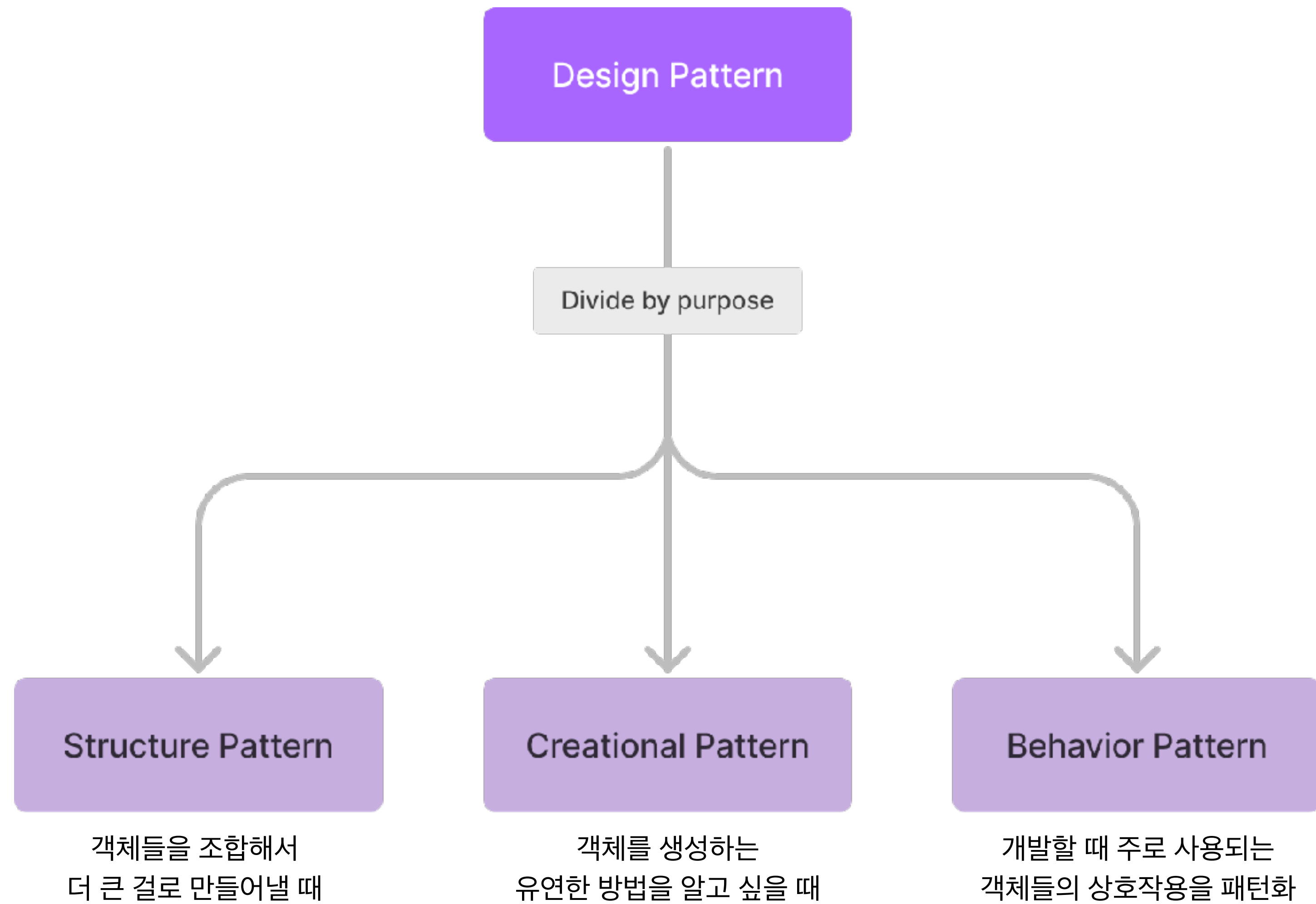


Deco, Adap

structure pattern을 곁들여서..

