[Project News] 리펙터링

- 진행: 황수재 컨설턴트
- 날짜: 2020.11.09
- 목차
  - 📣 라이브 방송 안내
  - 📣 최종평가 및 발표회 일정
  1. 리펙터링은 왜 하는가
  2. 리펙터링은 언제 하는가
  3. 리펙터링 어떻게 하는가
  4. 리펙터링 예시

📣 라이브 방송 안내

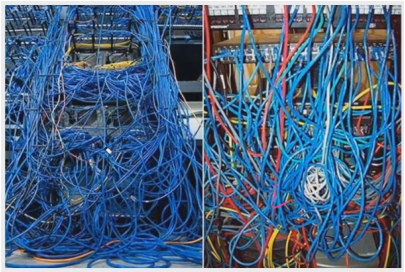
주차	일자	요일	시간	구분	주요내용
5주차	11월 9일	월	9시	Project News	클린코드 및 리펙토링
5주차	11월 11일	수	9시	IT Essential	웹서버 보안 방법
5주차	11월 13일	금	9시	Project Review	3기 자율 프로젝트 리뷰

📣 최종평가 및 발표회 일정

> 발표회와 평가 일정, 수료식 일정을 감안하여 다음 주 월요일이 반 발표일 입니다.
>
> 이번 주 정말 화이팅 입니다!

최종평가 산출물/UCC 제출	• 11월 16일 (월)
본선 발표회 - 서울 -	• 11월 18일(수)
본선 발표회 - 지역 -	• 11월 19일(목)
결선 발표회	• 11월 23일(월)

1. 리펙터링은 왜 하는가

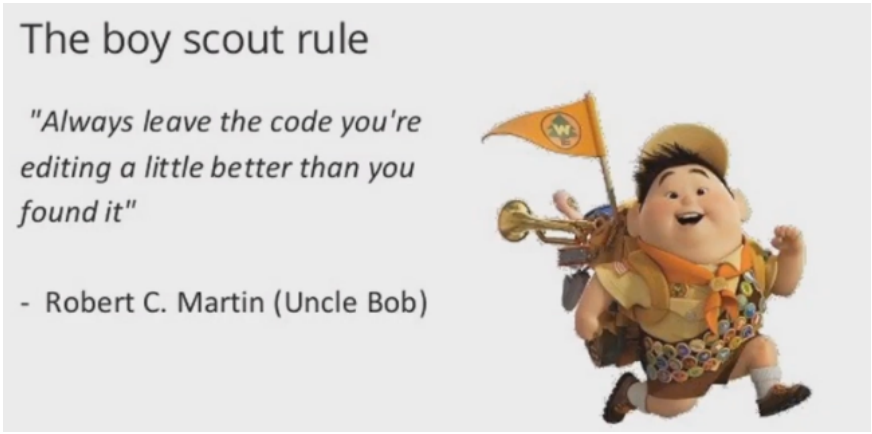



- 스파게티 코드라 하는 코드들이 있다면 버그 찾기도 어렵고, 새로운 기능을 추가하기도 어려우며 유지보수도 어렵습니다.
- Technical debt
    - **기술 부채**(technical debt, design debt, code debt)는 현 시점에서 더 오래 소요될 수 있는 더 나은 접근방식을 사용하는 대신 쉬운(제한된) 솔루션을 채택함으로써 발생되는 추가적인 재작업의 비용을 반영하는 소프트웨어 개발의 한 관점이다.

2. 리펙터링은 언제 하는가

> 마감이 1주일 남은 시점에서 기본 기능이 아직 구현되어있지 않다면 기능 구현을 먼저 한다.
>
> 발표를 할 때 기능 완성이 우선이며, 코드가 얼마나 깔끔한가는 이보다 후순위!

