Clean Code

a Summary of **Clean Code**

2장 : 의미 있는 이름

- 의도를 분명히 밝혀라
  - 변수, 함수, 클래스 등에 의도가 드러나도록 이름을 지어라
  - 의도가 드러나면 코드의 이해와 변경이 쉬워진다.
    - ex) 코드는 동일하지만 변수명만 바꿔 명확해진 코드 예시
- 그릇된 정보를 피하라
  - 다르게 해석될 수 있는 이름을 변수명으로 하지마라
    - ex) userlist(리스트 구조 아님)
  - 비슷한 이름 여러개를 쓰지마라
  - 유사한 개념은 유사한 표기법을 사용하지만(정보), 일관성이 떨어지면 그릇된 정보이다.
  - ex) 0, O, 1, l, I `(X)`
- 의미 있게 구분하라
  - 읽는 사람이 차이를 알도록,
  - 이름이 정보를 제공하도록 이름을 지어야 한다. (의미 없게 짓지 말아라)
    - ex) a1, a2, a3 / moneyamount, money `(X)`
    - ex) source, destination `(O)`
- 발음하기 쉬운 이름을 사용하라
  - 프로그래밍은 사회 활동이기에 발음하기 쉬운 이름도 중요하다.
- 검색하기 쉬운 이름을 사용하라
  - 문자 하나를 사용하든지 상수를 그냥 사용하면 검색하기 어렵다.
  - 여러 곳에서 사용한다면 검색하기 쉬운 이름이 적합하다.
    - 이름 길이는 범위 크기에 비례해야 한다.
    - ex) 5 : 상수, 검색 어려움 vs WORK_DATS_PER_WEEK : 검색 쉬움

- 인코딩을 피하라
  - 인코딩 = 구닥다리, 읽기 어려움
  - 써야 하더라도 알기 쉽도록 써야한다.
- 자신의 기억력을 자랑하지 마라
  - i, j, k같이 전통적인 것을 제외하고
  - 자신만이 이해하는, 기억하는 변수로 이름 짓지말아라, 타인이 이해하는 이름 지어라.
- 클래스 이름
  - 명사, 명사구가 적합하다.
  - ex) customer, account `(O)`
  - ex) data, info `(X)`
- 메서드 이름
  - 동사, 동사구가 적합하다.
  - ex) payment, save `(O)`
  - 접두사 get, set, is
- 기발한 이름은 피하라
  - 재치있는 이름보다 명료한 이름이 좋다.
  - 특정 문화를 모르면 이해하기 어렵다.
- 한 개념에 한 단어를 사용하라
  - 추상적인 개념에 한 단어만 사용해라
    - ex) controller = manager = driver 다 같음
- 말장난을 하지 마라
  - 같은 맥락에만 같은 단어를 사용해라
  - ex) add ⇒ 더해서, 이어서 / 반환하기 `(O)`
  - ex) add ⇒ 집합에 추가하기 `(X)`, insert가 옳음
- 해법 영역에서 가져온 이름을 사용하라
  - 코드 어차피 프로그래머가 읽는다.
  - 전산, 알고리즘, 패턴, 수학 용어 등 사용해도 된다. (프로그래머 용어)

- 문제 영역에서 가져온 이름을 사용하라
  - 문제 영역과 관련 깊다면 문제 영역에서 이름을 지어라
  - 나중에 코드 보수하는 프로그래머가 분야 전문가에게 의미 물어서 파악할 것이다.
- 의미 있는 맥락을 추가하라
  - 변수 스스로 의미가 분명하지 않다면
  - 맥락을 추가하여 의미를 구분하여 이해하기 쉽도록 한다.
- 불필요한 맥락을 없애라
  - 의미가 분명한 경우 짧은 이름이 더 좋다.
  - 불필요하게 맥락을 추가하지 마라
- 마침
  - 우리는 모든 이름을 암기하지 못하기에 암기는 도구에 맡기고
  - 우리는 코드가 읽히기 쉽도록 짜야한다.

3장 : 함수

- 작게 만들어라!
  - 함수는 작게 만들어야 좋다.
  - 블록과 들여쓰기
    - if/else, while문 등에 들어가는 블록은 한 줄이어야 한다.
    - enclosing function도 작아지고 블록 안에서 호출하는 함수 이름도 적절히 지으면, 코드 이해도 쉬워진다.
    - 중첩 구조가 1, 2단을 넘어서지 않도록 해야 한다는 것이다.(들여쓰기)
- 한 가지만 해라!
  - 함수는 한 가지를 해야 한다. 그 한 가지를 잘 해야 한다. 그 한 가지만을 해야 한다.
  - 한 가지 : 추상화 수준이 하나인 단계
- 함수 당 추상화 수준은 하나로!
  - 함수에서 추상화 수준이 동일해야 한다. (섞으면 안된다.)
  - 위에서 아래로 코드 읽기 : 내려가기 규칙
    - 코드는 위에서 아래로 이야기처럼 읽혀야 한다.

