Test-Driven Development :By Example

4장 - 프라이버시 & 5장 - 솔직히 말하자면 & 6장 - 돌아온 '모두를 위한 평등'

Date
2023.08.08

Presenter
황유란

테스트가 원하는 바

기존의 코드는 테스트가 정확히 원하는 바를 말하지 않는다.

- 객체의 값에 원하는 곱수만큼의 값을 가지는 객체를 반환한다

```
fun testMultiplication() {
  val five = Dollar(5)
  var product: Dollar = five.times(2)
  assertEquals(10, product.amount)
  product = five.times(3)
  assertEquals(15, product.amount)
}
```

```
fun testMultiplication() {
  val five = Dollar(5)
  var product: Dollar = five.times(2)
  assertEquals(Dollar(10), product.amount)
  product = five.times(3)
  assertEquals(Dollar(15), product.amount)
}
```

테스트가 원하는 바

비교대상을 객체로 변경하게 되어 product.amount 대신 직접 비교

- amount라는 값은 더이상 호출하지 않게 되어 private으로 변경 가능

```
fun testMultiplication() {
  val five = Dollar(5)
  assertEquals(Dollar(10), five.times(2))
  assertEquals(Dollar(15), five.times(3))
}
```

검토

- 테스트를 향상시키기 위해서만 개발된 기능을 사용
- 두 테스트가 동시에 실패하면 망한다
    - 동치성에 대한 테스트가 실패하면 곱셈 테스트도 실패한다.
- 위험 요소가 있음에도 계속 진행
    - 자신감 가지고 전진할 수 있을 만큼만 결합의 정도를 낮추기
- 테스트와 코드 사이 결합도를 낮추기 위해 테스트하는 객체의 새 기능을 사용

환율을 계산한 덧셈

- $5 + 10CHF = $10 (환율 2:1인 경우)
- 환율을 계산해서 덧셈하는 기능 테스트하기

환율을 계산한 덧셈

- 일단 franc(프랑)을 표현할 수 있는 객체가 필요
- Franc끼리의 덧셈을 먼저 테스트
- Dollar 테스트를 복사해서 사용

```
fun testFrancMultiplication() {
  val five = Franc(5)
  assertEquals(Franc(10), five.times(2))
  assertEquals(Franc(15), five.times(3))
}
```

중복해서 사용해도 되는 이유

- 여기서 4번까지는 빨리 진행해야 하고 여기까지는 어떤 죄든 저지를 수 있다.
- 하지만 5번 없이는 4단계도 제대로 돌아가지 않으므로 5번까지 꼭 해야한다
    - 네발짜리 에어론 의자는 자빠진다

TDD의 리듬

1. 테스트 작성
2. 컴파일되게 하기
3. 실패하는지 확인하기 위해 실행
4. 실행하게 만듦
5. 중복 제거

검토

- 바로 환율 계산 후 덧셈 테스트를 할 수 없다
  - Franc끼리의 덧셈이라는 작은 테스트부터
- Dollar를 복사해서 중복으로 만들고 조금 고쳐서 테스트를 작성
- 중복을 해결해야 한다.
- 새로운 할 일 발생
  - equals 일반화하기

중복 제거하기

- 5장에서 중복으로 통과시킨 테스트 청소하기
- 한 클래스가 다른 클래스를 상속하게 하기 (Dollar가 Franc을 상속하거나 Franc이 Dollar를 상속하거나)
    - 어떤 코드도 구원하지 못함
- 두 클래스의 상위 클래스를 찾아내기 (Money)
    - 시간이 걸리긴 하지만 정상 동작

Money 객체 만들기

- equals를 일반화해야 함
  - Money가 가지고 있으면 어떨까?
- equals를 상위로 올리기 위해 amount 변수도 같이 올림
  - 상속할 수 있게 protected로

```
open class Money() {
    open val amount: Int = 0
}

class Dollar() : Money() {
    override val amount: Int = 5
}
```

Money 객체 만들기

- 기존 Dollar의 equals부분에서 임시 변수 타입 Money로 변경
  - Money 클래스로 옮길 수 있게 됨

```
//Dollar class

fun equals(object: Any): Boolean{
  val money: Money = (Money) object
  return amount == money.amount
}
```

Franc의 equals

- Franc의 equals 테스트가 없는 상태
- 적절한 테스트가 없는 경우에서 TDD를 해야하는 경우
  - 있으면 좋을 것 같은 테스트를 작성
  - 그렇지 않으면 리팩토링하다가 뭔가 깨트릴 것
- Dollar의 equals 테스트를 복붙해서 Franc 테스트를 만듦

```
fun testEquality() {
  assertTrue(Dollar(5).equals(Dollar(5))
  assertFalse(Dollar(5).equals(Dollar(6))
  assertTrue(Franc(5).equals(Franc(5))
  assertFalse(Franc(5).equals(Franc(6))
}
```

Franc의 equals 중복 제거하기

- 복붙한 코드를 없애기 위해 Franc도 Money의 하위 클래스로 상속
- Money의 equals와 코드가 동일해지면서 Franc와 Dollar의 equals가 동일해짐

검토

- 공통된 코드를 Dollar(하위)에서 Money(상위)로 올림 (amount, equals())
- Franc도 동일하게 공통된 코드를 Money로 올림
- 불필요한 구현을 제거하기 전에 두 equals() 구현을 일치