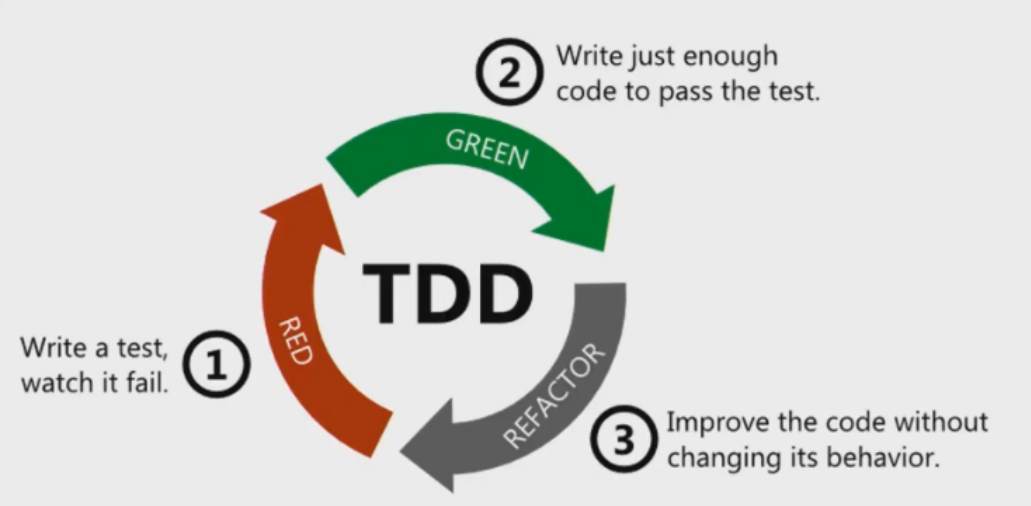
1) The Boy Scout Rule



- 내가 코드를 접했을 때, 버그를 접하거나, 유지보수를 해야하거나, 신규 기능을 추가해야 할 때, 항상 내가 접했던 코드보다 깔끔하게 만들어라.
- 기능 A 를 만들어야 한다면 A 를 빠르게 만들고, 이후에 정리하는 것

2) TDD

- Test Driven Development
    - **테스트 주도 개발**, 매우 짧은 개발 사이클을 반복하는 소프트웨어 개발 프로세스 중 하나이다. 개발자는 먼저 요구사항을 검증하는 자동화된 테스트 케이스를 작성한다. (RED) 그런 후에, 그 테스트 케이스를 통과하기 위한 최소한의 코드를 생성한다. (GREEN) 마지막으로 작성한 코드를 표준에 맞도록 리팩토링한다. (REFACTOR)



- '로그인이 되어야 한다' 라는 테스트 케이스(TC) 가 있다면 Green 에서 코드를 어떻게 짜든 일단 로그인이 되기만 하도록 만든다. 이후 Refactor 단계에서 실제로 로그인 이라는 기능은 건들지 않고, 기능에는 영향을 주지 않고 코드를 정돈한다는 마음으로 개선하는 것이다.

3. 리펙터링 어떻게 하는가

1) 리펙터링 중에는 새로운 기능을 추가하지 않는다. ☆

- 해야 하는 것을 작은 단위로 먼저 한다.
- Red, Green, Refactor 별로 커밋을 따로 만들 수 도 있다.

2) 리펙터링 후 기존에 존재하는 모든 테스트가 통과되어야 한다.

- 자동화 된 테스트 케이스가 없다 하더라도, 최소한의 테스트 케이스는 있을 것 (문서화 하면 더 좋습니다)
- 원래 로그인이 되었다면, 리펙터링 이후에도 로그인이 되어야 한다.
- 기능 수정은 다른 이야기이기 때문에, 제외 되어야 한다.

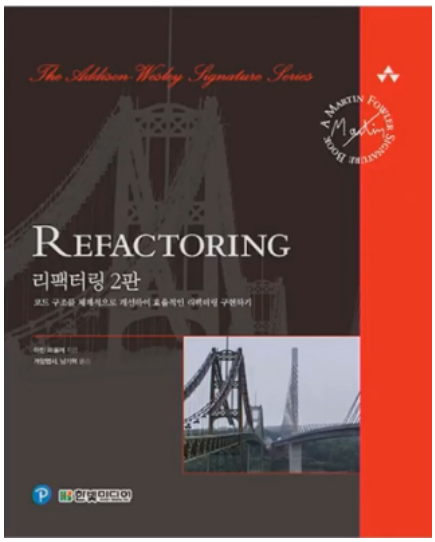
3) 코드의 품질을 높인다 (응집도, 결합도, 가독성 등)

- 응집도는 높을수록 좋으며 결합도는 낮을수록 좋다.

4) 추천 도서

- 마틴 파울러, "리펙터링 2판"

