실패는 좀 더 현명하게 다시 시작할 수 있는
기회다.

– 헨리 포드

CHAPTER 3

TDD 좀 더 잘하기

3장에서는 더 나은 TDD를 위해 필요한 요소들을 살펴본다. 테스트 케이스 클래스들의 위치를 어떻게 잡으면 좋을지에 대해 살펴보고, 테스트 메소드는 보통 어떻게 작성되고, 또 어떻게 작성하는 것이 좋을지 살펴본다. 그리고 TDD를 위해 만들어지는 테스트 케이스 작성은 어떤 식으로 해야 할지 테스트 케이스 접근 방식을 알아본 다음, TDD의 한계점은 무엇이며, 그에 어떤 식으로 대처하는지를 살펴본다.

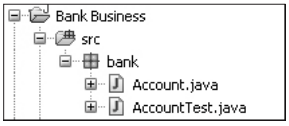
3장 체크리스트

- ☐ 테스트 클래스의 위치
- ☐ 테스트 메소드 작성 방식
- ☐ 테스트 케이스 접근 방식
- ☐ TDD의 한계

3.1 테스트 케이스 클래스의 위치

단위 테스트 케이스를 소스 구조 안에서 어디에 놓을 건지 결정해야 한다. 사용 가능한 몇 가지 방식이 있다. 어떤 방식이 있고, 각각의 장단점은 무엇인지 살펴보자.

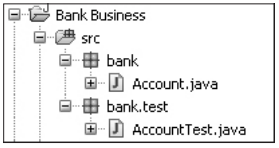
#1. 테스트 대상 소스와 테스트 클래스를 같은 곳에



테스트 클래스와 같은 패키지 내에 놓는, 가장 기본적인 형태다. 간단하게 테스트를 만들 때 외에는 잘 사용되지 않는다.

장점	클래스 이름 규칙을 잘 정했을 경우 테스트 클래스를 찾기가 쉽다.
단점	패키지 내에 제품 코드 클래스와 테스트 클래스가 함께 존재하기 때문에 혼란을 줄 수 있다. 배포 시에 테스트 클래스만 발췌해야 하는 불편함이 있다.
선호도	아주 낮음.

#2. 테스트 클래스는 하위 패키지로



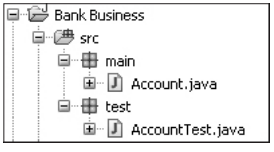
대상 소스의 하위에 테스트 패키지를 만드는 방식이다.

```
package bank;

public class Account {
    private int balance;

    public Account(int money)
…
…
```

```
package bank.test;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import bank.Account;
public class AccountTest {
…
…
```

장점	테스트 코드가 가까운 곳에 위치한다.
단점	이미 하위 패키지가 존재할 경우에는 혼란스러울 수 있다. 테스트 클래스와 제품 클래스를 따로 분리해서 배포하는 데 어려움이 있다.
선호도	낮음. 지금은 거의 사용되지 않는다.

#3. 최상위 패키지를 분리

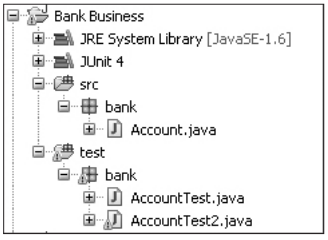


흔히 많이 사용하는 방식으로, 최상위 패키지부터 업무 코드와 테스트 코드를 분리해서 작성할 수 있게 만들어준다.

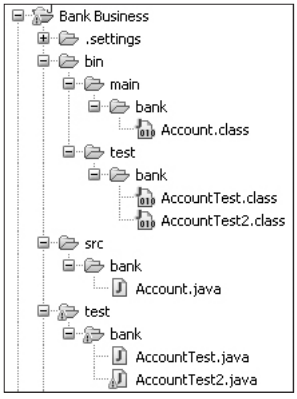
장점	업무 코드와 테스트 코드가 섞일 염려가 없다. 테스트 코드만 분리해내기 쉽다.
단점	default나 protected로 선언된 메소드들에 대해서는 테스트 코드를 작성할 수 없다. 어렵진 않지만 어쨌든 운영환경 배포 시나 제품 패키징 시에 test 패키지 이하를 따로 발췌해내야 하는 불편함이 있다.
선호도	보통

추천

#4. 소스 폴더는 다르게, 패키지는 동일, 컴파일된 클래스는 각각 다른 곳으로



패키지 구조



Navigator로 살펴본 실제 폴더 구조

대상 코드

```
package bank;

public class Account {
    private int balance;
…
}
```

