Chapter9

코드리뷰 : 마음가짐

더 나은 코드리뷰 문화를 위한 마음가짐

PR 기본 마인드

코드 리뷰이 (PR 요청자)

상대방에게는 추가 업무가 될 수 있기에
추가적인 업무 부하를 주지 않도록, 준비를 철저하게 하고 PR을 요청한다.

리뷰를 보며, 마음의 상처를 덜 받는다.

코드 리뷰어

상대가 마음의 상처를 입지 않도록, 정중한 표현을 사용하도록 노력한다.

코드 리뷰이 - 마인드

팀원의 코멘트에 상처를 덜 받자.

극단적인 마인드를 제어하자.

예시 1) 감히 (?) : 직급도, 나이도 어린 것이.. 감히 나에게 이런 지적을?
예시 2) 코멘트 주셔서 감사합니다. 아이고, 네네.. 무조건 맞습니다.

코멘트는 커뮤니케이션의 시작이다.

코멘트에 의문이 있다면,
충분히 분석 / 조사를 해본 후, 구체적인 내용에 대해 논의를 한다.

코멘트 내용을 받아들이지 않아도 된다.

받아들이지 않고 진행하더라도
해당 내용에 대해 충분히 검토한 것이기에, 품질에 도움이 된다.

코드 리뷰이 – PR 작성시

리뷰어를, 돈을 내고 책을 구매한 독자들이라고 생각해야 한다.

읽기 편한 문체와 전문용어 사용

- 리뷰어들이 이해하기 쉽게 써야 한다.

PR의 단위가 적절해야 한다.

너무 많은 파일과 많은 소스코드를 리뷰하라고 하면, 리뷰하지 않는다.
PR 코드가 딱 떨어지는 적은 정도

- 예시) Demo 노래 평가시, 노래를 이해하기 좋은 구절 정도가 PR 단위로 좋다.
- PR 단위 = Branch 단위

코드 리뷰이 – 셀프 리뷰

셀프 리뷰를 먼저 하고, 코드 리뷰를 요청한다.

이 PR을 보면서
내가 리뷰어라면 어떤 Comment를 달 것인가?를 예측하고,
이것에 대해 개선을 먼저 한다.

그리고 나서 다시 PR을 준비한다.

나에게 추가적인 일인 만큼,
상대방에도 추가적인 일이 되므로, PR은 꼼꼼한 대비가 필요하다.

코드 리뷰이 – 테스트 정보 제공

테스트 정보를 제공한다.

테스트를 어느정도 했고, 충분히 검증이 되었음을 명시한다.

코드 리뷰어가 품질에 대한 의심이 없도록 해주어야 한다.

Unit Test 결과
가능하다면, 더 높은 Level의 테스트 까지
가능하다면, 성능 분석까지
Self 코드리뷰 결과까지

코드 리뷰어가, 더 Deep한 숨은 버그 발생 요소 &

코드 개선점에 대해 논의할 수 있도록 테스트 정보를 제공한다.

코드 리뷰이 – 내용 정리

배경 설명

어떤 History에 의해서, 어떤 코드를 왜 바꿨는지,

미팅에 참석했으면 알 수 있는 정보라도, 다시 적는다.

변경 내용

기존 대비 변경 내용에 대해 내용을 적는다.

코드 보면 알 수 있어도, 이해에 도움이 되도록 적는다.

테스트 이력

어떠한 테스트를 했고, 성능 / 품질에 이상이 없음을 알린다.

정적 분석도구 결과 / 코드 커버리지 결과 등

코드리뷰어 – 도움을 위한 Review

코드 리뷰이와 다른 코드 리뷰어까지
모두가 도움이 될 수 있는 Review를 남긴다.

안 좋은 예시)

이 부분을 수정했으면 좋겠습니다.
요즘 이런 식으로 개발 안합니다.(X)

좋은 예시)

이 부분을 수정했으면 좋겠습니다.
이유는 이렇습니다,
고치는 방법은
1. 어떤 방법
2. 어떤 방법
인데 제 개인적으로는 1번을 더 추천합니다. 이러하기 때문입니다.
관련 자료는 이렇고, 검토를 부탁드립니다.

