프로그래밍 역량 강화 전문기관, 민코딩

SOLID

목차

SOLID 이해를 위한, OOP 주요 내용

OOP 주요 내용

객체지향프로그래밍

✓ Object-Oriented Programming이란

- 객체를 주된 관심으로 삼는 프로그래밍 방법론.
- 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.

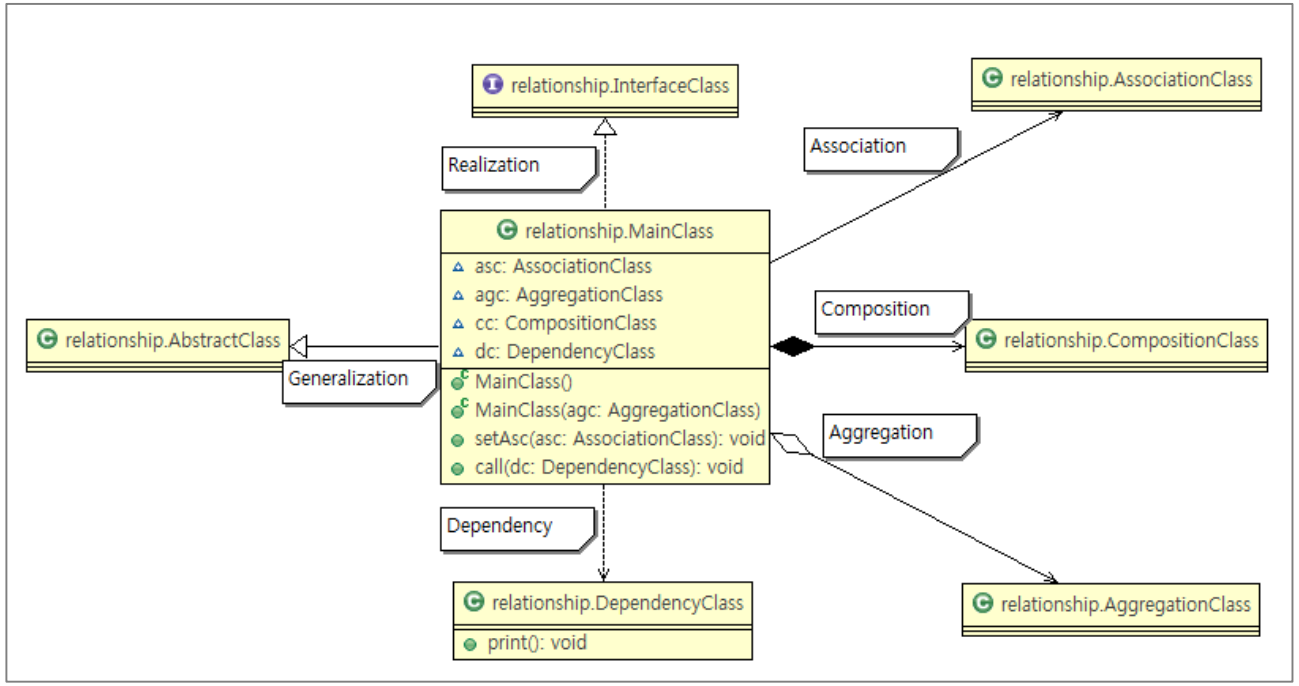
✓ **캡슐화(Encapsulation)**
- 구현방법에 대한 자세한 사항은 블랙박스화 하여 드러내지 않고 외부로 노출된 인터페이스를 통해서만 사용할 수 있게 하는 개념
- 인터페이스를 변경시키지 않는 한 사용자는 구현의 변경에 영향을 받지 않고 사용할 수 있고, 개발자는 내부 구조나 구현방법을 자유롭게 변경할 수 있다.

✓ **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

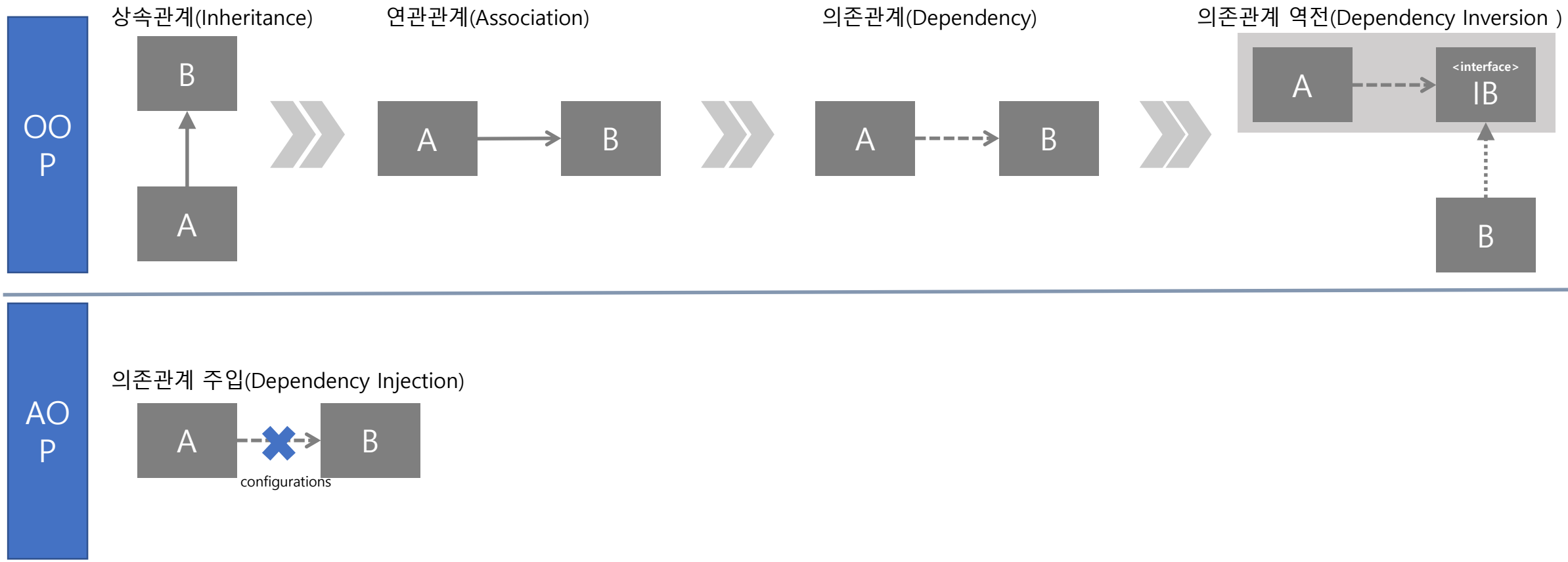
객체의 관계 표현 방법

✓객체의 관계 표현 방법은 결국 객체간의 의존성에 대한 설계이다.

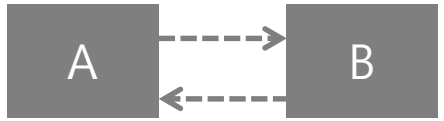


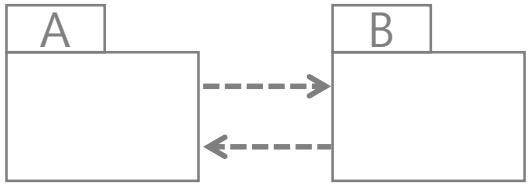
관계별 의존 정도 비교

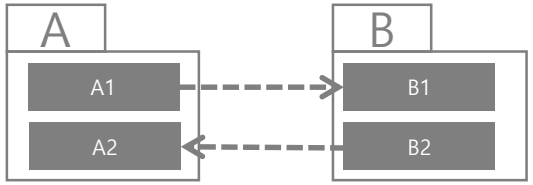
✓A가 B에 의존한다.(A depends on B.) : B가 변경될 때 A도 함께 변경될 수 있다.



