나는 실패한 것이 아니다. 단지 제대로 동작하지 않는 10000개의 방법을 발견했을 뿐이다.

– 토머스 에디슨

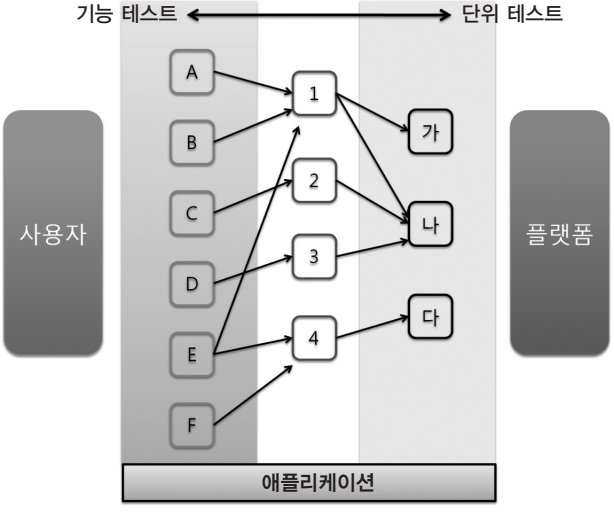
CHAPTER 9

자주 접하게 되는 질문들, FAQ

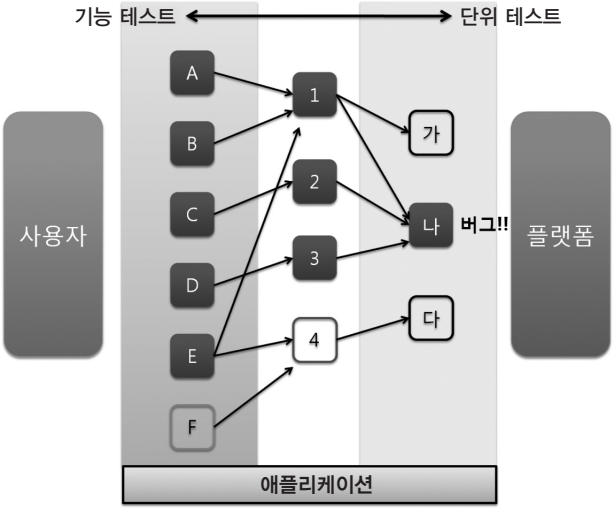
이번 장은 TDD를 적용하면서 흔히 발생하는 문제상황과 질문에 대한 대답을 실어봤습니다. 사실과 의견이 혼재되어 있는 부분인지라, 일부 내용은 논란의 소지가 있는 항목도 있으리라 봅니다. 절대적인 답변이라기보다는 하나의 의견으로 봐주셨으면 좋겠습니다. 그리고 어떤 질문들은 이미 앞에서 이야기된 내용도 있습니다만, 정리하는 차원에서 다시 소개되는 내용도 있습니다. FAQ만 읽게 되실 분들을 위한 배려라고 생각해주셨으면 좋겠습니다. 마지막으로, 몇 가지 이유로 인해 이번 장에 한해 공손체를 사용했습니다. 혹, 어투가 바뀌어서 어색하더라도 너그러운 마음으로 양해 부탁드립니다.

Q 저희는 이미 충분히 많은 기능 테스트를 하고 있습니다. 그런데도 따로 단위 테스트 케이스를 만들 필요가 있을까요?

결론부터 이야기하자면, 기능 테스트와는 별개로 단위 테스트 케이스는 만들어야 합니다.



앞의 그림은 기능 테스트와 단위 테스트의 위치를 표현해본 그림입니다. 기능 테스트(functional test)는 사용자 쪽에 더 가깝고, 단위 테스트는 애플리케이션이 동작하는 플랫폼(시스템)에 더 가깝게 위치합니다. 따라서 기능 테스트는 집중할수록 내부 클래스들의 동작 자체에는 깊게 관여하지 않습니다. 기능 테스트는 현상적으로 애플리케이션의 외부 동작 위주로 진행되기 때문입니다. 이를테면, 아이디와 패스워드를 넣고 로그인 버튼을 눌렀을 때 로그인이 안 되는 건 하나의 테스트가 실패한 상황입니다만, 정확히 어느 모듈로 인해 해당 테스트가 실패했는지는 바로 파악하기가 어렵습니다. 애플리케이션을 큰 단위로 묶어서 실행하는 블랙박스 테스트(black box test)[1]라고 볼 수도 있겠습니다. 그뿐 아니라, 기능 테스트는 내부 소스의 작은 변경에도 크게 영향을 받을 수 있습니다. 다음 그림과 같은 경우를 가정해보겠습니다.

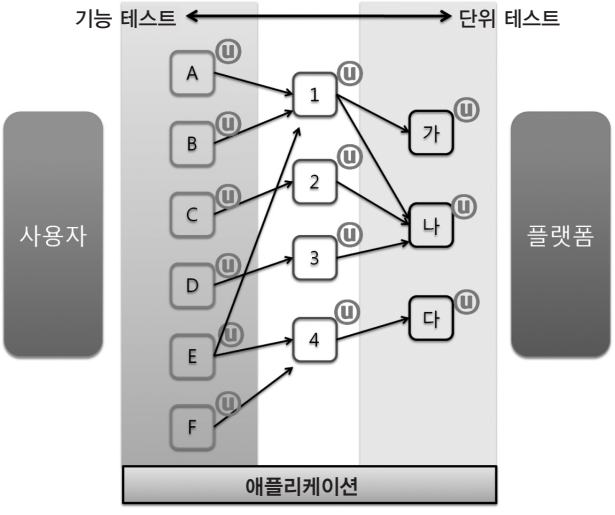


'나' 모듈[2]을 수정했는데 버그가 숨어 들어갔습니다. 이로 인해 많은 숫자의 기능 테스트가 실패하게 됩니다. 위 그림에서는 A~E까지, 약 5개의 기능이 정상적으로 동작하

1 블랙박스 테스트(black box test): 내부 구조를 모르는 상황에서 겉으로 보이는 부분만을 이용해 테스트를 진행하는 방법. 내부 구현이나 모듈 구성은 관심 없고, 외부와 통신할 수 있는 부분에서 의도된 대로 동작하면 된다. 보통, 사용자 테스트나 기능 테스트에서 많이 사용되는 기법이다. 반대되는 기법으로, 화이트 박스 테스트(white box test)가 있다.

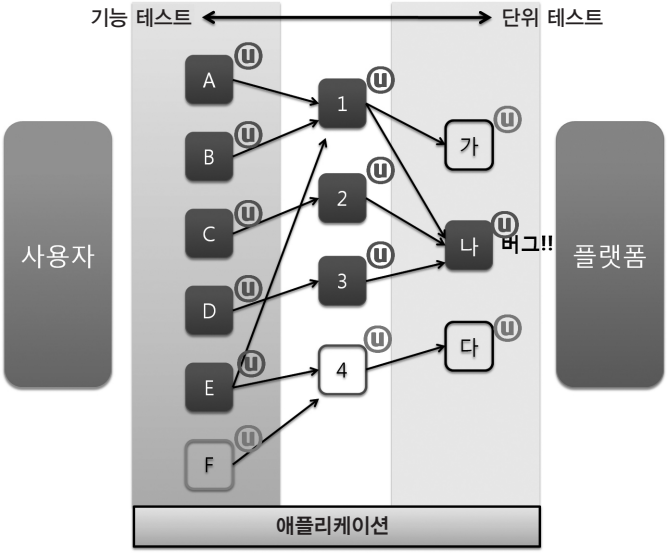
2 클래스나 메소드라고 가정해도 무방하다.

