Clean Code Summary

POSTECH CSE
20190218 김민결

Chapter 2: Meaning Names

본 챕터에서는 변수나 함수, 클래스, 파일, 디렉토리 등에 '좋은 이름'을 붙이기 위한, 즉 좋은 네이밍 전략을 위한 몇 가지 규칙을 설명하고 있다.

첫째, 이름에서 그 대상이 왜, 무엇에, 그리고 어떻게 사용될지에 대한 의도가 드러나야 한다. 만약 그러한 의도를 명확히 하기 위해 추가적인 주석이 필요하다면, 좋은 이름을 붙여준 것이 아니다. 만약 의도가 불분명한, 무의미한 이름을 짓게 된다면, 대여섯줄로 이루어진 단순한 코드의 가독성도 떨어지게 된다. 즉, 좋은 이름을 짓는 것은 코드의 가독성에 매우 중요한 요소이다.

또한 잘못 해석될 수 있는 여지가 있는 이름을 사용하면 안된다. 가령 다른 의미로 해석될 수 있는 약어(Abbreviation)를 사용하거나, List type도 아니면서 ~List라고 이름을 붙이는 것은 혼동의 여지를 줄 수 있기에, 절대로 좋은 네이밍이 아니다. 정말 미묘하게 다른 두 이름을 서로 다른 두 대상에 적용하는 것 또한 좋은 습관이 아니다. 이름만으로 서로 다른 두 대상의 차이가 한 눈에 보여야 한다.

종종 네이밍을 하다 보면, 다른 대상과 이름이 겹쳐서 한 쪽을 수정해야 하는 경우가 있는데, 그냥 컴파일 에러만 해결하겠다고 임의로 아무 의미 없는, 또는 숫자만 추가하거나 스펠링만 살짝 바꾼 이름으로 수정하는 것은 절대 좋은 습관이 아니다. 그러한 방법 대신, 앞 뒤로 추가적인 단어를 붙여 두 대상의 이름을 구별하는 것도 괜찮은 방법이라고 생각할 수 있겠지만, moneyAmount와 money간의 의미적 차이가 불분명한 것처럼, 오히려 의미 없고 불필요한 단어 추가에 그칠 수 있다.

또한 '발음할 수 있는' 이름을 붙이자. 막 abdr2231처럼 발음이 불가능한 단어로 이름을 붙이면, 그 이름의 대상에 대한 팀원과의 토론이 상당히 힘들어질 것이다. 즉, 적절한 영단어를 냅두고, 제대로 읽지도 못할 약어로 이름 붙이지 말자.

종종 코딩을 하다 보면 ctrl+F (or ctrl+shift+F)로 이름을 입력하여 특정 변수 또는 함수 등을 찾을 때가 있다. 그런데, 찾으려는 변수의 이름이 i, j처럼 알파벳 하나로 이루어져 있다면 절대로 방대한 크기의 파일 내에서 찾을 수 없을 것이다. 그러한 변수명은 traditional하게 쓰이고 있는 for문의 loop counter나 한 눈에 볼 수 있는 짧은 코드 내의 지역변수로나 사용되어야 하고, 하나 이상의 영단어로 이루어져 찾기 쉬운 이름을 사용하는 것이 바람직하다. 또한, 어떠한 정수형 또는 실수형 상수도 ctrl+F로는 찾기 어려운 요소이다. 그러한 상수도 찾아내기 좋은 변수명으로 선언하는 것이 더 바람직하다.

또한 type이나 scope와 같은 정보를 암호화(Encode)한 notation을 이름에 포함시키는 것은 오히

려 이름이나 type의 수정을 힘들게 하고 코드의 가독성을 떨어뜨리는 작업이다. 옛날에 컴파일러 자체에서 type check이 안되던 시절에는, 프로그래머들이 type을 기억하기 위해 그러한 annotation을 사용했을지언정, 컴파일러가 type check을 자동으로 하고 코드 에디터 자체에서 type error를 확인해주는 요즈음에는 그러한 것이 매우 불필요하다.

