20200429 김도영 소프트웨어설계방법 읽기과제

**클린 코드 Chapter 2: 의미 있는 이름**

프로그래밍에서 이름 짓기는 중요하다. 좋은 이름은 나중에 코드를 읽는 이에게 적절한 맥락을 제공해주고 코드의 의미를 명료하게 만든다. 나쁜 이름은 읽는 이로 하여금 코드가 어떤 의도를 가지고 쓰였는지 추측하게 만들고, 코드의 의미를 혼동하게 만든다. 클린 코드의 저자는 좋은 이름을 짓기 위한 몇 가지 원칙들을 다음과 같이 설명한다.

1. 의도를 분명히 밝혀라.

나쁜 이름을 짓고 주석을 달아 부연 설명하지 말고 처음부터 좋은 이름을 지어라. 수량을 나타내는 변수명일 경우에는 단위를 달아 변수의 의미를 명확히 해라. 코드를 이해하기 위한 맥락을 이름으로 제공해라.

2. 그릇된 정보를 피하라.

특별한 의미를 가진 단어를 변수명에 사용하여 읽는 이를 혼동하게 하지 않는다. 헷갈리기 쉬운 한 글자 이름은 피하고, 다른 객체, 자료형, 변수, 함수는 확실하게 다른 이름을 써서 혼동을 방지한다. 반대로 비슷한 종류의 대상들은 비슷한 이름을 써서 비슷한 일을 할 것임을 예측 가능하게 한다.

3. 의미 있게 구분하라.

군더더기 말, 혹은 불용어를 추가하여 이름 중복 문제를 손쉽게 해결하려 들지 않는다. 독자에게 의도를 잘 전달하는 이름 중 가장 간결한 이름을 사용한다. 같은 타입을 가진 함수 인수가 여러 개라 해서 숫자로만 구분 가능한 이름, 예를 들어 arg1, arg2, …과 같은 이름을 짓지 않는다.

4. 발음하기 쉬운 이름을 사용하라.

단어를 과도하게 축약하여 발음을 어렵게 만들지 않는다. 과도하게 생략된 이름은 읽는 이가 의미를 추측하기 어렵게 할 뿐만 아니라, 차후 의사소통에 지장을 준다.

5. 검색하기 쉬운 이름을 사용하라.

변수, 함수, 객체가 하는 일이 무엇인지 알아보기 위해 소스 코드를 검색하는 일은 흔하다. 과도하게 짧거나 서술적이지 못한 이름, 예를 들어 한 글자 알파벳이나 숫자 등으로 검색을 방해하지 않는다. 상수는 적절한 이름을 붙여 맥락을 제공하고 차후 값이 바뀔 때 모든 사용처를 찾아 일일이 확인 후 바꿔주는 수고를 예방한다.

6. 인코딩을 피하라.

헝가리안 표기법, 멤버 변수 접두사, 인터페이스 접두사 등을 변수 이름에 넣지 않는다. 최근의 IDE는 함수 혹은 변수의 선언부로 바로 이동하거나 들춰보는 기능이 있어 헝가리안 표기법은 더 이상 의미가 없으며, 새로운 타입이 생긴다면 이에 대응하는 새로운 접두사를 또 만들어야 하는 수고가 생겨난다. 게다가 변수의 타입이 바뀐다면 일반적으로 작성된 코드는 논리적 구현만 손보면 그만이지만, 이름에 타입 등을 인코딩한 코드는 이름까지 바꿔주어야 한다. 이 과정을 잊어버린다면 바로 그릇된 정보를 제공하는 이름이 된다.

7. 한 개념에 한 단어를 사용하되 말장난은 하지 마라.

일관적으로 이름을 지어 읽는 이가 헷갈리게 하지 않는다. 무언가를 덧붙이는 기능을 하는 메서드가 여러 개 있다면 append, 네트워크에서 무언가를 받아오는 기능을 하는 메서드가 여러 개 있다면 fetch 등과 같이 비슷한 개념은 같은 단어로 묶는다. 단, 이는 어디까지나 비슷하거나 같은 개념을 가진 대상에만 적용되며, 다른 속성을 가진 대상에 억지로 위를 적용하려 하지 않는다. 어떤 대상의 맨 끝에 무언가를 덧붙이는 게 아닌 중간에 삽입하는 메서드라면 append는 좋지 않은 변수 이름이다.