지 않는 것으로 판단됩니다. 물론, '나' 모듈 수정 후 회귀 테스트를 착실히 했다는 가정하에서입니다(회귀 테스트(regression test)를 제대로 안 했다면, 파악조차도 안 됐겠죠). 만일 단순하게 모듈 '나'로 인해서만이 아니라 '가' 모듈이나 '다' 모듈을 함께 수정했고, 그로 인한 버그가 추가로 존재할 경우라면, 기능 테스트만을 수행하는 시스템에서는 문제 부분을 정확히 잡아내기가 굉장히 어려워집니다. 결국 총체적 난국인 상황에 빠지게 됩니다. 물론 기능 테스트는 단위 테스트와는 또 다른 자신만의 가치 있는 의미를 지닙니다. 사용자에겐 이 부분이 가장 밀접하게 피부로 와 닿는 부분이니까요. 그러므로 지금 이야기하려는 부분은 기능 테스트는 효용이 낮다는 게 아니라, 시스템을 구축하고 유지하기 위해서는 기능 테스트만으로는 불충분하다는 이야기입니다. 그럼, 이번엔 모듈별로 단위 테스트를 작성해놓은 경우를 살펴보겠습니다.



단위 테스트 케이스가 작성되어 있는 시스템

현재 단위 테스트 케이스들은 모두 정상적으로 수행되도록 유지되고 있습니다. 앞의 경우와 마찬가지로 '나' 모듈을 수정하다 버그가 발생했다고 가정해보겠습니다. 다음은 해당 상황에서 단위 테스트 케이스의 수행 결과를 보여줍니다. 짙게 칠해진 모듈들의 단위 테스트 케이스들이 실패하게 됩니다.



단위 테스트 케이스들이 오류를 감지해내는 모습

'나'를 수정해서 문제가 생겼을 경우 영향받는 모듈들이 바로 파악이 되며, 버그 수정 시에도 바로 확인이 가능합니다. 현재 모습도 나쁘지는 않지만, 인터페이스나 Mock 객체 등을 이용해 최대한 독립적으로 동작할 수 있도록 단위 테스트 케이스를 만들어놓았다면, 위 그림에서 발생한 것보다 훨씬 적은 테스트 실패 전파(propagation)를 막을 수 있었을 것이고, 문제 확인도 더 간단해졌을 것입니다. 특히 단위 테스트가 잘 이뤄져 있다면, 여러 개의 모듈에서 수정이 일어난 후 발생한 경우에도 매우 유효하게 대응할 수 있게 됩니다. 또한 그림에서 보이는 것처럼, 일부 단위 테스트는 기능 테스트의 역할을 수행하는 수준까지 만들어질 수 있습니다. 단위 테스트를 잘 만들어놓으면, 결과적으로 단위 테스트에 비해 비용이 많이 드는 기능 테스트의 범위나 양을 줄일 수 있습니다.

참고

Evil Unit Testing, http://www.javaranch.com/unit-testing/

Q 개발 시에 결과값을 미리 예상할 수 없는 경우에는 그럼 어떻게 TDD를 진행하나요?

결과값(종료조건)을 모르고 개발을 시작하는 경우는 매우 드물고, 실제 개발 업무인 경우가 아닐 가능성이 높습니다. 이를테면, "레거시의 저 코드는 대체 뭘까? 입력값과 출력값은 대체 어떻게 되는 거야?"라며 메소드나 기능을 테스트해보는 경우가 그렇습니다. 혹은, 그런 모듈의 타 플랫폼 포팅(porting) 작업을 하게 될 때도 마찬가지입니다.

이런 경우에는 임의의 값을 만들어서 실패를 만들고, 실패 때 나온 값을 결과값으로 만들어서 현재 상태가 올바름을 알 수밖에 없습니다. TDD는 사실 개발자가 특정 메소드를 호출했을 때 어떤 값이 리턴될지 알고 있다는 것이 사전조건으로 가정되어 있어서 그렇습니다.

Q 저희는 특정 객체 생성 시 사용되는 값이 랜덤 값입니다. 이런 랜덤 값에 대한 테스트는 어떻게 만드나요?

만일 랜덤한 값이 범위를 가질 경우, 그리고 그 범위를 검사하는 일이 중요하다면, assertEquals가 아닌 부등호 비교와 assertTrue를 이용합니다. 그렇지 않고 분포가 중요하다면, 이를테면 주사위처럼 6개의 숫자가 고르게 나와야 할 경우에는, 오차값을 포함한 통계를 측정해서 판단합니다.

```
Dice dice = new Dice();                    // 주사위 객체를 생성한다.
int [] distribution = new int[7];          // 분포를 저장할 수 있게 배열을 잡는다.
for(int i = 0; i<60000; i++){              // 충분히 많은 숫자에 대해 동작시킨다.
    distribution[dice.roll()]++;           // 주사위 숫자에 해당하는 배열의 숫자가
}                                          // 증가한다.

assertEquals(10000, distribution[1], 100); // 특정 값의 분포가 오차 내에 있는지
                                           // 확인한다.
```

Q 만일 팀 내에 TDD를 통한 단위 테스트 케이스를 작성하지 않고도 이미 높은 수준의 코드를 작성하고 있다면 어떻게 해야 할까요?

비슷한 질문이 Yahoo TDD 그룹에서도 이야기된 적이 있습니다. 대부분이 공감할 만한 대답은 이러했습니다. 단위 테스트 케이스는 현재 소스코드가 정상적으로 동작함을 보장하는 방법 중 하나입니다. 그리고 소스에 이뤄지는 변경이 시스템 전반에 걸쳐 어떤 영향을 미치게 될지에 대한 피드백을 줍니다. 현재에만 주목하지 말고 미래를 보세요. 시간이 지날수록 계속 소스에 대한 수정이 이뤄질 것입니다. TDD를 통해 미래에 투자하세요. TDD의 진정한 가치는 긴 시간에 걸쳐서 얻게 될 겁니다.

Q 테스트 케이스(메소드) 하나에 여러 개의 테스트가 들어 있는데요, 하나가 실패하면 나머지는 실행조차 되지 않아서 불편합니다. 하나의 테스트 메소드 내에 실패하는 assert 문장이 있어도 이후 나머지 부분이 계속 실행되도록 만들고 싶은데 이럴 땐 어떻게 해야 하나요?

우선, 하나의 테스트 메소드는 하나의 목표만 테스트하는 것을 권장합니다. 이럴 경우 대상 메소드 하나에 대해 테스트 메소드는 몇 개가 만들어져서 테스트 메소드 숫자가 늘어나는 경향이 있긴 합니다만, 여러 가지로 이 경우가 적절합니다. 하지만 부득이하게 assert 실패에도 해당 테스트 메소드 진행을 중지하고 싶지 않다면 사용할 수 있는 방법이 몇 가지 있습니다.

우선 JUnit을 기준으로 말씀드리면, JUnit 4에서 제공하는 Rule 중에 ErrorCollector(에러 수집기)라는 기능이 있습니다. 이와 관련해서는 부록 A.1 '놓치기엔 아까운 JUnit 4의 고급 기능들'을 참고하세요.

다른 방법으로는, StringBuilder에 메시지를 쌓는 식으로 테스트하거나 Hamcrest의 allOf 등을 이용할 수도 있습니다. 이런 방식을 Soft assert라고 부릅니다. 또한 Groovy의 closure와 invokeMethod를 이용하는 방식도 있습니다. Groovy를 이용한 방식에 대해서는 다음 사이트에서 찾을 수 있습니다.

참고

"Groovy closures make unit testing with "soft asserts" simple"
http://naleid.com/blog/2009/06/25/groovy-closures-make-unit-testing-with-soft-asserts-simple/

Q 단위 테스트 케이스 코드를 작성하기가 여전히 어렵습니다. 옆 동료를 보니까, 전 잘 이해할 수 없는 코드를 이용해 어떻게든 테스트 코드를 만들면서 진행하고 있던데, 저는 어떻게 해야 할까요?

네. 이해합니다. 간단한 예제만 보다가 실제 업무를 TDD로 작성하는 건 쉬운 일이 아닙니다. 그런데, 제가 다시 질문을 해보겠습니다. TDD 시에 단위 테스트 케이스를 선행 작성해야 한다는 사실이 어렵게 하나요? 아니면 단위 테스트 케이스를 작성할 수 있게 프로그램 코드가 설계되어 있지 않아서 어려운 건가요? 때때로 우리는 '테스트 코드를 작성해야 한다'에 지나치게 집중해버린 나머지 TDD의 목표가 테스트 작성이 아닌, 좀 더 나은 설계에 있다는 사실을 종종 잊곤 합니다.

