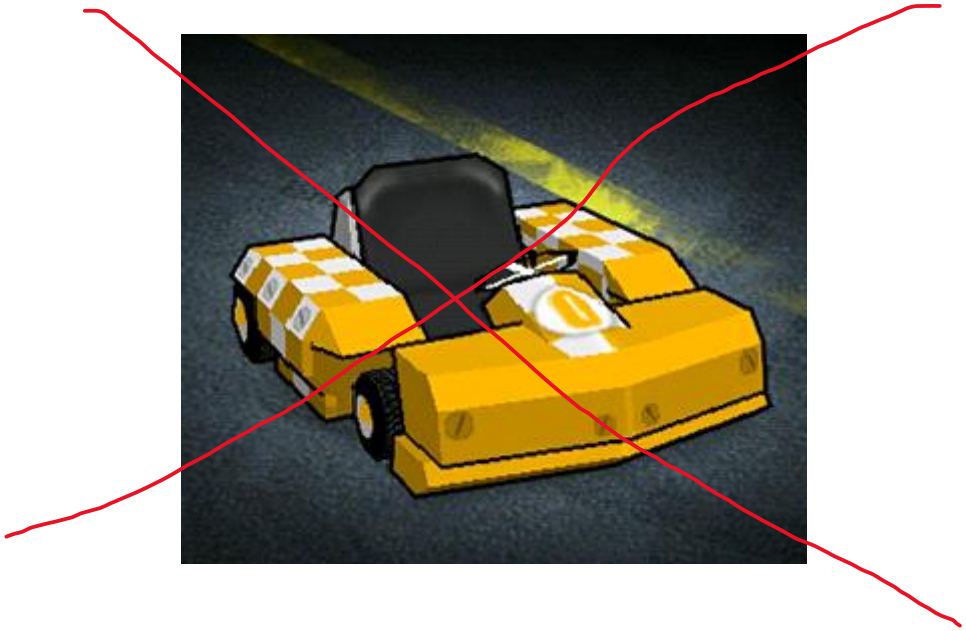


@daebalprime

객체지향 SOLID 원칙

객체지향 SOLID 원칙이란?

어느정도 입증된 객체지향 디자인 원칙으로써,
유지보수하기 쉽고 확장이 쉬운 소프트웨어를 만들 수 있도록 돕는 원칙.

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Single Responsibility Principle, 단일 책임 원칙

Single Responsibility Principle, 단일 책임 원칙

정의 : 하나의 클래스엔 하나의 책임만 부여한다.

- 클래스가 제공하는 모든 서비스는 단 하나의 책임을 수행하는데 집중되어야 한다.
- 결합도를 낮추고 응집도를 높이는 원칙.

어떻게 적용하는가?

- 단순하게 여러 책임을 맡고 있는 클래스를 여러 클래스로 분할하기(예제)
- 분리하는 두 클래스의 유사도가 높다면, Superclass를 만들어 상속받아 구현하기
- 클린 코드에서는 함수는 작게, 매개 변수 리스트는 짧게 하라고 권고하고 있습니다. 이렇게 작게작게 쪼개다 보면 일부 메서드만 사용하는 인스턴스 변수가 많아지는데, 이는 클래스를 쪼개야 함을 의미한다.
- 하나의 책임이 여러 클래스에 나눠지는 것도 한 곳에 모아야 한다.(Shotgun Surgery)

Single Responsibility Principle, 단일 책임 원칙

응집도(Cohesion)

- **클린 아키텍쳐 :** 모듈 요소간의 기능적인 연관
- **한국정보통신기술협회 IT용어사전 :** 하나의 프로그램을 구성하는 각각의 모듈이 그 고유의 기능을 잘 처리할 수 있는지를 나타내는 정도.
- **클린 코드 :** 클래스 메서드는 클래스 인스턴스 변수를 하나 이상 사용해야 한다. 일반적으로 메소드가 변수를 더 많이 사용 할 수록 메서드와 클래스의 응집도가 더 높다. . . . (중략) . . . 응집도가 높다는 말은 클래스에 속한 메소드와 변수가 서로 의존하며 논리적인 단위로 묶인다는 의미기 때문이다.(클린코드 177쪽)

결합도(Coupling)

- 프로그램 구성 요소들 사이가 얼마나 의존적인지 나타내는 척도. 쉽게 말해서 어떤 클래스를 수정할 때 다른 클래스를 얼마나 수정해야 하느냐를 의미.