코드리뷰어 – 명확한 표현

- 중립적인 표현은 하지 않는다.

  Approve인지, Request changes인지 명확하게 의견을 표현한다.
  리뷰를 했지만, Comment로 모호한 의견을 남기지 않는다.
  수정할 사항이 없으면 Comment가 아닌 Approve이다.

- 좋지 못한 예시)

  comment : 좋은 것 같지만, 다른 분 의견이 궁금합니다.
  코드 리뷰했지만, 무응답

- 좋은 예시)

  무엇을 왜 바꿔야 하는지가 명확해야 한다.
  Approve : 이 코드는 가독성이 있는 Clean한 코드로 생각합니다.
  저도 이런 코드 Style로 진행하고자 합니다.
  Request changes 이 부분에서 이런 부분은 이러한 risk가 있을 것으로 보입니다. 왜냐하면 ..,

코드리뷰어 – 재검토

리뷰를 남길 때, 다음 항목을 재검토 해보자.

1. 이 Review가 코드 리뷰어에게, 도움이 되는 내용인지 검토해본다.
2. 이 Review가 코드 리뷰어에게, 부정적인 지적 / 비난으로 들릴 수 있을지 생각해본다.
3. 모든 팀원들이, 내가 작성한 Review처럼 남길 때,
   긍정적인 코드 리뷰 문화가 만들어질 수 있을지 생각해본다.

[참고자료]

- 구글의 코드리뷰 문화
  - https://hanbit.co.kr/channel/category/category_view.html?cms_code=CMS3858769941
  - https://m.post.naver.com/viewer/postView.naver?volumeNo=30978428&memberNo=36733075

- 카카오 코드리뷰
  - https://tech.kakao.com/2022/03/17/2022-newkrew-onboarding-codereview/

- 구글 코드 리뷰 가이드
  - 영문 : https://google.github.io/eng-practices/review/
  - 한글번역 : https://soojin.ro/review/

mincoding

좋은 Unit Test
나쁜 Unit Test

실질적인 Unit Test 활용 팁

테스트 신뢰도가 없어지는 예시

UnitTest의 신뢰도가 잃어버려지는 상황 예시

1. 개발 방향이 자주 변경되면서 개발하였다.
2. 일부 변경, 리팩토링 시도 할 때 마다 유닛테스트 Fail이 되었다.
3. 일부는 진짜 Fail이지만, 대다수는 거짓양성이었다.
4. 처음에는 테스트 실패를 처리하려고 했지만 거짓양성이 주류를 이뤄 비활성화를 하곤한다.
5. 그러다 보니, 나중에 살펴볼 생각으로 일단 비활성화 부터 하곤 했다.
6. 이후에는 모든 테스트가 비활성화가 되었다.
7. 이후 비활성화된 유닛테스트 코드에는 아무도 손을 대지 않았다.

나쁜 TestCase를 만들게되면,
잦은 Fail 발생으로 (거짓양성)
점차 UnitTest의 신뢰성을 잃게 된다.

Tip 1. 유닛테스트는 회귀 방지 역할을 해야한다.

- **회귀 방지**

  회귀 버그 : 코드 수정으로 인해 기존 잘 되는 기능이 안돌아가는 버그

  테스트가 가능한 많은 코드를 실행하는 것을 목표로 해야,
  회귀 버그 방지 역할을 할 수 있다.

- **코드 커버리지가 높은 테스트 코드를 만들자.**

Tip 2. 유닛테스트는 리팩토링 내성을 가져야 한다.

- **리팩토링 내성**

  리팩토링을 하더라도 PASS가 잘 될 수 있는 테스트 코드 이어야 한다는 것이다.

  조금만 수정해도 매번 Fail이 발생하면, 거짓 양성이 비번해져 유닛테스트 신뢰를 잃을 수 있음. (거짓양성 : 실제 고장은 아니지만, 테스트가 빨간색으로 뜨는 것)

  유닛테스트에 거짓 양성이 없어야한다.

