실질적인 Unit Test 활용 팁

좋은 Unit Test
나쁜 Unit Test

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

중요한 부분을 테스트하는 코드 작성

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

2. 불필요한 유닛테스트가 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.

- 관리가 되지 않는 UnitTest들은
  지속적으로 Fail이 나오게 되며,
  개발자들이 Fail에 대한 심각성을 잃어버리게 된다.

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.

```
class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
    MOCK_METHOD(int, getGop, (int a, int b), ());
};

TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    {
        InSequence seq;
        EXPECT_CALL(mock_cal, getSum(1, 1));
        EXPECT_CALL(mock_cal, getGop(1, 2));
        EXPECT_CALL(mock_cal, getSum(1, 3));
        EXPECT_CALL(mock_cal, getGop(1, 4));
    }

    cout << mock_cal.getSum(1, 1) << "\n";
    cout << mock_cal.getGop(1, 2) << "\n";
    cout << mock_cal.getSum(1, 3) << "\n";
    cout << mock_cal.getGop(1, 4) << "\n";
}
```