8. 해법 영역과 문제 영역에서 가져온 이름을 사용하라.

여러 자료구조, 알고리즘, 디자인 패턴 등의 해법 영역과 코드가 해결하고자 하는 문제 영역에서 이름을 가져오는 것은 좋다. 읽는 이 또한 프로그래머이며 해당 문제에 대한 지식이 어느 정도 있을 것이라고 가정할 수 있기 때문이다.

9. 의미 있는 맥락을 추가하되 불필요한 맥락은 피하라.

이름의 의미가 불분명하다면 우선 해당 변수, 함수, 클래스를 상위 클래스, 함수, 이름 공간에 넣는 것을 고려해본다. lookup은 혼자 쓰이기엔 너무 광범위한 이름이지만 TranslationEnvironment.lookup은 무언가를 번역하기 위한 어떤 환경 내에서 (아마 인자로 주어질 무언가를) 찾는 기능을 하는 함수라는 것을 추측하기 쉽다. 모든 시도가 실패한다면 접두사를 추가하되, 코드 재사용성을 저해하는 무의미하거나 불필요한 맥락은 피한다.

10. 읽는 이가 자신과 같은 배경지식을 가지고 있을 것이라 가정하지 말라.

자신만 아는 유머, 자신만 아는 불분명한 변수 이름 등은 피한다. 읽는 이가 프로그래머에게 일반적인 전산 지식과 현재 프로그램이 해결하고자 하는 문제에 대한 지식 이외를 알 것이라고 가정하여서는 안된다.

위 원칙에 비추어서 내가 과거 작성했던 코드를 다시 한 번 살펴보았다. 가장 먼저 눈에 띈 것은 cnthp라는 함수 이름이다. 보자마자 의미를 추측하기는 꽤 힘든 이름이다. cnt가 가끔씩 count의 약어로 쓰인다는 것을 알고 있다면 뭔가를 세는 함수 이름이라는 것은 추측할 수 있겠다. 하지만 뒤의 hp는 휴렛-팩커드인지, 직각삼각형의 빗변인지, 혹은 심하게 축약된 단어인지 추측하기 어렵다. 심지어 전체가 어떤 다른 단어의 약어일 수도 있다.

이 이름이 의도한 원래 발음은 countHeap이었고, 함수의 시그니처는 cnthp (env: env) : int이다. 이 함수는 인자로 주어진 환경 내에 총 몇개의 힙 영역 변수가 있는지를 돌려주는 보조 함수였다. 솔직히 말하자면, 내가 작성한 코드임에도 불구하고 이 이름을 처음 맞닥뜨렸을 때 무엇을 하는 함수인지 추측해내는데 꽤 오랜 시간이 걸렸다. 만약 이 코드를 다시 짠다면 먼저 이 함수를 Environment 클래스의 메서드로 바꾸고, 이름 또한 하는 일을 더 명확히 파악할 수 있도록 countHeapVariable과 같이 바꾸는 편이 좋을 것이다.

**클린 코드 Chapter 3: 함수**

함수는 소스 코드의 가장 기본적인 단위로, 좋은 함수를 짜는 것은 좋은 코드를 짜는 것의 첫걸음이다. 나쁘게 설계된 함수는 읽는 이가 함수가 하는 일을 이해하는 것을 어렵게 만든다. 게다가 ‘더러운’ 함수는 침과 껌으로 범벅이 된 더러운 길바닥과도 같아서, 코드를 나중에 읽는 사람에게도 코드 한 줄을 더 추가해 함수를 더 더럽게 만들도록 충동질한다.

