Chaper 3. 함수

☑ Reviewed	☐
생성 일시	@2024년 9월 11일 오후 1:36

의도를 분명히 하는 함수를 어떻게 구현할 수 있을까?

- 작게 만들어라
  - 블록과 들여쓰기
    - fi문/else문/while문 등에 들어가는 블록은 한 줄이어야 한다.
    - 블록 안에서 호출하는 함수 이름을 적절히 짓는다면 코드를 이해하기도 쉬워진다.
    - 함수에서 들여쓰기 수준은 1단이나 2단을 넘어서면 안된다.
- 한 가지만 해라
  - 함수는 한 가지를 해야 한다. 그 한 가지를 잘해야한다. 그 한 가지만을 해야한다.
  - '한 가지'를 판단하는 방법?
    - 지정된 함수 이름 아래에서 추상화 수준인 하나인 단계만 수행하는지 판단
    - 단순히 다른 표현이 아니라 의미 있는 이름으로 다른 함수를 추출할 수 있는지 판단
    - 한 가지 작업한 하는 함수는 자연스럽게 섹션으로 나누기 어렵다.
- 함수당 추상화 수준은 하나로
  - 함수가 확실히 '한 가지' 작업만 하려면 함수 내 모든 문장의 추상화 수준이 동일해야한다.
  - 위에서 아래로 코드 읽기: 내려가기 규칙
    - 코드는 위에서 아래로 이야기처럼 읽혀야 좋다.
    - 위에서 아래로 프로그램을 읽으면 함수 추상화 수준이 한 번에 한 단계씩 낮아진다.
- Switch 문
  - switch 문은 작게 만들기 어렵다. '한 가지' 작업만 하는 switch 문도 만들기 어렵다.

    - switch 문을 완전히 피할 방법은 없으나, 각 switch 문을 저차원 클래스에 숨기고 절대로 반복하지 않는 방법이 있다. (polymorphism을 이용한다.)
    - 예시) 직원 유형에 따라 다른 값을 계산해 반환하는 함수

```
public Money calculatePay(Employee e)
```

- 서술적인 이름을 사용하라
    - 이름이 길어도 괜찮다. 이름을 정하느라 시간을 들여도 괜찮다(최신 IDE에서 이름 바꾸기는 아주 쉽다).
- 함수 인수
    - 함수에서 이상적인 인수 개수는 0개(무항)이다. 그 다음은 1개(단항), 그 다음은 2개(이항). 3개(삼항)은 가능한 피하는 편이 좋다.
    - 4개 이상(다항)은 특별한 이유가 필요하다. 특별한 이유가 있어도 사용하면 안된다.
    - 인수는 개념을 이해하기 어렵게 만든다.
    - 인수가 많아지면 테스트 케이스를 작성하기도 어려워진다.
    - 출력 인수는 입력 인수보다 이해하기 어렵다.
    - 최선은 입력 인수가 없는 경우이며, 차선은 입력 인수가 1개 뿐인 경우다.
    - 많이 쓰는 단항 형식
        - 인수에게 질문을 던지는 경우 (boolean fileExits("MyFile"))
        - 인수를 뭔가로 변환해 결과를 반환하는 경우 (InputStream fileOpen("MyFile"))
        - 함수 이름을 지을 때, 그 경우를 명확히 구분해야하고, 일관적인 방식으로 두 형식을 사용해야 한다.
        - 이벤트 함수: 출력 인수 없이 입력 인수만 있다. 입력 인수로 시스템 상태를 바꾼다.
            - passwordAttemptFailedNtimes(int attempts)
            - 이벤트라는 사실이 코드에 명확히 드러나야 한다.
            - 이름과 문맥을 주의해서 선택해야한다.
        - 위의 경우가 아니라면 단항 함수는 가급적 피한다.
    - 플래그 인수 사용을 자제하자.