비슷하게, 옛날 코드들을 보면 멤버 변수 앞에 prefix로 m_을 붙이는 경우도 있는데, 이 또한 클래스나 함수의 길이 자체가 짧아지고 에디터의 하이라이트 기능으로 쉽게 멤버 변수 등을 구별 가능한 요즈음에는 불필요한 작업이다. 특히 코드를 읽으면 이름에서 prefix나 suffix를 제외하고 보는 경우가 많다.

반면, interface와 해당 interface를 구현(implement)한 class를 작성하는 경우, 두 개의 구별을 위해 한 쪽 이름에 type을 encode해야 하는 경우가 있다. 이 책에서는 interface의 이름은 '그대로' 두고, 해당 interface를 구현한 class의 이름을 앞에 C를 붙이는 식으로 바꾸는 것을 추천하고 있다.

참고로 l(소문자)과 O(대문자)는 각각 에디터에서 1, 0과 혼동의 여지가 있으니 단일 알파벳 이름으로 사용하지 않도록 하자. 또한, 개개인의 사전 지식에 따라서 따로 해석될 여지를 남기지 않을 정도로 이름을 매우 명확히 지어야 한다. i.e. clarity is king.

Class와 object의 이름은 명사(Noun or Noun phrase)으로 이루어져야 하며, Manager, Processor, Data, Info와 같은 단어는 사용하지 않는 편이 좋다.

Method의 이름은 동사나 동사+명사(Verb or Verb phrase) 형식으로 지어야 하며, Accessor, mutator, predicate을 나타내는 method의 이름 앞에는 각각 get, set, is를 붙이는 것이 일반적이다.
생성자 오버로딩을 사용하는 경우에는, object를 생성할 때 그 생성자 자체를 바로 사용하기보다는, 그 생성자가 받는 argument의 특성을 이름에 포함한 static factory method를 따로 만들어, 해당 static factory method를 통해 새로 object를 생성하고 반환 받도록 하자. 이 내용은 Clean-Code 책 p.25에서 예제와 함께 확인할 수 있다.

또 일부만 아는 속어, 구어적 표현 (소위 말하는 드립이나 밈)을 이름에 포함하지 말아라. 네이밍에 중요한 것은 clarity이지 재미가 아니다.

그리고 네이밍에 있어서 일관성 있는 어휘를 사용하자. 가령 똑같이 Accessor로 작동하는 method인데, 어떤 class의 method는 get을 쓰고, 또 어떤 method는 fetch를 쓴다면 당연히 헷갈릴 수밖에 없다. 즉, 하나의 개념에 대해서는 하나의 용어로 통일하는 것이 좋다.
물론, 조금이라도 의미가 다른 두개의 개념을 하나의 용어로 쓰는 것은 피해야 한다. 줄곧 두 개의 수를 더하여 반환하는 함수 이름에 add를 사용하다가, 갑자기 새로운 element를 추가하는 함수에도 add를 사용한다면, 의미적으로는 말이 될지언정, 일관성을 잃게 된다.

경우에 따라, Solution Domain에서 네이밍을 할지 Problem Domain에서 네이밍을 할지를 결정해야 한다. 만약 프로그래머를 고려해야 한다면 Solution Domain에서, 즉 CS에서 많이 사용할 법한 전문 용어로 네이밍하는 편이 좋고, 그렇지 않다면 Problem Domain에서 네이밍해야 한다.

또한, 동일한 context에 있는 변수들은 같은 클래스나 함수, 네임스페이스에 묶어 두는 편이 좋다. 예를 들어, 집 주소와 관련한 여러 종류의 변수들(state, city, ...)을 Address라는 class의 멤버변수로 설정한다면, 해당 변수들이 'Address'라는 동일 context에 있음을 한 눈에 확인 가능하다. 정 그렇게 묶어 두는 것이 불가능하다면, 최후의 수단이긴 하지만 그 context에 관련한 동일 prefix를 붙여 각 변수들이 동일 context에 있음을 나타낼 수도 있다.

그러나 불필요한 context까지 이름에 포함할 필요는 없다. context가 명확히 전달되고 서로 구별만 되면 된다. 짧고 간단한 이름이 긴 이름보다 좋다는 것을 기억해 두도록 하자.