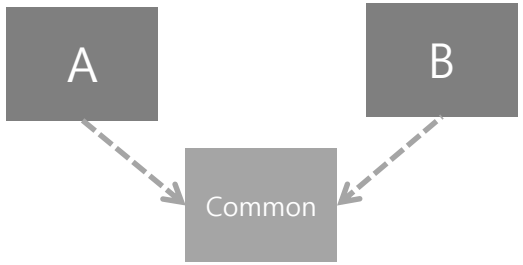
양방향 의존성일때, 리팩토링 방법



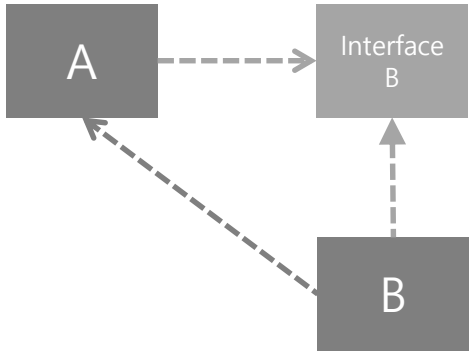


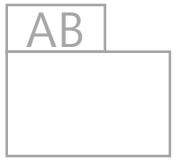


OR

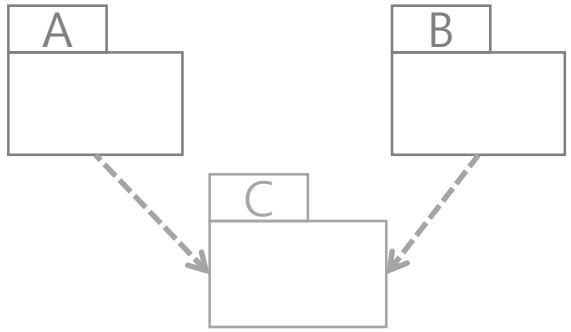


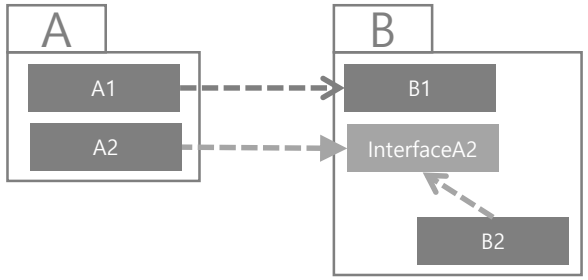
OR



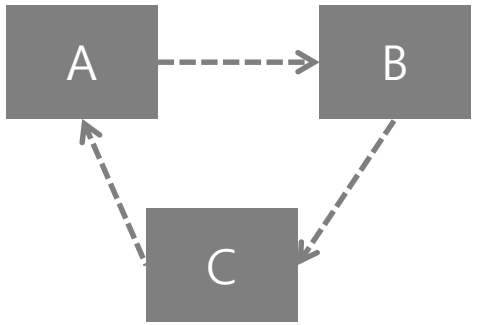


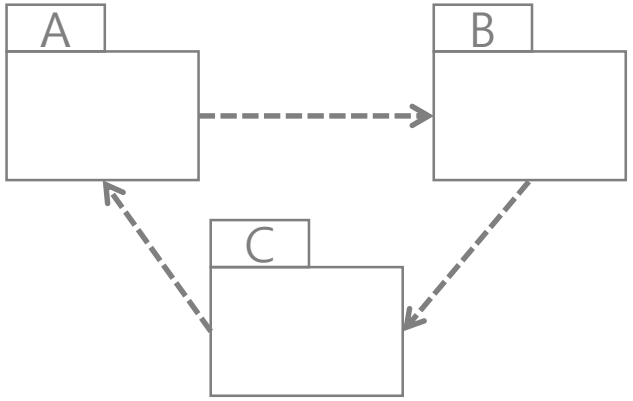
OR

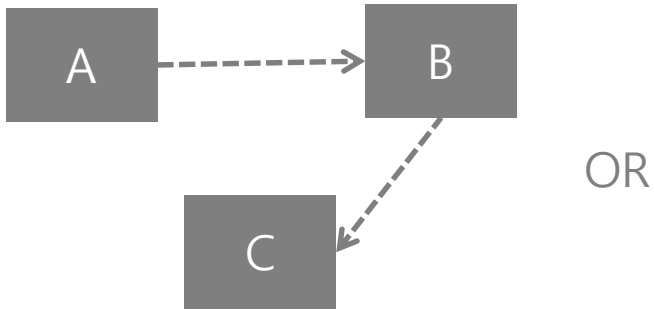




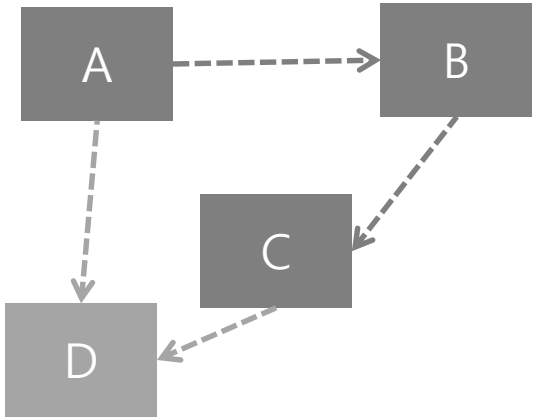
순환참조에서 리팩토링 방법

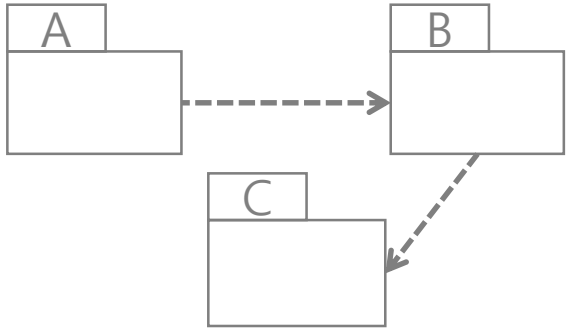




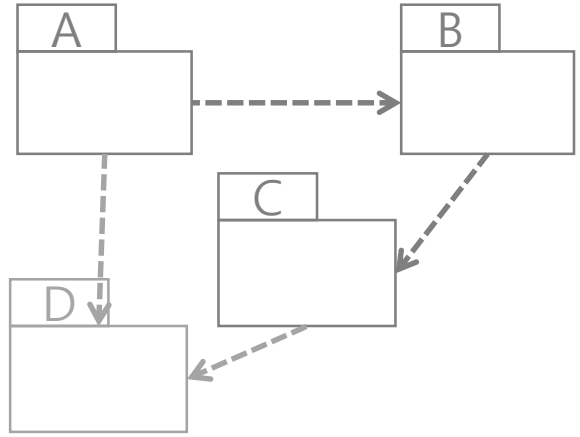


OR



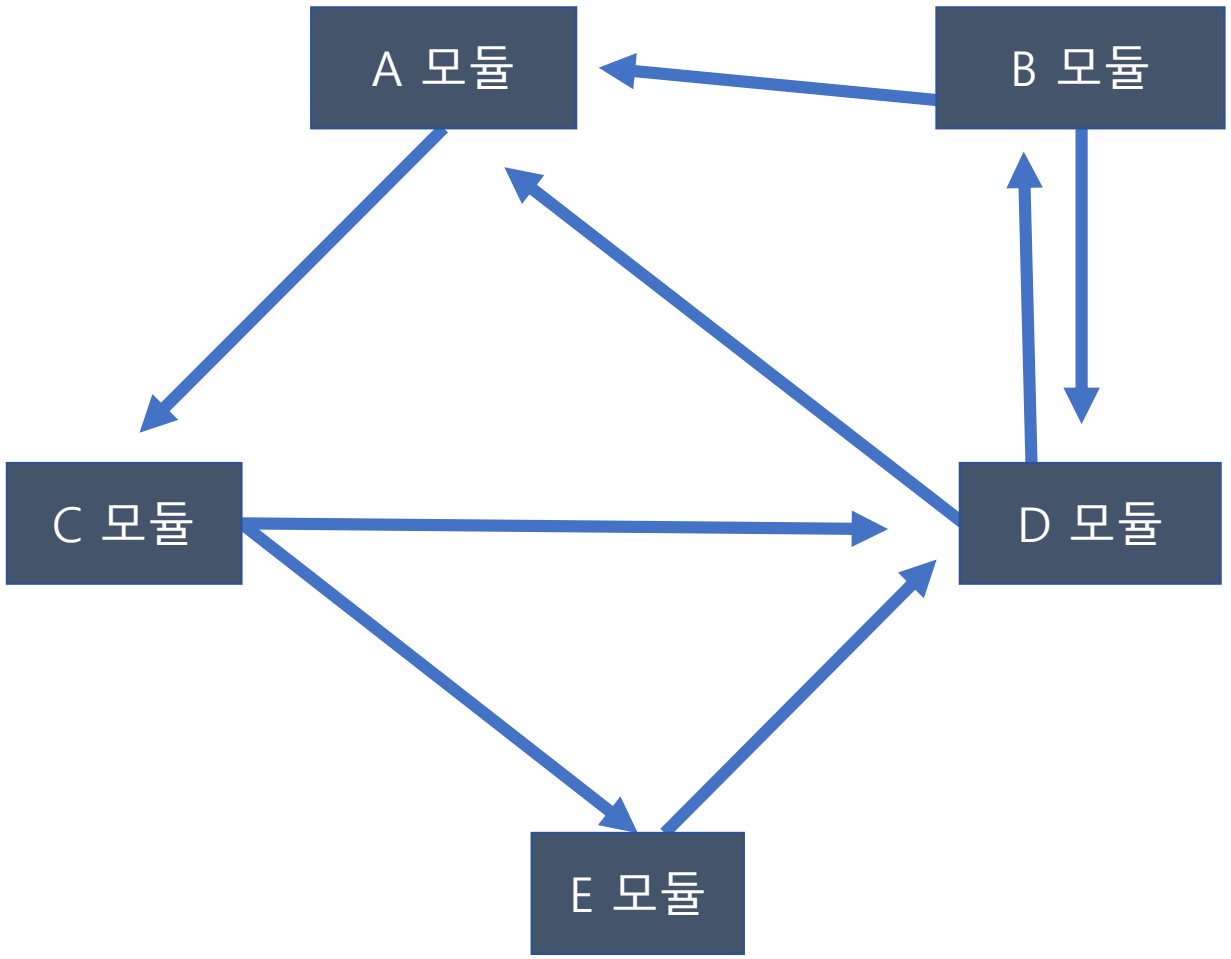


OR



의존성 싸이클, 순환참조의 문제점

✓한 모듈에 변경이 발생할 때,
유지보수가 어려워진다.



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

✓사내 주소

- https://github.samsungds.net/cra1-sec/SOLID

✓사외 주소

- https://github.com/mincoding1/SOLID
  - Kata 출처 1 (Vehicle 컨셉) : https://github.com/bsferreira/solid
  - Kata 출처 2 : https://github.com/mikeknep/SOLID

Single Responsibility Principle

SRP

SRP 정의, 리팩토링 적용 방법

✓Single Responsibility Principle (SRP)

- 모든 클래스는 하나의 책임만 가지며, 클래스는 그 책임을 완전히 캡슐화해야 한다.
- 클래스가 제공하는 모든 기능은 이 책임과 부합해야 한다.

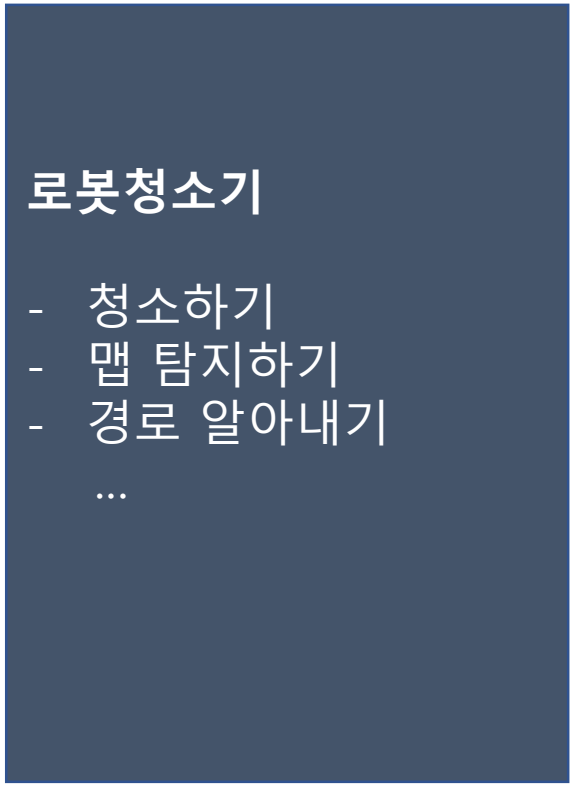
✓리팩토링 적용 방법

- Extract Class를 통해 혼재된 각 책임을 각각의 개별 클래스로 분할하여 클래스 당 하나의 책임만을 맡도록 함
- 책임만 분리하는 것이 아니라 분리된 두 클래스간의 관계 복잡도를 줄이도록 설계
- Extract Class된 각각의 클래스들이 유사하고 비슷한 책임을 중복해서 갖고 있다면 Extract Superclass를 사용
- 책임을 기존의 어떤 클래스로 모으거나, 이럴만한 클래스가 없다면 새로운 클래스를 만들어 해결

한 객체는 한 가지 책임(임무)를 수행

✓한 객체는 본인의 책임(임무)만 수행한다.

✓로봇청소기는로봇청소기의 임무만을 수행한다.



정상적인 모듈 예시

여러 역할을 하는 예시

이상한 점을 찾아보자.

근로자

- 업무하기
- 근로비용 받기
- 출근하기
- JSON 파싱 (?)
- BlutoothSend (?)

배트맨

- 추적하기
- 전투
- 랭킹비교하기(?)
- 로그화면출력하기 (?)

