[2기][item18] 상속보다는 컴포지션을 사용하라

💡 Key Points

#상속

#오버라이딩

#캡슐화

#컴포지션

#래퍼 클래스

#데코레이터 패턴(decorator pattern)

#Forwarding

#위임(delegation)

1. 상속의 경우 생길 수 있는 문제점

- 클래스가 인터페이스를 구현하거나, 인터페이스가 인터페이스를 상속받는 경우가 아닌 상속의 경우 생길 수 있는 문제점

- 상위 클래스가 어떻게 구현되느냐 따라 기존에 잘 동작하던 하위 클래스에 이상이 생길 수 있다.

  (+ 다음 릴리즈에서 부모 클래스가 얼마든지 변경될 수 있다, 추후 변경사항까지 고려해야 한다)

- 문제가 생길 수 있는 케이스 (간단히 말하면, 메소드 재정의로 인해 비롯된 문제)

  1. 상속하여 자기 사용(self-use) 메소드를 오버라이딩 하는 경우

     ex) HashSet 클래스의 addAll(), add()

  2. 부모 클래스에서 새로운 메서드가 추가되는 경우

     컬렉션에 추가된 모든 원소가 특정 조건을 만족해야 하는 클래스 B가 있다. 클래스 A는 클래스 B의 부모 클래스였는데 만약 부모 클래스 A에서 원소를 추가하는 메소드가 만들어진다면? 클래스 B는 A의 릴리즈에 맞춰서 매번 부모 클래스가 추가될 때마다 해당 메소드를 재정의하는 오버라이드 코드를 추가해야만 문제가 발생하지 않는다.

     ex) `Vector`를 확장한 `Stack`, `Hashtable`을 확장한 `Properties`

- 그렇다면 자식 클래스에서 메서드를 오버라이딩 하지 않고 새로운 메서드를 추가하면 되지 않을까?

  → 부모 클래스의 다음버전이 릴리즈 됐는데 하필이면 하위 클래스에서 추가한 메서드와 시그니처가 같을 수 있는 경우가 있다. 이때 반환 타입이 다르면 자식 클래스는 컴파일조차 되지 않고, 반환 타입이 같았으면 재정의 하는 꼴이 된다. 따라서 원론의 문제로 돌아가게 된다. (피할 수 없어 😱!)

- 이러한 점에서 상속은 오버라이딩을 통해 캡슐화를 깨뜨릴 수도 있다.

EX) 자식 클래스에서 자기 사용(self-use) 메소드를 오버라이딩 하는 경우

- `InstrumentedHashSet`이라는 클래스는 `HashSet`을 상속받아 만들어졌다.
- 처음 생성된 이후 원소가 몇 개 더해졌는지 알 수 있어야 한다
- `HashSet.size()` 로는 현재 크기만 알 수 있다.
  지워진 원소가 있다면 `size()` 가 반환하는 값의 크기도 줄어든다.
- `InstrumentedHashSet`내에 `addCount` 변수로 생성 이후 더해지는 원소의 갯수를 알아보려는 것

```
// Broken - Inappropriate use of inheritance! (Page 87)
public class InstrumentedHashSet<E> extends HashSet<E> {

    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

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

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

```
public static void main(String[] args) {
    InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
    s.addAll(List.of("Snap", "Crackle", "Pop"));
    System.out.println(s.getAddCount());
}
```

▼ 답은?

```
6
```

▼ 이유는?

HashSet의 addAll() 는 내부적으로 add() 를 호출한다 → `self-use`

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

생각해 볼 수 있는 다른 방안은 `addAll()` 에서 `HashSet.addAll()` 을 호출하지 않고 아래와 같이 작성하는 방법이다.