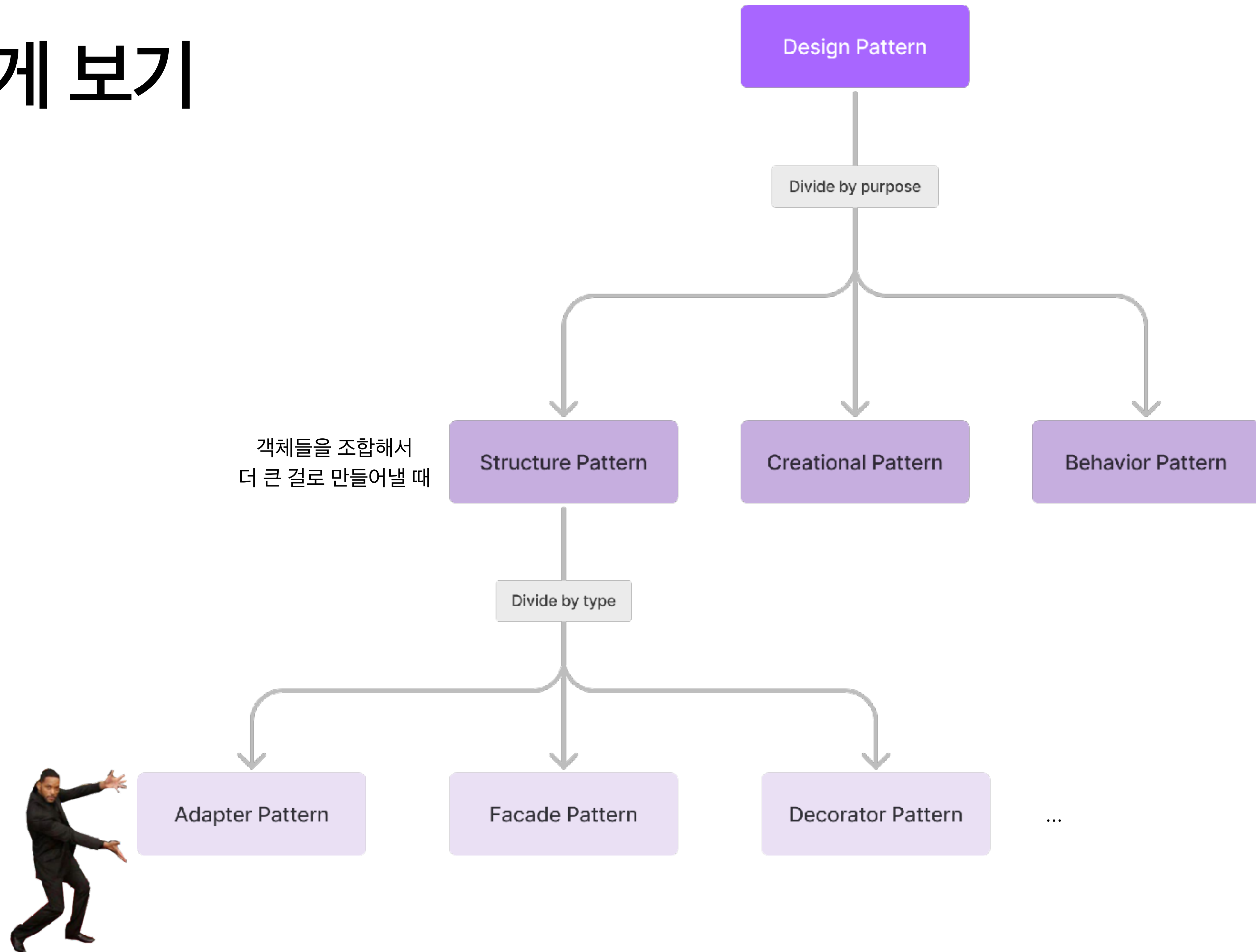
0. 크게 보기



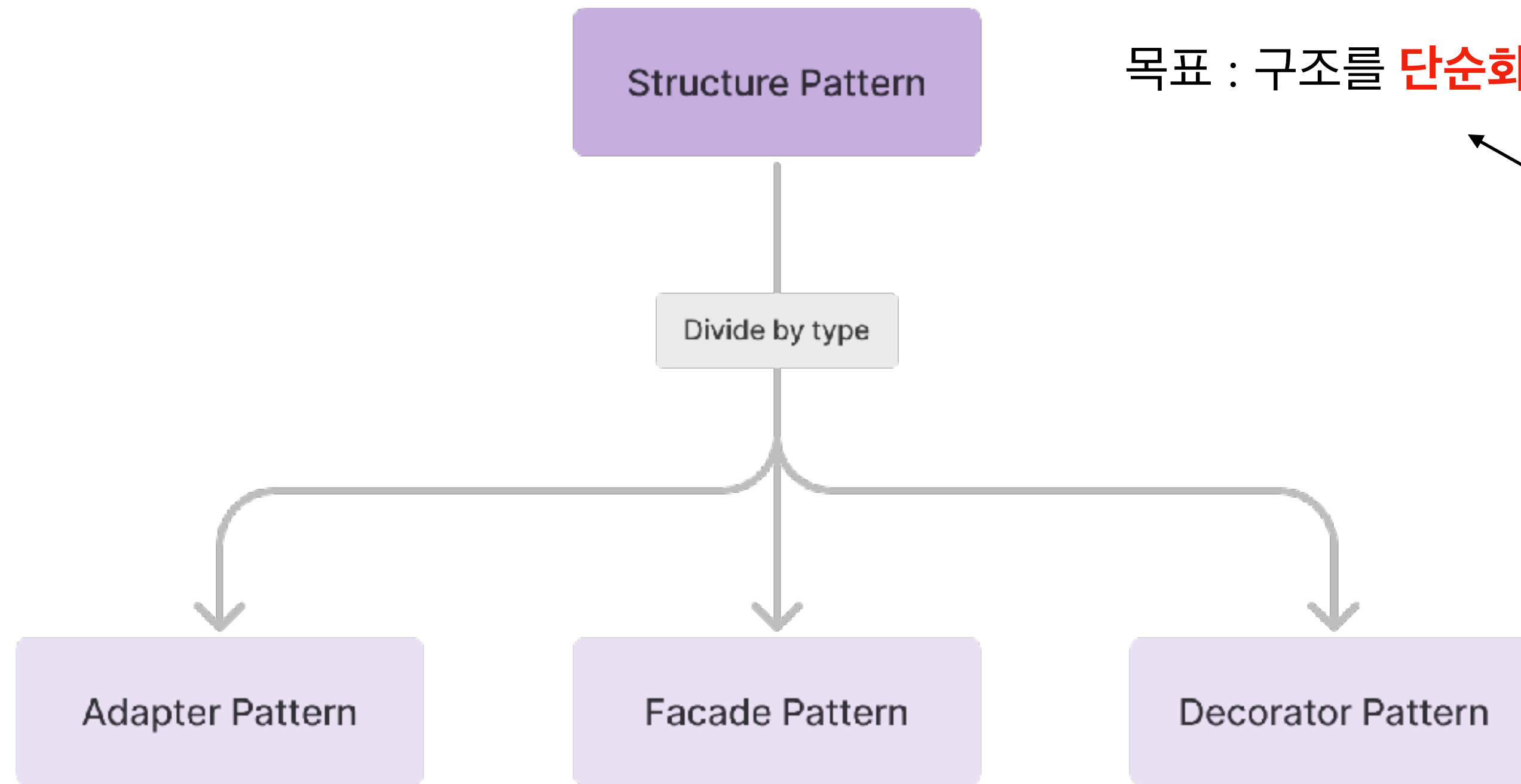
???



0. 크게 보기



0. 크게 보기



목표 : 구조를 **단순화**하는 것 (유연성 챙기면서)

HOW?

클라이언트 쪽에서 사용하는 여러 객체들 다
구체적으로 알게하지 말자

➡ 추상화된 형태로 접근하게 하자

HOW?