강아지

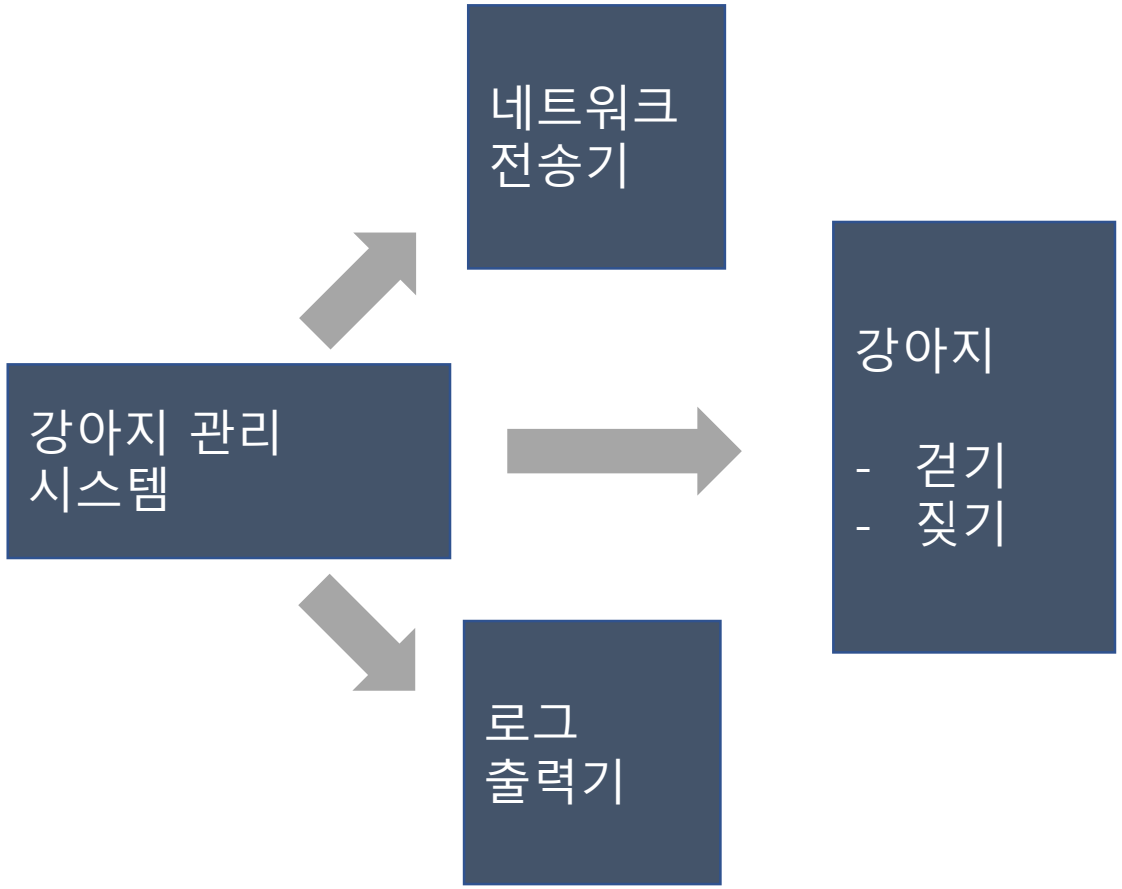
- 걷기
- 짖기
- 로그 화면 출력하기(?)
- 네트워크 전송(?)

책임이 많으면, 분리시키면 된다.

분리 후, 공통인 부분이 많다면 Super Class 추가

강아지

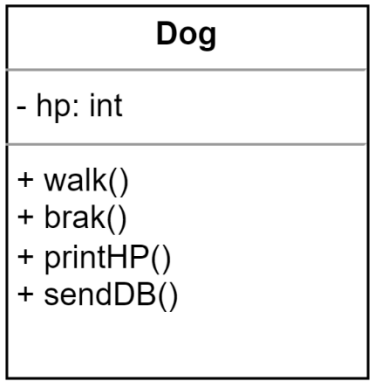
- 걷기
- 짖기
- 로그 화면 출력하기(?)
- 네트워크 전송(?)



[도전] 소스코드 수정해보기

한 가지 객체는 본인의 책임(임무)만을 수행하도록 코드를 수정한다.

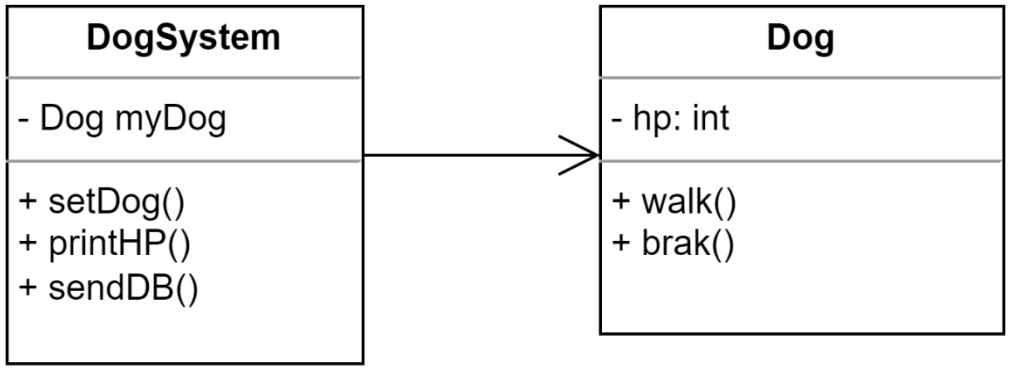
- 본인의 역할이 아닌 것은 다른 Class로 빼둔다.




```cpp
#include <iostream>
using namespace std;

class Dog {
public:
    void Walk() {
        cout << "걷다\n";
    }

    void Bark() {
        cout << "왕왕\n";
    }

    void PrintHP() {
        cout << "HP 값을 화면에 출력합니다\n";
        cout << "HP : " << hp_ << "\n";
    }

    void SendDB() {
        cout << "DB에 데이터 기록\n";;
    }

private:
    int hp_;
};
```

해결방안

클래스를 분리한다.



[도전] step1 : Vehicle

✓SRP를 어긴 항목이 있는지 찾아내고, 이를 개선하자.

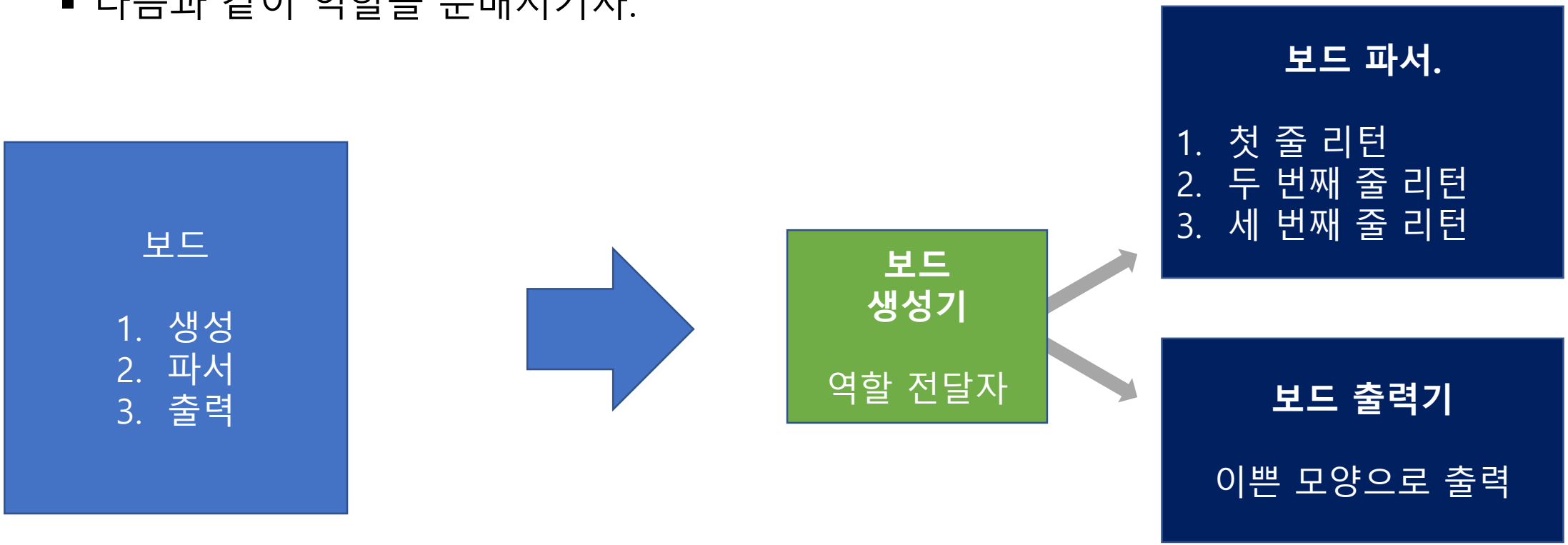
- KATA 스토리상 SRP를 어긴 곳은, 주석으로 표기되어 있음

Vehicle

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

[도전] step2 : TickTackTock

✓틱택톡 보드판의 정보를 읽는 프로그램
- 다음과 같이 역할을 분배시키자.