```
@Override
public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    for(E ele : c) {
        super.add(ele);
    }
    return true;
}
```

그러나 이 방법은 어렵고, 시간도 더 들고, 자칫 오류를 내거나 성능을 떨어뜨릴 수도 있다.

그리고 만약 HashSet.add() 메소드가 private 이었다면 애초에 사용할 수 없다.

2. 데코레이터 패턴

The Decorator Pattern in Java | Baeldung

A Decorator pattern can be used to attach additional responsibilities to an object either statically or dynamically. A Decorator provides an enhanced interface to the original

https://www.baeldung.com/java-decorator-pattern

Overview

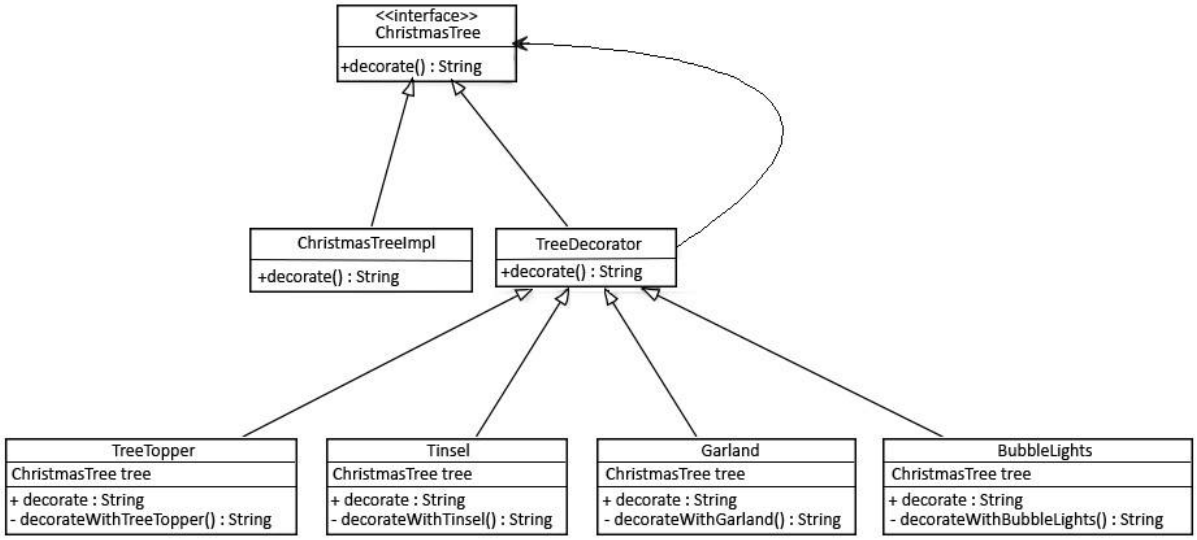
- 데코레이터 패턴을 이용하면 정적으로, 동적으로 어떤 object에 추가적인 일을 더 시킬 수 있다.

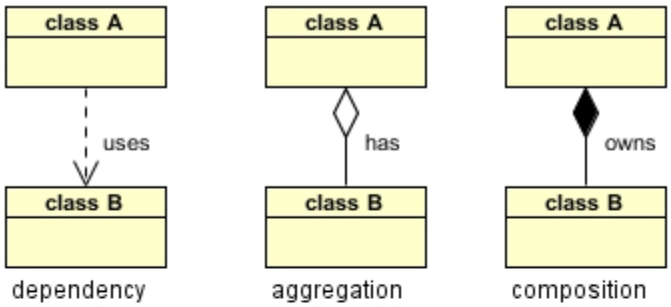
- 데코레이터 패턴은 기존에 존재하던 객체의 인터페이스에 특정 기능을 추가할 수 있다.
- 이 패턴을 구현할 때는 상속보다 컴포지션을 선호한다.

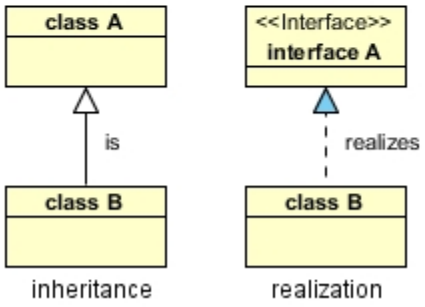
  그래야 데코레이팅 할 원소를 상속한 자식 클래스에서 끊임없이 발생할 수 있는 오버헤드를 줄일 수 있다.

Decorator Pattern Example

- 크리스마스 트리라는 object가 있고 이를 decorate 하려고 한다.
- 기존의 크리스마스 트리 object 자체를 수정하지 않고, 크리스마스 트리에 데코레이션 아이템을 붙여나간다








```
public interface ChristmasTree {
    String decorate();
}
```

```
public class ChristmasTreeImpl implements ChristmasTree {

    @Override
    public String decorate() {
        return "Christmas tree";
    }
}
```

- TreeDecorator 는 Christmas 인터페이스를 구현하면서 private 필드로 ChristmasTree 인스턴스를 참조한다.
- 인터페이스 메서드는 단순하게 멤버필드 ChristmasTree 인스턴스의 메서드를 호출하는 식으로 구현되어 있다.

