

쉽게 익히는 클린코드

이번주 콘텐츠

~냄새: 기능 편애, 데이터 뭉치, 기본형 집착
✱리팩토링: 기본형을 객체로 바꾸기, 타입 코드를 서브클래스
로 바꾸기, 조건부 로직을 다형성으로 바꾸기



전략파트너개발그룹

~ 냄새. 기능 편애

어떤 모듈에 있는 함수가 다른 모듈에 있는 데이터나 함수를 더 많이 참조하는 경우

적용 가능한 리팩토링 기법

함수 옮기기

함수 추출하기

- ✓ 만약 여러 모듈을 참조하고 있다면, 그 중에서 가장 많은 데이터를 참조하는 곳으로 옮기거나, 함수를 여러개로 쪼개서 각 모듈로 분산시킬수도 있다.
- ✓ 데이터와 해당 데이터를 참조하는 행동을 같은 곳에 두도록 하자.
- ✓ 예외적으로, 데이터와 행동을 분리한 디자인 패턴(전략 패턴 또는 방문자 패턴)을 적용할 수도 있다.

~ 냄새. 데이터 뭉치

여러 비슷한 데이터들이 여러 클래스/함수에 걸쳐 선언
되어 있는 경우

적용 가능한 리팩토링 기법

클래스
추출하기

매개변수 객체
만들기

객체 통째로
넘기기

✓ 항상 뭉쳐 다니는 데이터는 **한 곳**으로 모아두는 것이 좋다.

* 여러 클래스에 존재하는 비슷한 필드 목록

* 여러 함수에 전달하는 매개변수 목록

~ 냄새. 기본형 집착

도메인에 맞는 기본 데이터 타입을 만들지 않고 프로그래밍 언어가 제공하는 **기본 타입만을 사용**하려는 경우

✓ 기본형으로는 단위(인치 vs 미터) 또는 표기법을 표현하기 어렵다.

* 여러 클래스에 존재하는 비슷한 필드 목록

* 여러 함수에 전달하는 매개변수 목록

적용 가능한 리팩토링 기법

기본형을 객체로 바꾸기

타입 코드를 서브 클래스로 바꾸기

조건부 로직을 다형성으로 바꾸기

클래스 추출하기

매개변수 객체 만들기

*리팩. 기본형을 객체로 바꾸기

- ✓ 개발 초기에는 기본형(숫자 또는 문자열)으로 표현한 데이터가 나중에는 해당 데이터와 관련있는 다양한 기능을 필요로 하는 경우가 발생한다.
 - 예) 문자열로 표현하던 전화번호의 지역 코드가 필요하거나 다양한 포맷을 지원하는 경우
 - 예) 숫자로 표현하던 온도의 단위(화씨, 섭씨)를 변환하는 경우
- ✓ **기본형을 사용한 데이터를 감싸 줄 클래스**를 만들면, 필요한 기능을 추가할 수 있다.

```
orders.filter(o -> "high".equals(o.priority) ||  
                  "rush".equals(o.priority));
```



```
orders.filter(o -> o.priority.hightThan(  
                  new Priority("normal"));
```

*리팩. 타입 코드를 서브클래스로 바꾸기

- ✓ 비슷하지만 다른 것들을 표현해야 하는 경우, 문자열(String), 열거형(enum), 숫자(Int) 등으로 표현하기도 한다.

예) 주문 타입, "일반 주문", "빠른 주문"

예) 직원 타입, "엔지니어", "매니저", "세일즈"

- ✓ 타입을 서브클래스로 바꾸는 계기

- 조건문을 다형성으로 표현할 수 있을 때, 서브클래스를 만들고 "조건부 로직을 다형성으로 바꾸기"를 적용한다.
- 특정 타입에만 유용한 필드가 있을 때, 서브클래스를 만들고 "필드 내리기"를 적용한다.

*리팩. 타입 코드를 서브클래스로 바꾸기

```
Employee aEmployee = createEmp(type, name);
Employee createEmp(type, name) {
    return new Employee(type, name);
}
class Employee {
    private int type; // E, S, M
    private String name;
}
```



```
Employee aEmployee = createEmp(type, name);
Employee createEmp(type, name) {
    if(type == E ) return new Engineer(name);
    if(type == S ) return new SalesPerson(name);
    if(type == M ) return new Mananger(name);
}
class Employee {
    private String name;
}
class Engineer extends Employee { ... }
class SalesPerson extends Employee { ... }
class Manager extends Employee { ... }
```


*리팩. 조건부 로직을 다형성으로 바꾸기

- ✓ 복잡한 조건식을 상속과 다형성을 사용해 코드를 보다 명확하게 분리할 수 있다.
- ✓ **switch 문**을 사용해서 타입에 따라 각기 다른 로직을 사용하는 코드
- ✓ 기본 동작과 (타입에 따른) 특수한 기능이 섞여있는 경우에 상속 구조를 만들어서 기본 동작을 상위클래스에 두고 특수한 기능을 하위클래스로 옮겨서 각 타입에 따른 "차이점"을 강조할 수 있다.
- ✓ 모든 조건문을 다형성으로 옮겨야 하는가? 복잡한 조건문을 다형성을 활용해 좀 더 나은 코드로 만들 수 있는 경우에만 적용한다.



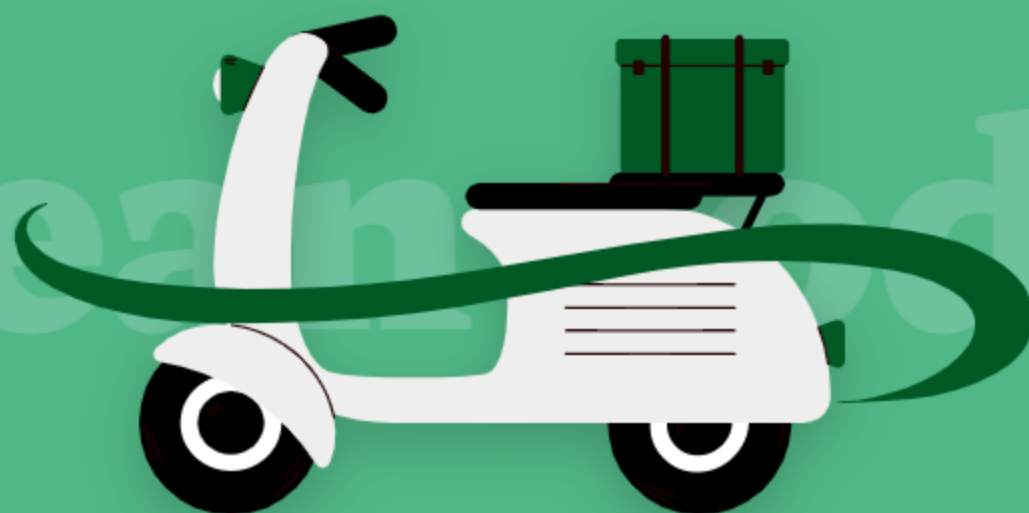
*리팩. 조건부 로직을 다형성으로 바꾸기

```
class Bird {  
    double getSpeed() {  
        switch(type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor();  
            case NORWEGIAN:  
                return (isNailed) ? 0 :  
                    getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("...");  
    }  
}
```



```
abstract class Bird {  
    abstract double getSpeed();  
}  
class European extends Bird {  
    double getSpeed() { return ...; }  
}  
class African extends Bird {  
    double getSpeed() { return ...; }  
}  
class Norwegian extends Bird {  
    double getSpeed() { return ...; }  
}
```

쉽고 빠르게 '클린코더'가 되는 법



구독하기

전략파트너개발그룹 - 양지용