Open Closed Principle

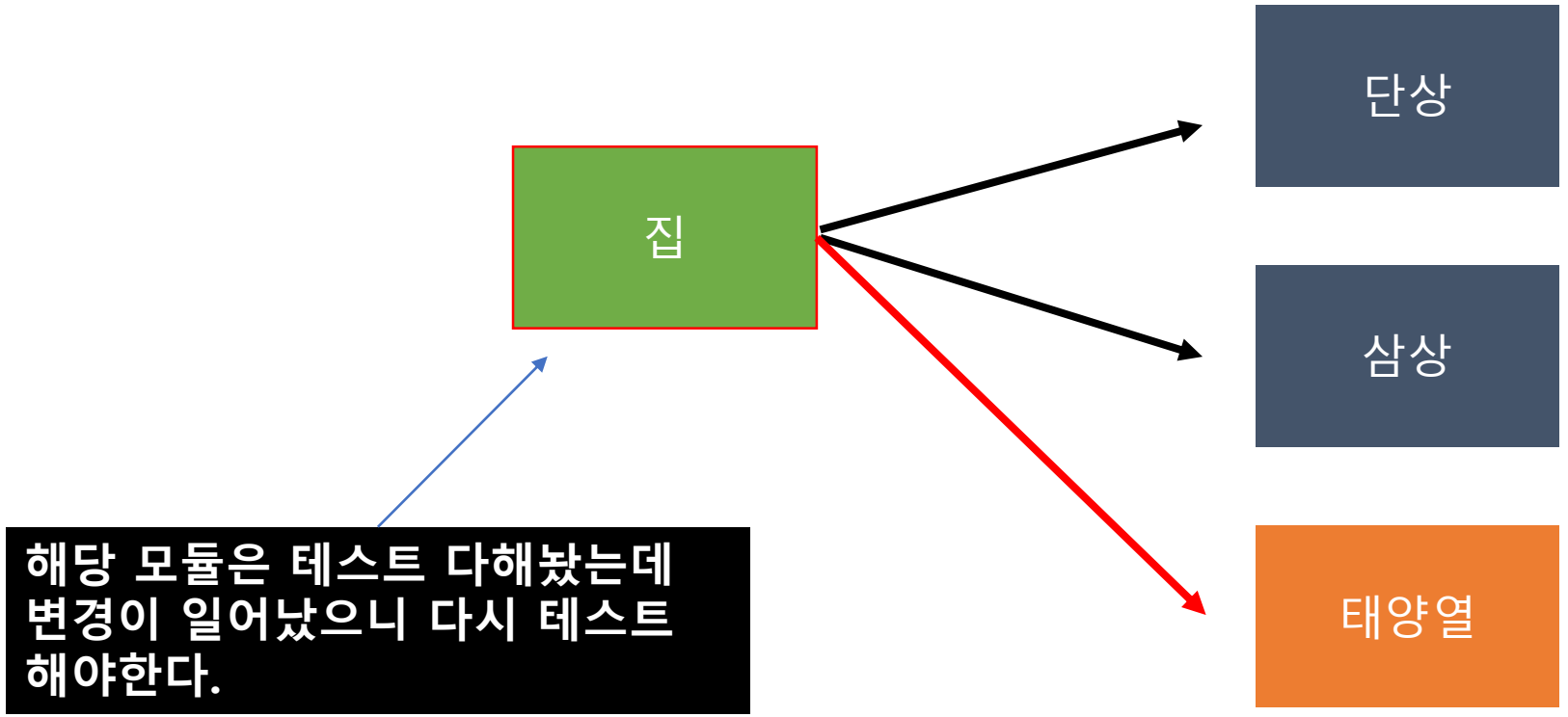
OCP

열린마음 닫힌마음

한 클래스가 다른 여러 클래스 의존하는 경우

✓기능이 하나 더 추가되는 경우, 집 Class에 변경이 일어난다.

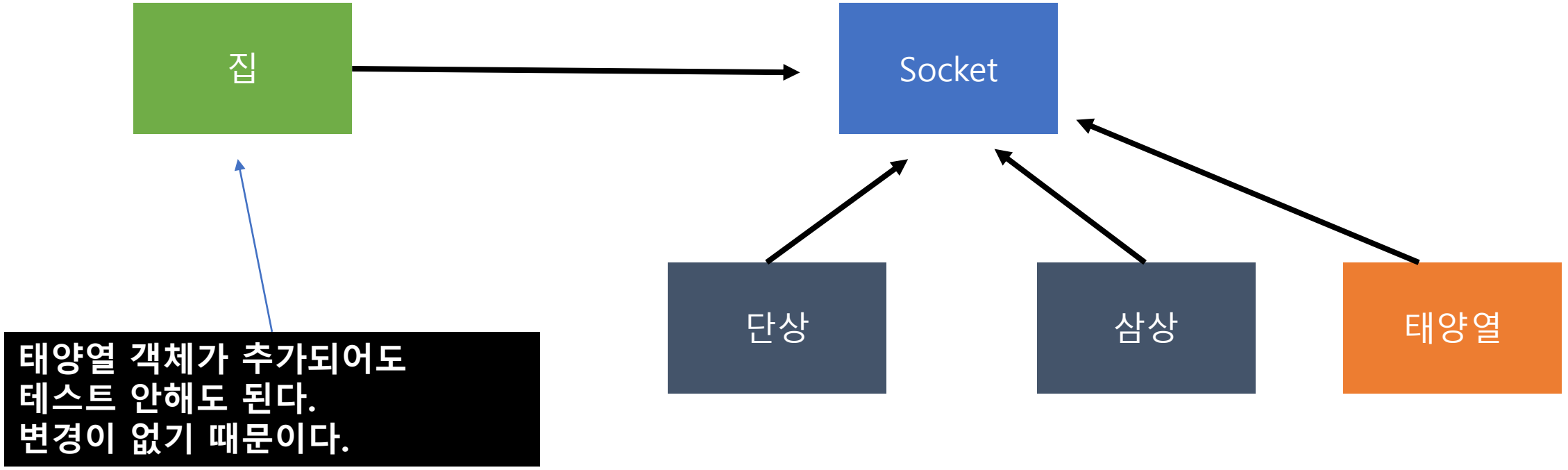
- 태양열 클래스 하나 더 추가시, '집' 객체는 변경이 일어남



Interface를 추가한다.

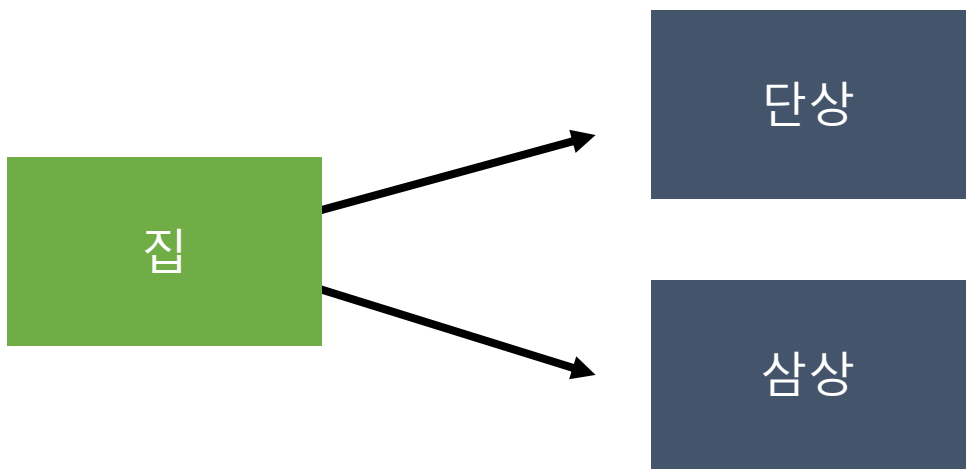
✓Interface를 추가하면, 집 Class는 변경이 없다.

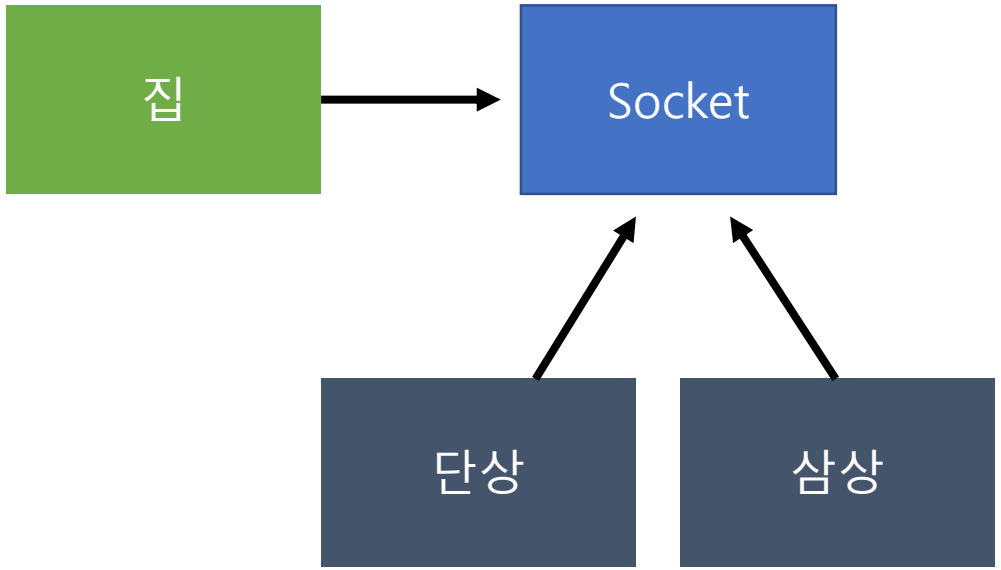
- 태양열이 하나 더 추가되더라도, 집 Class는 변경이 없다.



[도전] 직접 구현해보기

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

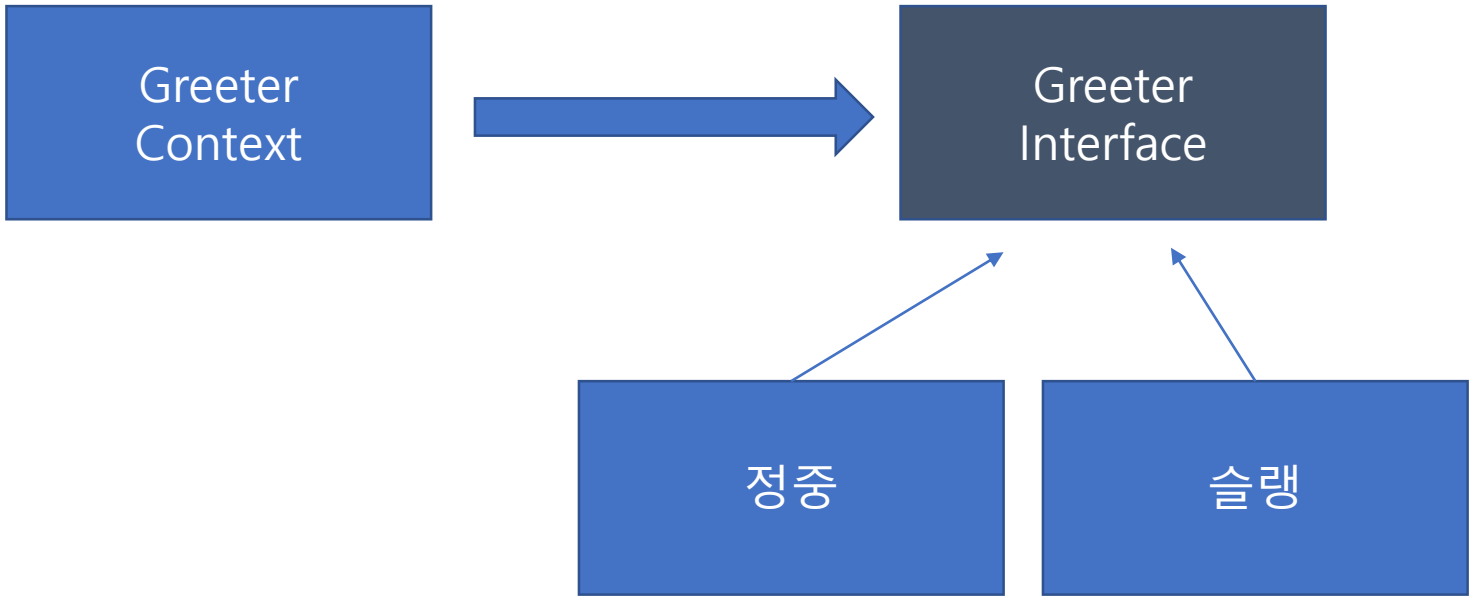




[도전] step1 : Greeter

✓직접 해결해보자.

