

쉽게 익히는 클린코드

이번주 컨텐츠는?

- ~냄새) 중재자
- *리팩토링) 중재자 제거하기, 슈퍼클래스를 위임으로, 서브클래스를 위임으로

~냄새. 중재자

Middle Man

캡슐화를 통해 내부의 구체적인 정보를 최대한 감출 수 있다.

그러나 어떤 클래스의 메소드가 대부분 다른 클래스로 메소드
호출을 위임하고 있다면 중재자를 제거하고 클라이언트가 해당
클래스를 직접 사용하도록 코드를 개선할 수 있다.

[사용 가능한 리펙토링 기술]

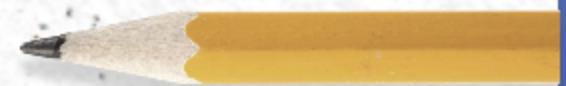
중재자 제거하기(*Remove middle man*)

함수 인라인(*Inline function*)

슈퍼클래스를 위임으로 바꾸기(*Replace Superclass with Delegate*)

서브클래스를 위임으로 바꾸기(*Replace Subclass with Delegate*)

*리팩토링. 중재자 제거하기



"위임 숨기기"의 반대에 해당하는 리팩토링 기법

필요한 캡슐화의 정도는 시간에 따라 그리고 상황에 따라 바뀔 수 있다.

캡슐화의 정도를 "중재자 제거하기"와 "위임 숨기기" 리팩토링을 통해 조절할 수 있다.

위임하고 있는 객체를 클라이언트가 사용할 수 있도록 `getter`를 제공하고, 클라이언트는 메시지 체인을 사용하도록 코드를 고친 뒤에 캡슐화에 사용했던 메소드를 제거한다.

`Law of Demeter`를 지나치게 따르기 보다는 상황에 맞게 활용하도록 하자

- 디미터의 법칙, "가장 가까운 객체만 사용한다."

*리팩토링. 중재자 제거하기



```
Person manager = aPerson.getManager();
class Person {
    Person getManager() {
        return getDepartment().getManager();
    }
    getDepartment() { ... }
}
class Department {
    getManager() { ... }
}
```



```
Person manager = aPerson.getDepartment().getManager();
class Person {
    getDepartment() { ... }
}
class Department {
    getManager() { ... }
}
```

*리팩토링. 슈퍼클래스를 위임으로 바꾸기



객체지향에서 "상속"은 기존 기능을 재사용하는 쉬우면서 강력한 방법이지만 때로는 적절하지 않은 경우도 있다.

서브클래스는 슈퍼클래스의 모든 기능을 지원해야 한다.

- Stack이라는 자료구조를 만들 때 List를 상속 받는 것이 좋을까?

서브클래스는 슈퍼클래스 자리를 대체하더라도 잘 동작해야 한다.

- 리스코프 치환 원칙

서브클래스는 슈퍼클래스의 변경에 취약하다.

*리팩토링. 슈퍼클래스를 위임으로 바꾸기



```
class List { ... }
```

```
class Stack extends List { ... }
```



```
class List { ... }
```

```
class Stack {
    Stack() {
        this.storage = new
        List();
    }
}
```

*리팩토링. 서브 클래스를 위임으로

어떤 객체의 행동이 카테고리에 따라 바뀐다면, 보통 상속을 사용해서 일반적인 로직은 슈퍼클래스에 두고 특이한 케이스에 해당하는 로직을 서브클래스를 사용해 표현한다.

하지만, 대부분의 프로그래밍 언어에서 상속은 오직 한번만 사용할 수 있다.

- 만약 어떤 객체를 두 가지 이상의 카테고리로 구분해야 한다면?
- 위임을 사용하면 얼마든지 여러가지 이유로 여러 다른 객체로 위임을 할 수 있다.

슈퍼클래스가 바뀌면 모든 서브클래스에 영향을 줄 수 있다. 따라서 슈퍼클래스를 변경할 때 서브클래스까지 신경써야 한다.

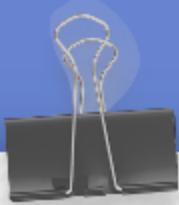
- 만약에 서브클래스가 전혀 다른 모듈에 있다면?
- 위임을 사용한다면 중간에 인터페이스를 만들어 의존성을 줄일 수 있다.

처음에는 상속을 적용하고 이후 상황에 따라서 해당 리팩토링 기법을 통해 위임으로 전환할 수 있다.

*리팩토링. 서브 클래스를 위임으로

```
class Order {  
    int daysToShip() {  
        return warehouse.daysToShip;  
    }  
}  
  
class PriorityOrder extends Order {  
    int daysToShip() {  
        return priorityPlan.daysToShip;  
    }  
}
```

```
class Order{  
    private PriorityOrderDelegate delegate;  
    int daysToShip() {  
        return (priorityDelegate != null ? delegate.daysToShip() :  
               warehouse.daysToShip);  
    }  
}  
  
class PriorityOrderDelegate {  
    int daysToShip() {  
        return priorityPlan.daysToShip;  
    }  
}
```



클린코드 끌팁을 놓치고 있다면?

클린코드 팁을
구독하세요:)

구독하기?!