    - 이항 함수
        - 이항 함수가 적절한 경우: 직교 좌표계 점-한 값을 표현하는 두 요소. 자연적인 순서도 존재.
        - 이항 함수를 사용했을 때 발생할 수 있는 위험성을 이해하고 가능하면 단항 함수로 바꾸도록 애써야 한다.
    - 삼항 함수
        - 단항, 이항 함수에 비해 이해하기 훨씬 어려워지므로 신중히 고려해야한다.
        - 예외)asserstEquals(1.0,amount,.001): 부동 소수점 비교가 상대적이라는 사실은 언제든 주지할 중요한 사항.
    - 인수 객체
        - 인수가 2-3개 필요하다면 일부를 독자적인 클래스 변수로 선언할 가능성을 짚어본다. 결국 개념을 표현하게 된다.
    - 인수 목록
        - 인수 개수가 가변적인 함수가 존재한다.
        - ex-String.format("%s worked %.2f hours", name, hours);
            - format의 선언부를 살펴보면 public String format (String format, Object ... args)로, 결국 이항 함수로 취급하는 것을 볼 수 있다.
        - 위와 같은 식으로 단항, 이항, 삼항으로 취급할 수 있음. 그러나 이를 넘어가는 인수 개수를 사용한다면 문제가 있다.
    - 동사와 키워드
        - 함수의 의도나 인수의 순서와 의도를 제대로 표현하려면 좋은 함수 이름이 필요하다.
            - 함수 이름에 키워드를 추가하여, 인수 순서를 기억할 필요 없게 만들 수 있다.
- 부수 효과를 일으키지 마라
    - 많은 경우 시간적인 결합(temporal coupling)이나 순서 종속성(order dependency)을 초래한다.
    - 출력 인수
        - 일반적으로 출력 인수는 피해야 한다.

        - 함수에서 상태를 변경해야 한다면 함수가 속한 객체 상태를 변경하는 방식을 택한다.
- 명령과 조회를 분리하라
    - 함수는 뭔가를 수행하거나 뭔가에 답하거나 둘 중 하나만 해야한다.
    - 객체 상태를 변경하거나 아니면 객체 정보를 반환하거나 둘 중 하나다. 둘 다 하면 혼란을 초래한다.
- 오류 코드보다 예외를 사용하라
    - 명령 함수에서 오류 코드를 반환하는 방식은 명령/조회 분리 규칙을 미묘하게 위반한다.
        - if(deletePage(page)==E_OK)처럼, 명령을 표현식으로 사용하기 쉽기 때문.
            - 여러 단계로 중첩되는 코드를 야기한다.
            - 오류 코드를 반환하면 오류 코드를 곧바로 처리해야한다는 문제점이 존재.
        - 오류 코드 대신 예외를 사용하면 오류 처리 코드가 원래 코드에서 분리되므로 코드가 깔끔해진다.
    - Try/Catch 블록 뽑아내기
        - 정상 동작과 오류 처리 동작을 분리하면 코드를 이해하고 수정하기 쉬워진다.
    - 오류 처리도 한 가지 작업이다.
    - Error.java 의존성 자석
        - 오류 코드를 반환한다는 이야기는, 어디선가 오류 코드를 정의한다는 뜻이다.
        - Error enum이 변한다면 Error enum을 사용하는 클래스 전부를 다시 컴파일하고 다시 배치해야한다.
        - 오류 코드 대신 예외를 사용하면 새 예외는 Exception 클래스에서 파생된다. 따라서 재컴파일/재배치 없이도 새 예외 클래스를 추가할 수 있다. (OCP: Open Closed Principle의 예시)
- 반복하지 마라 (DRY: Don’t Repeat Yourself)
    - 알고리즘이 변하면 중복된 곳을 모두 일일히 다 손봐야 한다.
- 구조적 프로그래밍
    - 에츠허르 데이크스트라(Edsger Dijkstra)의 구조적 프로그래밍 원칙
        - 모든 함수와 함수 내 모든 블록에 입구(entry)와 출구(exit)가 하나만 존재해야 한다. 즉, 함수는 return 문이 하나여야 한다.

        - 루프 안에서 break나 continue를 사용해서는 안되며, goto는 절대로 안 된다
    - 함수가 아주 클 때만 상당한 이익을 제공한다.
        - 함수가 작을 땐 return, break, continue를 여러 차례 사용해도 괜찮으며, 단일 입/출구 규칙보다 의도를 표현하기 쉬워진다. goto는 피해야한다.
- 함수를 어떻게 짜죠?
    - 처음에는 길고 복잡하고, 들여쓰기 단계도 많고 중복된 루프도 많다. 인수 목록도 아주 길다. 이름은 즉흥적이고 코드는 중복된다. 이를 모두 다 테스트하는 단위 테스트 케이스도 만든다.
    - 이후에 코드를 다듬고, 함수를 만들고, 이름을 바꾸고, 중복을 제거한다. 메서드를 줄이고 순서를 바꾼다. 때로는 전체 클래스를 쪼개기도 한다. (단위 테스트는 계속 통과한다.)
    - 처음부터 완벽하게 짜는 것이 아닌, 고쳐나가는 방식.