Chapter 3: Functions

해당 챕터에서는 함수를 작성하는 방법에 대하여 다루고 있다.

우선 함수는 짧으면 짧을수록 좋다고 한다. 한 눈에 어떤 일을 하는 함수인지 보일 정도로 짧고 명료해야 한다. 또한, if, else, while block은 한 줄 수준으로 끝나야 한다. 즉, 해당 block은 한 번의 function call 정도만을 포함하고 있어야 한다. 이는 indent의 수가 1~2번을 넘어가지 않는다는 면에서도 가독성을 높여주는 방식이다.

함수의 역할은 하나의 큰 일을 여러 단계로 나누는 것이다. 하나의 함수는 반드시 '하나의 일'을 하도록 작성해야 하며, 함수의 이름이 표현하는 선 내에서 작업이 이루어지면 하나의 일을 했다고 본다. 즉 어떠한 함수에 대하여, 함수 이름의 중복(restate)를 허용하지 않는 선에서 두 개 이상의 함수로 쪼갤 수 없는 상황이라면, 해당 함수는 '하나의 일'을 하고 있는 것이다.

또한, 하나의 함수를 구성하는 모든 구문의 level of abstraction이 동일해야 한다. 여기서 level of abstraction이란 system에 대한 detail이 얼마나 드러나는지를 나타내며, 낮은 level of abstraction에서 system detail이 더 많이 드러난다. level of abstraction이 한 함수 내에서 일관성 있어야, 각 함수의 역할이 더 명확해지는 듯하다.

술술 읽히는 코드를 구성하기 위해서는, level of abstraction이 낮아지는 순서대로 함수들을 읽어나갈 수 있어야 한다. 쉽게 읽히도록 함수들을 작성하기 위해서는, 하나의 함수가 일관된 level of abstraction 내에서 쓰여야 하며, 각 함수 내에서 바로 다음에 등장할 level of abstraction의 함수를 등장시켜야 한다. 책 p.37에 소개된 TO paragraph를 읽어보면 어떤 느낌인지 알 수 있다.

기본적으로 switch 구문은 함수처럼 '하나의 일' 만을 하도록 만들 수가 없으며, 여러 개의 case가 있는 만큼 코드 길이가 길어지는 것은 당연하다. 이러한 이유로 switch문은, 반복 작성을 막기 위하여 polymorphism을 통해 low-level 클래스에 작성하는 것이 좋다. switch문의 각 case에서 polymorphic object를 return하는 방식이며, 이 방식에 대한 예제는 책 p.39의 Listing 3-5에서 확인할 수 있다. 즉, switch문은 한 번만 작성하는 것이 최선이라는 것이다.

함수를 짧게 쓰는 것만큼 함수에 좋은 이름을 붙여주는 것도 중요하다. 이름은 해당 함수의 역할

을 제대로 설명할 수 있는 수준이어야 하며, 읽기만 쉽다면 길고 서술적인 이름을 붙이는 것이 짧게 압축된 이름보다 훨씬 낫다. 또한 Chapter2에서 설명한 것처럼, 일관성 있는 어휘를 사용하는 것이, 코드를 하나의 '이야기'로 표현할 수 있도록 해준다. 마지막에도 설명하겠지만, 하나의 순차적인 이야기처럼 읽을 수 있는 수준이어야 읽기 좋은 코드라고 할 수 있다.

함수의 Argument(인자)는 사실상 없는 것이 가장 좋다. Argument는 오히려 해석의 양을 더 늘릴 뿐만 아니라, 불필요한 포인트에서 제공되는 detail로 간주될 수도 있기 때문이다. 특히 argument의 수가 늘어날수록 고려해야 하는 test case의 수가 늘어난다는 점에서도 argument의 수가 적은 편이 좋다.

또한, argument는 일반적으로 함수의 input으로 인식된다는 점에서, 함수로부터 제공된 정보가 따로 반환되지 않고 argument에 담기는 방식인 output argument는 가독성을 방해하는 요소이다. 즉 argument를 아예 사용하지 않거나, 1개의 input argument를 사용하는 방식이 좋다.