```
public abstract class TreeDecorator implements ChristmasTree {
    private ChristmasTree tree;

    // standard constructors
    @Override
    public String decorate() {
        return tree.decorate();
    }
}
```

- 새 클래스의 인스턴스 메서드들은 기존 클래스의 대응하는 메서드를 호출해 그 결과를 반환한다. 이 방식을 ***전달(forwarding)***이라고 하며 새 클래스의 메서드들을 ***전달 메서드(forwarding method)***라고 부른다
- 그 결과 새 클래스는 기존 클래스의 내부 구현 방식의 영향에서 벗어나며 심지어 기존 클래스에 새로운 메서드가 추가되더라도 전혀 영향받지 않는다.
- 기존 클래스가 새로운 클래스의 구성요소로 쓰인다는 뜻에서 이러한 설계를 ***컴포지션(Composition)***이라고 한다.
- 다른 인스턴스를 감싸고 있다는 뜻에서 TreeDecorator 같은 클래스를 ***래퍼 클래스(Wrapper Class)***라고 한다.
- ***컴포지션***을 통해 ***전달***하는 조합을 넓은 의미로 ***위임(delegation)***이라고 부른다. 엄밀히 따지면 래퍼 객체가 내부 객체에 자기 자신의 참조를 넘기는 경우만 위임에 해당한다

```
public class BubbleLights extends TreeDecorator {

    public BubbleLights(ChristmasTree tree) {
        super(tree);
    }

    public String decorate() {
        return super.decorate() + decorateWithBubbleLights();
    }

    private String decorateWithBubbleLights() {
        return " with Bubble Lights";
    }

}
```

```
@Test
public void whenDecoratorsInjectedAtRuntime_thenConfigSuccess() {
    ChristmasTree tree1 = new Garland(new ChristmasTreeImpl());
    assertEquals(tree1.decorate(),
      "Christmas tree with Garland");

    ChristmasTree tree2 = new BubbleLights(
      new Garland(new Garland(new ChristmasTreeImpl())));
    assertEquals(tree2.decorate(),
      "Christmas tree with Garland with Garland with Bubble Lights");
}
```

- tree1 는 Garland 하나 붙이고, tree2 는 BubbleLight 하나와 Garland 2개를 붙였다.
- 데코레이터 패턴은 런타임 때 원하면 몇개든지 데코레이터를 추가할 수 있다는 유연함을 제공한다

Conclusion

다음의 경우 데코레이터 패턴이 유용하게 쓰일 수 있다.

- object 의 어떤 행위나 속성을 추가하거나 기능을 덧붙이거나 기존의 행위나 속성을 지우고 싶을 때
- 특정 object의 다른 기능은 남겨놓으면서 일부를 수정하고 싶을 때

3. InstrumentedHashSet의 문제를 데코레이터 패턴으로 수정해보자

```
import java.util.*;

public class InstrumentedSet<E> implements Set<E> {

    private final Set<E> s;
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) { this.s = s; }
```

```
    public void clear()               { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    public int size()                 { return s.size();      }
    public Iterator<E> iterator()     { return s.iterator();  }
    public boolean remove(Object o)   { return s.remove(o);   }
    public boolean containsAll(Collection<?> c) { return s.containsAll(c); }
    public boolean removeAll(Collection<?> c) { return s.removeAll(c);   }
    public boolean retainAll(Collection<?> c) { return s.retainAll(c);   }
    public Object[] toArray()          { return s.toArray();  }
    public <T> T[] toArray(T[] a)      { return s.toArray(a); }

    @Override public boolean equals(Object o) { return s.equals(o);  }
    @Override public int hashCode()    { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }

    public boolean add(E e) {
        addCount++;
        return s.add(e);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

}
```

```
public static void main(String[] args) {
    InstrumentedSet<String> s = new InstrumentedSet<>(new HashSet<>());
    s.addAll(List.of("Snap", "Crackle", "Pop"));
    System.out.println(s.getAddCount());
}
```

- Set 인터페이스를 구현하고 있다

  (HashSet 을 상속하지 않았음, 구현 클래스를 직빵으로 상속하면 위험해 질 수 있다)
- Set의 인스턴스를 인수로 받는 생성자를 하나 제공한다. 어떠한 Set이던 사용 가능하다
- 임의의 Set에 addCount 를 측정하는 기능을 추가하는 것이 이 클래스의 핵심
- 특정 조건하에서만 임시로 addCount 를 측정하게 할 수도 있다.

▼ 코드