다음과 같은 코드를 보겠습니다.

```
class Rental {
    Movie movie;
    Rental(Service service) {
        this.movie = service.getMovie();
    }
}
```

위와 같은 코드에 대한 단위 테스트 케이스는 어떻게 작성됐을까요? 아니, 그걸 떠나서 위와 같은 Rental 클래스 생성에 대한 테스트를 만들려면 어떻게 해야 할까요? Service로 인해 Rental 클래스는 커플링[3]이 급격히 증가되어 있습니다. 만일 Service가 다른 객체와 커플링이 있는 상황이라면 테스트 코드 작성이 점점 어려워질 겁니다. Mock 객체를 만들 수도 있지만, 앞에서도 말했지만, Mock은 최소한으로만 사용하세요. 이 같은 코드에서는 답이 아닙니다. 만일 Service를 제거해 의존관계를 없앨 수 있다면 어떨까요?

```
class Rental {
    Movie movie;
    Rental(Movie movie) {
        this.movie = movie;
    }
}
```

3 커플링(coupling): 모듈이 서로 의존적인 관계가 되는 정도

이렇게 되면, 아마 테스트 케이스 작성이 훨씬 쉬워질 겁니다.

TDD를 진행할 때, 억지로 테스트를 작성하는 것은 잘못된 접근입니다. 테스트를 작성하기 쉽게 코드를 작성하는 것이 맞습니다. 테스트를 만들기가 너무 어렵다면, 기본적으로 만들어진 코드 설계를 다시 살펴보세요. 많은 뛰어난 개발자들이 'CAN DO' 정신으로, 어떻게든 테스트 케이스를 작성해내기 위해 다양한 트릭을 사용합니다. 대개는 Mock 프레임워크를 대거 사용하거나 자신만의 테스트 유틸리티를 작성하는 식으로 말이죠. 본인은 어깨 으쓱 하는 식으로 자랑스러워할 수 있겠지만, 그렇게 작성된 테스트 케이스를 팀 전체가 끝까지 유지하기는 매우 어렵습니다. 다시 한번 스스로에게 질문해보세요. TDD 시에 테스트 케이스 작성이 어려운 이유가, 단위 테스트 케이스를 작성해야 한다는 사실 때문인가요, 아니면 단위 테스트 케이스를 작성하기 어렵게 프로그램 설계가 되어 있어서인가요?

참고

Breaking the Law of Demeter is Like Looking for a Needle in the Haystack, http://misko.hevery.com/2008/07/18/breaking-the-law-of-demeter-is-like-looking-for-a-needle-in-the-haystack/

Q 저희는 대부분의 코드를 TDD 방식으로 개발했습니다. 그리고 커버리지 100%의 자동화된 테스트도 갖고 있답니다. 자, 이젠 QA팀을 해체해도 되겠죠?

제가 알기로는 아직까지 세상에 버그 없는 프로그램을 만드는 100% 확실한 방법은 존재하지 않습니다. 아마, 그리고 대부분의 개발자는 테스트에 대한 전문적이고도 광범위한 교육과 경험을 갖고 있지 않으리라 생각합니다. TDD는 버그를 없애기 위한 테스트 방법론이 아닙니다. 품질활동은 단순히 자동화된 테스트가 존재하느냐, 얼마나 꼼꼼히 작성됐느냐의 수준을 넘어섭니다. 혹, 평소 미움의 대상이었던 QA팀 제거를 위해 상상할 수 있는 모든 케이스에 대해 미친 듯이 자동화된 테스트를 만드셨다 하더라도 이룰 수는 없는 바램인 것 같습니다. 전문가는 비 전문가가 하지 못하는 일을 하기 때문에 존재합니다.[4] 만일 누군가가 당신에게 특정 분야의 전문가는 더 이상 필요 없다고 이야기한다면, 둘 중 하나일 겁니다. 해당 분야 자체가 사라졌거나, 아니면 당신이 그 방면의 전문가이거나. 부디 잊지 마세요. "약은 약사에게, 품질은 품질전문가에게!"

4 테스팅을 주업으로 하는 QA는 제품의 요구사항에 존재하는 문제점을 찾아주기도 하고, TDD에 의한 단위 테스트가 지향해야 할 테스트 기법이나 방법 등을 제시하기도 합니다. - 베타리더 듬뿍님의 의견

Q 왜 이클립스의 각종 기능과 단축키 등을 사용해야, 혹은 배워야 하나요? 전 울트라 에디터(UltraEditor)나 심지어 때로는 메모장(notepad)으로도 충분히 잘할 수 있다구요! (게다가 TDD랑 무슨 상관?)

IDE[5] 툴에서는 우리가 미처 생각하지 못했던 부분까지도 자동화된 지원을 해줍니다. IDE 툴의 개발이 어렵고, 많은 시간과 자원과 비용이 들어가는 데는 다 이유가 있습니다. 평소에 인식하고 있는지 모르겠지만, 그 비용과 자원과 시간은 바로 개발자 당신의 시간을 줄여주기 위해 사용된 비용과 시간입니다. 매뉴얼을 읽으세요. 유용한 기능이 보이면 메모해두세요. 자주 쓰는 기능은 단축키가 있는지 확인해보세요. 당신이 당신의 소중한 시간을 효율적으로 사용할 수 있도록 준비된 환경이 IDE입니다. IDE를 통해 절약된 시간을, 프로그램에 대해 '생각할 시간'으로 몰아주세요. 결과적으로 코드의 품질을 높일 수 있습니다. 소프트웨어의 품질은 소스 작성에 들어간 시간이 아닌 소스(프로그램 구조)에 대해 생각한 시간만큼 올라가게 되어 있기 때문입니다(그리고 TDD시에 소요되는 지연 시간도 줄일 수 있답니다).

Q 테스트 케이스를 작성할 때 사용하게 되는 외부 모듈이 익숙지 않을 때는 어떻게 해야 하나요?

만일 외부에서 받은 라이브러리이거나 처음 접하는 프레임워크일 경우에는 테스트 케이스를 작성하는 일이 더욱 쉽지 않습니다. 집중해야 하는 업무뿐 아니라, 라이브러리 사용법도 배워나가야 하기 때문입니다. 이럴 경우 테스트 케이스 작성을 두 가지 경우로 생각할 수 있습니다.

Case 1 라이브러리 학습을 위한 테스트 케이스

Case 2 업무 코드 작성을 위한 테스트 케이스

Case 1 같은 경우에는 외부 모듈이 우리가 원하는 형태로 동작하는지 검증하고자 할 때입니다. 신뢰도 측정의 개념으로 작성되는 테스트 케이스라 할 수 있겠습니다.

Case 2의 경우가 우리가 좀 더 집중해보고자 하는 부분인데요, 외부 모듈의 알 수 없

5 통합개발환경(Integrated Development Environment, IDE): 이클립스(eclipse), 넷빈즈(NetBeans), IntelliJ IDEA, WSAD, Oracle JDeveloper 등이 있다.

는 API들이 신경에 거슬리더라도 우선은 배제하고 업무 시나리오에 맞게 테스트 케이스를 작성합니다. 이를테면 구글 캘린더를 이용한 일정 관리 프로그램을 작성해본다고 하겠습니다. 구글 캘린더 API 자체도 익숙하지 않고, 일정 관리 프로그램이라는 모듈도 만들어야 합니다. 일정 관리 프로그램에 대한 테스트 시나리오를 우선 작성해봅니다. 우선은 '일정등록'의 경우를 먼저 예로 들어보겠습니다.

> 일정 관리 프로그램 시작 → 로그인(인증) → 일정 등록 → 등록한 일정의 정상 등록 여부 확인

이 내용을 테스트 케이스로 작성해보면 아래와 같은 형태가 될 겁니다.