Single Responsibility Principle, 단일 책임 원칙

```
class Transportation {
    String type;
    //...
    public void fly(){};
    public void lock(){};
}

Transportation airplane = new Transportation();
Transportation bicycle = new Transportation();

bicycle.lock(); // 자물쇠 잠구기
airplane.fly(); // 날기

bicycle.fly(); // ???
airplane.lock(); // ???

/*
Transportation은 지구상 존재하는 모든 운송수단을 커버한다.
bicycle.fly()는 개념적으로 존재하지 않지만, 단일 클래스가
너무 많은 책임을 지고 있다면 고려하지 못한 운송수단이
날게 되거나 혹은 그 처리를 위해 지나친 분기문을 사용하여야 할 것이

너무 많은 책임을 지고 있기 때문에 별도의 클래스로 구현하거나,
공통적인 특성만을 추상화 클래스에 남기고 각각 상속하여 구현하자.
*/
```

Single Responsibility Principle, 단일 책임 원칙

```
class Transportation {
    String type;
    int capacity;
    int size;
}

class Bicycle extends Transportation {
    public void lock();
}

class Airplane extends Transportation {
    public void fly();
}
```

Open-Closed Principle, 개방-폐쇄 원칙

정의 : 확장에 대해서는 개방적이고, 수정에 대해서는 폐쇄적이어야 한다.

- 다형성(Polymorphism)의 원칙을 적극 이용하자.
- 수정에 폐쇄적이어야 하는 이유는 자그마한 수정사항이 생겨도 해당 클래스에 의존하거나 호출하는 모든 클래스들을 재컴파일 해야하고 큰 프로젝트에서는 시간을 많이 소모할 것

어떻게 적용하는가?

- 예시를 보며 설명드리겠습니다.

Open-Closed Principle, 개방-폐쇄 원칙

```
class Rectangle {
  int x, y, size;
  double angle;
  //...
}
class Drawer {
    public void draw(Rectangle r){...}
    //....
}

ArrayList<Rectangle> arr = new ArrayList<>();

for(Rectangle r : arr) {
    Drawer.draw(r);
}

// 만약 사각형 뿐 아니라 삼각형, 원, 별 등 여러가지 도형을
//그리고 싶다면?
// 코드 수정해야 할 부분이 많아진다.

class Shape {};
class Rectangle extends Shape{};
class Star extends Shape{};
class Circle extends Shape{};
class Triangle extends Shape{};

class Drawer {
    public void draw(Shape r){...}
    //....
}

ArrayList<Shape> arr = new ArrayList<>();

for(Shape r : arr) {
    Drawer.draw(r);
}
```

Open-Closed Principle, 개방-폐쇄 원칙

그러나 항상 OCP 원칙을 따르는게 만능은 아니다.

1. 개발비용 상승(추상화 클래스 별도 작성)
2. 인지과학적인 측면에서 개발자가 감당할 수 있는 추상화 수준은 정해져있다.

=> 자주 변경될 것 같은 요소를 추측하고, 선별적으로 원칙을 적용하기.

Liskov Substitution Principle, 리스코프 치환 원칙

정의 : 어떤 클래스를 상속받은 하위 클래스의 인스턴스는, 상위 클래스의 인스턴스를 대체할 수 있어야 한다.