- new는 client에서 수행!



[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.

Liskov Substitution Principle

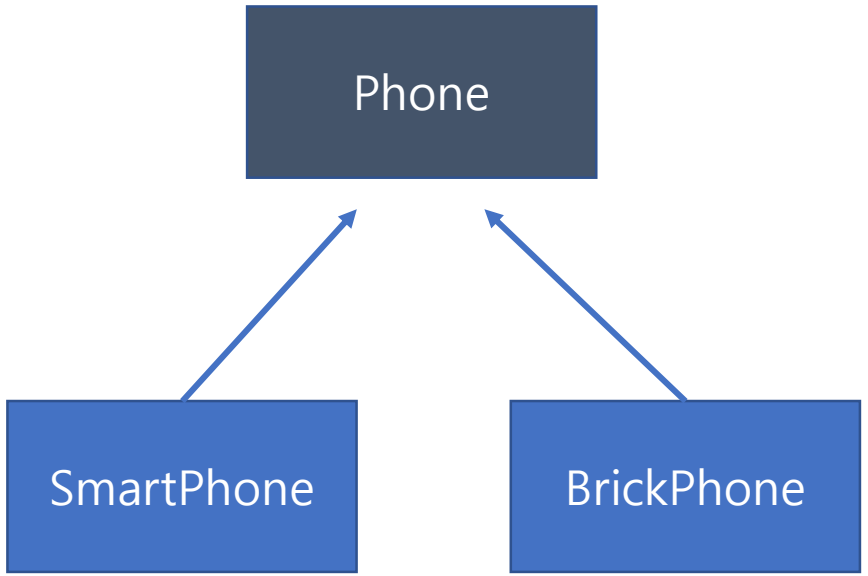
LSP

리스코프 님의 원칙

LSP (리스코프 Principle)

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

- 어떤 sub 클래스가 매개변수로 들어오던지, tryDate 메서드는 잘 동작해야한다.



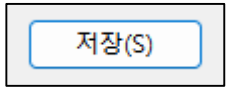

```
class GoodDay {
public:
    void TryDate(Phone *p) {
        p->Call();
    }

    void Calling() {
        TryDate(new SmartPhone());
        TryDate(new BrickPhone());
    }
};
```

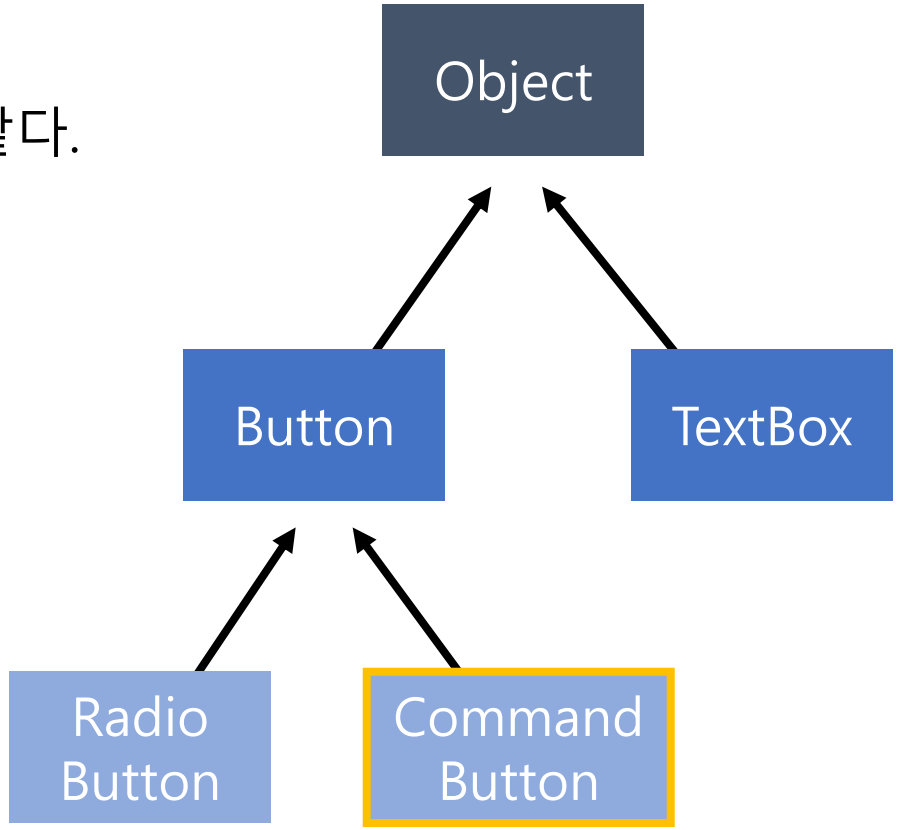
O / X 퀴즈

아래와 같은 getName 이라는 메서드를 확인했다.
그리고 객체들의 상속관계를 확인하였더니 오른쪽과 같다.

그렇다면 Command 객체를 Object에 넣으면
무조건 동작해야 하는가?

```
string GetName(Object *obj) {
    return obj->GetObjectName();
}
```



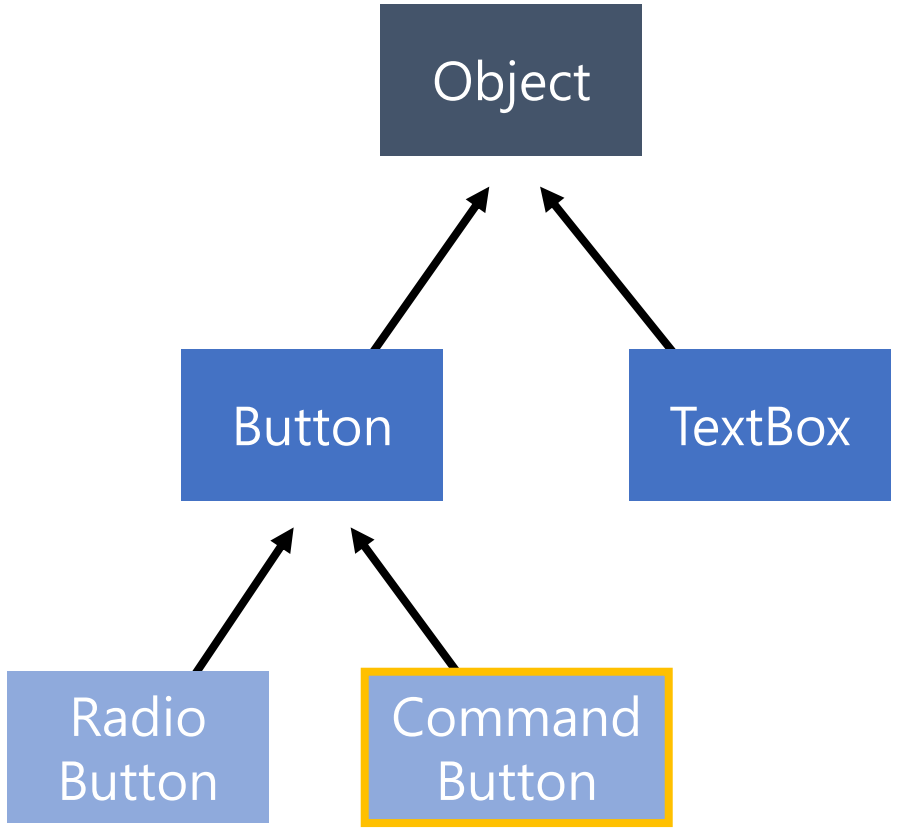


그렇다!

Base로 만들어진 파라미터에, Sub 객체를 넣으면 반드시 동작되어야 한다.

→이것이 LSP 원칙을 지킨 코드

```
string GetName(Object *obj) {
    return obj->GetObjectName();
}

void Run() {
    Command* cmd = new Command("저장(S)");
    cout << GetName(cmd);
}
```



벽돌폰

Call 이 안되는 장난감 BrickPhone

- 기능 추가 요구사항이 들어옴 : BrickPhone

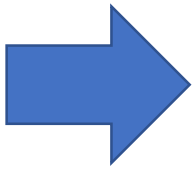
```
class GoodDay {
public:
    void TryDate(Phone *p) {
        p->Call();
    }

    void Calling() {
        TryDate(new SmartPhone());
        TryDate(new BrickPhone());
    }
};
```

```
class Phone {
public:
    virtual void Call() {
        cout << "CALL\n";
    }
};

class SmartPhone : public Phone {
};

class BrickPhone : public Phone {
public:
    void Call() override {
        cout << "ERROR\n";
        exit(1); //Application 종료
    }

    void FakeCall() {
        cout << "FAKE CALLING CALLING\n";
    }
};
```

해결방법 : Server Code 수정하기

오버라이딩 한 메서드가 모두 잘 동작되도록 구현한다.

```
class BrickPhone : public Phone {
public:
    void Call() override {
        cout << "ERROR\n";
        exit(1); //Application 종료
    }

    void FakeCall() {
        cout << "FAKE CALLING CALLING\n";
    }
};
```



```
class BrickPhone : public Phone {
public:
    void Call() override {
        cout << "알림 : FakeCall만 가능하니, Call을 FakeCall로 대신 수행함\n";
        FakeCall();
    }

    void FakeCall() {
        cout << "FAKE CALLING CALLING\n";
    }
};
```

[도전] step1 : Car vs Plane

가정

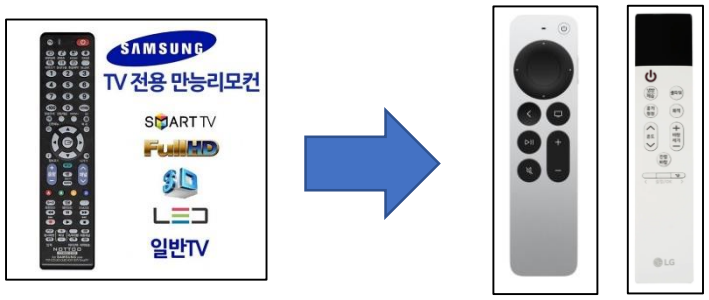
- 자동차는 Drive 모드에서 멈추지 않고 즉시 후진은 불가하다.
- 그리고 후진도중에 즉시 Drive 모드로 전환 불가능하다.
- 그런데 Plain은 Drive 도중 바로 후진이 가능하다.

해결방법

- 불가능한 요청시 자동차를 stop 시킨다.

ISP

Interface Segregation Principle



큰 Interface 보다는
전용 Interface를 선호한다.

