[Java] 컴포지션(Composition)

> 컴포지션 : 기존 클래스가 새로운 클래스의 구성요소가 되는 것
> 상속(Inheritance)의 단점을 커버할 수 있는 컴포지션에 대해 알아보자

우선 상속(Inheritance)이란, 하위 클래스가 상위 클래스의 특성을 재정의 한 것을 말한다. 부모 클래스의 메서드를 오버라이딩하여 자식에 맞게 재사용하는 등, 상당히 많이 쓰이는 개념이면서 활용도도 높다.

하지만 장점만 존재하는 것은 아니다. 상속을 제대로 사용하지 않으면 유연성을 해칠 수 있다.

구현 상속(클래스→클래스)의 단점

1. 캡슐화를 위반
2. 유연하지 못한 설계
3. 다중상속 불가능

오류의 예시

다음은, HashSet에 요소를 몇 번 삽입했는지 count 변수로 체크하여 출력하는 예제다.

```
public class CustomHashSet<E> extends HashSet {
    private int count = 0;

    public CustomHashSet(){}

    public CustomHashSet(int initCap, float loadFactor){
        super(initCap,loadFactor);
    }

    @Override
    public boolean add(Object o) {
        count++;
        return super.add(o);
    }

    @Override
    public boolean addAll(Collection c) {
        count += c.size();
```

```
        return super.addAll(c);
    }

    public int getCount() {
        return count;
    }

}
```

add와 addAll 메서드를 호출 시, count 변수에 해당 횟수를 더해주면서, getCount()로 호출 수를 알아낼 수 있다.

하지만, 실제로 사용해보면 원하는 값을 얻지 못한다.

```
public class Main {
    public static void main(String[] args) {
        CustomHashSet<String> customHashSet = new CustomHashSet<>();
        List<String> test = Arrays.asList("a","b","c");
        customHashSet.addAll(test);

        System.out.println(customHashSet.getCount()); // 6
    }
}
```

`a, b, c`의 3가지 요소만 배열에 담아 전달했지만, 실제 getCount 메서드에서는 6이 출력된다.

이는 CustomHashSet에서 상속을 받고 있는 HashSet의 부모 클래스인 `AbstractCollection`의 addAll 메서드에서 확인할 수 있다.

```
// AbstractCollection의 addAll 메서드
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

해당 메서드를 보면, `add(e)`가 사용되는 것을 볼 수 있다. 여기서 왜 6이 나온지 이해가 되었을 것이다.

우리는 CustomHashSet에서 `add()` 와 `addAll()`를 모두 오버라이딩하여 count 변수를 각각 증가시켜줬다. 결국 두 메서드가 모두 실행되면서 총 6번의 count가 저장되는 것이다.

따라서 이를 해결하기 위해선 두 메소드 중에 하나의 count를 증가하는 곳을 지워야한다. 하지만 이러면 눈으로 봤을 때 코드의 논리가 깨질 뿐만 아니라, 추후에 HashSet 클래스에 변경이 생기기라도 한다면 큰 오류를 범할

수도 있게 된다.

결과론적으로, 위와 같이 상속을 사용했을 때 유연하지 못함과 캡슐화에 위배될 수 있다는 문제점을 볼 수 있다.

그렇다면 컴포지션은?

상속처럼 기존의 클래스를 확장(extend)하는 것이 아닌, **새로운 클래스를 생성하여 private 필드로 기존 클래스의 인스턴스를 참조하는 방식**이 바로 컴포지션이다.

> forwarding이라고도 부른다.

새로운 클래스이기 때문에, 여기서 어떠한 생성 작업이 일어나더라도 기존의 클래스는 전혀 영향을 받지 않는다는 점이 핵심이다.

위의 예제를 개선하여, 컴포지션 방식으로 만들어보자

```
public class CustomHashSet<E> extends ForwardingSet {
    private int count = 0;

    public CustomHashSet(Set<E> set){
        super(set);
    }

    @Override
    public boolean add(Object o) {
        count++;
        return super.add(o);
    }

    @Override
    public boolean addAll(Collection c) {
        count += c.size();
        return super.addAll(c);
    }

    public int getCount() {
        return count;
    }

}
```

```
public class ForwardingSet<E> implements Set {

    private final Set<E> set;
```

```
    public ForwardingSet(Set<E> set){
        this.set=set;
    }

    @Override
    public int size() {
        return set.size();
    }

    @Override
    public boolean isEmpty() {
        return set.isEmpty();
    }

    @Override
    public boolean contains(Object o) {
        return set.contains(o);
    }

    @Override
    public Iterator iterator() {
        return set.iterator();
    }

    @Override
    public Object[] toArray() {
        return set.toArray();
    }

    @Override
    public boolean add(Object o) {
        return set.add((E) o);
    }

    @Override
    public boolean remove(Object o) {
        return set.remove(o);
    }

    @Override
    public boolean addAll(Collection c) {
        return set.addAll(c);
    }

    @Override
    public void clear() {
        set.clear();
    }

    @Override
    public boolean removeAll(Collection c) {
        return set.removeAll(c);
    }
```

```
    @Override
    public boolean retainAll(Collection c) {
        return set.retainAll(c);
    }

    @Override
    public boolean containsAll(Collection c) {
        return set.containsAll(c);
    }

    @Override
    public Object[] toArray(Object[] a) {
        return set.toArray();
    }
}
```

CustomHashSet은 Set 인터페이스를 implements한 ForwardingSet을 상속한다.

이로써, HashSet의 부모클래스에 영향을 받지 않고 오버라이딩을 통해 원하는 작업을 수행할 수 있게 된다.

```
public class Main {
    public static void main(String[] args) {
        CustomHashSet<String> customHashSet = new CustomHashSet<>(new HashSet<>
());
        List<String> test = Arrays.asList("a","b","c");
        customHashSet.addAll(test);

        System.out.println(customHashSet.getCount()); // 3
    }
}
```

CustomHashSet이 원하는 작업을 할 수 있도록 도와준 ForwardingSet은 위임(Delegation) 역할을 가진다.

원본 클래스를 wrapping 하는게 목적이므로, Wrapper Class라고 부를 수도 있을 것이다.

그리고 현재 작업한 이러한 패턴을 데코레이터 패턴이라고 부른다. 어떠한 클래스를 Wrapper 클래스로 감싸며, 기능을 덧씌운다는 의미다.

상속을 쓰지말라는 이야기는 아니다. 상속을 사용하는 상황은 LSP 원칙에 따라 IS-A 관계가 반드시 성립할 때만 사용해야 한다. 하지만 현실적으로 추후의 변화가 이루어질 수 있는 방향성을 고려해봤을 때 이렇게 명확한 IS-A 관계를 성립한다고 보장할 수 없는 경우가 대부분이다.

결국 이런 문제를 피하기 위해선, 컴포지션 기법을 사용하는 것이 객체 지향적인 설계를 할 때 유연함을 갖추고 나아갈 수 있을 것이다.

[참고 자료]

- 링크
- 링크