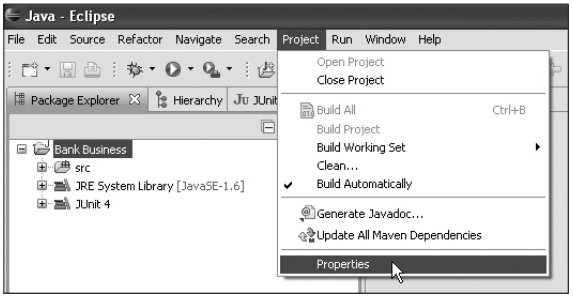
테스트 클래스 코드

```
package bank;

public class AccountTest {
    private Account account;
…
}
```

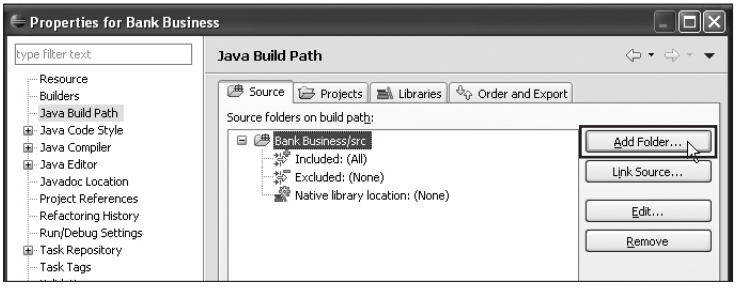
장점	대상 클래스와 테스트 클래스를 동일한 패키지로 선언할 수 있다. 접근 범위(scope) default나 protected로 선언된 메소드도 테스트 케이스로 작성할 수 있다. 컴파일된 대상 클래스와 테스트 클래스의 위치가 최상위 폴더부터 다르게 만들어지기 때문에, 서로 섞일 염려가 없다. 배포 시에도 분리가 쉽고 빠르게 패키징이 가능하다.
단점	환경 구성 방법이 Eclipse IDE에 다소 의존적이다.
선호도	**매우 높음.** 대부분의 사람에게 가장 권장하는 스타일이다.

환경 구성 방법

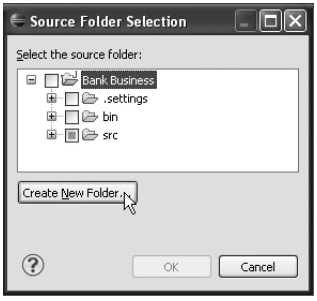
01 대상 프로젝트에서 Properties(속성값)를 선택한다.



02 프로젝트 설정창에서 Java Build Path 항목을 선택한 다음 Source 탭의 Add Folder 버튼을 누른다.

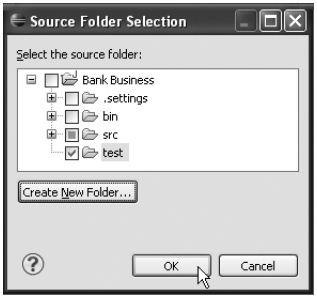


03 프로젝트 이름(현재는 Bank Business)을 선택하고 Create New Folder(새 폴더 만들기)를 누른다.

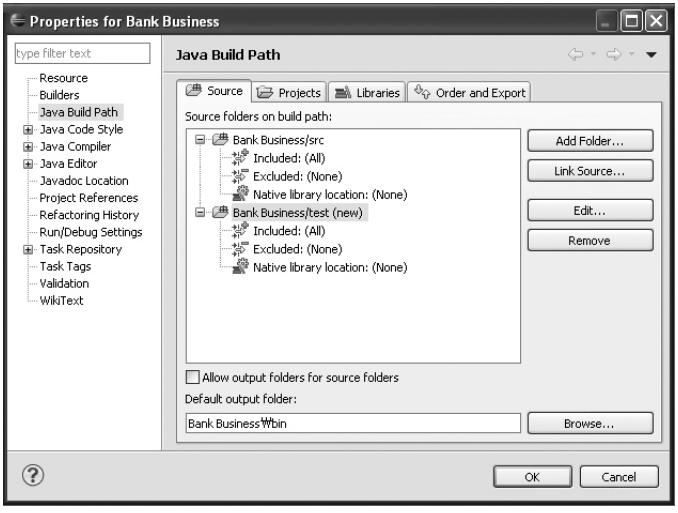


04 테스트 클래스들이 위치할 폴더 이름을 정한다.

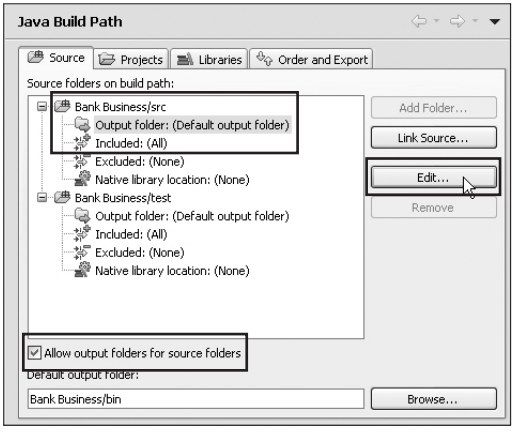
05 OK를 선택한다.