- 마틴 파울러: "클린 코드"의 저자

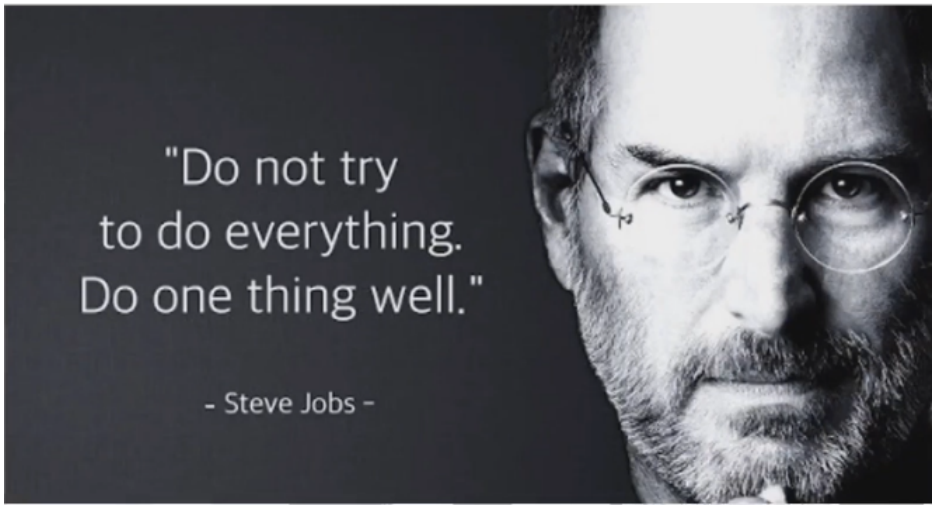


4. 리펙터링 예시

1) Long Method

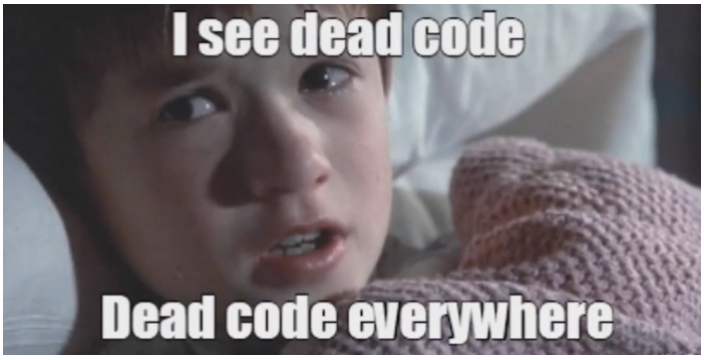
- Long Method
  - 메소드가 지나치게 긴 경우
  - 이러한 메소드는 기존의 매소드에 계속해서 기능을 추가하는 경우 발생하곤 한다.
- **해결책**
  - 보이스카웃 룰에 따라, 자신이 작성한 코드를 되돌아 보면서 기능을 분리하거나, 코드를 정리하고 넘어가도록 하자.
  - 하나를 잘 하는 Method 를 만든다. Method 가 더 작아질 수 있을 것 같다면 작은 단위로 모듈화 한다.

    + 객체지향프로그램의 Solid, 단일 책임 원칙
  - Method 에 너무 많은 기능을 부여하지 말고, 작은 Method 를 합쳐가며 기능을 만든다.



2) Dead Code

- Dead Code
  - 주석처리 하거나, 사용하지 않는 코드. 프로그래밍의 동작과 크게 관련이 없는 코드
  - 넓은 범위로는 사용되지 않는 파일을 포함한다.
  - 코드베이스를 잘 모르는 사람이 그 코드를 보게 된다면 이를 이해하기 위한 오버헤드가 발생한다.



- **해결책**
  - 주석, 사용하지 않는 변수의 선언 등은 지우면 된다! 이를 위해 Git 등 형상관리 툴을 쓴다.
  - 불필요한 파일을 삭제한다.
  - 정적 분석 도구를 활용한다. (SonarQube, Linter 등)
  - 좋은 IDE 를 사용한다.

3) Shotgun Surgery

- Shotgun Surgery
  - 하나의 기능을 수정하고자 할 때, 연결 된 다른 Classes 에서도 수정을 해야하는 경우
- **해결책**
  - 계좌에서 출금/이체/수수료를 지급하는 Class, 다음 코드에서 문제점을 생각해보자.

```
class SavingsAccount {

  withdraw(amount) {
    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(WITHDRAWAL_MIN_BALANCE);
      return;
    }
    // implementation
  }

  transfer(amount) {
    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(TRANSFER_MIN_BALANCE);
      return;
    }
    // implementation
  }

  processFees(fee) {
    this.balance = this.balance - fee;

    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(MIN_BALANCE_WARNING);
    }
  }
}
```

- 동일한 조건이 3번 반복되고 있다. 조건문을 변경해야 한다면, 매 조건을 찾아서 수정해주어야 한다.
- 다음과 같이 하나의 책임을 가진 메서드로 만든다.
- 한 번의 변경으로 모든 곳에 반영 될 수 있다. 유지보수에 용이하다.

```
class SavingsAccount {

  withdraw(amount) {
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(WITHDRAWAL_MIN_BALANCE);
      return;
    }
    // implementation
  }

  transfer(amount) {
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(TRANSFER_MIN_BALANCE);
      return;
    }
    // implementation
  }

  processFees(fee) {
    this.balance = this.balance - fee
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(MIN_BALANCE_WARNING);
    }
  }

  accountIsUnderMinimum() {
    return this.balance < MIN_BALANCE;
  }
}
```