전략파트너개발그룹

쉽게 익히는 클린코드

제04호

~냄새. 긴 함수

짧은 함수 vs 긴 함수

- 함수가 길 수록 더 이해하기 어렵다. vs 짧은 함수는 더 많은 문맥 전환을 필요로 한다.
- "과거에는" 작은 함수를 사용하는 경우 서브루틴 호출로 인한 오버헤드가 있었다.
- 작은 함수에 "좋은 이름"을 사용했다면 해당 함수의 코드를 보지 않고도 이해할 수 있다.
- 어떤 코드에 *"주석"을 남기고 싶다면*, 주석 대신 함수를 만들고 함수의 이름으로 "의도"를 표현해보자.

~냄새. 긴 함수

사용할 수 있는 리펙토링 기술

- 99%는 "함수 추출하기(Extract Function)"로 해결할 수 있다.
- 함수를 분리하면서 해당 함수로 전달해야 할 매개변수가 많아진다면 다음과 같은 리펙토링을 고려해볼 수 있다.
  - 임시 변수를 질의 함수로 바꾸기
  - 매개변수 객체 만들기
  - 객체 통째로 넘기기
  - 함수를 명령으로 바꾸기
- "조건문 분해하기"를 사용해 조건문 분리할 수 있다.
- 같은 조건으로 여러개 Switch문이 있다면, "조건문을 다형성으로 바꾸기"을 사용할 수 있다.
- 반복문 안에서 여러 작업을 하고 있어서 하나의 메소드로 추출이 어렵다면, "반복문 쪼개기"를 적용할 수 있다.

03호를 확인해주세요

*리펙토링. 함수를 명령으로 바꾸기

Replace Function with Command

함수를 독립적인 객체인, Command로 만들어 사용할 수 있다.

커맨드 패턴을 적영하면 다음과 같은 장점을 취할 수 있다.

1. 부가적인 기능으로 undo 기능을 만들 수 있다.
2. 더 복잡한 기능을 구현하는데 필요한 여러 메소드를 추가할 수 있다.
3. 상속이나 템플릿을 활용할 수도 있다.
4. 복잡한 메소드를 여러 메소드나 필드를 활용해 쪼갤 수 있다.

대부분의 경우에 "커맨드"보다는 "함수"를 사용하지만, 커맨드 말고 다른 방법이 없는 경우에만 사용한다.

*리펙토링. 함수를 명령으로 바꾸기

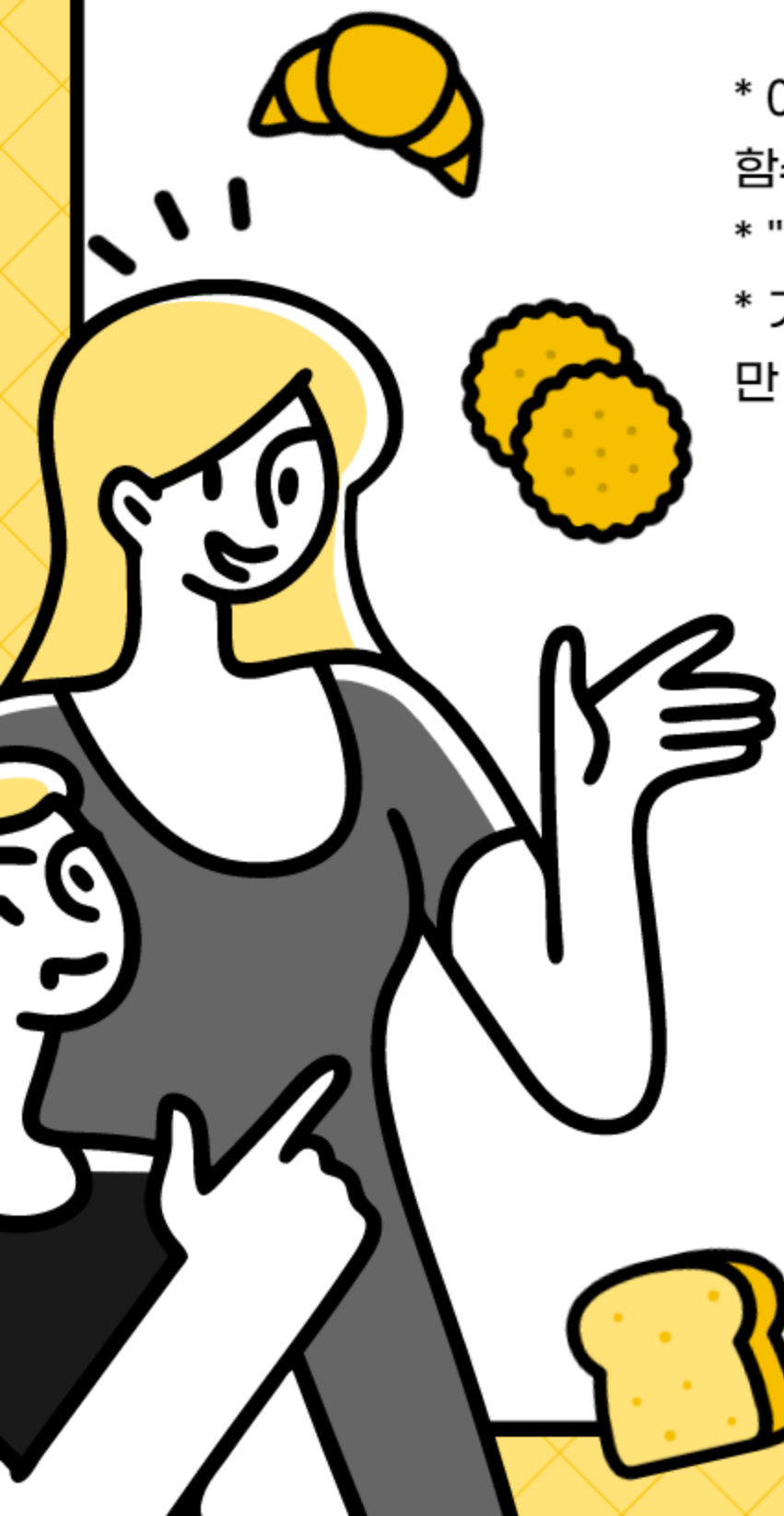
Replace Function with Command

```
double score(candidate, medicalExam, scoringGuide) {
    double result = 0;
    double healthLevel = 0;
    // Perform long computation
}
```