Large Interface vs Small Interface 사용

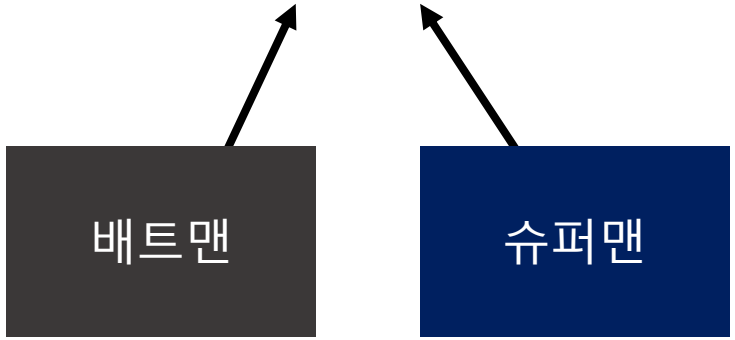
✓어떤 것이 더 좋은 것일까?

- 배트맨 : 걷거나 뛰어다님
- 슈퍼맨 : 걷거나 뛰어다니거나 날아다님

```
#define interface struct
```

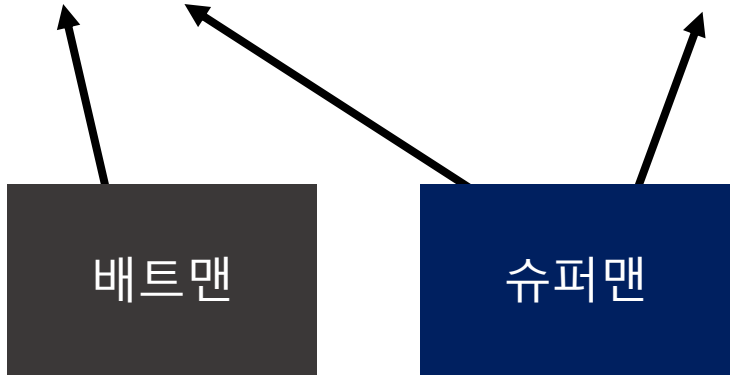
interface 키워드는 struct로 간주한다.

```
interface Move {
    virtual void Walk() = 0;
    virtual void Run() = 0;
    virtual void Fly() = 0;
};
```



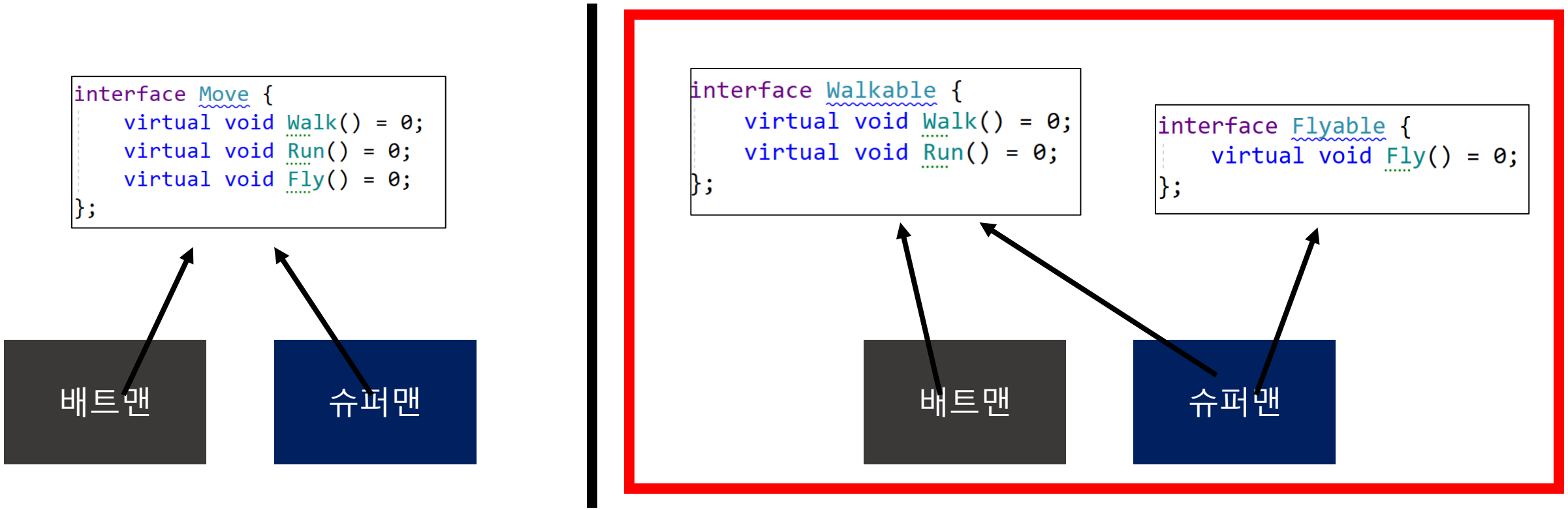

```
interface Walkable {
    virtual void Walk() = 0;
    virtual void Run() = 0;
};
```

```
interface Flyable {
    virtual void Fly() = 0;
};
```



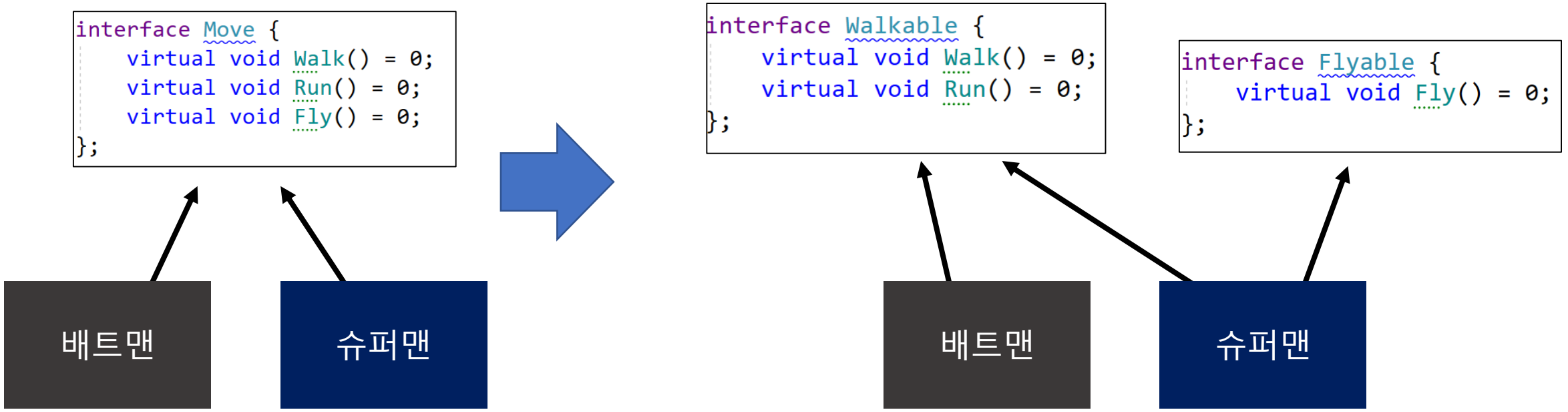
정답은 작은 Interface 사용

배트맨은 fly( ) 가 필요하지 않기 때문



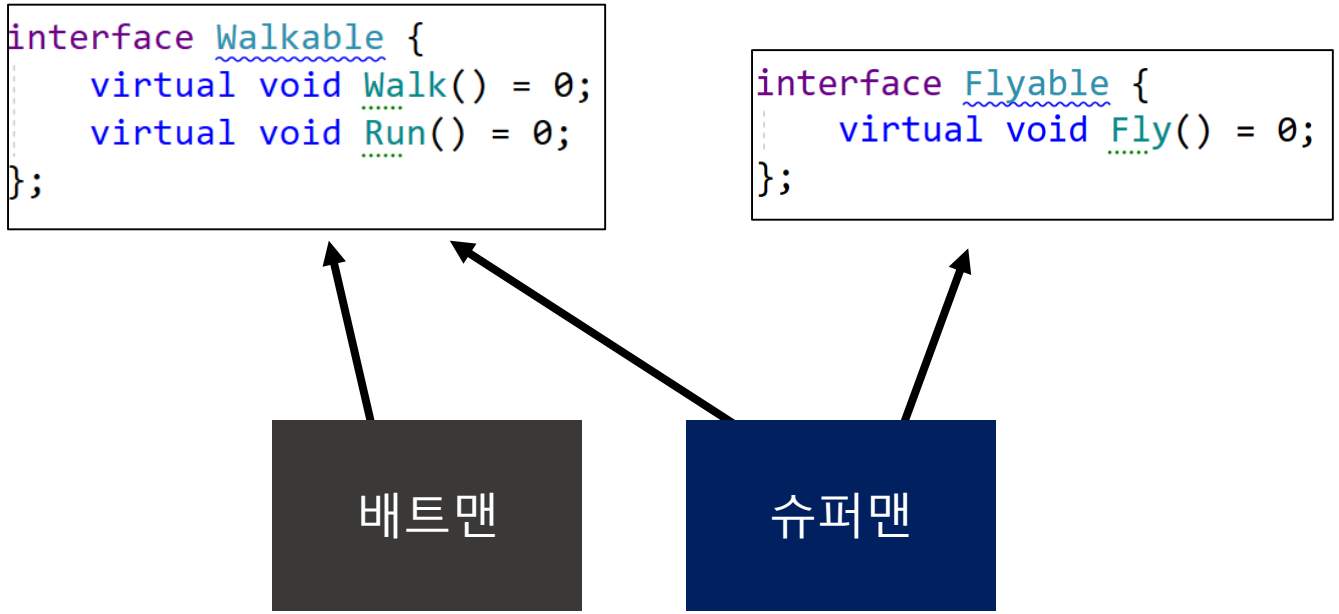
ISP란?

✓사용하지 않을 메서드를 의존하게 하지 않도록 Interface를 분리시킨다.



[도전] 직접 구현해보기

Client Code 는 간단히 테스트할 수 있는 코드로 작성한다.



Larget Interface 사용시

배트맨은 Larget Interface 에서 fly 를 사용하지 않음
결국 LSP 을 위배하게 됨

```
interface Move {
    virtual void Walk() = 0;
    virtual void Run() = 0;
    virtual void Fly() = 0;
};
```

```
class Batman : public Move {
public:
    void Walk() override {
        //걷는다.
    }

    void Run() override {
        //뛴다.
    }

    void Fly() override {
        cout << "ERROR";
    }
};
```

```
class Commander {
public:
    void Go(Move* move) {
        move->Fly();
    }

    void Order() {
        Go(new Batman());
    }
};
```

해결방법

작은 Interface 로 쪼갠다.

Interface 다중 상속을 사용한다.