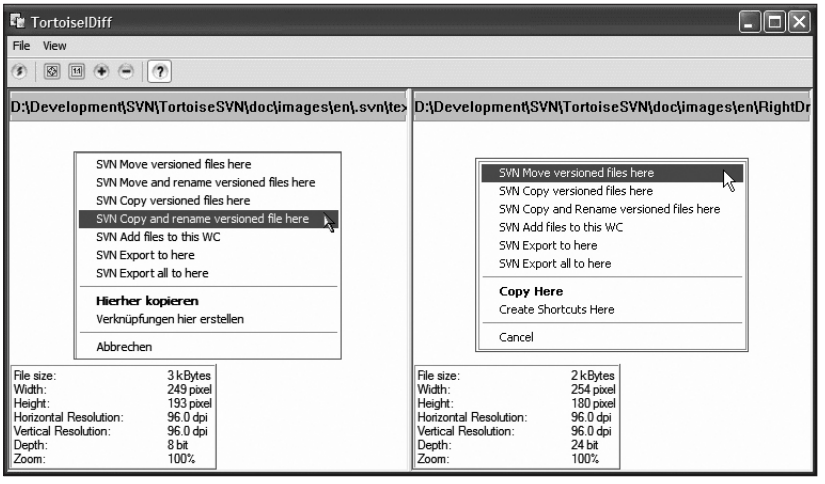
```
@Test
public void testCreateEvent() throws Exception {
    MyCalendar myCalendar = new MyCalendar();
    myCalendar.login();

    Event event = new Event("8pm 동기모임");
    myCalendar.createEvent(event);
    Event searchEvent = myCalendar.search("동기모임");
    assertEquals(event.toString(), searchEvent.toString());
}
```

사실 위 소스에서 작성한 부분들은 굳이 저렇게 자신만의 클래스로 덧씌우지 않더라도 지원해주는 기능들이 모두 있습니다. 그럼에도 구글 API를 하나하나 찾아서 테스트 케이스를 작성하지 않은 것은 구글 API는 단지 재료로 쓸 뿐이라는 생각으로 먼저 위 내용처럼 시나리오 베이스로 테스트 케이스를 먼저 작성했습니다. 이후에 테스트 케이스 완성과 리팩토링을 통해 구조는 바뀔 수 있겠습니다만, 제가 보기엔 이 모습도 괜찮다고 생각합니다.

Q 저희는 화면 UI가 매우 중요합니다. 그중에서 이미지 관련한 부분이 특히 중요합니다. 고정 이미지도 있고, 어도비 플래시를 이용한 움직이는 이미지도 있습니다. 이럴 경우에는 어떻게 TDD를 진행할 수 있나요?

포토샵 등의 이미지 툴을 사용해 목표 이미지를 먼저 만듭니다. 그 다음 실제 동작한

상태에서의 화면 이미지를 캡처한 후 diff를 이용하는 방법이 있습니다. 이미지를 바이트배열로 만들어서 비교하는 방법도 있지만, TortoiseIDiff[6]라는 유틸을 사용하면 좀 더 간단히 일치 여부를 확인할 수 있습니다.



Q private 메소드도 테스트 케이스를 만들어야 하나요?

우선 먼저 'private 메소드가 왜 생기나?' 혹은 '언제 private 메소드를 만들게 되나?'를 생각해볼 필요가 있습니다. Java에서의 private은 접근 범위(access scope)로 봤을 때 오직 자신의 클래스 내에서만 접근 가능한 속성이나 메소드입니다. 즉, 외부에 알리고 싶지 않은 부분이라는 전제하에 만들어집니다. 하지만 클래스의 효용 가치는 그 클래스가 갖고 있는 정보와 그 정보를 기반으로, 다른 클래스와 상호작용한다는 데 있습니다. 이를 모듈화, 독립화라고도 하는데, 다른 클래스 입장에서는 자신이 원하는 기능을 올바로 해주기만 하면 더 이상 신경 쓰지 않아도 되는 겁니다. 그리고 그 외부에 제공하는 기능이 제대로 동작하는지 보장해준다는 의미로 테스트 케이스를 만드는 거죠.

우리는 자동차를 살 때 특정 전선들이 정상 동작하는지, 모터의 회전수가 일정 수준 이상으로 유지되는지를 확인하지 않습니다. 자동차를 이용하는 사용자 측면에서는 헤드

6 tortoisesvn.tigris.org

라이트가 잘 켜지고 꺼지고, 핸들이 정확히 동작하는지를 테스트해보고, 만족하면 되는 겁니다.

이야기가 좀 돌아왔습니다만, 이 문제에 대해서는 많은 사람이 다양하게 논의를 해왔고, 근래에는 private 메소드는 테스트하지 않아도 무방하다는 것이 주류입니다. 그리고 public 메소드를 테스트하면 그에 딸린 private 메소드는 자연스럽게 테스트되었다고 간주합니다. 물론 굳이 private 메소드에 대한 테스트 케이스 수행을 원한다면 리플렉션 등을 이용할 수는 있습니다만, 리플렉션이라는 기술의 특성상 유리처럼 부서지기 쉬운 테스트 코드가 될 가능성이 큽니다.

Q 사용자가 직접 수행하는 수동 테스트와 자동화된 테스트, 굳이 TDD에 한정하지 않고 봤을 때, 어느 것이 더 중요한가요?

TDD의 가치를 크게 두 개만 꼽는다면, 지속적인 설계 개선과 자동화된 테스트라고 할 수 있습니다. 자동화는 우리가 지향해야 하는 목표가 맞습니다. 그런데 자동화는 대부분의 경우에 장점이 되지만 모든 경우에 무조건 그런 건 아닙니다. 우선, 자동화하는 데는 많은 노력이 들어갑니다. 자동화의 비율이 높을수록 비용이 많이 듭니다. 따라서 자동화된 테스트를 작성하고자 할 때는 ROI를 고려해야 합니다. 당장 오픈이 낼 모래인데, 테스트 자동화를 위해 노력하는 건 어리석은 일일 수 있습니다. 하지만 자동화는 확실히 다양한 이점을 제공합니다. 특히 오랜 기간을 놓고 볼 때는 더욱 그렇습니다. 수동 테스트의 단점이 자동화된 테스트의 장점인 경우가 대부분이므로, 반대로 수동 테스트의 단점을 살펴보면 이렇습니다.

첫째, 회귀 테스트가 팀에 큰 부담이 됩니다. 개발자가 기능을 추가/변경할 때마다 매번 기능 테스트를 수행해야 하는데, 누락되거나 지나치게 단순화된 형태로만 테스트를 진행하게 될 가능성이 큽니다.

둘째, 실패/성공, 테스트 커버리지 등 다양한 보고서를 손쉽게 볼 수 없습니다.

결론은 적절한 수준에서의 자동화 테스트를 결정해야 합니다. 때로는 전체 자동화(full automation)가 불가능할 경우가 있습니다. 그렇다고 자동화를 포기하지는 마세요. 수동 테스트를 진행하게 될 때, 수동 테스트를 좀 더 쉽게 만들어주는 도구만으로도 매뉴

얼 테스트의 부하를 줄여줄 수 있다면 적극 사용해야 합니다. 화면 녹화 기능이나 매크로 도구를 바로 그런 경우에 효과적으로 사용할 수 있습니다. 물론 이런 경우에는 1분마다, 메소드 개발마다 수행하기는 어렵겠지요. 그래도 수동으로 하는 경우보다는 간격을 줄여줄 수 있습니다.

Q TDD를 적용하기 어려운 대표적인 이유는 무엇입니까?

TDD는 이름처럼 테스트가 주도적으로 개발을 이끌어나가는 것이 목표입니다.