↓

```
double score(candidate, medicalExam, scoringGuide) {
    return new Scorer(candiate, medicalExam, scoringGuide);
}
class Scorer {
    public Scorere(candidate, medicalExam, scoringGuide) {
        this.cadidate = candidate;
        ...
    }
    double execute() {
        this.result = 0;
        this.healthLevel = 0;
        // Perform long computation
    }
}
```

*리펙토링. 조건문 분해하기



Decompose Conditional

* 여러 조건에 따라 달라지는 코드를 작성하다보면 종종 긴 함수가 만들어지는 것을 목격할 수 있다.
* "조건"과 "액션" 모두 "의도"를 표현해야 한다.
* 기술적으로는 "함수 추출하기"와 동일한 리팩토링이지만 의도만 다를 뿐이다.

```
if(date.before(SUMMER_START) || date.after(SU
MMER_END))
   charge = quantity * winterRate + winterServic
eCharge;
else
   charge = quantity * summerRate;
```

↓

```
if(isSummer(currentDate))
    charge = summerCharge(quantity);
else
   charge = winterCharge(quantity);
```

*리펙토링. 반복문 쪼개기

Split Loop

* 하나의 반복문에서 여러 다른 작업을 하는 코드를 쉽게 찾아볼 수 있다.
* 해당 반복문을 수정할 때 여러 작업을 모두 고려하여 코딩을 해야한다.
* 반복문을 여러개로 쪼개면 보다 쉽게 이해하고 수정할 수 있다.
* 성능 문제를 야기할 수 있지만, "리팩토링"은 "성능 최적화"와 별개의 작업이다. 리팩토링을 마친 이후에 성능 최적화를 시도할 수 있다.

```
double averageAge = 0;
double totalSalary = 0;
for(Person p of people) {
   averageAge += p.age;
   totalSalary += p.salary;
}
avergaeAge = averageAge / people.length;
```

↓

```
double totalSalary = 0;
for(Person p of people)
    totalSalary += p.salary;
double averageAge = 0;
for(Person p of people)
   averageAge += p.age;
averageAge = averageAge / people.length;
```

*리펙토링. 조건문을 다형성으로 바꾸기

Replace Conditional with Polymorphism

여러 타입에 따라 각기 다른 로직으로 처리해야 하는 경우에 다형성을 적용해서 조건문을 보다 명확하게 분리할 수 있다. (예, 책, 음악, 음식 등 ...) 반복되는 switch문을 각기 다른 클래스를 만들어 제거할 수 있다.

공통으로 사용되는 로직은 상위클래스에 두고 달라지는 부분만 하위 클래스에 둠으로써, 달라지는 부분만 강조할 수 있다.

모든 조건문을 다형성으로 바꿔야 하는 것은 아니다.

```
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 ...; }
}
```

클린코드 꿀팁을 놓치고 있다면?

클린코드 팁 구독해요!

전략파트너개발그룹 - 양지용