보통 하나의 argument를 사용하는 경우는 크게 두 가지로, argument에 대한 상태를 boolean 반환 값으로 확인하거나, argument를 처리하여 원하는 정보를 반환 받기 위한 용도로 사용된다. 자주 사용되지는 않지만 하나의 argument를 사용하여 system state를 변경하는 event도 그의 한 종류이다. 이러한 세 가지 용도가 명확히 드러나도록 함수의 이름을 짓는 것이 중요하다.

종종 boolean argument의 값을 다르게 넘겨 함수의 동작을 다르게 하기 위해 Flag argument를 사용하는 경우가 있는데, 이러면 하나의 함수가 두 가지 이상의 일을 하게 되는 것이기에, 절대로 좋은 방식이 아니다. 차라리 하나의 Flag argument에 따른 두 개 이상의 동작을 개별 하나의 함수로 쪼개는 것이 더 좋다.

Arguments 두 개를 사용하는 것은 해석의 양을 늘릴 뿐만 아니라 두 arguments의 순서에 충분한 혼동이 올 수 있다는 점에서, 하나를 사용하는 것보다 훨씬 가독성을 낮춘다. 다만, Point(x, y)처럼 두 arguments가 통상적인 ordering을 가진 경우에는 크게 문제되지 않는다.
두 개의 Arguments를 하나로 줄이는 방법에는 여러 가지가 존재한다. 둘 중 하나가 class object라면, 다른 하나를 argument로 받는 동일한 기능의 함수를 해당 class object의 멤버로 만들 수 있을 것이다. 또한 argument를 2개 받는 멤버 함수의 경우, 하나의 argument를 class의 멤버로 만들어버리는 방법도 있다.

Arguments 세 개를 사용하는 경우, arguments 간 ordering 경우의 수가 6가지일 정도로, 앞서 말한 문제들이 훨씬 더 복잡해진다. 다만, 책 p.42에서 언급된 Floating point equality 함수처럼 3개의 arguments를 쓰는 것이 합당한 경우도 있음을 알아두자.

앞서 언급한 것 외에도 arguments 수를 줄일 수 있는 또 다른 방법들이 있다. 하나의 그룹으로 생각할 수 있는 두 개 이상의 arguments를 class 하나의 멤버변수로 포함하는 방법이 있으며, 동일하게 간주될 수 있는 두 개 이상의 arguments를, String.format처럼 하나의 arguments list(...)로 간주하는 방법도 있다.

Chapter2에서 언급한 것처럼 함수의 이름에는 동사를 사용하는 것이 일반적이며, 함수와 argument의 이름이 서로 동사+명사(목적어)의 짝을 이룬다면 더 가독성을 높아진다. 또한, argument의 속성을 함수의 이름에 간단히 언급하는 것도 좋으며, arguments 두 개의 ordering을 함수 이름에 언급하는 것도 앞서 말한 ordering 혼동 문제를 줄일 수 있는 좋은 방법이다.

당연히 함수를 실행했을 때, 함수의 이름에서 언급된 것 이외의 예상치 못한 동작을 한다면 큰 문제가 생길 수 있다. 물론 앞서 말한 것처럼 하나의 함수는 '하나의 일'을 하는 것이 바람직하지만, 정말 불가피한 이유로 두 개의 동작을 해야 한다면, 두 동작 모두 함수의 이름의 언급해주어야 한다.

앞서 언급한 것처럼, 함수에 output argument를 쓰는 것은 코드의 가독성을 떨어뜨린다. output argument가 class object라면, 차라리 output argument에 해당하는 class에 동일한 기능을 하는 멤버 함수를 만들어 쓰는 편이 가독성에 훨씬 낫다. 책 p.45에 있는 예시를 보자.