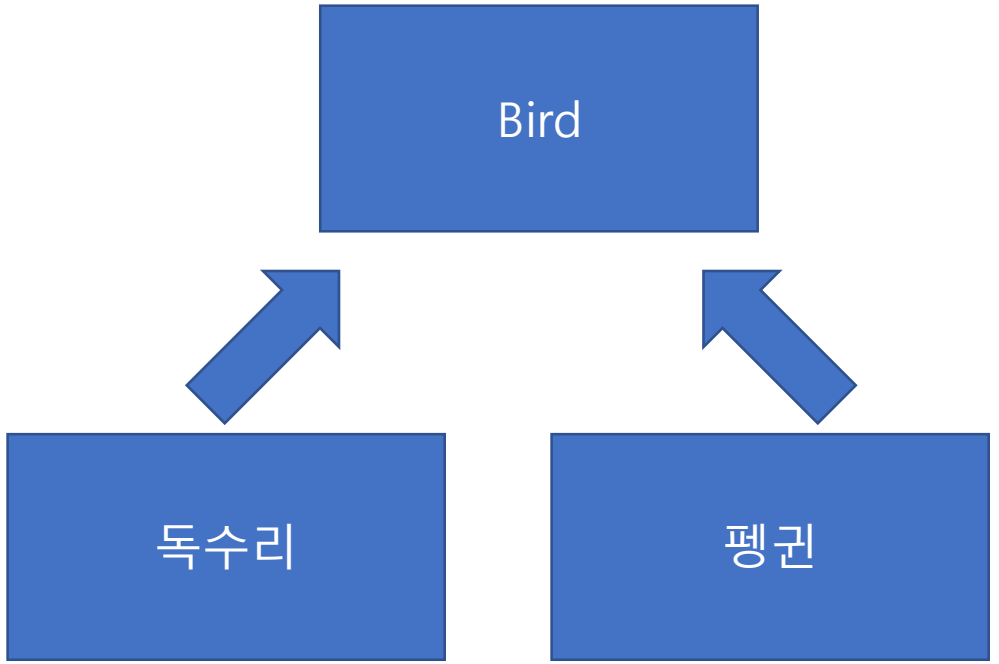
```
class SuperMan : public Walkable, public Flyable {
public:
    void Walk() override {
        //걷는다.
    }

    void Run() override {
        //뛴다.
    }

    void Fly() override {
        //난다.
    }
};
```

```
class Batman : public Walkable {
public:
    void Walk() override {
        //걷는다.
    }

    void Run() override {
        //뛴다.
    }
};
```

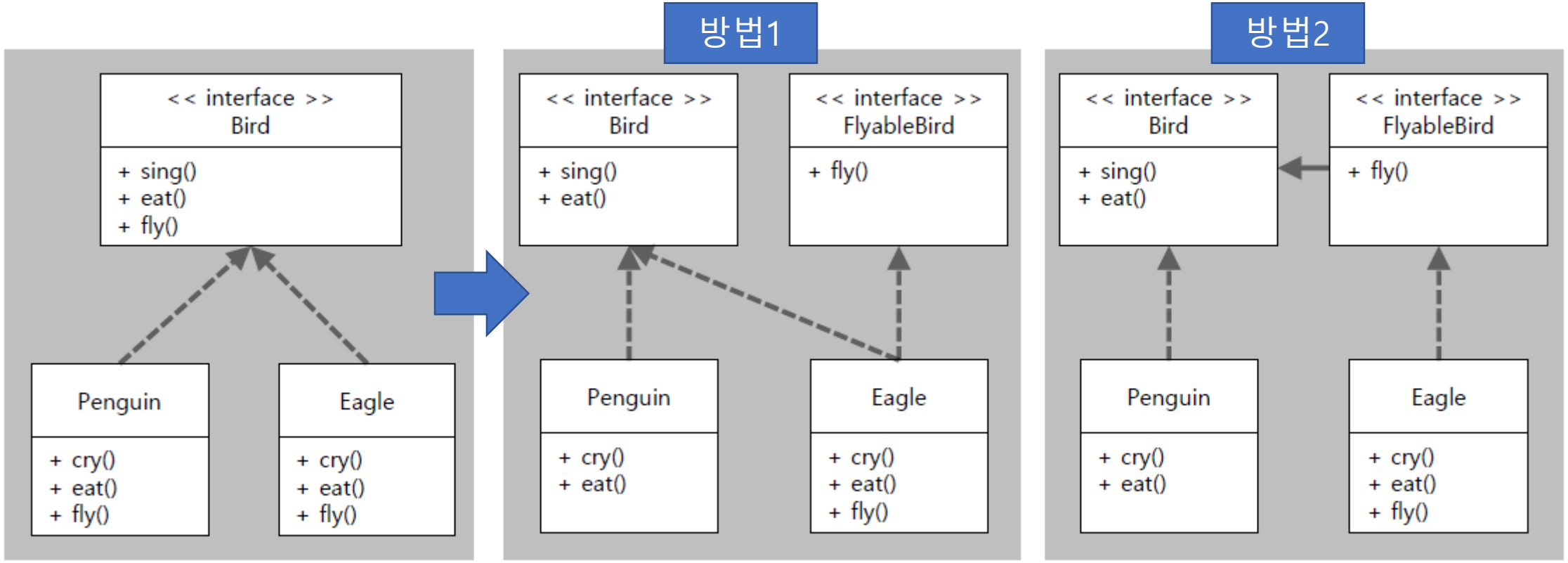
```
class Commander {
public:
    void GoWalk(Walkable* move) {
        move->Walk();
    }

    void GoFly(Flyable* move) {
        move->Fly();
    }

    void Order() {
        GoFly(new SuperMan());
        GoWalk(new Batman());
    }
};
```

[도전] step1 : 팽귄과 독수리

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



두 가지 해결방법


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

[도전] step2 : 자동차와 드론

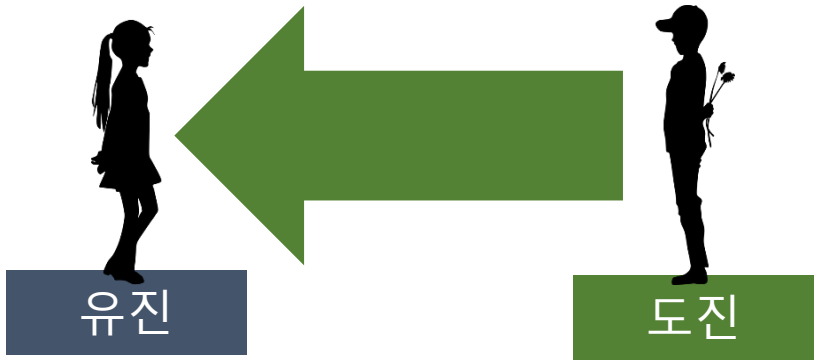
✓자동차 (Vehicle 상속받음)
- 라디오 ON/OFF 기능 있음
- 카메라 ON/OFF 기능 없음

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

거대한 Vehicle Interface를
분할해보자.

Dependency Inversion Principle

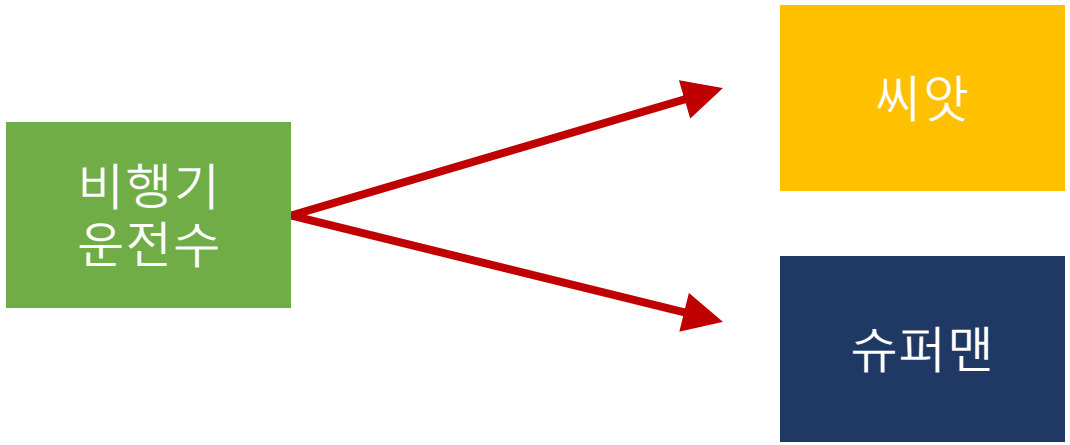
DIP



의존관계 역전

객체들의 의존성 관계

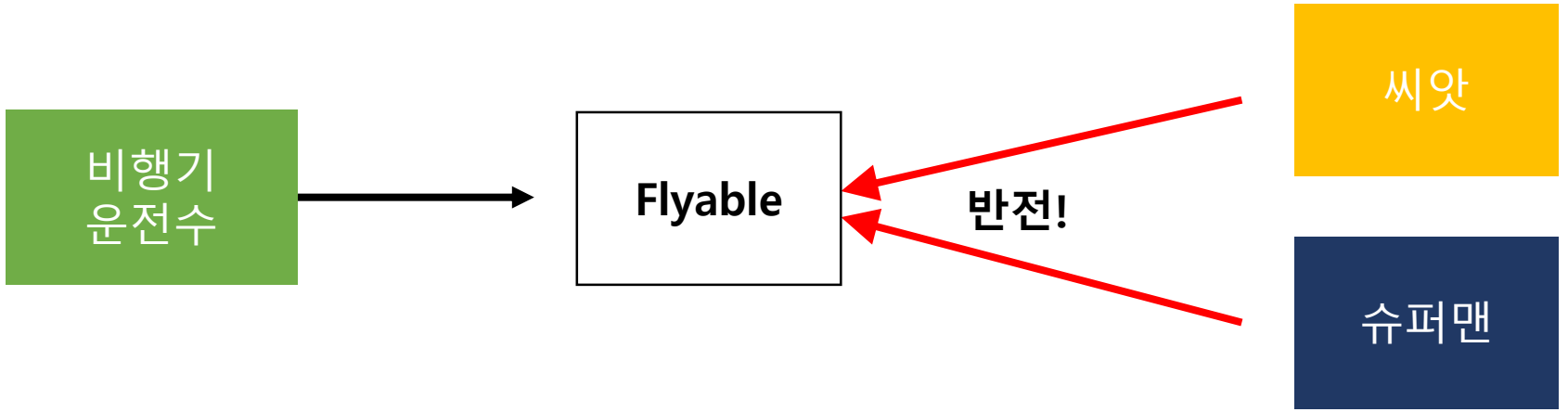
한 모듈이 씨앗과 슈퍼맨에 의존성을 갖는다.



의존성 방향을 반전시킨다.