그 방법론에 관한 이야기가
Decorator, Facade, Adapter Pattern

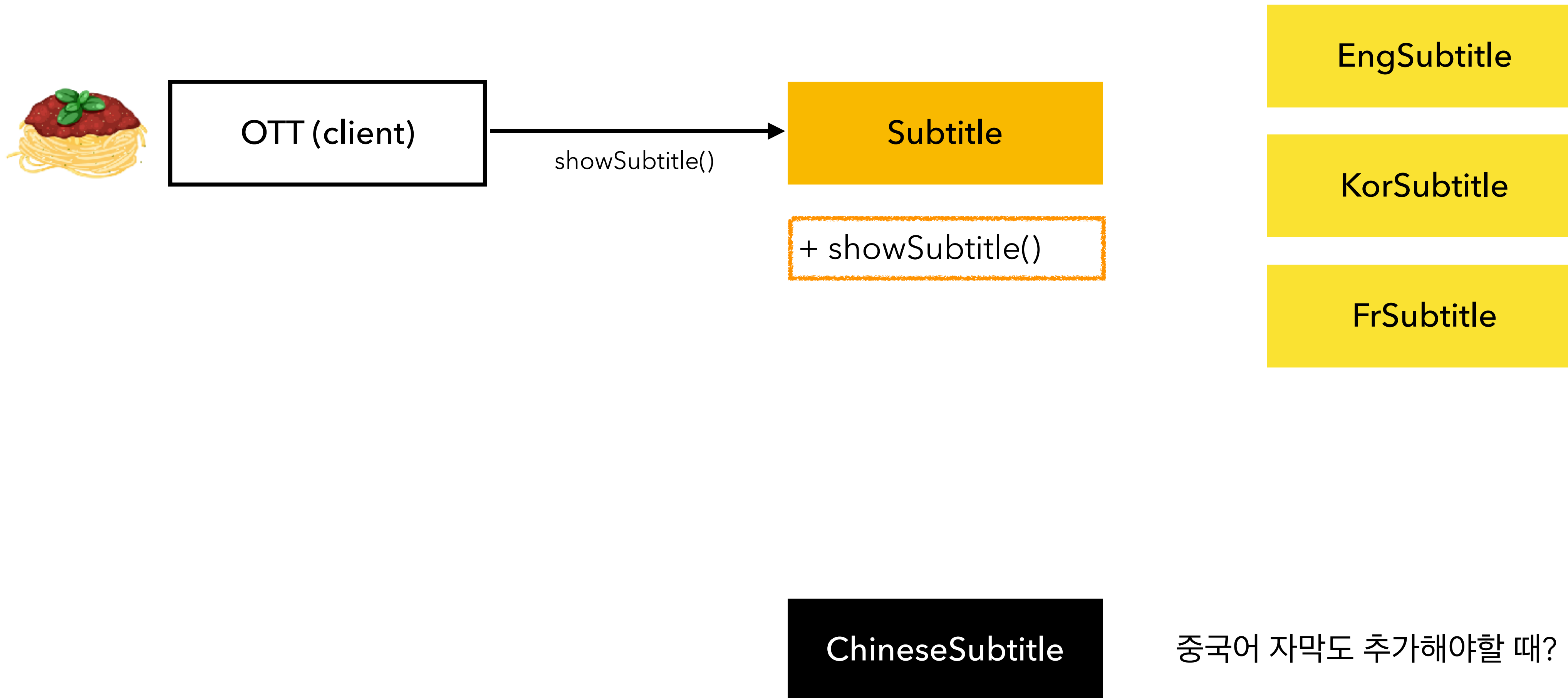
1. Adapter Pattern



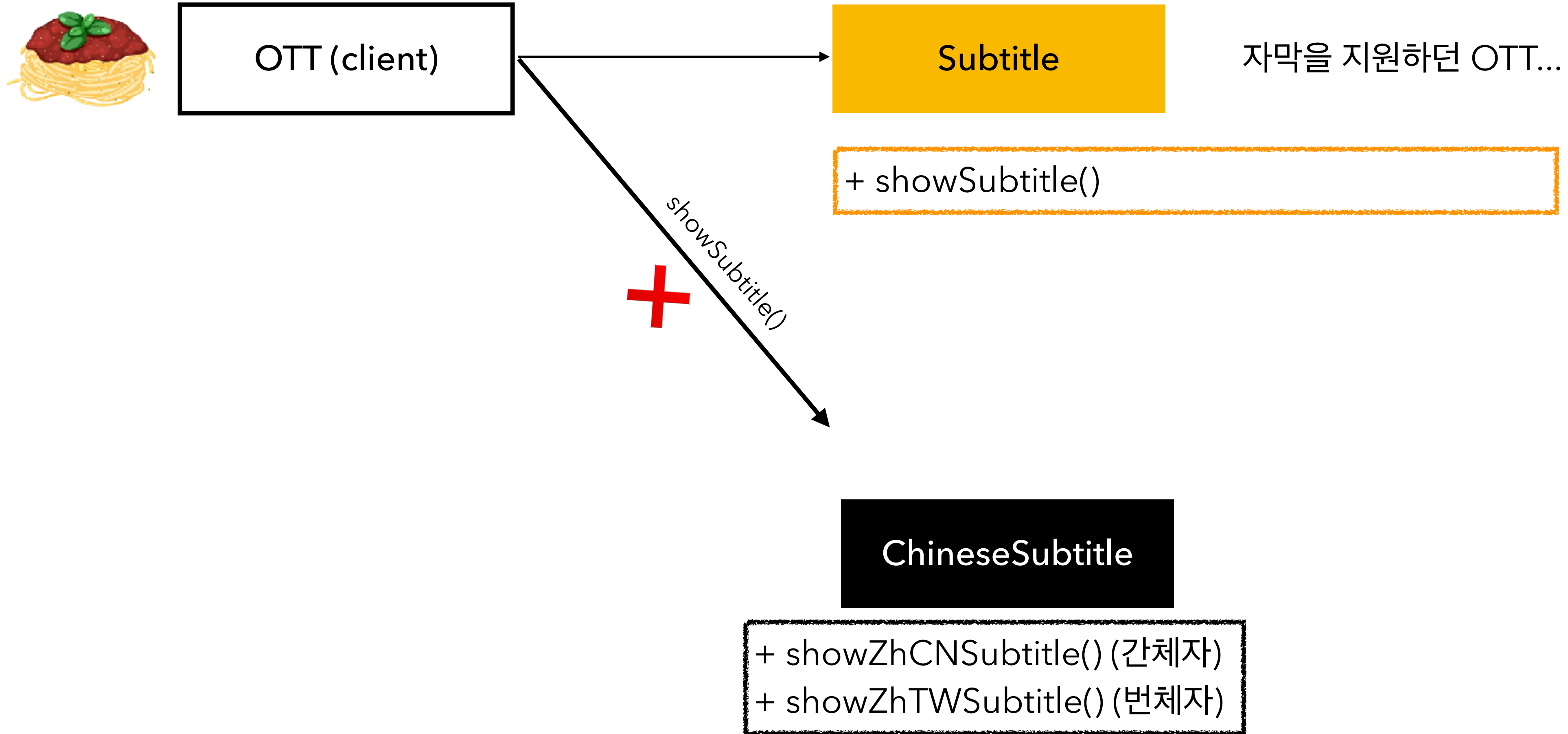
Adapter

:: 원래 사용하던 규격에 맞지 않는 규격도 사용하고자 할 때!