```
// 해당 메서드 내에서만 Set<Dog> 을 ForwardingSet으로 감싸서 addCount 측정 가능
static void walk(Set<Dog> dogs) {
    ForwardingSet<Dog> iDogs = new ForwardingSet<>(dogs);
    ...
}
```

- 재사용할 수 있는 전달 클래스를 인터페이스당 하나씩 만들어두면 원하는 기능을 추가한 전달 클래스들을 아주 손쉽게 구현할 수 있다

  ex) 구아바는 모든 컬렉션 인터페이스용 전달 메서드를 전부 구현해 뒀다

  ▼ 코드

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()               { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    public int size()                 { return s.size();      }
    public Iterator<E> iterator()     { return s.iterator();  }
    public boolean add(E e)           { return s.add(e);      }
    public boolean remove(Object o)   { return s.remove(o);   }
    public boolean containsAll(Collection<?> c)
                                  { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c)
                                  { return s.addAll(c);      }
    public boolean removeAll(Collection<?> c)
                                  { return s.removeAll(c);   }
    public boolean retainAll(Collection<?> c)
                                  { return s.retainAll(c);   }
    public Object[] toArray()          { return s.toArray();  }
    public <T> T[] toArray(T[] a)      { return s.toArray(a); }
    @Override public boolean equals(Object o)
                                       { return s.equals(o);  }
    @Override public int hashCode()    { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }
}
```

```
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
```

```
        super(s);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
```

```
public static void main(String[] args) {
    InstrumentedSet<String> s = new InstrumentedSet<>(new HashSet<>());
    s.addAll(List.of("Snap", "Crackle", "Pop"));
    System.out.println(s.getAddCount());
}
```

4. 데코레이터 패턴의 단점

- 콜백 프레임워크와는 잘 어울리지 않는다

▼ code

```
interface SomethingWithCallback {

    void doSomething();

    void call();

}
```

- doSomething() 이 호출되면 call() 콜백 메서드가 호출되기를 바라는 상황

```
final class SomeService {

    void performAsync(SomethingWithCallback callback) {
        new Thread(() -> {
            perform();
```

```
            callback.call(); // (3)
        }).start();
    }

    void perform() {
        System.out.println("Service is being performed.");
    }

}
```

```
class WrappedObject implements SomethingWithCallback { // HashSet

    private final SomeService service;

    WrappedObject(SomeService service) {
        this.service = service;
    }

    @Override
    public void doSomething() {
        service.performAsync(this); // (2)
    }

    @Override
    public void call() {
        System.out.println("WrappedObject callback!");
    }
}
```

```
 public static void main(String[] args) {
    SomeService   service       = new SomeService();
    WrappedObject wrappedObject = new WrappedObject(service);
    wrappedObject.doSomething(); // (1)
}
```

- wrapperObject.doSomething → service.performAsync(this) → callback.call()
- performAsync() 파라매터 this 는 wrapperObject 이다.

```
class Wrapper implements SomethingWithCallback { // InstrumentedHashSet

    private final WrappedObject wrappedObject;
```

```
    Wrapper(WrappedObject wrappedObject) {
        this.wrappedObject = wrappedObject;
    }

    @Override
    public void doSomething() {
        wrappedObject.doSomething();
    }

    @Override
    public void call() {
        System.out.println("Wrapper callback!");
    }

}
```

```
 public static void main(String[] args) {
    SomeService   service       = new SomeService();
    WrappedObject wrappedObject = new WrappedObject(service);
    Wrapper       wrapper       = new Wrapper(wrappedObject);
    wrapper.doSomething();
}
```

▼ 콘솔

```
Service is being performed.
WrappedObject callback!
```

- wrapper.doSomething → wrappedObject.doSomething() → service.performAsync(this) → callback.call()
- performAsync() 파라매터 this 는 wrapperObject 이지 Wrapper 가 될 수 없다

- 전달 메서드가 성능에 주는 영향이나 래퍼 객체가 메모리 사용량에 주는 문제는 별로 없다고 밝혀졌다.

5. 상속을 사용해도 괜찮은 경우

- 부모 클래스 A 를 자식 클래스 B 에서 상속하고 있다면, 클래스 B가 클래스 A 와 is-a 관계여야만 한다
- B가 정말 A인가? A의 API에 아무런 결함이 없는가? 결함이 있다면 이 결함이 클래스 B의 API까지 전파되어도 괜찮은가?
- 상속은 상위 클래스의 API를 그 결함까지도 그대로 승계한다
- 위의 질문에 대한 대답이 아니라면 컴포지션과 전달을 사용하자