        - 위에서 아래로 추상화 수준이 낮아져야 한다.
- Switch 문
    - 추상 팩토리를 통해 switch문을 숨긴다.
- 서술적인 이름을 사용하라!
    - 함수를 읽으면서 짐작했던 기능을 그대로 수행한다면 좋은 코드이다.
    - 이름이 길어도 좋다. 일관성이 있어야 한다.
- 함수 인수
    - 함수 인수는 적을수록 좋다. (없는 것이 가장 좋다)
        - 적을수록 읽는 사람이 이해하기 쉽다.
    - 단항 인수
        - 질문을 던지는 경우
        - 변환해 결과를 반환하는 경우
        - 이벤트 함수인 경우
        - 위 3가지를 제외하고 가급적 피한다.
    - 플래그 인수
        - 함수가 한꺼번에 여러 가지를 처리한다는 것
        - 추함
    - 이항 함수
        - 이해하기 어려움
        - 실수 위험
    - 삼항 함수
        - 더 이해하기 어려움
    - 인수 객체
        - 인수가 2-3개면 독자적인 클래스로 변환하여 선언하라.
    - 인수 목록
        - 가변인수 : list형 인수 하나로 취급
    - 동사와 키워드

        - 함수의 의도나 인수의 순서와 의도를 제대로 표현하기 위해 좋은 이름이 필요하다.
        - 동사 + 키워드 형태로 적는다.
- 부수 효과를 일으키지 마라!
    - 한 가지만 하도록 해야 한다.
    - 다른 작업이 꼭 필요한 경우 함수 이름에 분명히 명시해야 한다.
    - 출력 인수
        - 쓰면 안된다.
        - 객체 지향 언어에서 this를 쓴다.
        - 함수에서 상태를 변경해야 한다면 함수가 속한 객체 상태를 변경하는 방식을 택한다.
- 명령과 조회를 분리하라!
    - 함수는 뭔가를 수행하거나 뭔가에 답하거나 둘 중 하나만 해야한다.
        - 객체 상태 변경 or 객체 정보 반환
    - 명령과 조회를 분리하여 혼란을 뿌리뽑는다.
- 오류 코드보다 예외를 사용하라!
    - try/catch 블록 뽑아내기
        - try/catch 블록은 별도 함수로 뽑아내라
        - 정상동작과 오류처리 동작을 분리하면 코드를 이해하고 수정하기 쉬워진다.
    - 오류 처리도 한 가지 작업이다.
        - 오류 처리도 한 가지 작업이므로 오류를 처리하는 함수도 오류만 처리해야 한다.
        - try/catch 깔끔하게 써라
    - 오류코드 의존성 자석
        - 오류코드가 변한다면 다 재컴파일 재배치를 해야하므로 번거롭다.
        - 오류코드 대신 예외를 사용해라
- 반복하지 마라!
    - 중복되면 나중에 손 봐야 할 곳이 중복된다는 것이다.

    - 중복이 없으면 가독성도 높아진다.
- 구조적 프로그래밍
    - 함수를 작게 만들면 단일 입출구 규친 안지켜도 된다.
    - goto는 쓰지마라
- 함수를 어떻게 짜죠?
    - 글짓기 처럼 짜라
    - 초안은 규칙도 어기고 지저분하지만,
    - 다듬어 가면서 완성한다.
- 결론
    - 함수는 동사고 클래스는 명사다.
    - 시스템은 풀어나아갈 이야기이다.
    - 함수가 분명해야 이야기를 풀어나가기 쉬워진다.

4장: 주석

: 잘 달린 주석은 유용하지만, 주석은 잘못된 정보를 제공할 수 있으므로 주의해야 한다.

: 코드만이 진실을 담는다.

: 주석은 코드를 유지 보수하면서 방치되는 경우가 많으므로 주석을 줄여야한다.

- 주석은 나쁜 코드를 보완하지 못한다.
    - 나쁜(이해하기 어려운)코드를 짜놓고 주석으로 포장하지 말아라.
    - 코드를 이해하기 쉽게 짜려고 노력해라.
- 코드로 의도를 표현하라!
    - 좋은 함수, 변수 이름으로 코드를 깔끔하게 표현해라.
    - 이게 주석보다 더 깔끔하다.
- 좋은 주석
    - 법적인 주석
        - 저작권, 소유권을 표시
    - 정보를 제공하는 주석

        - 기본적인 정보를 제공하는 것은 유용하다.
        - 그러나 함수 이름으로 표현하는 것이 더 낫다.
    - 의도를 설명하는 주석
        - 이해도 도와주고 작성자의 의도까지 설명한다.
    - 의미를 명료하게 밝히는 주석
        - 인수, 반환 값을 명료하게 코드를 짜는 것이 더 낫다.
        - 어쩔 수 없는 경우(표준 라이브러리, 변경 못하는 코드 등)에 주석을 달아라 .
        - 잘못된 주석을 달지 않도록 주의해라.
    - 결과를 경고하는 주석
        - 함수 등을 사용할 때 주의를 요하는 경우(오래걸림, 위험 등)에 사용하면 좋다.
        - 요즘은 테스트 케이스 끌 때 @Ignore로 처리한다.
    - TODO 주석
        - 앞으로 할 일을 남겨둘 때 유용하다.
        - 떠넘기는 용으로 사용하지 말아라.
    - 중요성을 강조하는 주석
        - 중요한 것 강조할 때
    - 공개 API에서 Javadocs
        - 이것도 잘못된 정보 제공 주의해라.
- 나쁜 주석 : 대다수
    - 주절거리는 주석
        - 의무감으로 아무거나 달지 말아라.
        - 타인이 알아들을 수 있도록 달아라.
    - 같은 이야기를 중복하는 주석
        - 코드 내용을 그대로 중복하면 읽는 시간만 오래걸린다.
        - 코드보다 부정확하다. 의미있는 정보를 제공하지 않는다.
    - 오해할 여지가 있는 주석
        - 주석에 잘못된 정보가 담기면 안된다.
    - 의무적으로 다는 주석