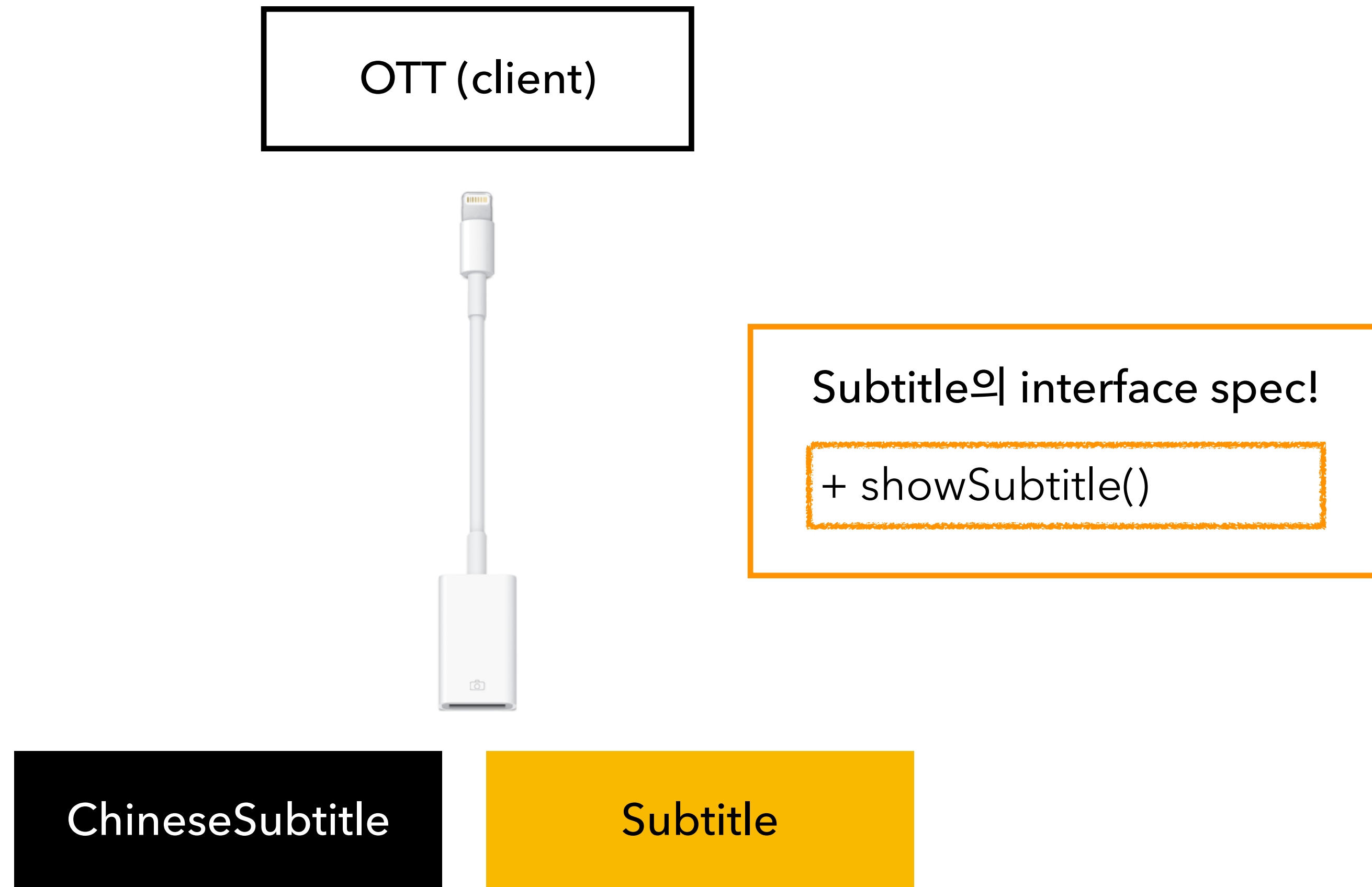
1. Adapter Pattern



1. Adapter Pattern



1. Adapter Pattern

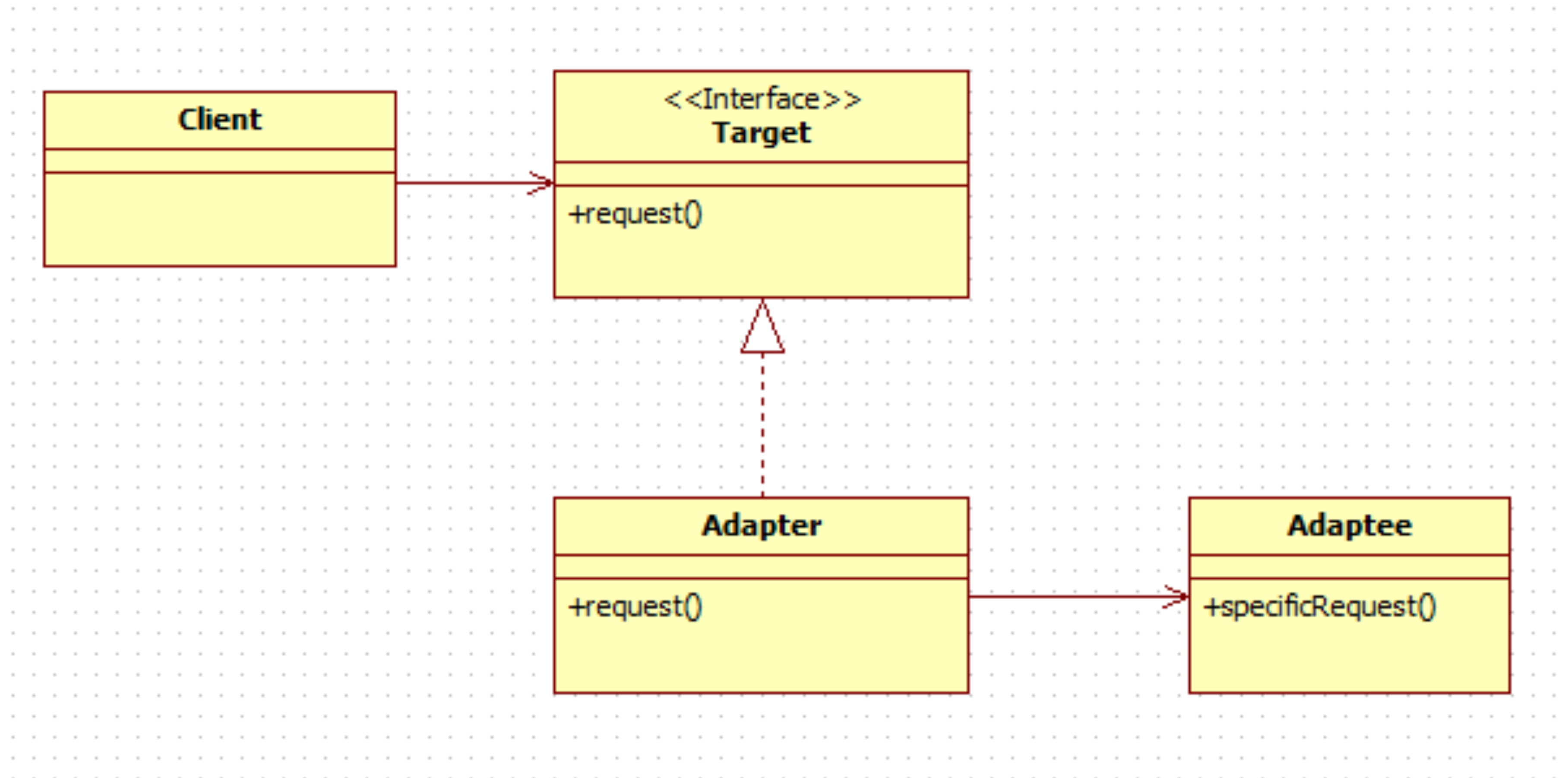


Interface Spec만 맞춰줄 수 있다면?
OTT client 객체의 큰 변경 없이 번체, 간체 자막도 지원 가능함

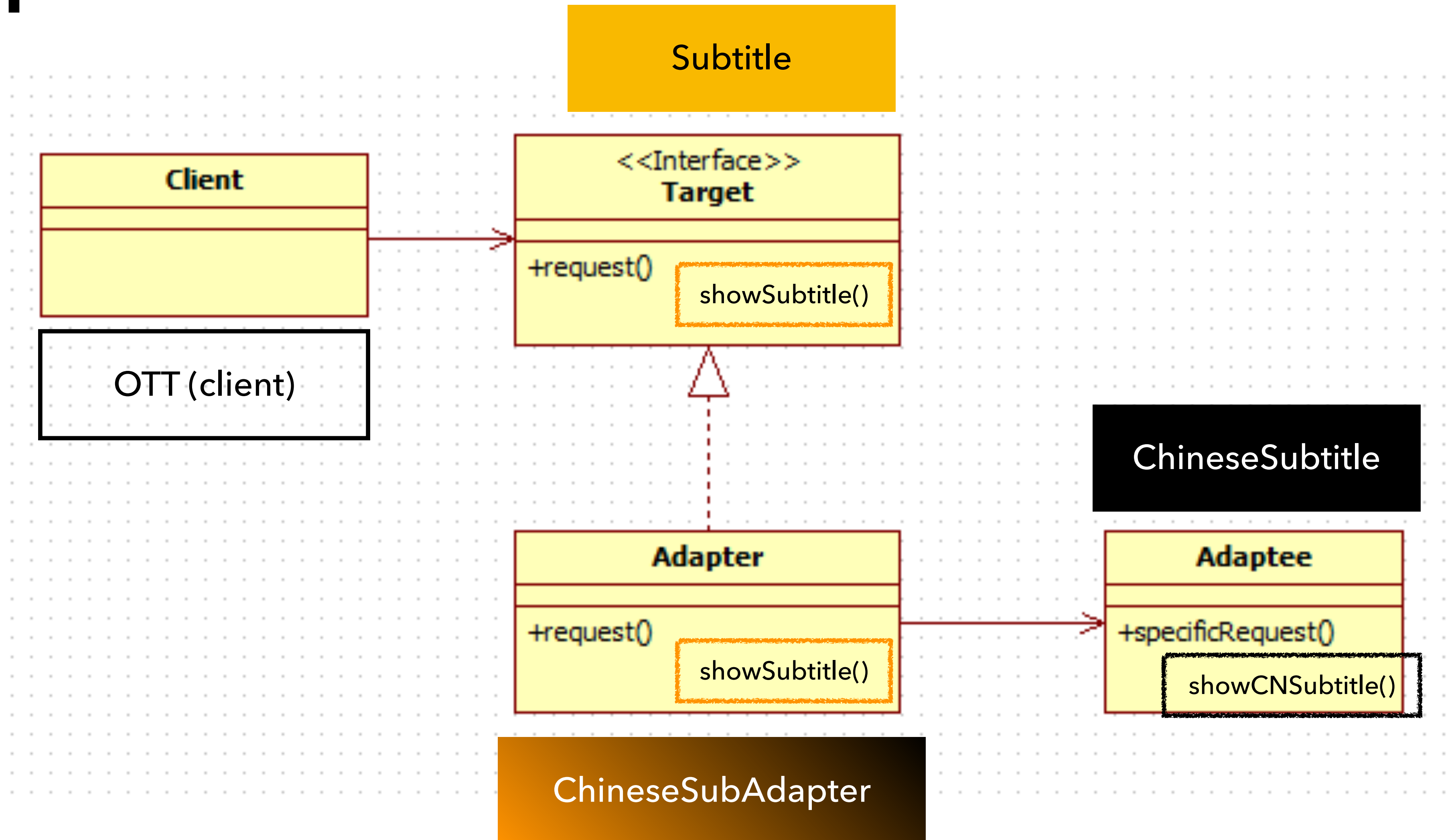
1. Adapter Pattern

```
class ChineseSubAdapter(  
    // property  
    private val adaptee: ChineseSubtitle,  
) : Subtitle {  
  
    override fun showSubtitle() {  
        adaptee.showZhTWSubtitle()  
    }  
}
```

1. Adapter Pattern



1. Adapter Pattern



1. Adapter Pattern

장점 : OCP

클라이언트의 코드를 수정하지 않으면서,
기존에 사용하지 않던 spec의 객체를 사용할 수 있음.

i.e. 수정에는 닫혀있지만 확장에는 열려있는 구조가 됨.

1. Adapter Pattern

단점 : 완벽한 100% 대응은 불가능

target에만 존재하거나, adaptee에만 존재하는 인터페이스는?