작성해야 할 기능(ToDo)이 있으면 코드를 작성하기 전에 테스트 케이스를 작성해서 진행하는 것이 일반적인 순서입니다. 그런데 문제는 해야 할 기능이나 기능 리스트가 명확하지 않으면 테스트 코드를 작성하기가 매우 어려워진다는 점입니다. 실제로 필자가 속한 회사 내에서도 TDD가 너무 어렵다며 포기한 사례가 많습니다. 물론 TDD 튜토리얼에서 제시하는 코드들이 있습니다. 하지만 예제 코드의 수준이 현업의 업무 상황이나 레벨과 차이가 나는 경우가 많습니다. 또한 개발자들의 연습이 되어 있지 않다는 점도 큰 이유입니다. 그리고 TDD를 효과적으로 적용하기 위한 체계적인 가이드가 없다는 것도 큰 문제입니다. 어찌어찌 어렵게 테스트 코드를 작성해나가지만, 생각보다 꽤 고통스러운 과정이라고 느끼게 됩니다. 그리고 결과적으로 이를 통해 만들어진 코드들이 어지럽게 펼쳐져서는 한곳으로 모이지 못하고, 즉 TDD가 지향하는 설계의 개선으로 흐름이 이뤄지지 않고, 결국 TDD를 포기해버리는(진행이 붕괴돼버리는, 그래서 전통적인 개발 방식으로 다시 회귀해버리는) 케이스도 적지 않습니다. 또한 일부 사람들은 TDD를 하면, 설계를 하지 않고 테스트 케이스 작성과 리팩토링을 통해 설계를 만들어간다고 너무 강력하게 믿기에 요구사항 그 자체를 ToDo로 보고 무작정 TDD를 시작하는 경우도 있습니다. 그런데 TDD 본래의 지향점은 설계를 하지 않는 데 있진 않습니다. 완벽한 설계가 이뤄진 다음 개발에 들어가는 것이 아니라 설계를 일부 먼저 하고, 그 이후에는 TDD를 통해 불완전한 설계 구조를 지속적으로 개선해나가는 방식을 취합니다. 결과적으로는 고객의 다양한 요구사항을 필요한 수준에서 어렵지 않게 만족시킬 수 있는 더 나은 구조로 설계 모습을 갖게 되는 겁니다.

따라서 선행 설계는 반드시 필요합니다. 다만, 그 깊이와 목표가 전통적인 방법론에서

이야기하는 내용과는 다소 다릅니다. 애자일에서는 설계는 하되, 노력 대비 효용이 높은 '최소 비용 설계'를 지향합니다. 그럼, 여기서 말하는 최소 비용 설계란 무엇일까요? 또는 어떻게 해야 할까요?

설계의 첫 시작은 우선 요구사항을 정련하는 것에서부터 시작해야 합니다. 그 다음에 종이가 됐든, 설계 프로그램이 됐든, 참여한 개발자들이 의견일치(concensus)할 수 있도록, 사용자 스토리를 만족시킬 수 있는 최소한의 개발 구조(설계, 디자인)를 도출하고, 그 구조를 바탕으로 정련된 요구사항에서 ToDo를 발췌해나갑니다. 그 이후 각각의 ToDo에 대응하는 테스트를 하나씩 작성하고 만족시키는 식으로 개발을 진행합니다. 따라서 (정말 안타깝지만) TDD는 개발 언어 사용기술만 갖고서는 원활한 진행이 다소 어려울 수 있습니다(불가능한 건 아니지만, 다시 앞 단계로 돌아가서 설계를 고치는 일이 반복적으로 생기게 됩니다). 요구사항 도출 정련 기법과 기본 설계 개념에 대해서는 미리 어느 정도 이상의 지식이 있어야 합니다. 이러한 사실이 다소 부담이 될 순 있지만, 고무적인 일은, 개발 1년차 개발자와 함께 요구사항을 정련해서 간략한 설계를 도출한 다음 ToDo를 찾아내서 시작할 수 있도록 유도하는 작업을 했던 적이 있습니다. 개발자가 개발에 어느 정도 감이 있었기에, 요구사항 도출과 정련, 그리고 개략적인 설계에 대한 몇 가지 기본 원칙과 방법[7]을 전달해준 것만으로도, 두어 시간 만에 해당 개발자의 개발 방식이 크게 향상될 수 있었습니다. 처음에는 막막해하던 개발자가 ToDo 리스트에서 무엇부터 할지를 선택해 TDD를 진행하며 스스로 어떤 부분이 잘못 돼가는 것 같다는 판단과 함께, 이후 어떻게 진행할 것인지 생각하면서 개발할 수 있었습니다. 다시 말해, 짧은 시간의 도움만으로도 TDD를 할 수 있다는 자신감과 함께 개발 진행 스타일에 대해 어느 정도나마 감을 잡을 수 있게 됐습니다.

Q 이러저러한 노력에도 불구하고 TDD가 잘 안 되고 서툽니다. 어떻게 해야 하나요?

글의 머리부분에도 적었지만, (저를 포함하여) 개발을 시작했던 무렵에 TDD라는 개념

7 소프트웨어 공학은 애자일 실천 기법(agile practice)이 적대시하는 분야가 절대 아니며, 또한 무시될 수 있는 것이 절대 아닙니다. 업계에서 기울인 수십 년의 노력은 충분히 가치가 있으며, 실제 애자일 전문가라 불리는 사람들은 전통적인 소프트웨어 공학에서 말하는 개발 방법들을 사용해봤거나 현재 체득하고 있는 사람들이 대부분입니다. 켄트 벡, 마틴 파울러, 로버트 C. 마틴의 이력을 살펴보세요. UML의 창시자 중 한 명인 그래디 부치(Grady Booch)가 애자일 컨설턴트 활동을 하는 걸 알고 계세요?

조차 들어본 적 없는 사람들에겐 TDD에 적응하기가 결코 쉽지 않을 수 있습니다. 노력을 기울이고, 생각을 하면 할수록, 개발 속도도 저하되고, 테스트가 제대로 되는지도 확신이 없으며, 그리고 무엇보다 테스트 케이스를 만드는 일 자체가 쉽지 않을 수 있다는 걸 발견하게 됩니다. 그럼, 어떻게 해야 할까요? 안타깝습니다만, 뾰족한 방법은 없습니다. '영어에는 왕도가 없다'는 표어가 다소 핑계가 될까요? 아니면, '피아노는 딱 노력한 만큼 칠 수 있다'는 말이 도움이 될까요? 크게 위안은 안 되겠지만, 마찬가지로 TDD를 능숙하게 다루는 데도 시간이 필요합니다. 적응하는 시간, 익숙해지는 시간, 요령이 생기고 능숙해지는 시간 등이 필요합니다. 하지만 많은 사람이 이야기하고 있듯이, 노력이 드는 만큼 TDD가 우리에게 줄 수 있는 가치 또한 적지 않습니다. "우리는 왜 일하는 시간에 비해 이렇게 월급이 쥐꼬리만큼인가?"를 곧잘 외치는 본인의 동료 K군은 가끔 이렇게 말을 하곤 합니다. "품질을 핑계 삼아, 모든 프로젝트에 TDD를 반드시 적용하도록 강제화하는 거야! 어때? 그러면 IT의 진입장벽이 높아져서, 결국 우리의 몸값도 올라가지 않겠어?" 네. 물론 농담으로 하는 이야기입니다. 다음은 TDD가 잘 안 된다고 할 때 제가 종종 권하는 방법입니다.

책상을 비웁니다. 최대한 넓게 만듭니다.

혹시 봐야지 하는 생각으로 위에만 올려져 있는 책이 있다면 치워버리세요. 머릿속에서 그 책의 존재 자체가 사라질 것 같다면, 책 제목과 저자를 적어서 책상 벽면에 붙입니다. 최근에 쓴 적이 없는 필기구, 지금 당장 처리해야 하는 문서가 아니라면 마찬가지로 치웁니다. 기억해야 할 것은 벽면 종이에 적으세요. 책상 위가 깨끗이 정리됐다면, 이제 개발에 필요한 ToDO 리스트를 A4에 적습니다. 컴퓨터에 적을 수도 있는데, 사각거리는 느낌을 줄 수 있는 종이와 마음에 드는 펜을 이용해서, 넓고 깨끗한 흰 종이에 목표와 단계를 적어봅니다. 자, 이제 키보드와 화면과 ToDo에 해당하는 종이. 이 세 가지에만 집중합니다. 진행이 잘 안 될 때, 이런 방식이 제게는 도움이 되었습니다.

함께 합니다. 옆 사람에게 도움을 청하세요. 자신이 하려는 일을 이야기하고 함께 작업해줄 수 있는지를 물어봅니다. 네. 그렇습니다. 짝 프로그래밍을 하는 겁니다. 혹시 동료가 환경, 방식 등을 낯설어한다면 몇 분 정도의 시간을 투자해서 '규칙'을 알려주세요. 그리 오래 걸리지 않을 겁니다.

Q 저는 SI 프로젝트 위주로 일하고 있습니다. SI에서 TDD를 적용하는 것이 바람직할까요?