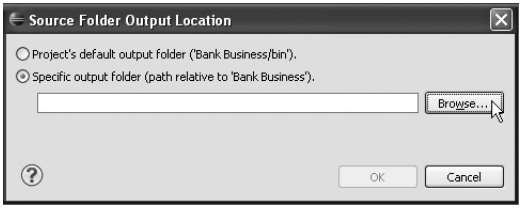
06 이제, 위에서 만든 test 폴더를 소스 폴더로 지정해보자. 아래 화면에서 OK를 누른다.



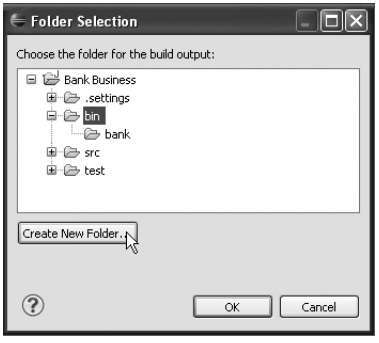
07 각각 다른 곳으로 컴파일된 파일이 위치할 수 있도록, 'Allow output folders for source folders(소스 폴더들에 대해 출력 폴더를 지정할 수 있도록 허용)'을 선택한다. 'Allow output folders for source folders'를 선택하면, 이전에는 보이지 않았던 Output 폴더를 지정할 수 있는 항목이 나타난다. src 폴더의 Ouput 폴더가 Default로 되어 있는 것이 보인다. 선택한 다음 Edit 버튼을 눌러서 변경해보자.



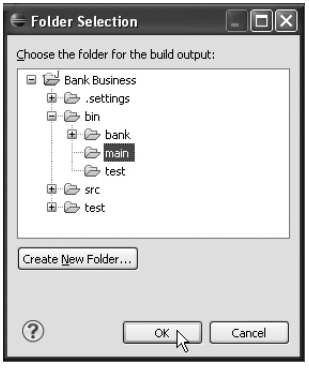
08 아래 'Specific output folder(출력물 폴더 지정)' 라디오 버튼을 선택한 다음 Browse(찾아보기) 버튼을 누른다.



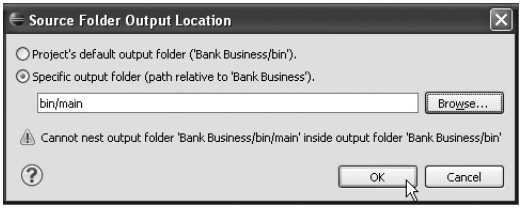
09 기존의 기본 ouput 폴더인 bin을 선택하고 'Create New Folder(새 폴더 생성)'을 눌러서 bin 하위에 main과 test 폴더를 각각 만들자.



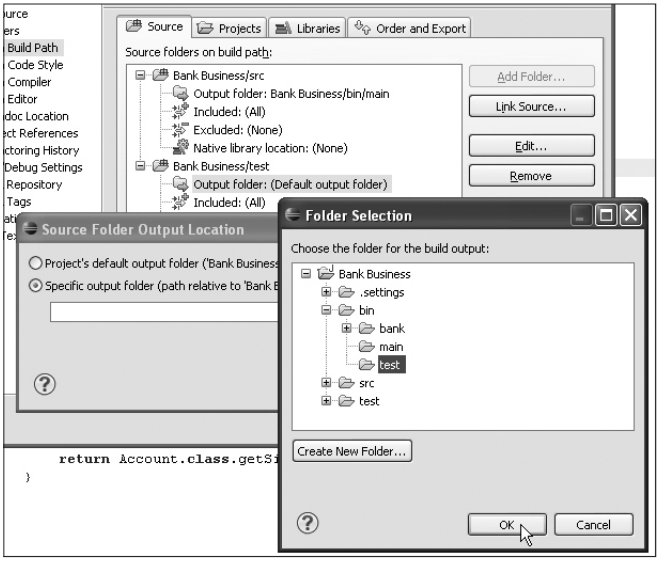
10 우선 테스트 대상 클래스가 컴파일될 src 폴더 쪽이니까 main을 선택한 다음 OK 버튼을 누른다.