Subtitle

+ showSubtitle()
+ set(font: Font)

ChineseSubtitle

+ showZhCNSubtitle() (간체자)
+ showZhTWSubtitle() (번체자)

+ setFont(font: Font) { throw Exception() }

2. Decorator Pattern



Decorator Pattern

:: 확장하고 싶은 기능을 데코처럼 추가하는 패턴

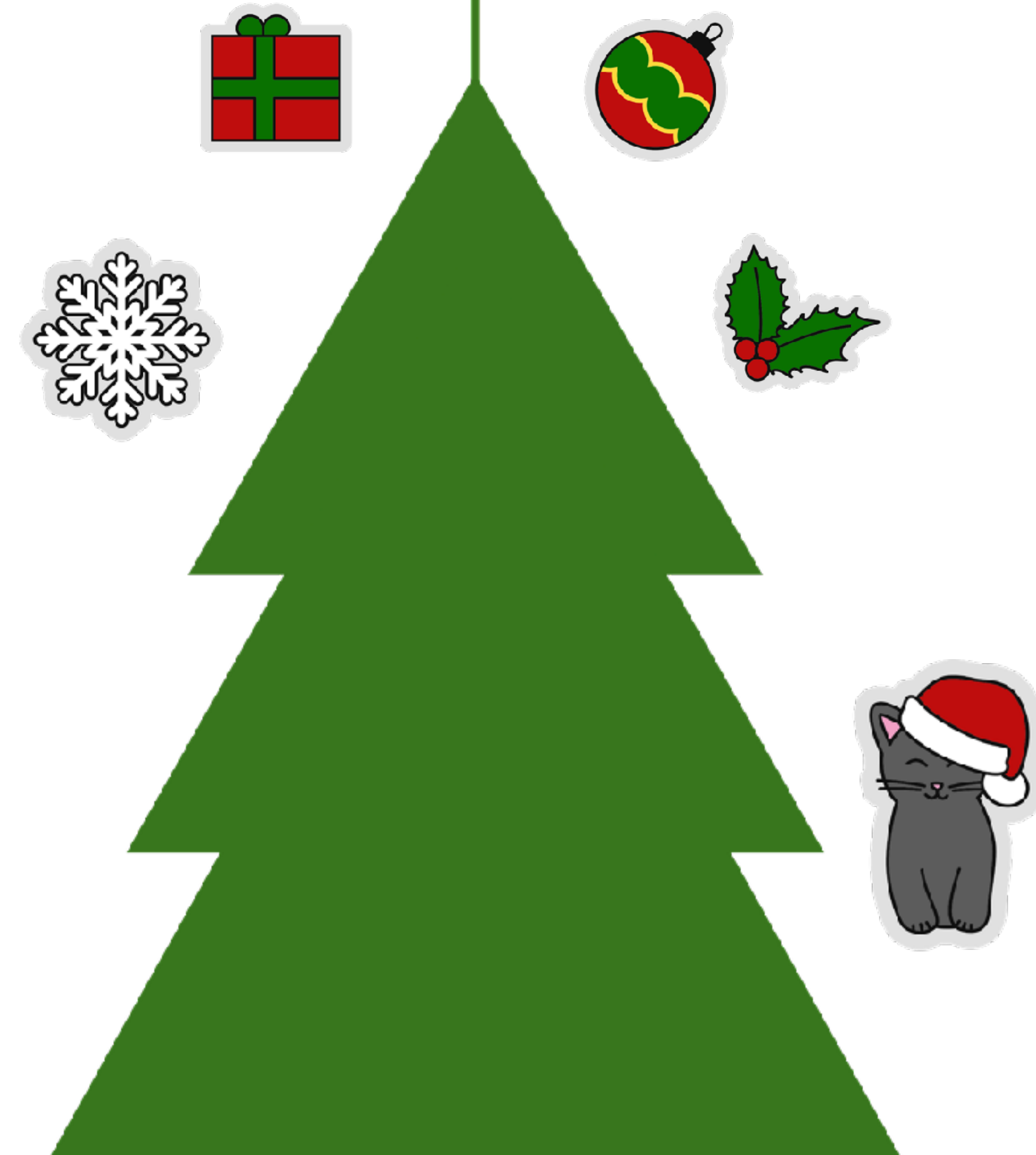
:: **본질은 바뀌지 않게** 추가된다는 점이 포인트!

기능 추가가 되어도 본질은 바뀌지 않는다는 점을 살리지 못한 개발...

오너먼트를 뭘 쓰느냐(데코레이터를 뭘 쓰느냐)에 따라
모든 트리를 다 다른 객체로 파악하는 경우

```
class TreeWithCookie()  
class TreeWithSnow()  
class TreeWithCat()  
class TreeWithCookieCat()  
class TreeWithCookieSnowCat()  
// ...
```

➡ 유지보수를 포기하자!



2. Decorator Pattern

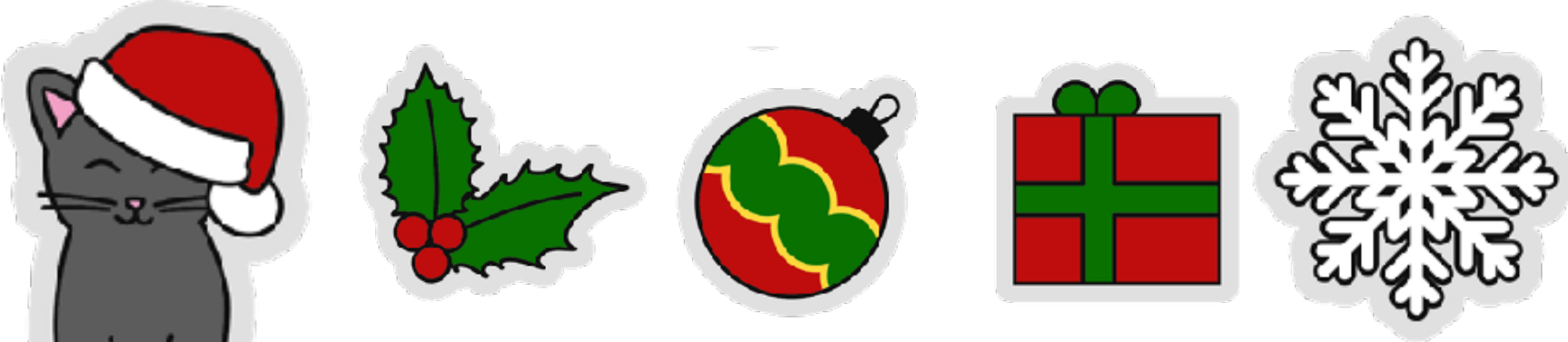
```
class Tree {  
    var isCookieExist = false  
    var isSnowExist = false  
    var isCatExist = true  
    // ...  
  
    fun draw() {  
        if (isCookieExist) { drawCookie() }  
        if (isSnowExist) { drawSnow() }  
        if (isCatExist) { drawCat() }  
        // ...  
    }  
}
```

Kotlin

1. flag로 추가 기능을 설정하겠어요.

침엽수를 트리의 베이스로 사용하면..?
SRP는 과연 지킨걸까..?