굉장히 어려운 질문입니다. "SI에 애자일 도입이 적절한가요?" 수준까지는 아니더라도 만만치 않은 수준의 논쟁과 메아리를 만들 수 있는 질문이라 생각합니다. SI는 전통적으로 개발과 유지보수가 분리되어 있습니다. 그리고 대개, 개발은 납기일자와 비용이 미리 정해져 있습니다. 앞에서도 말했지만 TDD(혹은 단위 테스트 케이스)는 해당 방법을 유지해나가기 위해 추가적인 비용이 발생합니다. TDD를 사용한다고 해도 개발 기간이 드라마틱하게 줄어드는 일은 발생하지 않으며 오히려 일정이 지연되는 경우도 종종 발생합니다. (네? 뭐라고요?) 프로젝트를 진행하는 사람의 입장에서 보면, 개발의 지연은 곧 프로젝트의 실패를 의미할 수 있습니다. 그런데 말입니다. 프로젝트는 단순히 기간 내에 고객이 원하는 기능이 구현완료됐는지만으로 평가받아야 하는 걸까요? 프로젝트가 끝나고 혹은 프로젝트 진행 도중에 참여했던 사람들이 해당 프로젝트에 환멸을 느끼고 떠난다든가, 이런 프로젝트는 다시는 참여하고 싶지 않다든가, 인간적인 존중을 받지 못한 채 일했다고 느낀다든가, 혹은 스스로의 성장감 없이 그저 사용됐다는 느낌을 받았다면, 그래도 그 프로젝트는 성공적인 프로젝트라고 할 수 있을까요?

그럼 TDD가 줄 수 있는 건 무엇이지 다시 생각해볼 필요가 있습니다. TDD는 제품(서비스)의 품질을 증가시킵니다. 품질이라고 하면 다소 추상적이고 넓은 의미로 인해 오히려 모호해질 수 있습니다만, 제품 자체의 오류 감소, 변경 비용 감소 같은 비용 감소 효과를 가집니다.

국내 SI에서 넓게 사용되고 있는 X-internet 기반의 제품을 사용할 경우 View 영역의 TDD를 적용하기가 어려울 수 있습니다. 독자적인 프레임워크를 사용하는 그룹에서도 마찬가지로 TDD를 적용하기 어려울 수 있습니다. 하지만 그럼에도 TDD가 줄 수 있는 장점이 크기 때문에 많은 이들이 해당 한계를 넘어설 방법을 모색하기 위해 노력하고 있습니다. 마치 외국에서는 미친 듯이 TDD를 적용할 것만 같지만, 그렇지 않습니다. 하지만 장점에 대해서는 국내보다는 더 많이 알려져 있지 않을까 생각합니다.

참고

Is unit testing doomed? http://www.javaworld.com/javaworld/jw-08-2008/jw-08-unit-testing-doomed.html

Q TDD는 개발 기법으로 봐야 하나요? 아니면, 테스팅 기법으로 봐야 할까요?

XP의 창시자이며 『Test Driven Development by example』의 저자이기도 한 켄트 벡은 TDD와 동일한 개념인 Test First Development에 대해 "테스트 우선 코딩(Test First Coding)은 테스팅 기법이 아니다"라고 말했습니다. 실제로도 TDD로 개발을 하다 보면 테스트를 작성함으로써 클래스 간의 결합도, 중복, 인터페이스 등 디자인 영역에 대해 좀 더 고민하게 됩니다. 따라서 TDD는 문제영역에 대한 분석과 소프트웨어 설계에 대한 영역을 개선하는 실용 디자인 기법에 가깝습니다. 하지만 테스팅 기법(균등분할, 경계값 분선, 탐색적 테스팅 등)을 학습할 수 있으시면, 테스트 케이스를 좀 더 효율적이고 코드의 안전성을 높일 수 있는 형태로 작성할 수 있게 됩니다.

Q TDD를 하면 TDD를 하지 않은 경우에 비해 확실히 품질도 좋아지고, 개발도 빨라지는 거, 맞죠?

핵심을 바로 찌르는 좀 미묘한 질문인 것 같습니다. TDD를 사용하면 확실히 결함이 적은 우수한 품질의 소프트웨어를 만들어낼 수 있고, 지속적으로 개선할 수 있는 바탕을 마련해준다는 데는 대부분의 참여자가 동의를 합니다. 하지만 수치적인 %를 지칭해서 "얼마만큼 확실히 개선됩니다"라고 말하기는 다소 어려운 게 사실입니다. 대부분의 TDD 사용 그룹은 효과를 봤다고 말을 합니다만, 일각에서는 "TDD를 사용해 소프트웨어를 개발했다는 것 자체가, 그 개발팀의 열의와 각오, 수준이 일정 수준 이상이라는 뜻이기 때문에 일반 평범한 개발팀과 비교하는 건 다소 불합리하지 않느냐?"고 말하기도 합니다.

또한 TDD를 사용했다고 말하는 그룹들의 대부분은 개발 진척이 다소 늦어지는 것 같다고 하기도 합니다. 이유는 테스트 테이스를 작성하고 유지하는 데 노력이 들어가기 때문입니다. 숙련된 개발자를 놓고 봤을 때, 약 20% 정도 가까이 진척이 늦어질 수 있습니다. 그리고 다소 논란의 여지는 있지만 TDD로 개발할 때의 결함률과 진척도에 관련해서는 일련의 연구 결과가 공개되어 있습니다(Realizing quality improvement through test driven development: results and experiences of four industrial teams, 2008, Pankaj Jalote).

Q 테스트 케이스를 최대한 정교하게 작성하려고 노력하고 있습니다. 그런데, 그러다 보니 테스트 대상 객체나 코드가 조금만 변경이 일어나도 테스트가 와장창 깨져서 유지보수하는 데 비용이 많이 듭니다. 이젠 솔직히 TDD에 대한 회의가 들 정도에요. 뭐가 잘못된 걸까요?

민감한 센서일수록 오탐(false positive)률이 더 높은 게 보통입니다. 사실, 질문하신 내용이 TDD의 어떤 한계점이라고 이야기하는 사람들도 있습니다. 아쉽지만, 뾰족한 답은 없습니다. 다만, 모든 테스트 케이스를 다 정교히 작성하기보다는, 중요하거나 민감하게 반응해야 하는 모듈에 대해서만 정교한 테스트 케이스를 작성하는 것도 하나의 방법입니다. 정확히 지키긴 어렵지만, '딱, 필요한 만큼만 테스트한다'는 원칙을 세워보면 어떨까요?

Q 면접 시에 TDD에 대해 물어보는 건 어떻게 생각하시나요?

만일 제가 면접관이라면 "TDD를 해본 적이 있습니까? 있다면 장점과 단점은 무엇이라고 생각합니까?"라고 물어볼 것 같습니다. 단순히 "관심은 있습니다"보다는 "적용해본 적이 있습니다" 쪽이 실력은 둘째 치고라도 개발에 대해 더 열의가 있는 편이라고 생각할 수 있을 것 같습니다. 그리고 TDD를 안다고 말한다면, 꼭 TDD의 단점에 대해 물어보세요. 대답은 뭐가 되어도 괜찮습니다. 단점을 이야기하는 게 장점을 물어보는 것보다는 더 면접자의 진실을 알려줄 수 있거든요.

Q 팀에 TDD를 도입하고자 할 때 초반에 TDD 외에 다른 어떤 활동을 같이 하면 좋을까요?

TDD를 도입하면 초반에 개발자들이 익숙해질 때까지는 다 함께 코드 리뷰를 진행할 것을 권장합니다. 작성 시 유의점과 어려움에 대해 함께 이야기하는 것으로 팀원들이 TDD를 진행하는 데 어려움을 줄여줄 수 있습니다.

또한 새로운 기능을 추가하거나 버그를 찾을 때 테스트 케이스를 먼저 만들도록 어느 정도 강제화할 필요가 있습니다. TDD는 습관이 중요하므로, 한두 번은 넘어가는 식으로 체계가 무너지면 곤란합니다. 어느 정도 TDD가 익숙해지면 CI 서버(지속적인 통합 서버) 등을 함께 사용할 것을 권장합니다.