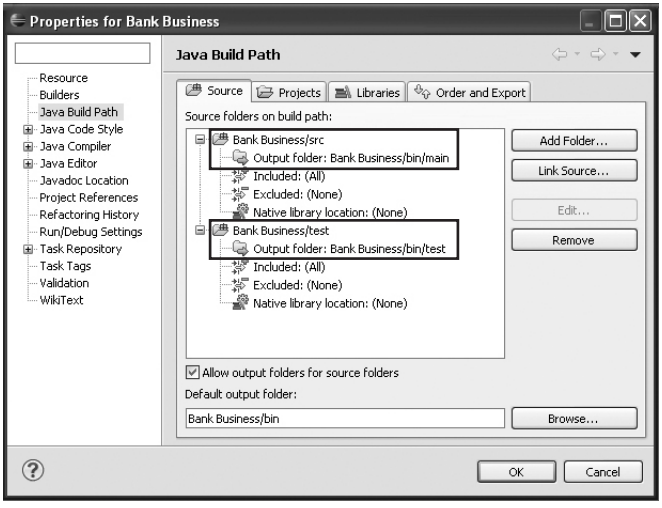
11 Output Location을 변경할 때 경고가 표시될 수 있지만, test 폴더의 output까지 변경하면 해결되니까 우선은 무시한다.



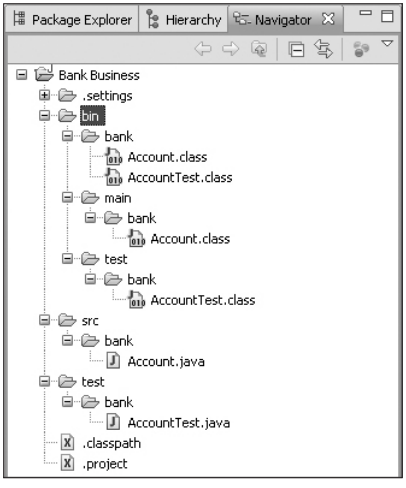
12 마찬가지로 test 소스의 ouput 위치를 test로 선택한 다음 OK를 누른다.



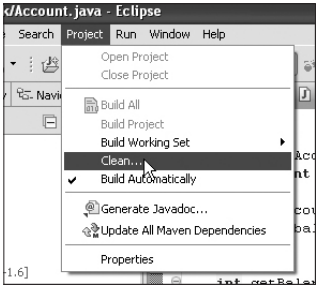
13 최종적으로 다음과 같은 모습이 되면 정상이다.



14 Navigator 뷰를 선택해서 변경된 위치로 클래스 파일들이 정상적으로 생성됐는지 확인한다.

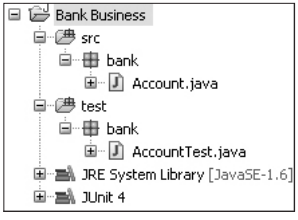


15 만일 정상적으로 생성되지 않았다면, 이클립스 메뉴에서 [Project] → [Clean]을 선택한다.

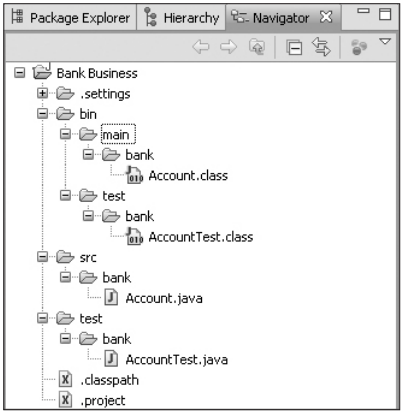


이클립스 프로젝트 메뉴에서 Clean을 선택

16 자, 정리를 위해 bin 폴더 아래에 컴파일 파일이 쌓였던 기존 bank 폴더를 지운다. 최종적으로 만들어진 구조는 다음과 같다.

