230911 데코레이터 패턴

객체에 추가 요소를 동적으로 더할 수 있다. 데코레이터를 사용하면 서브클래스를 만들 때 보다 훨씬 유연하게 기능을 확장할 수 있다.

스타버즈가 만난 문제

문제

- 음료별로 가격이 다름
- 우유, 두유, 휘핑, 초콜릿… 옵션이 들어갔을 때의 가격 계산(cost())을 무한 확장

문제해결

- 음료 / 음료를 확장한 첨가물
- 음료를 중심으로 첨가물이 계속 wrap 하는 구조

데코레이터 패턴 단점

- 자잘한 클래스가 너무 많아진다

- 특정 형식에 의존하는 클라이언트 코드에 적용하기 어렵다
- 구성 요소 초기화하는데 더 많은 코드가 필요해 복잡하다.

함께보기

패턴 사용 시기

- 객체 책임과 행동이 동적으로 상황에 따라 다양한 기능이 빈번하게 추가/삭제되는 경우
- 객체의 결합을 통해 기능이 생성될 수 있는 경우
- 객체를 사용하는 코드를 손상시키지 않고 런타임에 객체에 추가 동작을 할당할 수 있어야 하는 경우
- 상속을 통해 서브클래싱으로 객체의 동작을 확장하는 것이 어색하거나 불가능 할 때

패턴 장점

- 데코레이터를 사용하면 서브클래스를 만들때보다 훨씬 더 유연하게 기능을 확장할 수 있다.
- 객체를 여러 데코레이터로 래핑하여 여러 동작을 결합할 수 있다.
- 컴파일 타임이 아닌 런타임에 동적으로 기능을 변경할 수 있다.
- 각 장식자 클래스마다 고유의 책임을 가져 **단일 책임 원칙(SRP)Visit Website**을 준수
- 클라이언트 코드 수정없이 기능 확장이 필요하면 장식자 클래스를 추가하면 되니 **개방 폐쇄 원칙(OCP)Visit Website**을 준수
- 구현체가 아닌 인터페이스를 바라봄으로써 **의존 역전 원칙(DIP)Visit Website** 준수

패턴 단점

- 만일 장식자 일부를 제거하고 싶다면, Wrapper 스택에서 특정 wrapper를 제거하는 것은 어렵다.
- 데코레이터를 조합하는 초기 생성코드가 보기 안좋을 수 있다. `new A(new B(new C(new D())))`
- 어느 장식자를 먼저 데코레이팅 하느냐에 따라 데코레이터 스택 순서가 결정지게 되는데, 만일 순서에 의존하지 않는 방식으로 데코레이터를 구현하기는 어렵다.

예제

- 문자열 꾸미기 예제

https://youtu.be/UTmY_oB4V8I?si=3g2w0E4t-Jat26E6

Item.java

```java
public abstract class Item {
    public abstract int getLinesCount();
    public abstract int getMaxLength();
    public abstract int getLength(int index);
    public abstract String getString(int index);

    public void print() {
        int cntLines = getLinesCount();
        for(int i=0; i<cntLines; i++) {
            String string = getString(i);
            System.out.println(string);
        }
    }
}
```