Q 팀에서 여전히 JDK 1.4 버전을 쓰고 있어서 짜증이 나요. JUnit 4는 저 하늘 멀리 있고요, 스프링의 최신 버전 기능도 못 쓰고, Google Guice[8]도 못 쓰고, 심지어 FindBugs 최신 버전도 전혀 쓸 수가 없어요. 그런데도 이 인간들은 "꼭 Java 5를 써야 해?"라든가, "기존 시스템이 JDK 1.4를 쓰는데 같이 쓸려면 어쩔 수 없어. 1.4 쓰자!"라고 말합니다. 제가 보기엔 다 핑계이고요, Java 5의 신기능(맙소사! 이미 썬에서는 Java 5의 서비스 지원종료 전환기간에 돌입했다구요!)을 공부하기 귀찮아서인 게 뻔하다구요. 정말이지 개발할 의욕이 안 난다니깐요(JUnit 4 쓰고 싶단 말이에요). 여기까지는 넋두리고요, 이제 진짜 질문입니다. 팀에서 별로 달가워하지 않는데도 굳이 TDD를 쓰자고 계속 주장해야 할까요?

우선 흥분을 가라앉히세요. 심경은 충분히 이해합니다. 하지만 이런 일로 개발에 대한 의욕마저 접는 건 안 됩니다. 좋은 환경에서 좋은 도구로 소프트웨어를 개발하는 건, 열악한 환경과 제한된 자원에서 개발하는 것보다 여러모로 좋은 점이 많은 건 사실입니다. 업무 효율이나 품질이 좋아질 가능성도 높고요. 하지만 어쩌면 자신의 진짜 실력을 가늠해볼 수 있는 건 좀 더 열악한 환경하에서일 수도 있습니다. JUnit 4의 @BeforeClass, @AfterClass를 못쓰는 상황에서 같은 동작을 하려면 어떻게 해야 할까? JDK 1.4이기 때문에 테스트 케이스를 작성하지 않는다든가 하는 이야기는 어쩌면 핑계일 수 있습니다. 어느 정도 개발을 한 개발자임에도 이클립스 같은 IDE가 없는 서버 환경에서는 Java 파일을 제대로 컴파일조차 하지 못하는 모습을 보고 충격받으면서도, 한편으론 이해도 되는 상황인 적이 있습니다.

Toad나 PL/SQL developer나 오렌지[9] 같은 소프트웨어가 없으면, DB 서버에 접속조차 못하는 개발자들을 적지 않게 봤습니다. 툴이 제한되면 좀 더 원활한 활동에 제약이 있을 순 있지만, 그렇다고 해당 업무를 못한다는 건 핑계라고 생각합니다. 안타깝지만, 당신을 위한 완벽한 개발환경은 주어지지 않을 수 있습니다. 기간이 부족하거나 사무실 자리가 비좁거나, 냉난방이 엉망이거나, 돈을 조금 주거나, 사용하는 라이브러리가 맘에 안 들거나, 생각 없는 관리자가 당신을 전생의 원수로 보는 건지 이유 없이 괴롭힌다거나, 자바 전문가인데 닷넷을 하란 소릴 하지 않나… 매번 못하는 이유를 댈

8 구글 엔지니어가 만든 의존성 주입(dependency injection) 프레임워크

9 국산 DB 관리 / SQL 개발 툴

수는 없습니다. 할 수 없는 이유가 반복된다면, 그건 '핑계'라 불려야 하는 게 맞는 것일 수 있습니다.

현재 상황에서 할 수 있는 최선을 하세요. 하지만 설득해서 안 된다고 싸우진 마세요. 원하는 대로 되지 않는다고 상대방에 대해 안 좋은 감정을 가질 필요는 없습니다. 당신이 개발에 열정을 갖는 것처럼, 다른 사람은 다른 일에 열정을 가질 수 있기 때문입니다. 다시 한번 말하지만 TDD는 혼자 하는 개발이 아닌 '팀' 개발에서 빛을 발하는 기법입니다. 당신이 당신 옆 사람을 미워하거나 폄하하는 순간 당신은 팀을 부수는 일을 하는 겁니다.

가랑비에 옷 젖듯이 물들이세요. 당신이 모범을 보이고, 다른 사람이 따르고자 하는 의욕이 충분히 생길 정도로 옆 사람을 도와주세요. 공부하지 않는다 비난하기 전에, 당신이 고생해서 얻은 지식을 옆 사람이 쉽게 받아들일 수 있도록 기회를 주세요. 어쩌면, 옆 사람들도 배우고 싶지만 손을 먼저 내밀기가 부끄러워서 스스로에게 필요 없다고 되뇌이고 있는 상황일 수도 있으니까요.

Q void 메소드를 테스트 가능하게 만들려면 어떻게 해야 할까요?

우선 void 메소드가 언제 사용되는지에 대해 먼저 고민해봐야 합니다. 우리는 흔히 리턴값이 없는 메소드를 void라고 생각하고 개발을 하는 경우가 많지만, 엄밀히 따지면 이렇습니다.

- 리턴값이 있는 메소드: 기능(function)
- void 메소드: 절차(procedure)

메소드의 리턴값이 없다는 의미는 로직의 절차를 기술했거나, 프로그램 로직 트리의 맨 끝단에 해당하는 기능 위임 작업(이를테면 화면 출력, 데이터 등록 같은 작업)을 목표로 하겠다는 의미입니다. 즉, 해당 void 메소드의 이름이 의미하는 작업을 그 메소드에서 끝을 내겠다는 뜻입니다. 이런 메소드는 Mock 객체를 설명했던 부분에서 이야기했던, 행위 기반 테스트를 해야 합니다. 가장 간단한 방법은, 상태(state)를 확인할 수

있는 다른 메소드로 결과를 테스트하는 것입니다. 만일 상태를 확인할 인터페이스[10]를 제공받지 못하는 상황이라면, 리플렉션 같은 방법을 사용할 수도 있습니다. 하지만 그다지 권장하진 않습니다. 누누히 이야기 하지만 리플렉션으로 작성된 테스트 코드는 특히나 깨지기 쉬운 테스트가 돼버리곤 하기 때문입니다. 가급적이면 상태를 확인할 수 있는 isClicked(), isReady() 같은 상태확인 인터페이스를 만들어주세요. 아니면 Mock 프레임워크를 사용할 수도 있습니다. 관련해서는 4.1절 'Mock 객체'를 참고해주세요. 다시 한번 강조하지만, 테스트 가능한 설계가 곧 좋은 설계라는 걸 잊지 말아주세요.

Q 보통 이야기할 때 TDD와 단위 테스트를 혼용해서 쓰는데, 둘이 같은 거 맞죠?

엄밀히 말하면 TDD는 개발 방식을 의미합니다. 그리고 단위 테스트는 '자동화된 테스트 케이스'를 말합니다. 보통 단위 테스트는 TDD로 개발한 작업의 부산물이 됩니다. 따라서 굳이 TDD를 하지 않더라도 단위 테스트를 만들 수 있습니다. 하지만 이미 만들어져 있는 소스에 대한 단위 테스트 작성은 곧잘 귀찮고 지루한 작업이 되곤 합니다. 그리고 현재 잘 동작하는 코드에 대한 테스트 코드를 만들려고 하거나, 리팩토링을 통해 테스트가 가능한 코드로 만들어가는 작업을 하겠다고 고객에게 이야기했을 때도 대부분의 경우 일정상의 문제로 인해 긍정적인 반응을 얻기가 쉽지 않습니다. "요즘 한가한가 보이? 다른 할 일이 그렇게 없어? 일 좀 줄까?" 같은 반응도 요원한 일만은 아닙니다. 그렇기에 차라리, 할 수만 있다면 개발 초기부터 TDD를 통해 단위 테스트용 테스트 케이스를 작성하는 습관을 갖는 것이 큰 도움이 됩니다.

Q TDD로 개발을 하려면, 제가 사용하는 언어에 맞는 테스트 프레임워크를 먼저 찾는 게 급선무겠죠?

