6. 빈틈없는 코드 만들기

6. 빈틈없는 코드 만들기

- 6.1 함수형 프로그래밍과 단위 테스트
- 6.2 명령형 프로그램 테스트의 어려움
- 6.3 함수형 코드를 테스트
- 6.4 속성 기반 테스트로 명세 담기
- 6.5 코드 실행률로 효율 측정

6.1 함수형 프로그래밍과 단위 테스트

테스트의 분류

- 단위 테스트
- 통합 테스트
- 인수 테스트

위쪽으로 갈 수록 코드에 미치는 영향이 갈수록 확대

테스트 코드

- 도구 : Qunit
- 테스트 코드는 메인 APP 코드와 별개의 자바스크립트 파일에 두며, 대상 함수를 전부 들여와서(import)하여 테스트한다.
- 기본 구조

```
QUnit.test('사람 검색 기능을 테스트한다', (assert) => {
  const ssn = '444-44-4444';
  const p = findPerson(ssn);
  assert.equal(p,ssn, ssn);
});
```

6.2 명령형 프로그램 테스트의 어려움

부수효과와 변이 발생, 전역 상태와 변이에 의존함

- 식별은 물론 간단한 작업으로도 분해하기도 어려움
- 결과를 들쭉날쭉하게 만드는 공유자원에 의존
- 반드시 평가 순서를 미리 정해야함

6.3 함수형 코드를 테스트

함수형 코드 테스트 시 이점

- 함수를 블랙박스처럼 취급
- 제어 흐름 대신 비지니스 로직에 집중
- 모나드 격리를 통해 순수한 부분과 불순한 부분을 분리
- 외부 디펜던시를 모의 : 명령형에서 외부 자원을 사용하는 부분이 있다면 테스트가어렵다. 하지만 함수형에서는 자유롭게 모킹하고 해당 부분만 테스트하면 그만임~

> 결국 참조 투명성!

함수를 블랙박스처럼 취급

순수성
FP에서는 APP의 다른 부분에 구애받지 않고 느슨하게 입력값을 결합하는 함수를 독립적으로 작성. 이런 함수는 부수효과가 없고 참조 투명하므로, 임의의 순서로 몇 번이고 실행하더라도 결과가 동일하고 예측 가능한 테스트를 작성할 수 있다.

> 주어진 입력에 맞는 출력을 내는지만 집중

제어 흐름 대신 비지니스 로직에 집중

> 람다JS 같은 라이브러리 함수들은 이미 완벽히 검증을 마쳐서 curry , compose 같은 조합기를 사용하여 함수를 만드는게 좋다.

모나드 격리를 통해 순수/불순 코드 분리

명령형 `showStudent()` 라는 함수 가 있다. 다양하게 조합한 SSN 입력값을 넣고 실행해도 문제없는지, 이 하나만을 확인하려고 해도 매번 전체 프로그램을 실행하여 테스트할 수 밖에 없으니 매우 비효율적이고 생산적이지 못하다. 모든 구문이 단단히 결합되어 테스트를 철저하게 할 수 없는 것도 문제이다.

함수형 `showStudent()`

```
const showStudent = R.compose(
        map(append('#student-info')),
    liftIO,
    getOrElse('학생을 찾을 수 없습니다!'),
    map(csv), // 테스트
    map(R.props(['ssn', 'firstname', 'lastname'])),
    chain(findStudent), // 테스트
    chain(checkLengthSsn), // 테스트
    lift(cleanInput)
);
```

함수형은 명령형을 잘개 쪼개 합성 및 모나드를 이용해 재조립한다. 따라서 테스트 가능한 영역이 늘어나고 불순한 코드에서 순수함수를 떼어낼 수 있다.

6.4 속성 기반 테스트로 명세 담기

- 하스켈의 퀵체크(속성 기반 테스트의 표준 구현체)
- JS는 퀵체크를 모방한 JSCheck 가 있음 by god 더글라스 크락포드 행님
- 대체 무엇?
    - 속성을 무작위로 생성하여 프로그램 명세 및 속성을 테스트함
    - 순수프로그램의 명세를 프로그램이 만족해야하는 속성 형태로 설계하면, 수많은 케이스를 자동 생성하여 테스트를 실행 후 보고서를 생성함

6.5 코드 실행률로 효율 측정

- 코드 실행률은 프로그램을 단위 테스트했을 때 실제로 실행된 코드 라인 수의 비율(%)로 측정한다.
- 도구로는 Blanket.js로 계산한다.
- 다음과 같은 단계로 작동한다.
    i. 소스파일 읽기
    ii. 추적기 라인을 추가해서 코드를 장착
    iii. 테스트 실행기에 걸어 상세 실행률 정보를 산출

코드 실행률

- 긍정 테스트 (정상적 입력)
  - 명령형 : 80%
  - 함수형 : 99%
- 부정 테스트 (잘못된 입력)
  - 명령형 : 40% // 잘못된 입력이 들어오면 훌쩍 건너뜀
  - 함수형 : 80% // 말도 안되는게 들어와도 다음 함수에서 진행

정리

- 아주 단순한 함수들을 결합하는 추상화로 프로그램을 모듈화할 수 있다.
- 순수함수에 기반을 둔 모듈적인 코드는 테스트하기 쉽고, 속성 기반 테스트처럼 더 엄격한 테스트 방법론을 적용할 수 있다.
- 테스트 가능한 코드가 되려면 제어 흐름이 직관적이어야 한다.
- 제어 흐름을 단순화하면 전체 프로그램의 복잡도가 줄어든다. 복잡도는 각종 지표를 통해 정량적으로 측정할 수 있다.
- 복잡도가 줄면 프로그램을 읽고 이해하기 쉽다.

마무리 및 느낀점

- 걍 함수형 쓰셈
- 명령형보다 테스트 도입하기 쉬움
- 테스트 도구는 `Qunit`, `Blanket.JS`, `JSCheck`를 적절히 버무려서 사용하도록
- 그냥 함수형이 좋음
- 이유있이 함수형이 좋음
- 함수형은 깨끗함
- 명령형은 불순함

> 마지막으로 한 마디 함수형 또 쓰셈