Base 종류 🌲



꾸미기 종류 🌟



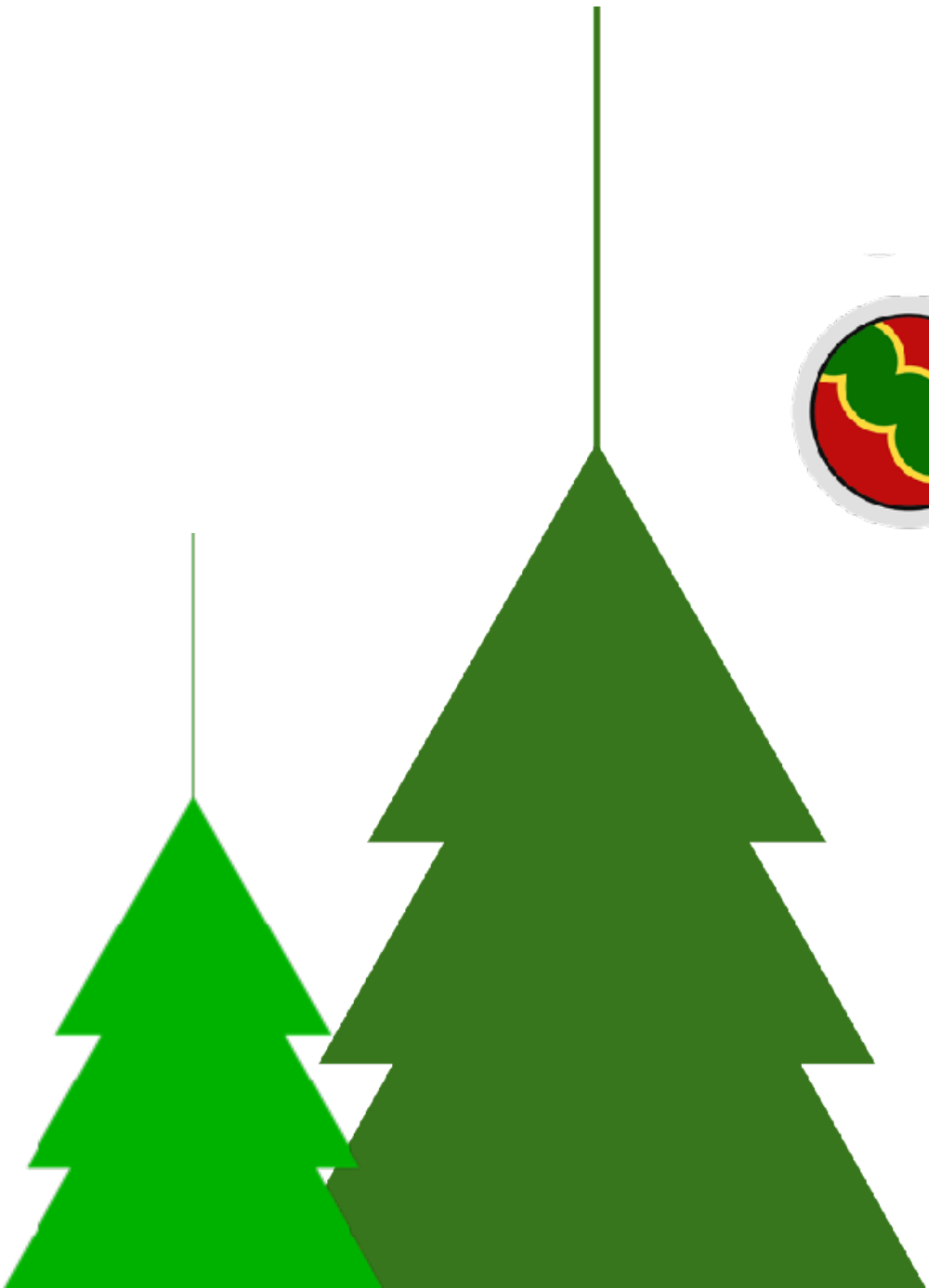
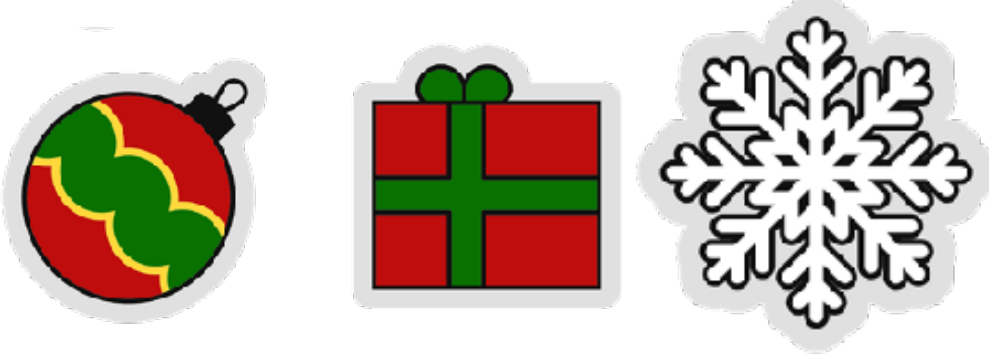
본질이 바뀌지 않는 포인트를 살린 방법론
데코를 뭘 써도, 베이스를 뭘 써도
결국 만들어지는 건 X-mas Tree!

2. Decorator Pattern

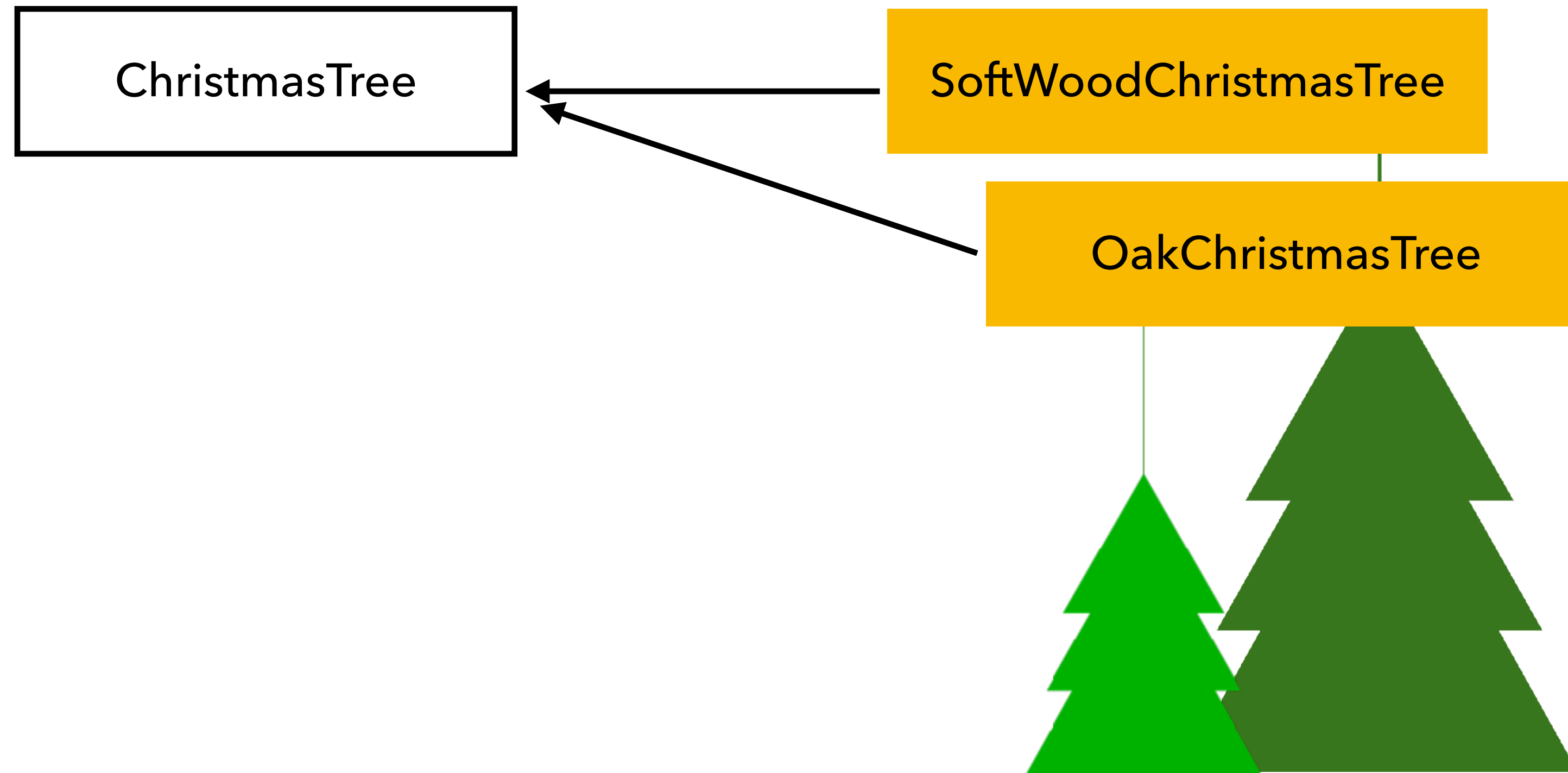
ChristmasTree

본질이 같으려면?