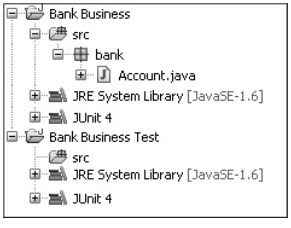


Package Explorer에서 보이는 모습



Navigator Explorer에서 보이는 모습

#5. 테스트를 프로젝트로 분리

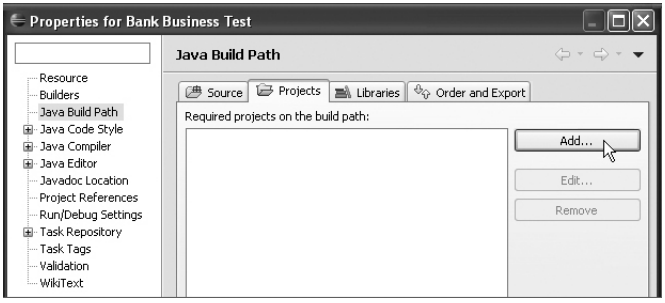


아무리 폴더를 변경한다 하더라도 동일 프로젝트 내에 제품 코드 클래스와 테스트 클래스가 함께 존재하면, 각자 사용하는 라이브러리를 클래스패스로 공유하게 된다. 이렇게 되면 제품 코드 동작을 위해 필요한 외부 라이브러리와 테스트를 위해 필요한 외부 라이브러리를 구별해내기 어려워진다. 이럴 경우 테스트 수행 시에만 필요한 라이브러리들을 제품 배포 시에 일일이 구분해줘야 하는 불편함이 발생한다(아니면, 업무 코드에서는 불필요함에도 함께 배포되기도 한다). 따라서 이런 외부 라이브러리 의존관계를 좀 더 확실하게 구분하고자 할 때는, 프로젝트 단위로 분리해놓기도 한다. 이클립스 플러그인 개발이나 안드로이드 앱 개발 등의 컴포넌트식 개발과 테스트에 흔히 쓰이는 방식이다.

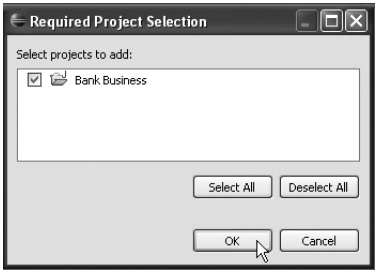
장점	제품 코드와 테스트 케이스 코드의 외부 라이브러리 의존관계를 명확히 분리해낸다. 즉, 테스트를 위해 사용되는 라이브러리는 어떤 것이고, 순수하게 제품 코드를 위한 라이브러리는 어떠한 것인지 구분할 수 있다. 이클립스 IDE 프로젝트 차원에서 분리되므로, 제품 코드와 테스트 코드를 프로젝트 레벨로 개별적 배포가 용의하다.
단점	환경 구성 방법이 Eclipse IDE에 다소 의존적이 된다. Ant 스크립트 등의 외부 툴을 함께 쓸 경우, 클래스패스 관련해서 스크립트 작성이 번거로워진다.
선호도	프로젝트 라이브러리에 대해 상세화된 전권행사(Detailed Full control)를 원하는 개발자에게 추천.

환경 구성 방법

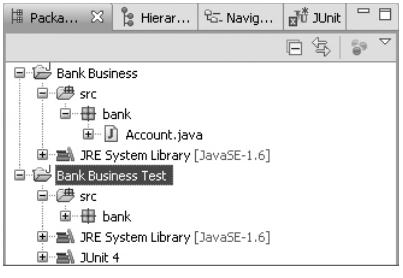
테스트 전용 프로젝트를 하나 만든다. 예제에서는 Bank Business Test라는 이름으로 생성했다. 프로젝트가 만들어졌으면, Bank Business Test 프로젝트 프로퍼티(Properties) 창을 띄운 다음 Java Build Path 항목의 Projects 탭에서 Add를 누른다.



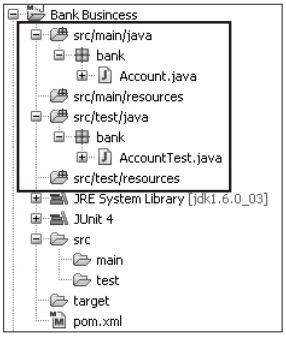
프로젝트 참조로 Bank Business를 선택한다.



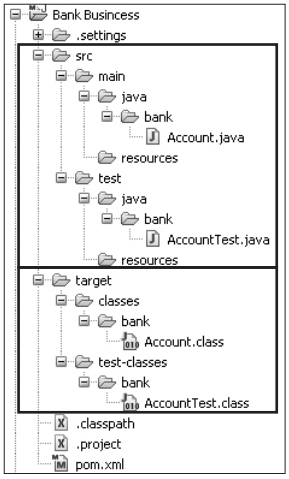
테스트 케이스를 작성한다. 보면 알겠지만, 테스트 대상 프로젝트에는 JUnit 라이브러리조차 필요 없어진다.



#6. 메이븐(Maven) 스타일



패키지 구조



실제 폴더 구조

CoC[1] 사상 기반의 Maven[2]을 사용할 경우 기본적으로 구성되는 구조다. 기본적으로 source 폴더가 네 개로 구성된다.

/src/main/java	제품 코드가 들어가는 위치
/src/main/resources	제품 코드에서 사용하는 각종 파일, XML 등의 리소스 파일들
/src/test/java	테스트 코드가 들어가는 위치
/src/test/resources	테스트 코드에서 사용하는 각종 파일, XML 등의 리소스 파일들

1 CoC(Convention over Configuration): 설정보다는 관례. 환경 구성을 위해 설정 작업을 일일이 따져 만들기보다는 일반적인 관례에 따라 보편적인 애플리케이션의 환경 구성을 그대로 이용한다는 사상. CoC를 기본 사상으로 삼고 있는 Ruby 언어 프레임워크인 RoR(Ruby On Rails)과 함께 널리 퍼지게 됐다. 여담을 추가하자면 RoR을 만든 37Signals의 데이비드 하이네마이어 핸슨(David Heinemeier Hansson)은 2005년에 RoR로 그해의 해커상을 타게 된다. 그렇다. RoR이란 그런 거다. 으..응?

