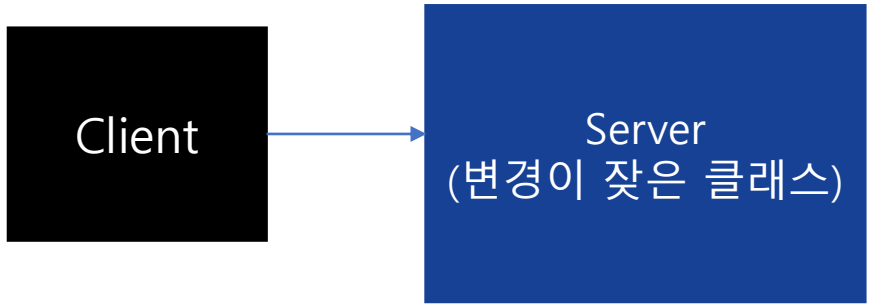
로봇 본체에 의해 인터페이스 결정
-> 인터페이스가 로봇 본체에 영향을 받는다



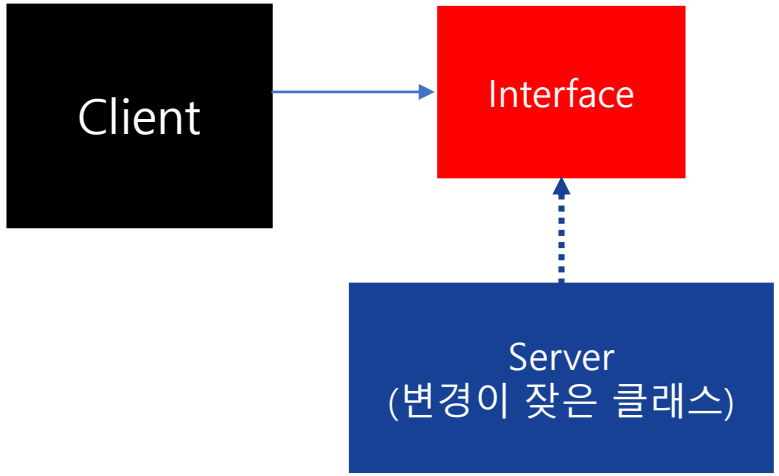
로봇 팔에 의해 인터페이스 결정
-> 인터페이스가 로봇 팔에 영향을 받는다

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

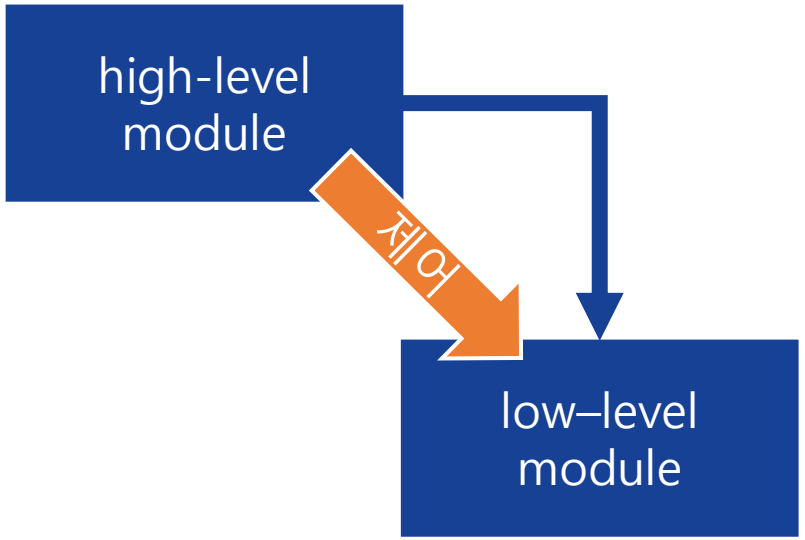


Client 는 인터페이스에 의존하여 변경이 잦은 클래스에 영향을 최소화 할 수 있다.

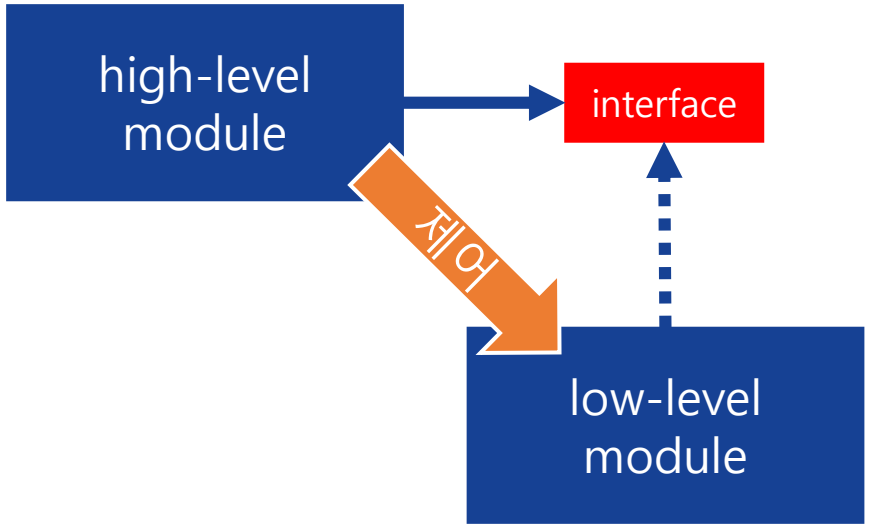
의존성 역전이라 하는 이유

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.