이 모든 것들이 다 공통 인터페이스를 갖춰야 함.



2. Decorator Pattern

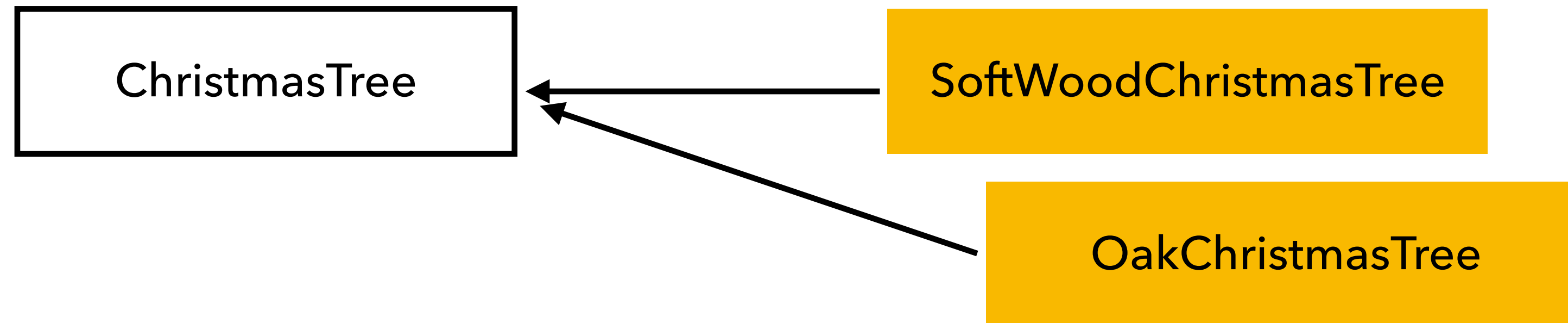


베이스🌳로

침엽수를 쓰든, 오크를 쓰든,
활엽수를 쓰든, 소나무를 쓰든,...

➡ 다 크리스마스 트리 🎄

2. Decorator Pattern



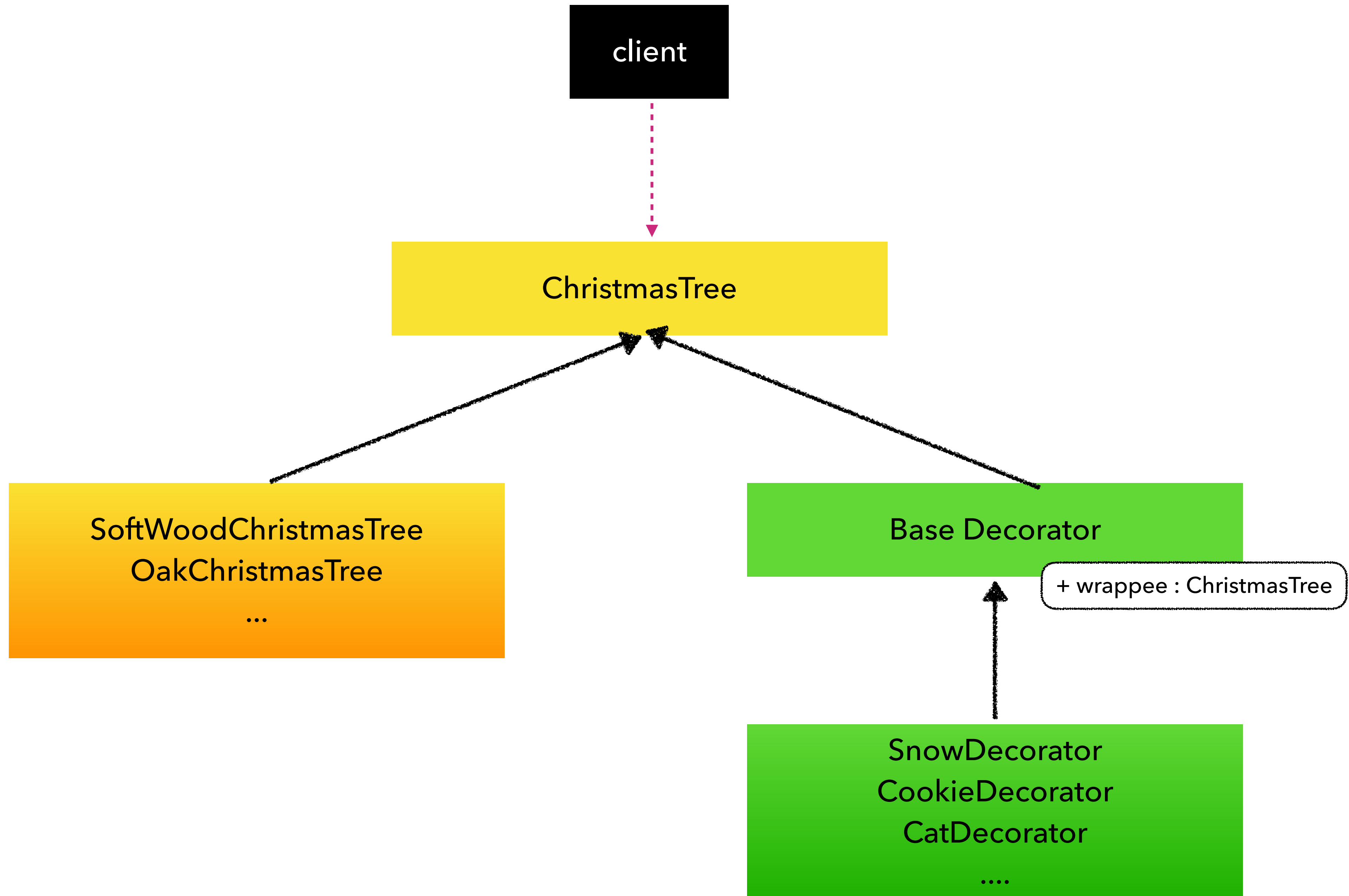
데코로

고양이 장식을 쓰든,
쿠키를 쓰든,
눈을 쓰든,...

모두 결과물은 크리스마스 트리!

그러나 데코는 베이스에만 달릴 수 있음.





☞ 외부에서는 애가 여러겹의 덩어리인지 단일인지 알 방법이 없음.
그냥 문제 해결을 위한 public interface만 사용하면 되는 거임.

client

☞ 하나의 문제 해결이 목적이라는 뜻

Component (Interface)