2 정식 명칭은 Apache Maven. 현재는 Maven이라고 하면 Maven 2 버전을 지칭한다. Java 프로젝트 관리와 빌드 자동화를 위한 툴로서, 마찬가지의 일을 하는 Ant와 기능적으로 많은 부분이 겹친다. 하지만 Maven은 빌드뿐 아니라, 개발 전체의 라이프 사이클까지 관여하며, 의존성 모듈 관리 기능까지 겸하고 있기 때문에 컨셉은 조금 다르다. 만일 단순히 의존성 관리만이 필요한 경우 Ant의 하위 프로젝트인 Apache IVY를 사용하기도 한다. 하지만 Maven 쪽이 이클립스 플러그인 구성이 잘 되어 있기 때문에, 개인적으로는 Maven을 더 추천한다. Maven의 전체 기능을 이용하지 않고 의존성 모듈 관리에만 사용해도 유용하다.

수많은 오픈소스 라이브러리가 이 구조를 따르고 있기 때문에, 익숙해지면 다양한 이점이 뒤따른다.

장점	제품 코드에 필요한 리소스와 테스트에 사용하는 리소스를 분리해서 관리하기에 편하다. 메이븐 방식으로 구성된 프로젝트를 접할 때 어색하지 않다.
단점	이런 형태의 구조를 처음 접하는 유저의 경우 익숙해질 때까지는 구조가 불편하게 느껴지거나 환경에 적응하는 데 시간이 걸릴 수 있다. 프로젝트 팀 전체가 이 구조로 가기 위해서는 교육과 의지가 동반되어야 한다.
선호도	보통. Maven 사용자에겐 거의 회피할 수 없는 구조이지만, Maven 사용자가 국내엔 아직 많지 않다. 하지만 조금씩 사용자가 늘어가고 있는 추세이고, 국내 전자정부 프레임워크에서 표준 관리툴로도 채택됐기 때문에 향후 확산될 가능성이 높다.

저자 한마디

제품 코드와 테스트 코드를 함께 배포하기

보통 업무 코드와 테스트 코드를 최대한 분리해서 배포하는 것이 일반적이다. 그런데 경우에 따라서는 함께 배포하는 편이 더 유용할 때가 있다. 모듈의 이식성을 테스트하는 경우가 그 대표적인 예다. JDK 버전이나 OS 등이 다른 경우, 아니면 특정 장비안에서도 동일하게 동작하는지를 보장하고 싶을 때 테스트 코드를 함께 배포해서 테스트를 한다. 이 아이디어는 하드웨어 개발자들이 회로를 만드는 것에서 비롯됐다. 대부분의 하드웨어는 기동시점에서 자가(self) 테스트를 수행하도록 만든다. PC를 켜면 처음에 뭔가 글자들이 나오면서 메시지가 쑥~ 나왔다가 사라지는 걸 본 적이 있을 것이다. 예전에는 메모리를 테스트한다면서 띠리리릭 하면서 숫자가 올라가곤 했다. 만일 못 봤다면 집 PC에서 키보드를 빼고 전원을 켜보면 테스트 에러가 나는 걸 볼 수 있을 것이다. 이런 식의 자가 테스트는 오히려 하드웨어 개발자들에겐 아주 당연한 절차다. 그래서 일부 하드웨어 개발자들은 그런 작업을 하지 않는 소프트웨어 개발자들이 오히려 이상하다고 생각한다. "테스트도 없이 프로그램을 바로 시작하다니 불안하지 않아?"라고 묻기도 한다. TDD로 만들어진 자동화된 테스트 케이스를 하드웨어의 셀프 테스트 수준으로 만들 순 없겠지만, 적어도 새로운 플랫폼에 이식하게 될 때 테스트 코드를 함께 배포해서 테스트를 하는 건 충분히 고려해볼 만한 내용이다.

3.2 테스트 메소드 작성 방식

테스트 클래스 내에 테스트 메소드를 추가하는 방법에는 몇 가지가 있다.