저자는 좋은 함수를 만들기 위한 한 가지 휴리스틱으로 ‘함수의 크기’를 꼽는다. 즉, 함수가 작을수록 경험적으로 더 좋은 함수일 확률이 높다는 이야기이다. 또한, 저자는 함수는 단 한가지 일 만을 하는 것이 이상적이라 말한다. 그렇다면 함수의 크기를 줄이고, 한 가지 일만을 하게 하기 위해서는 어떻게 해야 할까? 거대한 함수를 단순히 여러 함수로 쪼갠다고 자동으로 좋은 함수가 되지는 않는다.

좋은, 작은 함수를 만드는 원칙을 설명하기 위해 저자는 추상화 수준이라는 개념을 설명한다. 프로그램에는 여러가지 추상화 수준이 있다. 예를 들어 문자열을 다루는 프로그램이라면 문자열에 직접 문자를 추가하는 과정은 가장 낮은 추상화 수준에 위치할 것이고, 문자열을 모은 페이지라는 개념은 꽤 높은 추상화 수준에 위치할 것이다. 중요한 점은, 추상화 수준이 동일한 여러 루틴을 모아 놓은 함수는 읽기 쉽지만, 추상화 수준이 뒤섞인 코드는 읽기 어렵다.

어떤 로봇을 제어하는 프로그램을 설계한다고 생각해 보자. 로봇에게 자판기에 가서 사이다 한 캔을 사오라 명령하는 함수를 설계할 때, 모든 로봇의 움직임을 한 함수에 몰아넣는 것은 좋은 생각이 아니다. ‘무릎을 180도로 펴기’, ‘왼 다리를 디딘 상태에서 오른발을 10cm 들기’ 같은 추상화되지 않은 작업과 ‘건물 문까지 이동하기’, ‘출입 태그를 찍어 문을 열기’, 건물 밖으로 나가 자판기까지 이동하기’와 같은 고수준으로 추상화된 작업을 한 함수로 몰아넣는다면 읽는 이는 무릎을 펴는 것과 자판기까지 이동하는 것이 맥락상 무슨 관계가 있는 것인지 머리를 싸맬 것이다.

함수의 이름을 잘 짓는 것도 좋은 함수를 설계하기 위해 중요하다. 변수나 객체와 같이 함수도 이름은 최대한 명료하고 서술적으로 짓는 것이 좋다. 여러 함수, 또는 메서드가 하는 일이 비슷하다면 비슷한 이름으로 서로 연관되어 있음을 읽는 이에게 주지시킨다.

함수의 인수 개수 또한 또 다른 관심사 중 하나이다. 저자의 의견은 ‘함수의 인수 개수는 적을수록 좋다’이다. 무항 함수가 단항보다, 단항 함수가 이항보다 나으며, 삼항 이상의 함수는 되도록 피한다. 인수가 적으면 적을수록 함수가 하는 일을 이해하기 쉽기 때문이다. 특히 불리언 플래그를 함수에 넘겨 플래그에 따라 다른 일을 하게 만들거나, 인자로 출력 변수를 넘겨 함수가 값을 ‘반환’하는 것이 아닌 출력 인자를 ‘바꾸도록’ 만드는 함수는 나쁜 패턴 중 하나이다.

함수는 또한 부수 효과(side effect)가 없어야 한다. 부수 효과를 포함한 함수는 읽는 이가 함수의 시그니처만 보고 함수가 무슨 일을 하는지 추측하기 어렵게 만든다. 부수 효과는 또한 함수를 호출할 때 순서에 따른 종속성을 만들어내, 함수가 반드시 ‘적절한 상황’에서만 호출되어야만 하도록 만든다. 나중에 코드를 보는 이는 부수 효과가 있는 함수를 부주의하게 사용하다가 치명적인 버그를 만들어낸다.

함수의 실행 결과를 나타내는 불리언 값 혹은 열거형을 반환하는 관례는 피한다. 보통 이러한 관습은 상위 루틴의 if 조건문 안에서 함수를 호출하고 반환 값을 검사하는 패턴과 함께 쓰이는데, 자연스럽게 읽히지 않을 뿐만 아니라 함수가 한 번에 하나의 일만을 해야 한다는 원칙에 어긋난다. 만약 함수 실행이 실패할 수도 있다면 오류 코드를 반환하기보다는 예외를 사용한다.