Liskov Substitution Principle, 리스코프 치환 원칙

```java
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override public boolean equals(Object o) {
        if (o == null || o.getClass() != getClass())
        //getClass()는 해당 인스턴스의 클래스에 대한 메타데이터를 가지고 있다.
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }
}

public class CounterPoint extends Point {
    private static final AtomicInteger counter =
            new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated() { return counter.get(); }
}

Set<Point> unitCircle = Set.of(
        new Point( 1, 0), new Point( 0, 1),
        new Point(-1, 0), new Point( 0,-1));
unitCircle.contains(new CounterPoint(0,1)); // false

// CounterPoint와 Point의 getClass()는 다른 값을 반환한다.
```

CounterPoint는 Point 인스턴스를 대체하지 못한다.
LSP 위반

이펙티브 자바에서 컴포지션 등을 소개

Interface Segregation Principle, 인터페이스 분리 원칙

정의 : 클라이언트는 자신이 이용하지 않는 기능에 영향받지 않아야 한다.

- 프린터, 스캐너, 복사가 모두 가능한 프린터를 이용할 때, 스캔을 떠서 그림파일로 바꾸고자 하는 사람은 잉크가 떨어졌는지, 기계적인 고장은 없는지 등에 신경을 쓰지 않고 영향을 받아서는 안된다.

예시

- CRUD(Create, Read, Update, Delete)를 지원하는 흔한 커뮤니티 게시판을 가정
- 비회원, 회원, 관리자의 권한은 각각 다를 것
- 비회원 클라이언트와 관리자 클라이언트가 모든 기능을 담고 있는 인터페이스를 공통으로 사용한다면 ISP 원칙 위반.
- 비회원 전용 인터페이스에는, 오로지 글 읽기 메서드만 제공하면 된다.
- 관리자 전용 인터페이스는 모든 메서드를 담아야 한다.

Dependency Inversion Principle, 의존 역전 원칙

정의 : 의존 관계를 맺을 때 자주 변화하는 것보다는 변화가 없는 것에 의존

- 구체 클래스에 의존하는 대신에, 추상 클래스나 인터페이스와 관계를 맺자.
- 아래 원칙을 따르되 너무 과몰입하지는 말자.

1. 어떤 변수도 구체 클래스에 대한 포인터나 참조 값을 가져서는 안된다.
2. 어떤 클래스도 구체 클래스에서 파생되어서는 안된다.
3. 어떤 메서드도 그 기반 클래스에서 구현된 메서드를 오버라이드해서는 안된다.

- 구체 클래스가 변경될 일이 거의 없으면서 유사한 파생 클래스도 만들어지지 않을 것이라면, 구체 클래스에 의존하는게 문제가 될까?
- 원칙을 위한 원칙 대신에, 유지보수와 코드 재사용성을 높이는 시각에서 원칙을 따르자.

