**Clean Code**

20220019 안재영

**Chapter 2. Meaningful Names – 좋은 이름을 짓는 것이 중요하다**

**“이름”을 지을 때에는 의도를 분명히 밝혀야 한다.**

theList보다는 gameBoard가 더 좋다.

**의도를 분명히 하는 것에는 잘못된 이해를 막는 것도 중요하다.**

자료형이 List가 아닌 오브젝트를 accountList라고 명명하는 것을, 0과 O, 1과 l같이 혼동을 줄 수 있는 이름을 사용하는 것을 피해야 한다.

**발음하기 쉬운 이름을 사용하는 것도 좋다.**

말하기 쉬운 이름은 기억하기에도, 이해하기에도, 대화하기에도 더 적합하다.

**검색하기 쉬운 이름을 사용하라.**

짧은 이름이 좋은 이름은 아니다. WORK\_DAYS\_PER\_WEEK은 의미도 분명하며 검색하기도 쉬운 이름이다. E와 같은 한 글자 문자는 로컬 변수에만 사용한다.

**오브젝트의 종류에 따라 적합한 종류의 단어를 사용하라.**

클래스, 객체의 이름은 명사나 명사구가 적합하다. 동사를 사용하지 않고, data, info 등의 단어를 피하고 customer, account와 같은 단어가 좋다.

메소드 이름은 동사나 동사구가 적합하다. deletePage, save 등이 좋다.

**전체 코드에서 한가지 개념을 나타내는 단어를 통일하다.**

동일한 기능의 메소드를 클래스마다 fetch, retrieve, get으로 제각각 부르는 것을 피해라.

반대로, 한 단어를 두 가지 목적으로 사용해서도 안 된다.

**함수 이름에 맥락을 부여하라.**

firstName, lastName, houseNumber라는 이름으로는 맥락을 파악할 수 없지만, 그 앞에 addr라는 접두어를 추가하면 변수가 더 큰 구조인 “주소”에 속함이 분명해진다.

**다만, 불필요한 맥락은 제거해야 한다.**

accountAddress, customerAddress 클래스 인스턴스로 사용해서는 좋겠지만, 클래스 이름으로는 Adress가 더 좋다.

**Chapter 3. Functions – 짧고 명확한 함수를 만들어라**

**함수는 작게 만들어야 한다.**

중첩 구조가 없이 if, else, while 문 안에 함수를 호출하는 한 블록만이 들어가게 코드를 짜고 함수의 이름까지 적절하게 짓는다면 코드를 이해하기가 훨씬 편해진다.

**함수는 한 가지 일만을 해야 한다.**

함수가 이 조건을 만족하는지 의문스럽다면, 의미 있는 다른 함수를 추출해낼 수 있지 않은지 확인하고, 함수 내의 모든 문장이 동일한 추상화 수준에 있도록 한다.

이를 통해 함수가 확실히 한 가지 작업만 하도록 할 수 있고, 코드를 읽는 사람의 혼동을 막을 수 있다.

함수 내에 별도의 부수 효과를 구현하면 함수의 호출을 오히려 제한할 수 있다.

**함수의 기능을 잘 드러내는 서술적인 이름을 사용하라.**

다양한 함수들에 일관성 있게 명확한 이름을 붙인다면 그 길이는 중요하지 않다.

**함수의 인수는 적을수록 좋다.**

인수는 개념을 이해하기 어렵게 만든다. writeField(name)은 writeField(outputStream, name)보다 이해하기 쉽다.

인수가 여러 개라면, assertExpectedEqualsActual(expected, actual)과 같이 명확한 함수 이름으로 인수에 대한 이해를 도울 수 있다.

이상의 조건들을 갖춘 짧고 이름이 명확하며 체계적인 함수는 한번에 완성되지 않으므로, 여러 번의 수정을 거치는 것을 서슴지 않아야 한다.

**Chapter 4. Comments – 주석을 가능한 줄이도록 노력하라**

**“좋은, 혹은 필요한” 주석들**

<법적인 주석>

- 계약 조건이나 저작권, 소유권 등 법적 정보를 파일 첫머리에 주석으로 적기도 한다.