☞ Deco를 해줄 베이스 객체

Concrete Component

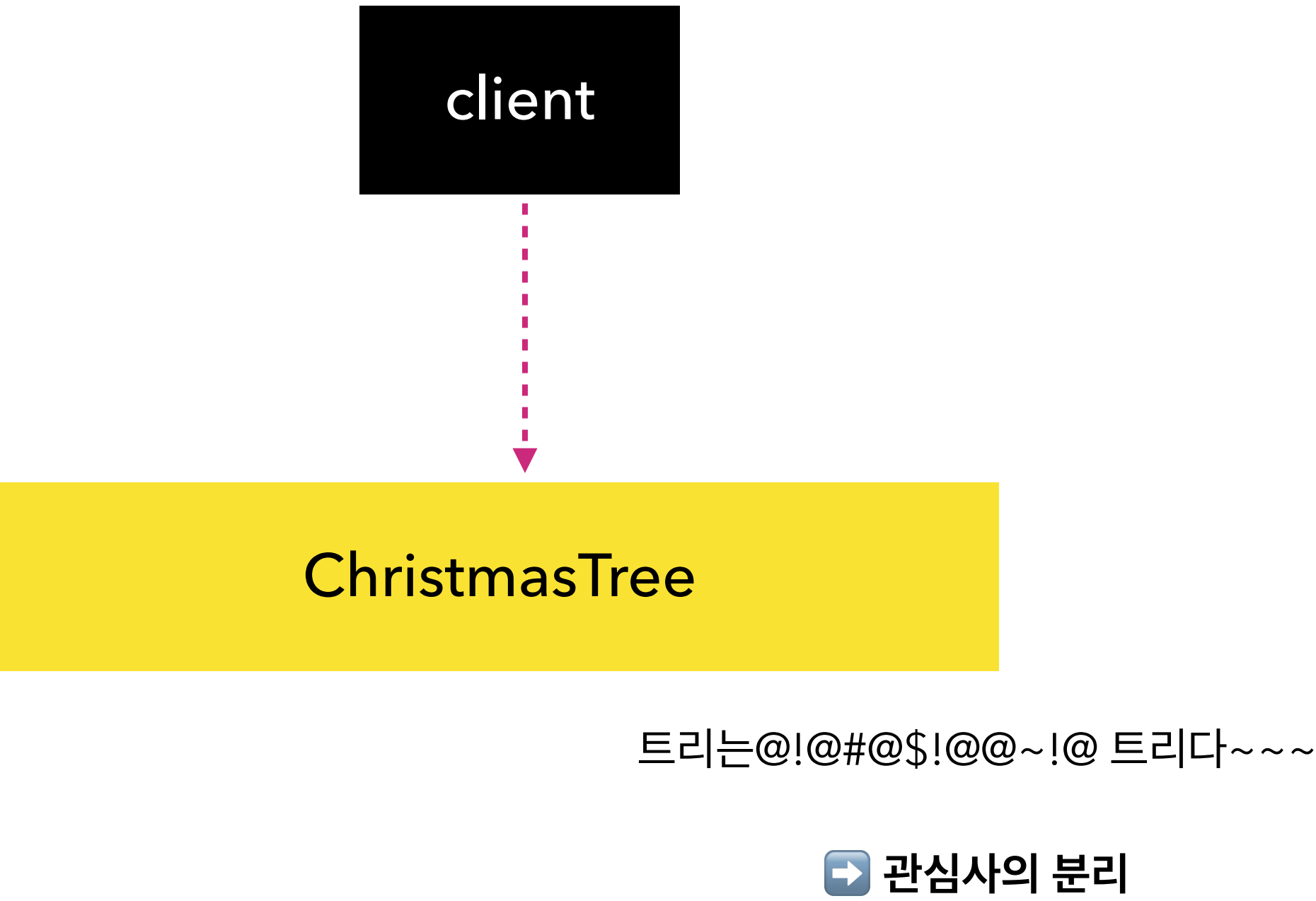
☞ 다른 Deco로 래핑되거나 베이스 감싸는
말 그대로 Decorator!

Base Decorator


+ wrappee : Component



Concrete Decorator








2. Decorator Pattern

 Sign-up for a new account!

 Full Legal Name 

 E-mail address

 Password 



✖ Password can't be blank

✖ Password must contain at least 1 lowercase

✖ Password must contain at least 1 uppercase

✖ Password must contain special character

✖ Password must contain at least one number

 Password Confirmation 

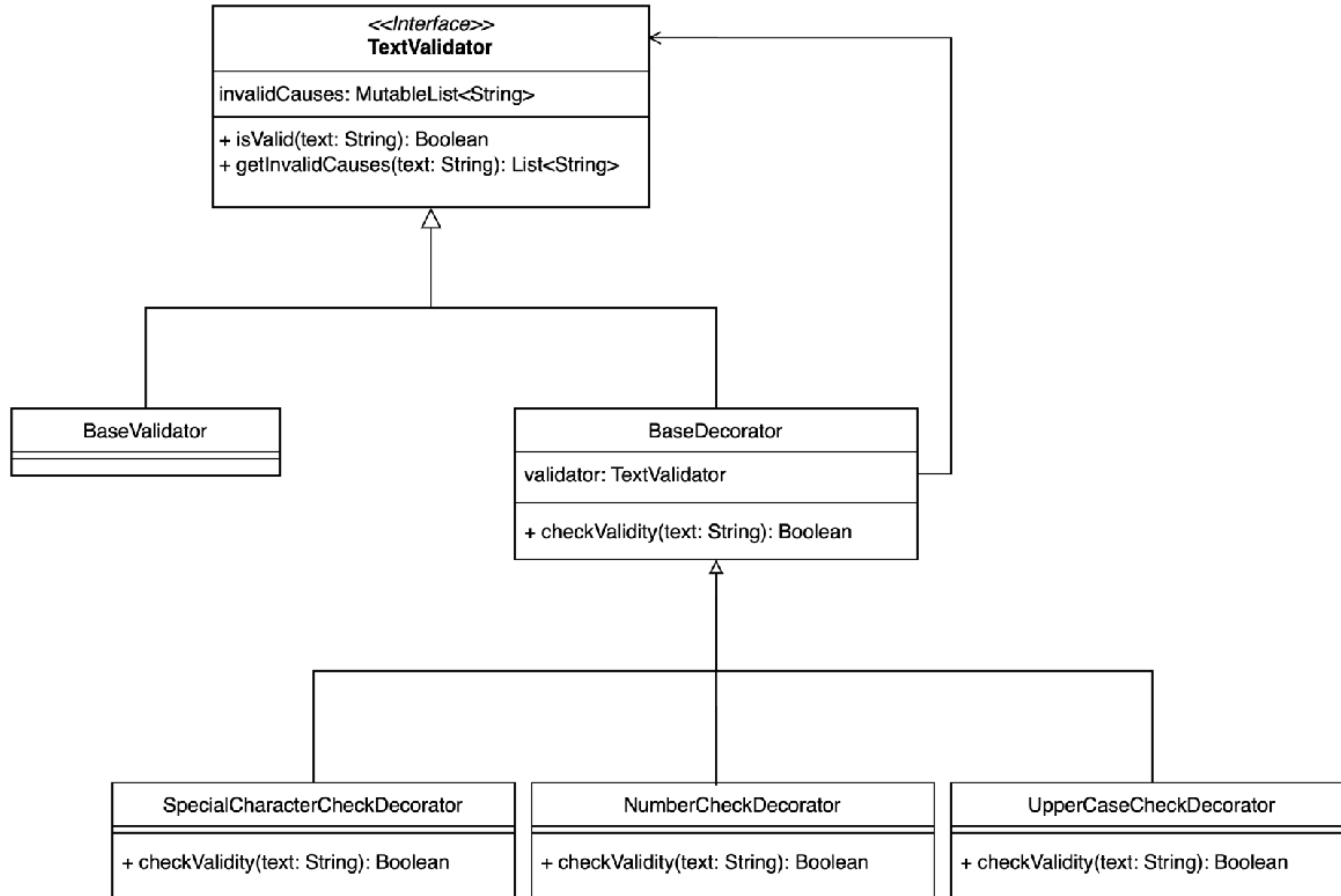
Signup

Already have an account? [Log In](#)

클라이언트 쪽에서 텍스트를 입력 받다 보면 여러 케이스가 생길 수 있음.

1. 어떤 글자가 와도 다 ok
2. 숫자(number)는 안 돼요
3. 특수문자(special symbol)는 안 돼요
4. 대문자(upper charactor)는 안 돼요
5. 2~4번중 n개 선택해서 안 돼요

2. Decorator Pattern



2. Decorator Pattern

장점 : SRP

베이스 컴포넌트, 데코레이터 관심사의 분리

➡ 문제가 생겼을 때 원인 파악을 각각에서 해주면 됨

2. Decorator Pattern

단점 : 데코의 순서 불명...

뭐가 들어가도 다 트리로만 인식

데코의 순서를 알 수가 없음 😓

순서가 중요한 거라면 이를 Client에서 꼭꼭 지켜줘야 함.

나중에 원치 않는 데코를 뺄 수 있는 방법도 없음. 😓