        - 잘못된 정보를 제공할 여지만 만든다.
    - 이력을 기록하는 주석
        - 로그는 git에 다 나온다. 필요없다.
    - 있으나 마나 한 주석
        - 주석은 새로운 정보를 제공해야한다.
        - 쓸모 없는 주석 달 시간에 코드개선에 노력해라
    - 무서운 잡음
        - Javadocks도 필요할 때만 써라.
        - 의무감으로 달지 말아라
    - 함수나 변수로 표현할 수 있다면 주석을 달지 마라
        - 주석이 필요 없도록 코드 짜는 게 좋다.
    - 위치를 표시하는 주석
        - //////////////////////////// ←이거 아주 가끔씩만 써야한다.
    - 닫는 괄호에 다는 주석
        - 중첩이 심한 경우에만 쓸모 있다.
        - 그러나 중첩 심한 코드를 짜면 안되기에 무의미하다.
        - 중첩을 줄이려 노력해라
    - 공로를 돌리거나 저자를 표시하는 주석
        - 쓸모없다.
    - 주석으로 처리한 코드
        - 남이 지울 때 머뭇거리게 한다.
        - 쓰지말고 지우고 git에 올려라
    - HTML주석
        - 쓰레기
    - 전역 정보
        - 주석을 달 때는 근처에 있는 코드만 기술해야한다.
    - 너무 많은 정보
        - 쓸모 없는 장황한 정보 적지 말아라.

    - 모호한 관계
        - 독자가 뭔 소리인지 알 수 있도록 달아라.
    - 함수 헤더
        - 짧고 한 가지만 수행하며 이름을 잘 붙인 함수가 주석보다 훨씬 좋다.
    - 비공개 코드에서 Javadocks
        - 공개 코드가 아니면 쓸모 없다.

7장: 오류 처리

: 오류 처리는 프로그램에 반드시 필요한 요소이다.

: 오류 처리는 매우 중요하며 오류 처리를 할 때도 깨끗하게 코드를 짜야한다.

- 오류 코드보다 예외를 사용하라
    - 오류 코드 + if else로 처리하지 말아라.
    - try catch 형태 + 함수 명 알기 쉽도록 코드를 짜서 처리한다.
    - 개념을 분리해서 이해하기 쉽도록 짜야한다.
        - 디바이스 종료 / 오류 처리 분리
- Try-Catch-Finally 문부터 작성하라
    - 범위를 정의하는 행위
    - 강제로 예외를 일으키는 테스트 케이스부터 작성한 후 테스트를 통과하게 코드를 작성해라.
- unchecked 예외를 사용하라
    - checked예외가 없는 언어도 많다.
    - 함수 최하단에 위치한 checked예외를 수정해야하면 많은 시간이 들 수 있다.
- 예외에 의미를 제공하라
    - 오류 메시지에 정보를 담아서 예외와 함께 던져라.
- 호출자를 고려해 예외 클래스를 정의하라
    - 오류를 잡아내는 방법에 관심을 가져라.
    - 예외에 대응하는 방식이 동일하면 클래스로 감싸서 한꺼번에 반환해라.

    - 외부 API를 사용하면 감싸기 방식이 좋다.
- 정상 흐름을 정의하라
    - 외부 API를 감싸 독자적인 예외를 던지고, 코드 위에 처리기를 정의해 중단된 계산을 처리한다.
    - 특수 사례 패턴을 처리할 때 클래스를 만들거나 객체를 조작해 처리한다.
- null을 반환하지 마라
    - null을 반환하지 않도록 코드를 짜라
    - null반환 확인하는 것은 일거리를 늘릴 뿐이다.
    - null반환하는 것이 필수적일 때
        - 감싸기, 예외 던지기, 특수 사례 객체 반환
        - null반환 안 하도록 수정하는 것이 좋다.
- null을 전달하지 마라
    - 메서드로 null을 전달하는 것은 반환보다 안 좋다.
    - 예외 유형을 만들어 던지거나
    - assert문을 사용해라.
- 결론
    - 깨끗한 코드는 읽기 좋아야 하며 안정성도 높아야 한다.
    - 오류 처리를 프로그램 논리와 분리해 독자적인 사안으로 고려하면 튼튼하고 깨끗한 코드를 작성할 수 있다.