# 좋은 코드란 무엇인가?

* 좋지 않은 코드가 없는 코드
* 코드 간의 의존성을 고민 하자
* 합의된 규칙으로 일관성있게 작성하자
* 적절하게 확장 가능하도록 설계하자
* 응급 조치로 작성된 코드는 주석과 함께 격리하자

**좋은 코드?**

사람마다 경험에 의해 좋은 코드는 다양하게 정의 된다.

읽기 쉬운 코드, 중복 없는 코드, 테스트가 쉬운 코드...

**읽기 쉬운 코드 == 좋은 코드 ?**

* 주석이 많은 코드

주석만 읽어도 함수가 어떻게 사용 되며 변수가 어떤 의미를 가지는 지 알 수 있다.

몇몇 잘못된 주석으로 오히려 코드를 이해하기 힘들수 있다.

관리가 매우 힘들다. 또한 주석의 내용이 실제 함수의 작동을 보장하진 못한다.

함수가 수정 될 때 주석이 수정되지 않을 수 있다.

* 내가 읽기 쉬운 코드

사람마다 다른 배경 지식을 가짐. 그러다 보디 자신이 읽기 쉽게 작성 되었다는 코드가 다른 사람에게는 읽기 어려울 수 있다.

* 그럼 왜 읽기 쉬운 코드를 작성 해야 할 까
  + 동료와의 협업을 위해.
  + 수정 사안이 생겼을 때 쉽게 수정 할 수 있게.
  + 코드 재활용으로 생산성 증대를 위해
  + 테스트가 용이하게

**좋지 않은 코드는 왜 작성되는 가?**

* **쓰이지 않는 코드**
  + 거리

코드가 정의 된 곳과 코드가 동작 하는 곳의 물리적 거리가 멀면 파악하기 힘듬

* + 순수하지 않은 함수

함수 외부의 값을 기반으로 동작하는 함수의 경우 역효과를 파악하기 쉽지 않음

def getWage(name):

return getMySalary(name)/getTotalTime(tiㅡ

* **응급처치를 한 코드**

중복으로 빼두었지만 사용하는 곳에서 다른 로직을 추가 해줘야 할 때를 종종 마주친다.

역효과를 경계해 함수에 입력을 추가 하거나 옵션 값을 추가해서 억지로 내부에서 수정하면 좋지 못한 코드가 된다. 응급 처치를 몇번 거치면 누구도 알아볼 수 없는 코드가 탄생하기 때문

* + 기술부채

응급 처치는 긴박한 문제를 해결할 때 유용. 현재 서비스에서 발생하는 버그를 잡아야하는 데 전체 구조 리팩토링을 할 수 없기 때문. 다만 이런 응급조치한 코드는 부채가 되고 상환 의무를 져야함. 기술 부채가 과도해지면 유지 보수하는 데 큰 비용이 들어가기 때문에 적정선을 찾는 게 중요.

**좋지 않는 코드 줄이기**

    좋은 코드란 좋지 않는 코드가 없는 코드. 작성된 모든 코드는 유지 보수 비용이 필     요하다.

    점진적으로 개선이 가능해야한다.

* **의존성을 드러내기 위한 추상화**

        함수를 분리할 때는 그 함수의 역할을 인지하고 하나의 역할 만 하도록 정의.

        즉 의존성을 드러내기 위함이 추출의 목적.

        한 파일에 여러 로직이 얽혀있을 때 각 코드 조각 중 서로 의존 관계에 있는         것들을 추출 해야한다. 이렇게 추상화가 된 함수는 하나의 목적(역할)을 갖게         되고 의미 있는 추출(추상화)이 이루어 진다. 무엇을 분리할지 결정 한다음         어디에 분리할지 결정해야한다. 목적에 따라 맞는 위치에 정의 해주는 것이         중요하다.

* **삭제하기 쉬운 코드와 어려운 코드의 분리.**

        비지니스 요구사항 맞추다 보면 어쩔 수 없이 복잡한 코드가 작성됨.

        이런 코드는 읽기도 어렵고 삭제도 어려움. 복잡한 요구 사항을 담고 있는 코드는         변경에 유연하지 못하기 때문에 별도로 분리해둬야 한다.

        좋지 않는 코드는 제대로 관리 될 수 있도록 격리 해야한다.

        주석과 함께 격리해둬면 기존의 코드의 흐름을 끊지 않고 복잡한 코드를         이 해 하는 데 도움을 줄 수 있다.

* **일관성 있는 코드**

일관성은 합의된 규칙을 기반으로 만들어진다. 코드에 일관성이 지켜진다면 예측이 가능하다. 즉 어느 곳에 어떤 코드가 위치하는 지 예상 할 수 있다. 최소한의 가독성을 보장 할 수 있으므로 프로젝트 팀원 간의 그라운드 룰이 필요하다.

* + 네이밍

변수명 네이밍 뿐만 아니라 함수의 네이밍에도 규칙이 있으면 일관성을 지킬수 있다. 예를 들어 react hooks API도 hooks임을 네이밍에서부터 드러내기 위해 use-\* prefix를 사용한다.

* + 디렉터리 구조

어디에 분리할지와 관련있다.

* + - 역할에 따라 분리한 디렉터리

애플리케이션이 커지고 복잡해지면 코드가 정의된 곳과 코드 사용되는 곳이 먹어지는 문제가 발생

* + - 도메인 영역에 따라 디렉터리 분리. 애플리케이션 전반에 공통으로 사용 되는 것들만 Top Level의 @shared 디렉터리에서 관리하고 나머지는 응집도 높게 각각의 디렉터리에서 관리. 서로 참조하고 맥락을 공유하는 코드들이 근처에 있어 관리하기 편하다는 장점이 존재.

참고 <https://ahnheejong.name/articles/package-structure-with-the-principal-of-locality-in-mind/>

* **확장성 있는 코드**

확장이 어려운 코드는 내부에서 많은 변경이 발생하며 이것은 코드를 읽기 어렵게 만든다. 그러나 확장에 너무 많이 열려있으면 내부 API를 수정하기 어려워지는 문제도 발생 할 수 있다. 라이브러리 확장성이 높으면 많은 use-case에 대응 할 수 있겠지만 내부적으로 큰 변화가 있을 때, Breaking Change가 생길 가능성이 높아진다. 적정선을 찾아야 한다.