멤버 함수는 크게 object의 상태를 바꾸거나, object 상태를 반환하는 일을 하는데, 이 두 가지 일을 개별 함수로 분류하는 것이 중요하다. 가령 멤버 변수를 변경하고 변경이 불가하다면 false를 return하는 멤버 함수를 생각해보면, 해당 멤버 함수를 if 조건문에서 호출하는 것만으로 멤버 변수를 변경 가능한 상태인지 확인하는 동시에 실질적인 변경이 가능하니, 코드 줄 수의 측면에서 이득이라고 생각할 수 있지만, 오히려 코드 해석의 모호함을 불러일으키는 요소이다. 즉 코드 해석을 명료히 하기 위해서는 멤버 변수가 변경 가능한 상태인지 확인하는 함수, 그리고 멤버 변수를 변경하는 함수를 따로 만들어 사용해야 한다.

비슷한 맥락에서 어떠한 일을 수행하고 해당 일을 제대로 끝마치지 못하면 Error Code를 반환하는 함수를 종종 찾아볼 수 있다. 이러한 함수 역시 if 조건문에 바로 쓰일 수 있지만, 그러한 함수가 연속하여 두 번 이상 실행되어야 하는 이유로 여러 번에 걸쳐 Error Code를 확인해야 한다면, 코드의 가독성을 현저히 낮추는 nested if문을 사용해야 한다. 때문에 try-catch를 이용하여, catch 문에서 한 번에 Error Code를 처리하는 편이 가독성의 측면에서 낫다.

또한, try-catch문에서 수행하는 Error Handling은 또 다른 종류의 '하나의 일'로 간주되는 만큼, Error Handling을 하는 함수 내에서는 try-catch문 이외의 추가적인 동작을 하지 않는 것이 좋으며, try block과 catch block 안에 있는 구문들은 각각 개별 함수로 빼내는 편이 훨씬 좋다.

보통 Error Code는 별개의 enum class로 정의되어 있는데, 만약 새로운 유형의 Error 처리가 필요해서 기존의 enum class에 새 Error Code를 추가해버리면, 해당 enum class를 import해서 사용하고 있는 수많은 파일들의 빌드를 다시 해야 한다. 이러한 이유로, Error Code를 사용하기보다는, 부모 exception class로부터 상속받아 새로운 exception class를 만들어 사용하는 편이 좋다고 한다.

중복되는 코드를 없애는 것은 코드의 가독성뿐만 아니라 유지보수의 측면에서도 중요한 요소이다. 만약 중복되는 코드를 남겨둔다면, 나중에 코드를 수정해야 하는 상황에서 중복된 코드를 모두 일일이 찾아 수정해줘야 하는 불편함이 생긴다. 책 p.50~51에 소개된 Listing 3-7 코드 예제에서

확인할 수 있 듯, include를 이용하면 중복된 코드를 줄일 수 있다.

Edsger Dijkstra의 rules of structured programming에 따르면, 모든 함수와 함수 내의 모든 block은 하나의 entry와 하나의 exit만을 가져야 한다. 즉 return은 단 한 번만 쓰여야 하며, goto문과 loop 내의 continue, break문은 사용하면 안 된다고 한다. 이 규칙은 긴 코드에 한해서는 도움이 되나, 오히려 짧은 코드에서 크게 도움되는 부분은 아니라고 한다. 앞서 강조한대로 한 함수의 길이를 짧게 유지한다면, 오히려 조건 분기적인 여러 번의 return이나 break, continue를 쓰는 편이 더 좋다고 한다. 다만, 길이가 긴 함수에서 주로 사용되는 goto문의 사용은 짧은 함수에서 피해야 한다.

위에서 설명한 길고 긴 규칙들을 맨 처음부터 지키면서 작성하려고 하면 안 된다. 우선 규칙과 무관하게 함수를 쭉 작성하고, 함수 전체에 대한 unit tests를 작성한 후에, unit tests가 통과하는 선 상에서 규칙에 맞게 코드를 정리해 나가는 방법이 바람직하다고 한다.

마지막으로 다시 한 번 강조하지만, 단순히 프로그래밍을 하는 것을 넘어서, 딱 보고 하나의 이야기를 서술할 수 있는 수준의 깔끔하고 명료한, 가독성 좋은 함수를 작성해야 한다. 앞서 정리한 규칙들을 잘 참고하여, 짧고 체계적이며 좋은 이름을 가진 함수를 작성할 수 있도록 노력해보자.