테스트 대상 메소드와 이름을 1:1로 일치

테스트 대상 코드	테스트 코드
public int getBalance( ) { … }	@Test public void testGetBalance( ){ … }

장점	테스트 메소드의 숫자가 적어져서 보기가 편하고, 대상이 되는 클래스의 메소드와 1:1로 연관지어 생각할 수 있다.
단점	추가적인 테스트 케이스가 하나의 테스트 메소드 내에 전부 존재하게 만들 경우, 메소드 내의 초반 테스트 단정문이 실패하면, 뒤쪽 테스트 케이스들은 실행되지 않는다. 이럴 경우 성공하는 케이스와 실패하는 케이스가 중간 중간 섞여 있을 수도 있지만, 구별해낼 수 없다.

테스트 대상 메소드의 이름 뒤에 추가적인 정보를 기재

테스트 대상 코드	테스트 코드
public void withdraw(int money) {… }	@Test public void testWithdraw_마이너스통장인출( ){ … } @Test public void testWithdraw_잔고가0원일때( ){ … }

기본적으로는 테스트 대상 코드의 메소드 단위로 작성하게 되고, 케이스별로 테스트 메소드를 추가하는 방식이다. 가장 권장되는 방식이다. 보통 상세 케이스들은 메소드 이름 뒤에 밑줄(_)을 붙여 구별하는데, 이때 한글을 사용해도 무방하다. 필자 개인적으로는 한글 사용을 적극 권장한다. 테스트 메소드 목록을 뽑은 다음에 엑셀에서 _로 구분해넣으면 자연스럽게 테스트 케이스 항목이 표로 만들어지기 때문이다.

장점	더 다양한 케이스별로 성공/실패를 알 수 있다. 케이스마다 독립적으로 수행할 수 있어 오류의 가능성이 좀 더 줄어든다.
단점	테스트 케이스만큼 메소드를 만들기 때문에 메소드 숫자가 많아진다. 테스트 메소드 이름을 짓는 데 상당한 요령이 필요하다. 케이스마다 리소스 초기화(setup) 작업이 필요하다.

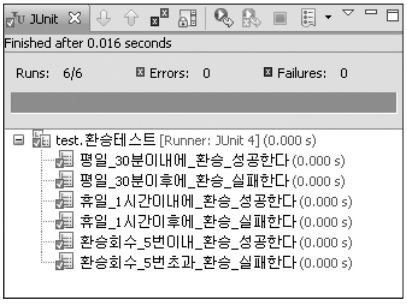
테스트 시나리오에 집중

테스트 대상 코드	테스트 코드
특정한 메소드를 대상으로 하기보다는 테스트 시나리오가 대상이 된다.	@Test public void VIP고객이_인출할때_수수료계산(){ … } @Test public void 일반고객이_인출할때_수수료계산(){ … }

통합 테스트(integration test)나 사용자 테스트(acceptance test)에 가까운 형태로 테스트 케이스를 만들 필요가 있을 때 사용한다. 작성 대상 클래스의 메소드 구현을 위해 사용한다기보다는 테스트 시나리오에 집중해서 대상 클래스 자체의 통합적인 기능을 테스트할 때 사용한다. 단, 이때 단순히 '한글로 테스트 메소드 이름을 짓는다'라고 정하기보다는 일정한 형식을 갖고 이름을 짓도록 유도하면 좀 더 도움이 된다. 통합 테스트나 사용자 테스트는 그 특성상 일반적으로 '선조건 → 수행 → 예상결과' 식의 테스트 시나리오를 갖는다.

```
public class 환승테스트 {

    @Test
    public void 평일_30분이내에_환승_성공한다(){
        // 테스트 케이스
    }

    @Test
    public void 평일_30분이후에_환승_실패한다(){
        // 테스트 케이스
```

```
        }
    …
```



테스트 수행 결과 화면은 위와 같아진다. 이런 경우에는, 테스트 결과만 정리해도 하나의 어엿한 테스트 문서가 될 수 있다.

장점	테스트 케이스를 시나리오에 따라 체계적으로 작성할 수 있다. 테스트 클래스를 하나의 업무단위 테스트 단위처럼 문서화할 수 있다.
단점	테스트 대상 클래스의 단위 메소드 구현 시에 사용하기에는 다소 무리가 따른다.

3.3 테스트 케이스 작성 접근 방식

테스트 클래스의 구성뿐만이 아니라, 테스트를 작성하는 데 있어서 어떤 형태로 접근할 것인가에 대한 논의가 필요하다. 보통 테스트 케이스를 작성할 때 기본적으로 케이스를 작성하는 기준은 '빠짐없이!'인데, 지나치게 집중하다 보면, 개발자가 개발자인지 QA 테스터인지 경계선이 모호해질 수 있다. 물론, 역할을 나누어서 그건 내가 할 영역이 아니니깐~ 하는 식의 접근도 좋은 건 아니지만, 그래도 각자 전문 분야라는 게 있는 것 아니겠는가.