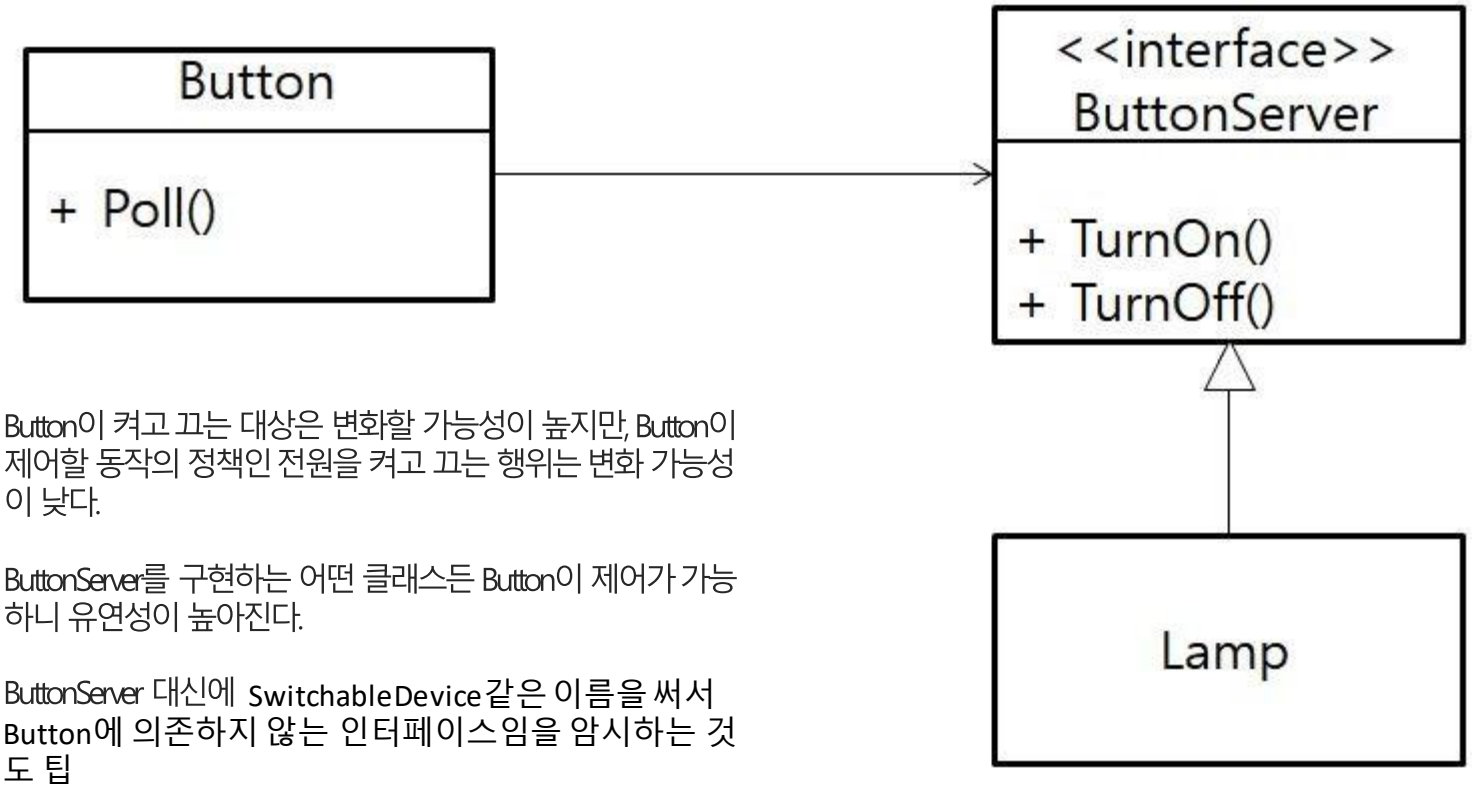
Dependency Inversion Principle, 의존 역전 원칙


```
public class Button{
    private Lamp itsLamp;
    public void poll(){
        if (/* 어떤 조건 */){
            itsLamp.turnOn()
        }else{
            itsLamp.turnOff()
        }
    }
}
```

Button 클래스는 Lamp 클래스에 의존

만약 Button 클래스가 제어하고자 하는 대상이 Lamp 뿐 아니라 Electricity, Computer 등 여러가지라면?

Dependency Inversion Principle, 의존 역전 원칙



Button이 켜고 끄는 대상은 변화할 가능성이 높지만, Button이 제어할 동작의 정책인 전원을 켜고 끄는 행위는 변화 가능성이 낮다.

ButtonServer를 구현하는 어떤 클래스든 Button이 제어가 가능하니 유연성이 높아진다.

ButtonServer 대신에 SwitchableDevice 같은 이름을 써서 Button에 의존하지 않는 인터페이스임을 암시하는 것도 팁

ButtonServer는 이름만 보면 Button 클래스와 관계를 맺을 것 같지만, 실질적으로는 이를 구현하는, 버튼에 의해 통제받을 객체의 소유이다. (소유의 역전)

Reference

https://siyoon210.tistory.com/88
https://server-engineer.tistory.com/224
https://m.blog.naver.com/jwyoon25/221615569649
Effective Java 3/E, Joshua bloch
https://velog.io/@kyle/객체지향-SOLID-원칙-이란