// Copyright © 2003, 2004, 2004 by Object Mentor, Inc. All rights reserved

<필요한 정보를 제공하는 주석>

- 코드에서 사용된 표현식(예를 들어, \\d\*:\\d\*:\\d\*가 kk:mm:ss 형식이라고 설명)에 대한 정보나 구현의 바탕에 깔린 의도, 애매한 인수나 반환 값의 명확한 의미 등을 밝혀주면 좋다.

<다른 프로그래머에게 전달하기 위한 주석>

- 시간이 오래 걸리는 특정 테스트 케이스와 같이 결과를 경고하거나, TODO를 사용해 앞으로 구현해야 할 기능을 알려주거나, 특정 부분의 중요성을 강조하는 용도로 주석을 사용하기도 한다.

**“나쁜” 주석들**

- 특별한 이유 없이 의무적으로 작성한 주석, 코드의 내용을 그대로 반복한 주석, 지극히 당연한 내용을 담은 주석 등 “필요 없는” 내용에 주의해야 한다.

- 코드 수정의 이력을 기록하는 주석, 저자를 표시하는 주석은 과거에는 일반적이었을 지 몰라도 이제는 주석이 아닌 방법으로 충분히 남길 수 있다.

- 주석이 필요한 상황이라도, 필요 이상의 정보를 담아 길어지는 것에 주의해야 한다.

**“주석은 나쁜 코드를 보완하는 도구가 아니다”**

품질이 나쁜 코드를 주석으로 보완하려고 할 것이 아니라, 코드를 정리해야 한다.

코드의 의도를 분명히 하기 위해서 주석을 추가하고 싶다면, 적절한 이름의 변수와 함수를 사용하는 것으로 대부분 대신할 수 있다.

**Chapter 7. Error Handling**

**– 오류 처리는 중요하지만, 프로그램 이해를 방해하면 안 된다**

**Try-Catch-Finally 구문을 사용하라**

- 이제는 많은 프로그래밍 언어들이 예외를 지원한다. 조건문을 사용해 오류를 처리하는 코드를 알고리즘과 섞어서 작성하는 것 대신, 오류를 발견하면 예외를 던지도록 하여 두 기능을 명확히 분리할 수 있다.

- 아래와 같이 오류 처리 코드를 메인 알고리즘인 tryToShutDown의 논리와 분리해 깔끔하게 할 수 있다.

try {

tryToShutDown( );

} catch (DeviceShutDownError e) {

Logger.log(e);

}

- 먼저 강제로 예외를 일으키는 테스트 케이스를 작성한 후 테스트를 통과하는 코드를 작성하면, 자연적으로 try 블록의 트랜잭션 범위부터 구현하게 되므로 해당 범위 내에서의 트랜잭션 본질 유지가 쉬워진다.

**예외를 고려하여 코드를 작성하라**

- 예외를 던질 때 전후 상황을 충분히 덧붙이고 오류 메시지에 명확한 정보를 담아 전달하면, 추후 해당 오류를 해결하는 데에 도움이 된다.

- 예외 처리는 주로 일정한 방식으로 구성되므로, 이 경우에는 예외 처리가 포함된 라이브러리를 감싸는 클래스를 구현해 예외를 잡아서 유형을 반환하도록 하면 코드를 간결하게 할 수 있다.

**특수 사례 패턴으로 예외 처리를 대신할 수도 있다**

- 예외 처리가 코드의 논리 이해를 방해할 경우, 클래스를 수정해 발견되지 않은 요소에 대한 기본값을 지정함으로써 예외 처리 코드를 대신할 수 있다.

public class PerDiemMealExpenses implements MealExpenses {

public int getTotal( ) {

// 기본값으로 일일 기본 식비를 반환한다.

}

}

**null의 사용을 피해라**

- 메소드가 null을 반환하거나 전달하는 방식으로 구현하는 것은 좋지 않다.

- 메소드를 사용하면서 null 확인을 빠트린다면 문제를 일으킬 수 있으며, null 확인이 너무 많아지는 것 역시 좋지 않다.

- 대신 예외를 던지거나 특수 사례 객체를 반환하도록 구현하는 것이 좋다.

public double xProjection(Point p1, Point p2) {

if (p1 == null || p2 == null) {

throw InvalidArgumentException(“Invalid argument”)

}

}