Tip 3. 적시에 테스트 코드 작성

개발할 때 테스트 작성

- 나중에 테스트 작성하면, 중요한 테스트 포인트를 잊어버린다.

Tip 4. 최소한의 유지비로, 최대한의 가치

핵심 로직, 결함 발생 가능성 높은 부분 등의

중요한 부분을 테스트하는 코드 작성

1. 최소한의 유지비로 최대한의 가치를 끌어내야 한다.
   (회귀 방지가 잘 되는 코드)

2. 불필요한 유닛 테스트가 무리하게 많으면,
   유지보수 할 것도 많아진다.

3. 가장 중요한 부분을 테스트하는지 체크하자.

4. 가치가 없는 테스트는 삭제하자.

Tip 5. 이해하기 쉬운 테스트 코드

- 테스트코드는 모듈의 사용설명서로 사용되기 때문에
  이해하기 쉽게 작성해야한다.

- 개발을 시작하기 위해서는
  기존에 있는 유닛테스트가 무엇을 하는지, 의도를 정확히 이해해야한다.
  (이해하기 어려우면, 업무를 시작하는데 지연 발생)

- 의도 파악이 되야,
  모듈 수정 후 유닛테스트로 검증하거나, 유닛테스트를 유지보수할 수 있다.

Tip 6. UnitTest도 지속적인 리팩토링

- 테스트도 지속적으로 관리해야 한다.

  비슷한 테스트 코드들이 많아지는 경우, 이런 테스트 코드들을 추가하기 쉽도록 리팩토링 해준다.

Tip 7. 잘 설명되는 실패

- Fail시 원인과 문제점을 명확하게 설명해야 한다.
- 잘 알려주지않으면, 담당자가 많은 시간을 낭비할 수 있다.
  ex) 리턴값이 2가 아니라, 3이라 Fail 발생 → ???? 이해하기 어렵다.

- 해결방법
  1. 실패 메세지를 정확하게 적는다. 그리고 이것이 유용한지 생각해보자.
  2. 어떤 테스트인지 정확히 테케 이름을 서술

Tip 8. 테스트하기 쉽고, 빠르게 실행 가능해야 한다.

- 단위테스트가 빨라야, 일상 작업중에 자주 실행한다.

Tip 9. AAA 패턴, 각 구절의 적당한 크기

- Arrange
  가장 길다.
  만약 너무 크다면 별도의 팩토리 / 테스트 메서드를 추가해두는 것이 좋다.

- Act
  하나의 실행 구절
  만약 실행 구절이 두 줄 이상인 경우,
  기능 구현 코드 자체 문제 이슈 or 캡슐화를 덜 했는지 확인 해보자.

- Assert
  하나의 Behvior 의미를 갖는 Assertion 문으로 구성
  만약 한 테스트 코드에서 여러 의미를 갖는검증을 하는 경우,
  Unit Test테스트가 아닌 통합 테스트이다.

Tip 10. 이럴 때, 테스트 더블을 고민해보자.

- Test → A → B → C 의존할 때
  B나 C 이상으로 더 내부까지 신경쓸 필요가 없기에, 이때 목을 쓴다.
  예를 들어, 은행출금이 실제로 일어나는 부분은 더블로 대체한다.

- 난수발생기에 의존하는 경우는 더블을 쓰는 것이 더 좋다.
  함수를 호출할 때 마다 결과가 달라지는 경우, 테스트 결과를 신뢰하기 어렵다.

Tip 11. Test Double의 단점을 이해한다.

- 구현의 세부사항을
  테스트 코드에서 직접적으로 명시하게 된다.

→ 리팩토링 내성이 낮아진다.

```python
import pytest
from unittest.mock import Mock, call

def test_method_calls_in_order():
    mock_object = Mock()

    mock_object.method_a("hello")
    mock_object.method_b(42)
    mock_object.method_a("world")

    mock_object.assert_has_calls([
        call.method_a("hello"),
        call.method_b(42),
        call.method_a("world"),
    ])
```

감사합니다.