**클린 코드 Chapter 4: 주석**

클린 코드를 처음 읽을 때, 내가 가지고 있던 직관과 가장 대치되던 챕터가 바로 주석 챕터였다. 프로그래밍을 처음 배울 때, 우리는 주석이 코드의 설계 의도를 설명해주고, 변수나 인자가 가진 의미를 명확하게 해주는 ‘선한 것’이라는 인식을 갖게 된다. 게다가 일반적인 프로그래밍 과정에서, 코드를 평가하는 기준에는 ‘주석의 사용’이 들어가게 마련이고, 주석을 언제, 어디서, 얼마나 써야 할 지 모르는 학생들은 점수를 지키기 위해 코드 전체를 주석으로 도배하다시피 코드를 작성한다.

클린 코드의 저자가 주석에 대해 챕터 처음부터 설명하는 바는 사뭇 다르다. 주석은 거짓말을 하고, 코드의 변화를 따라가지 못하며, 읽는 이에게 쓸모없는 정보를 주어 가독성을 해친다. 저자는 주석을 사용하는 대신, 코드 그 자체로 코드 작성자의 의도를 드러내고 코드에 내재된 논리를 설명할 것을 권한다. 모든 메서드, 함수, 클래스에 주석을 달아야 한다는 의무감에 주석을 달거나, Javadocs와 같이 주석을 문서화하는 도구를 사용하기 위해 모든 메서드와 클래스에 주석을 다는 일은 피하도록 강력히 권한다.

저자는 좋은 주석을 달고 나쁜 주석을 피하기 위한 여러 사례를 제시했지만, 모든 사례들은 사실하나의 문장으로 요약 가능하다. 저자가 제시한 좋은 주석을 달기 위한 원칙들을 하나로 요약하자면 다음과 같을 것이다. ‘좋은 주석이란 읽는 이가 코드를 이해하고 사용하기 쉽게 만드는 주석이다.’ 예를 들어, 메서드에 숨겨진 부수 작용을 주석이 설명한다면 이는 좋은 주석이다. 읽는 이가 코드를 사용할 때 자칫 빠질 수 있는 함정에 대해 알리기 때문이다. 코드에서 언뜻 이상해 보이는 설계상 결정을 주석이 설명해준다면 이는 좋은 주석이다. 읽는 이가 코드를 이해하는데 도움을 줄 뿐더러 나중에 그 설계를 망치지 않도록 주석이 돕기 때문이다.

위를 거꾸로 하면 나쁜 주석이란 읽는 이가 코드를 이해하고 사용하는데 오히려 방해가 되는 주석이라 할 수 있을 것이다. 주석이 코드가 말하는 것과 같은 이야기를 하여 읽는 이의 시간을 낭비한다면 이는 나쁜 주석이다. 주석이 형상 관리 도구를 이용해 알 수 있는 정보를 부연하여 읽는 이의 시간을 낭비한다면 이는 나쁜 주석이다. 주석이 불가해하여 읽는 이가 코드를 이해하는데 어떤 도움도 주지 못한다면 이는 나쁜 주석이다.

주석 챕터를 읽고 과거 객체지향프로그래밍 및 프로그래밍기초 과목에서 내가 짰던 코드들을 다시 찾아보았다. (변명을 좀 하자면) 두 과목 모두 평가 기준에 주석이 있어 코드의 거의 모든 줄에 주석을 달았던 기억이 있기 때문이다. 아니나 다를까, 위에서 나쁜 주석의 예시로 들었던 거의 모든 유형이 두 과목에서 짰던 코드에 들어있었다. 다음은 프로그래밍기초 과목에서 짰던 코드의 한 부분이다.

int \*\*board = (int \*\*)calloc(row, sizeof(int \*));

for(i = 0; i < row; i++)

\*(board + i) = (int \*)calloc(col, sizeof(int));

//보드를 저장할 이차원 배열을 선언합니다.

