쉽게 익히는
클린코드

이번주 컨텐츠

~냄새: 반복되는 switch 문, 반복문, 성의없는 요소

*리팩토링: 반복문을 파이프라인으로, 계층 합치기

전략파트너개발그룹

01

~ 냄새. 반복되는 switch 문

- 예전에는 swith문이 한번만 등장해도 코드 냄새로 판단되어 다형성 적용을 권장했다.
- 하지만 최근에는 다형성이 꽤 널리 사용되고 있으며, 여러 프로그래밍 언어에서 보다 세련된 형태의 switch문을 지원하고 있다.
- 따라서 오늘날은 "반복해서 등장하는 동일한 switch 문"을 냄새로 여기고 있다.
- 반복해서 동일한 switch 문이 존재할 경우, 새로운 조건을 추가하거나 기존의 조건을 변경할 때 모든 switch 문을 찾아서 고쳐야 할지도 모른다.

적용 가능한 리팩토링 기법
- switch를 Polymorphism으로 변경 -

함수 추출하기

조건부 로직을 다형성으로 바꾸기

타입 코드를 서브클래스로 바꾸기

함수 옮기기

02

~ 냄새. 반복문

- 프로그래밍 언어 초기부터 있었던 반복문은 처음엔 별다른 대안이 없어서 간과했지만, 최근 다양한 언어에서 함수형 프로그래밍을 지원하면서 반복문에 비해 더 나은 대안책이 생겼다.

- "**반복문을 파이프라인으로 바꾸기(Replace Loop with Pipeline)**" 리팩토링을 적용하면 필터나 맵핑과 같은 파이프라인 기능을 사용해 보다 빠르게 어떤 작업을 하는지 파악할 수 있다.

- **Collection 탐색**을 위해 반복문을 사용하는 경우 파이프라인 기법으로 적용하면 된다.

적용 가능한 리팩토링 기법

반복문을 파이프라인으로 바꾸기

*리팩. 반복문을 파이프라인으로 바꾸기

- ✓ Collection 파이프라인(Java-Stream, C#-LINQ)
- ✓ 고전적인 반복문을 Pipelin operation을 사용해 표현하면 코드를 더 명확하게 만들 수 있다.
  - **필터(Filter)**: 전달받은 조건의 true에 해당하는 데이터만 다음 오퍼레이션으로 전달
  - **맵(Map)**: 전달받은 함수를 사용해 입력값을 원하는 출력값으로 변환하여 다음 오퍼레이션으로 전달

```
List<String> names = new ArrayList<>();
for(Person p: people) {
        if("programmar".equals(p.job))
                names.add(p.name);
}
```

↓

```
List<String> name = people.stream()
                .filter(p → "programmar".equals(p.job))
                .map(p → p.name)
                .collect(Collectors.toList());
```

~ 냄새. 성의없는 요소

- 여러 프로그래밍적인 요소(변수, 메소드, 클래스 등)를 만드는 이유
  * 나중에 발생할 변화를 대비해서
  * 해당 함수 또는 클래스를 재사용하려고
  * 의미있는 이름을 지어주려고
- 가끔 그렇게 예상하고 만들어 놓은 요소들이 기대에 부응하지 못하는 경우가 있는데 그런 경우에 해당 요소들을 제거해야 한다.

적용 가능한 리팩토링 기법

함수 인라인

클래스 인라인

계층 합치기

*리팩. 계층 합치기

- 상속 구조를 리팩토링하는 중에 기능을 올리고 내리다 보면 하위 클래스와 상위 클래스 코드에 차이가 없는 경우가 발생할 수 있다. 그런 경우에 그 둘을 합칠 수 있다.

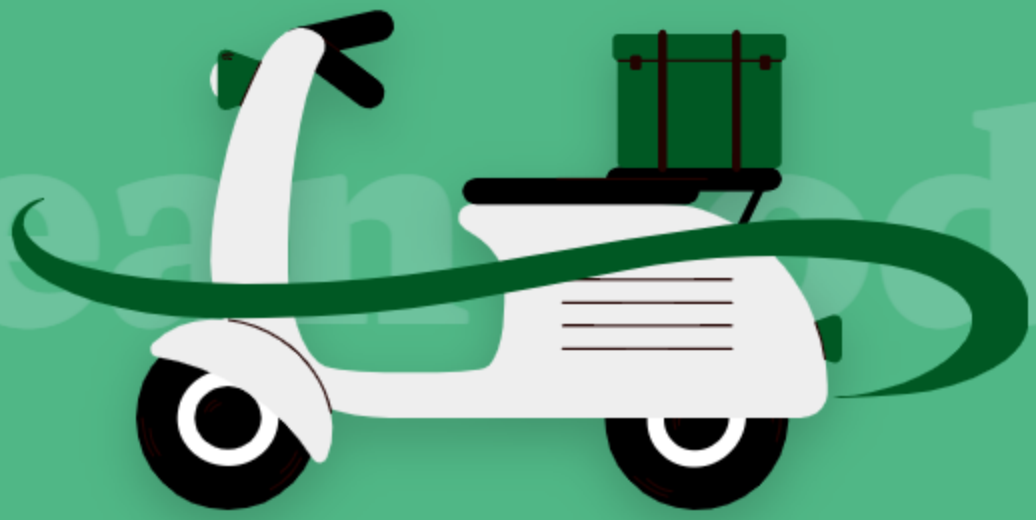
- 하위클래스와 상위클래스 중에 어떤 것을 없애야 하는가?
  - 둘 중에 보다 이름이 적절한 쪽을 선택하지만, 애매하다면 어느 쪽을 선택해도 문제없다.

```
class Employee {
        // A
}

class Salesman extends Employee {
        // B
}
```

```
class SalesEmployee {
        // A + B
}
```

쉽고 빠르게
'클린코더'가 되는 법



구독하기

전략파트너개발그룹 - 양지용