TDD 이야기를 하면서 일면 조심스러운 부분이 있는데, 그건 바로 TDD와 JUnit 같은 xUnit 테스트 프레임워크를 혼동하기 쉽다는 점입니다. TDD는 개발 방식이고, 그걸 도와주는 도구가 xUnit 테스트 프레임워크입니다. 따라서 특정 프레임워크를 사용했느냐 하지 않았느냐가 TDD 방식으로 개발됐느냐, 그렇지 않느냐를 판단하는 건 맞지 않

10 자바의 interface 키워드를 의미하는 것이 아니라, 객체의 상태에 접근할 수 있는 메소드나 객체 등을 이야기합니다.

습니다. 결과치를 미리 예상하고 그걸 자동화된 형태로 판별해낼 수 있으며, 빠른 주기 내에서 반복 실행이 가능하다면 어떤 모양이 됐든, 마찬가지로 TDD를 사용했다고 말해도 좋습니다.

Q 저희 팀은 테스트 커버리지 100% 도달이 목표입니다만, 커버리지 100% 달성은 좀 처럼 이뤄지지가 않습니다. 어떻게 해야 할까요?

테스트 커버리지는 기본적으로 두 가지 의미를 갖습니다.

첫째, 커버리지 비율이 낮을수록 테스트 케이스가 부족하게 작성되어 있다. 따라서 추가적인 테스트 케이스를 작성할 필요가 있다.

둘째, 커버리지 100%가 곧 버그 박멸(bug free)를 의미하지 않는다. 적어도 한 번은 해당 로직을 흐름에 맞추어 테스트 케이스로 진행해봤다는 것, 그 이상의 의미를 갖진 못합니다.

일반적으로 테스트 커버리지 100%는 달성에 많은 시간과 에너지를 소비하게 만듭니다. 2:8의 법칙이라고 했던가요? 마지막 2를 채우기 위해 8의 노력이 필요하곤 합니다. 또한 그 노력을 기울여 100을 채웠어도 여전히 버그는 존재할 수 있습니다. 따라서 테스트 커버리지를 TDD로 작업할 때 사용하는 안전 그물의 간격이라고 보면 좋을 것 같습니다. 촘촘할수록 좋지만, 그렇다고 무리하진 마세요. 해당 프로그램의 특성, 도메인의 성격에 맞는, 그리고 작업하는 팀에 맞는 테스트 커버리지 비율을 찾으세요. 앞서도 말했지만, 자동화된 테스트 케이스는 TDD의 부산물이자, 리팩토링을 위한 안전장치에 더 가깝습니다. 그 자체를 순수 목적으로 삼게 되면, 더 이상 TDD의 영역이 아니게 됩니다. 그건, 테스트의 영역이 된답니다.

Q 저희는 옆 팀처럼 바보같이 테스트 커버리지 100%에 도전하고 있지 않습니다. 저희 팀은 큰 부담 없이 작성할 수 있는 비율이 80%라는 걸 알고 살짝 도전적으로 85%를 목표로 잡았습니다. 잘하고 있는 거겠죠?

아이러니하게도, 단순히 커버리지만으로는 잘하고 있다고 판단하기가 어렵습니다. 이를테면, 커버리지 82%와 89%의 차이는 어떤 걸까요? 82%는 나쁘고, 89%는 좋

은 걸까요? 테스트 커버리지는 단순히 수치만으로 판단하기엔 곤란한 부분이 존재합니다. 오히려 분기 커버리지(Branch Coverage)[11]가 더 중요할 수도 있습니다. 예외(exception)도 사실 분기 커버리지에 들어갑니다. 하지만 실제로 일어나기 어려운 예외일 경우, 단순히 분기 커버리지 안에 포함시키기 어렵곤 합니다. 따라서 좀 더 복합적이고 전략적인 커버리지율이 필요할 수 있습니다.

기본적인 프로그램 목표 커버리지, 예외로 인한 Branch 테스트 여부, 일반 조건 분기문 테스트 여부 등을 따져서 가이드해야 합니다. TDD를 만든 켄트 벡은 이렇게 말했다고 합니다. "테스트를 어느 정도까지 작성하면 될까요? 불안이 지겨움이 될 때까지 하면 됩니다."

Q TDD로 개발을 진행해나간다고 하면, UML은 언제 쓰나요?

이 질문에 대한 의견은 『UML과 패턴의 적용(Applying UML and Patterns)』(크레이그 라만, 프렌티스홀)이라는 책에 나오는 문구로 대신합니다.

> "소프트웨어 개발에 있어서 가장 중요한 설계 도구는 UML이 아니며 어떤 특정 기술을 말하는 것도 아니다. 다름 아닌, 설계 원칙에 따라 잘 교육된 마음가짐이다. (중략) UML의 경우 핵심사항은 UML 자체가 아니라 시각적 모델링을 하는 것이다. 즉, 글자로 이뤄진 것보다는 좀 더 시각적으로 알아볼 수 있게 해주는 언어를 사용하는 것이다."

Q 전 C++ 개발자인데요, 추천할 만한 테스트 프레임워크는 없나요?

우선 전 C++ 개발자가 아닙니다. 하지만 C++ 개발자로부터 추천을 받은 사이트가 있습니다. iPhone 인디 개발자인 Noel이라는 사람이 자신의 블로그에 대표적인 C++ 단위 테스트 프레임워크의 장단점을 비교해 적은 글이 있습니다. 제목은 'Exploring the C++ Unit Testing Framework Jungle'입니다. 간략하게 정리하자면, 쓰기 편하고 간단한 프레임워크로 CxxUnit을 추천하고 있습니다. 그리고 CppUnit은 가장 인지도도 높고 많은 기능이 준비되어 있는 프레임워크라고 칭하며, 정말 간편하게 쓰

11 if 문 같은 조건절에 의해 로직이 분기되는 경우 각각에 대한 테스트 케이스 작성 여부를 의미

고 싶다면 CppUnitLite를 쓰길 권장하고 있습니다. 필요하다면, 제목으로 검색창에서 찾아보시길 권장합니다. 또한 함께 쓸 Mock 프레임워크로는 구글에서 발표한 GoogleMock(googlemock.googlecode.com)이 있습니다. jMock과 Hamcrest 개념까지 들어가 있는 편리한 Mock 프레임워크입니다.

FAQ 정리

혹시라도 TDD가 어떤 결정적인 해법이라는 생각이 들었다면 필자가 설명을 잘 못했기 때문일 겁니다. TDD는 소프트웨어 개발에 있어 결코 절대적인 해법은 아닙니다. 하지만 소프트웨어 개발 전쟁에서 사용할 수 있는 매우 유용한 무기입니다. 전쟁이란, 총으로만 싸울 수도 없고 미사일이나 탱크로만으로 싸울 수도 없습니다. 이길려면 가용한 자원과 기술이 다양하고 적절한 곳에 사용할 수 있어야 하고, 그럴 능력이 돼야 합니다. 그러면, 그렇지 않은 경우보다 성공 확률이 더 올라갑니다. TDD도 마찬가지입니다. TDD는 개인에게나 팀에게나 매우 유용한 기술이고, 보탬이 되는 기법입니다. 그뿐 아니라 배워놓으면 꼭 보답해주는 몇 안 되는 기본 기술 중 하나입니다.

앞에서 살펴본 FAQ 외에, 기타 궁금한 사항이나 질문이 있으면 tddbook@gmail.com으로 메일 주세요. 바로 대답해드릴 수 있는 내용은 바로 답변해드리고, 대답이 어려운 내용은 찾아서 알려드리도록 하겠습니다.

노력한다고 모두 성공하는 건 아니지만, 성공한 사람 중 노력하지 않은 사람은 없다고들 하죠. TDD를 겁내기보다는 한번 몸을 바쳐 뛰어들어 보면 어떨까 싶습니다. 물에 반만 몸을 담구고 뜨지 않더라 말하지 말고, 한번 흠뻑 젖을 정도로 뛰어들어 볼 필요가 있습니다. 그럼, 개인적으로 좋아하는 문구를 마지막으로 FAQ를 마무리하겠습니다.

"우물쭈물하다가 내 이럴 줄 알았다." – 버나드 쇼(George Bernard Shaw)의 묘비명에서 발췌