위 코드에서 주석이 과연 필요한가? 주석에 있는 정보는 이미 코드에 다 나와있다. board는 누가 봐도 보드를 저장할 것이라 예상할 수 있으며, 이중 포인터인 것도 변수의 선언에서 바로 알 수 있다. 심지어 board는 이차원 배열이 아니라 할당된 메모리를 가리키는 이중 포인터를 이차원 배열처럼 쓰는 것이므로 거짓말까지 한다. 이 코드에 주석을 달아야 한다면 보드를 저장할 이차원 배열을 선언한다 따위의 동어반복이 아니라, 왜 이중 포인터가 가리키는 메모리를 할당할 때 malloc이 아닌 calloc을 사용했는지가 더 가치있는 정보일 것이다. 읽는 이 입장에서 메모리를 할당하며 모든 원소를 0으로 초기화하는 이유는 당연하지 않은 사실이기 때문이다.

다음은 객체지향프로그래밍 과목에서 짰던, 과목을 나타내는 클래스 Subject의 메소드 선언부 중일부이다.

QString name(void) const;

//과목의 이름을 반환하는 함수입니다.

QString location(void) const;

//과목의 수업 장소를 반환하는 함수입니다.

subject.name()과 subject.location()의 의미를 이해하는데 주석이 과연 필요할까? 영어를 모르는 사람한테는 주석이 유용할 수도 있겠다.

**클린 코드 챕터 7: 오류 처리**

앞의 ‘주석’ 챕터가 내가 이미 가지고있던 직관에 대치돼서 받아들이기 힘들었다면, 오류 처리 챕터는 오류 처리를 해본 적이 없어서 이해하기 힘들었던 챕터였다. 솔직히 말해서, 나는 아직도 왜 예외를 사용하고 Try-Catch문을 사용하는지 완전히 이해하지 못했으며, 저자가 제안하는 몇 가지 권고사항들에 대해서도 왜 그래야 하는지 완벽히 이해하진 못했다.

어찌되었든 저자는 오류 처리에 전통적인 (그리고 낡은) C 식의 오류 코드 반환이나 플래그 설정 등의 방식을 쓰지 말라고 주문한다. 오류 처리 코드는 함수의 반환값을 바로 확인해야 해 비즈니스 논리와 오류 처리 과정이 뒤섞인다는 것이 그 이유이다. 확실히 Try-Catch문은 If를 통해 바로 반환값을 확인하는 것에 비해 가독성이 좋긴 하지만, 예외를 사용하는 이유로는 살짝 부족한 감이 있다.

저자는 이에 만족하지 않고, 예외를 던지기보다는 특수 사례 패턴을 사용하여 특수한 사례를 일반적인 사례의 일종으로 캡슐화하여 그 캡슐화된 객체를 반환할 것을 권장한다. 이의 일환으로, 저자는 null을 메서드나 함수의 반환값으로 사용하지 말 것을 강하게 주장한다. 무언가가 없거나 찾지 못했음을 의미하기 위해 null을 반환하는 것은 호출자에게 반환값을 검사할 것을 강제하여 오류 처리 논리와 비즈니스 논리를 뒤섞어버리고 검사를 깜빡했을 경우 치명적인 결과를 낳는다. 때문에, null을 반환할 상황에는 그 상황을 나타내는 더 특수한 객체를 반환할 것을 권한다. 대수적 데이터 타입을 지원하는 언어에서는 null 대신 Option 타입을 반환하여 호출자가 반드시 반환된 객체를 풀어헤쳐 오류 상황을 처리하도록 만드는 것이 차선책이 될 수 있겠다.

예외를 처리하기 유용한 한 가지 패턴으로 저자는 예외를 그대로 외부에 노출시키지 말고, 예외가 발생하는 외부 라이브러리를 사용할 때는 그 라이브러리를 감싸는 래퍼(wrapper) 클래스를 선언하여 호출자에게 유용한 예외만 노출시키도록 권한다.