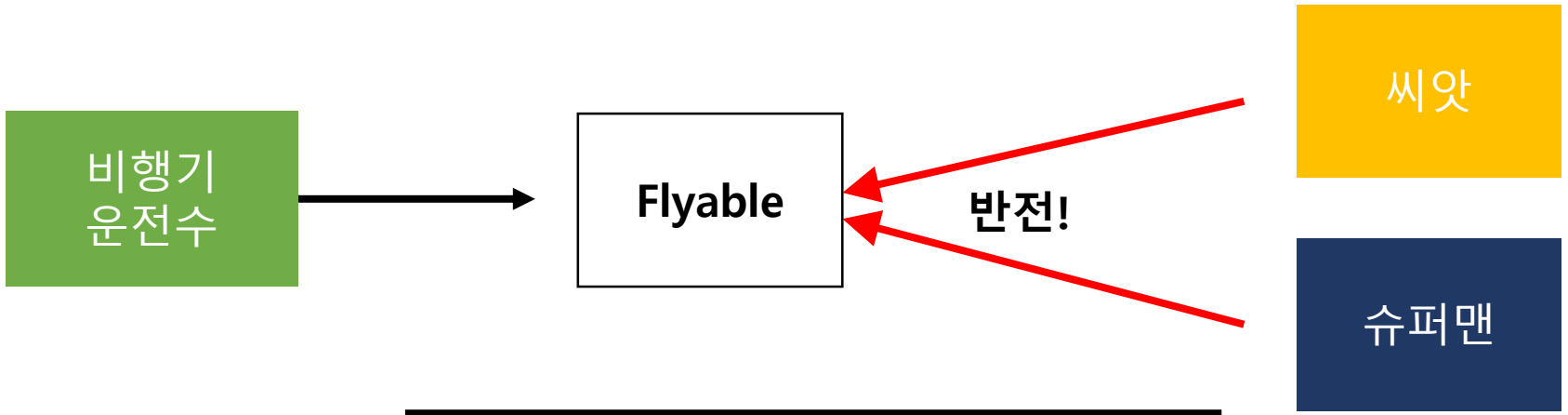
씨앗과 슈퍼맨은 Flyable Interface에 의존성을 갖도록 반전한다.

의존성을 반전시켜, 각 모듈들은 Interface에 의존하도록 하는것을 Dependency Inversion 원리라고 한다.



왜 Inversion 시키는 것일까?

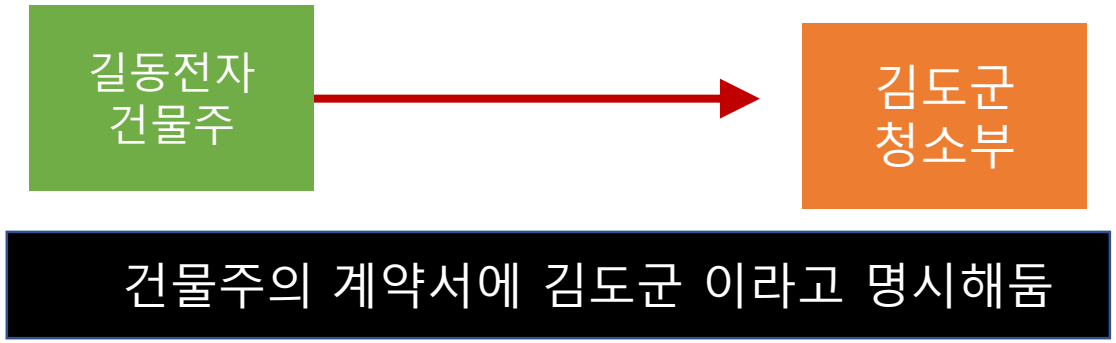
✓Interface를 두고, 의존성 방향을 Inversion 시키는 이유는?



Interface를 두는 것이
왜 의존성을 낮추는 행동일까?

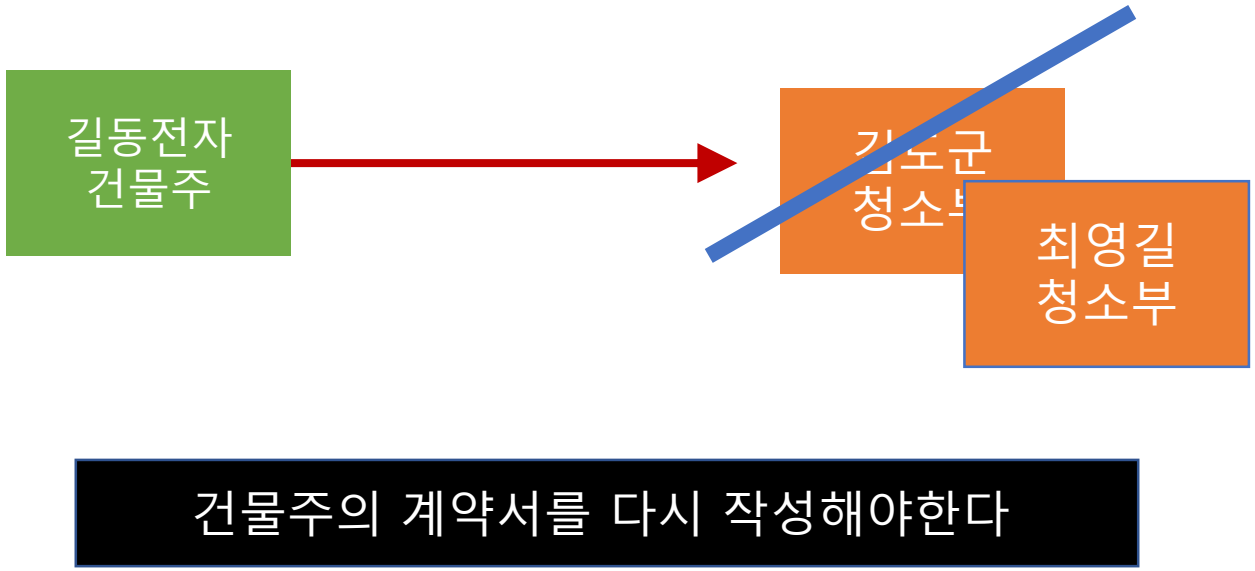
Interface를 두는 것이
왜 의존성을 낮추는 행동인지 알아보자.

한 객체가, 특정 객체를 의존한다.



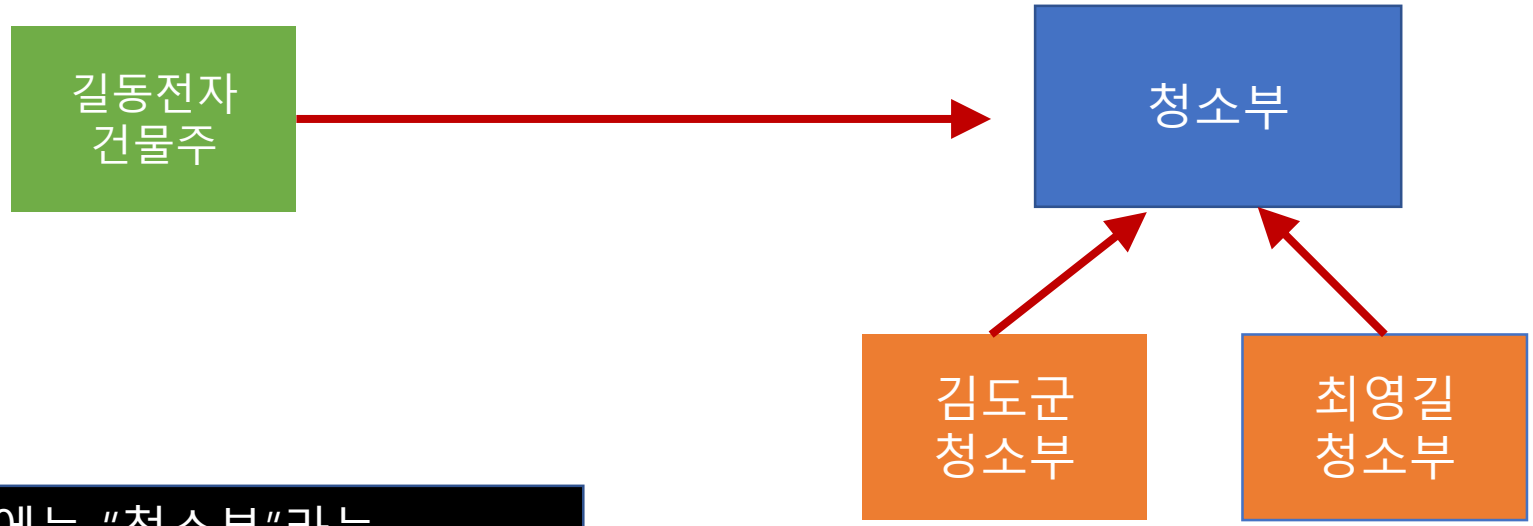
객체들의 의존성 관계

특정 객체에만 강한 의존성을 보일 때,
특정 객체에 변경이 발생하면, 건물주에도 변경의 가능성이 크다.



객체들의 의존성 관계

만약 특정 객체가 아닌,
청소부라는 '추상화된 객체'에 의존성을 보이도록 개발을 했다고 가정한다.
그룹에 속한 특정 객체의 변경으로 인해, 영향을 끼칠 가능성이 줄어든다.



건물주의 계약서에는 "청소부"라는
역할을 적어두면, 청소부가 변경해도
계약서에는 변경이 없다.

DIP vs OCP

의존성을 Interface에 두는 것을 DIP 라고 한다.

DIP와 OCP는 소스코드가 같지만, 개념적으로 차이가 있다.

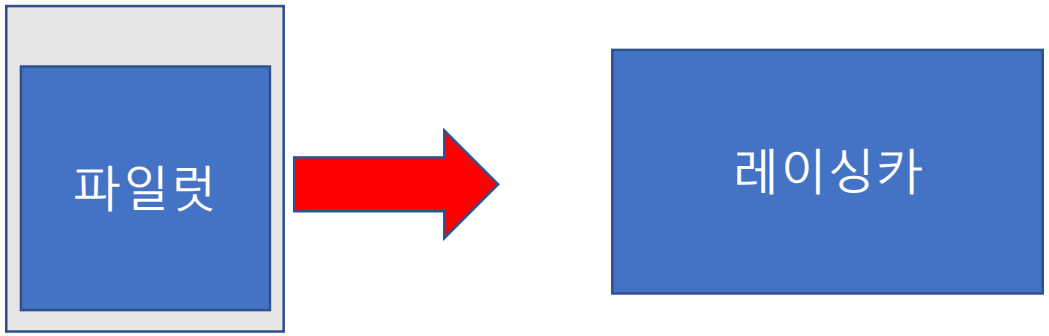
- DIP : Interface를 두어, 강한 의존성을 낮추는 것에 포커스
- OCP : **확장성을 고려하여**, 기능 추가에 모듈 변경을 최소화

DIP가 OCP를 포함한 개념

[도전] step1 : 강한 결합 낮추기

✓**파일럿은 태어나자마자 특정 레이싱카에 의존성을 강하게 보인다.**

- Dependency를 역전시켜, 의존도를 낮춰보자.



[도전] step2 : Notifier로 의존성 역전하기

인터페이스를 두어, 의존성을 낮추어 보자.