설계자와 개발자가 분리되어 있는 경우

만일 설계자와 개발자가 분리되어 있는 경우라면, 설계자가 설계문서에 테스트 케이스를 작성해놓아야 한다. 이 경우엔 코드가 아닌 말 그대로의 '테스트 시나리오'를 적어서 개발자에게 전달해야 한다. 그렇지 않으면 개발자는 잘 동작한다고 믿는 코드의 모습 그대로 테스트 케이스를 작성하게 될 것이다. 설계 문서를 받아서 개발하는 개발자 입장에서는 머리를 혹사시켜 가면서 테스트 케이스를 작성할 여력이나 의지를 갖기 어렵다. 개발자의 문제라기보다는 상황 자체가 종종 그렇게 만든다는 걸 이해해야 한다. 대부분의 경우, 안 그래도 모자란 개발 시간, 그것 말고도 해야 할 업무들이 서로 자기 먼저 진행해달라면서 번호표 뽑고 대기하고 앉아 있다는 걸 알아야 한다.

개발자가 설계와 개발을 함께 하는 경우

물론 개발자가 설계에 적극 참여해서, 제품 자체에 주인의식을 갖고 스펙을 작성할 수 있다면, 혹은 애자일에서 이야기하는 것처럼 사용자 스토리를 작성해서 스토리 카드 뒷면에 테스트 케이스 조건까지 적고 그것이 고객에게까지 합의가 가능할 정도로 능동적으로 참여할 수 있는 상황이라면 테스트 작성 범위에 대해 크게 고민할 필요가 없다. 알고 있는 그대로, 해당 케이스에 대해 작성하면 되니까.

[스토리 카드 앞면 예]

Backlog Item #34 **09. 기능 공통 : 공통 : 첨부파일** 파일 업로드/다운로드 기능을 구현한다.	스토리 포인트 **8** 스프린트 #1

[스토리 카드 뒷면 예]

테스트 케이스

– hwp, doc, xls, ppt만 첨부 가능하다.

– exe, bat, com 등의 확장자를 등록하려 할 때 보안 모듈을 호출한다.

– 10M 이상의 파일은 업로드할 수 없다.

무엇을 테스트 케이스로 작성할 것인가?

이런저런 것 다 걷어내고, '그래, 뭐가 됐든 테스트 케이스를 찾아내서 우선 개발 좀 하자'라는 일반적인 상황이라면, 초창기에 어떤 것을 테스트로 삼을 건지, 바로 찾아내기가 쉽지 않을 때가 많다. 이럴 때 사용할 수 있는 흔한 접근 방식이 몇 가지 있다. 우선, 시나리오식 접근 방법을 들 수 있다. 해피데이 시나리오(happy day scenario)라고도 하는 방식으로, 정상적인 흐름일 때 동작해야 하는 결과값을 선정해놓는 방식이다. 그 다음에는 발생할 수 있는 예외나 에러 상황에 대한 결과값을 적은 블루데이 시나리오(blue day scenario) 방식[3]이 있다. 그리고 또 삼각측량법이라는 것이 있는데, 이를테면 '두 숫자 곱하기(a,b)' 메소드는 a*b와 같은지 비교해보고, a를 b번 더한 것과도 같은지 비교해보는 방식이라고 생각하면 되겠다. 에지 케이스(Edge Case)라 불리는 경계조건 찾아내기는 양수, 음수, 0, 아주 큰 값, 소수점 등을 떠올리면 된다.

무엇을 테스트할 것인가는 꽤 어렵고도 중요한 부분이다. 이에 대해 『실용주의 프로그래머를 위한 단위 테스트 with JUnit』(인사이트)라는 책에서는 다음과 같은 질문을 해보라고 권하고 있다.

- 결과가 옳은가?
- 모든 경계조건이 옳은가?
- 역(inverse)관계를 확인할 수 있는가?

3 에러나 오류가 발생하는 상황. 예외처리나 정상적인 경우와는 다른 분기를 타게 되는 경우를 지칭한다. 말 그대로 우울(blue)한 시나리오에 해당한다.

- 다른 수단을 사용해서 결과를 교차확인할 수 있는가?
- 에러 조건을 강제로 만들어낼 수 있는가?
- 성능이 한도 내에 있는가?

간단하지만, 테스트 메소드 작성에 있어 가이드가 되는 질문들이다. 만일 테스트 접근 방법에 대해 좀 더 알고 싶다면 서점에 가서 가장 얇은 테스트 관련서를 한 권 찾아보자. 대신, 욕심내지 말고 부디 제일 얇은 걸로 고르길 바란다. 무리하게 욕심을 내면, 사놓고 읽지 않아 책이 쌓여만 가는 북 쇼퍼(book shopper)가 될 수 있다는 점을 유의하자.

3.4 TDD의 한계

■ 동시성 문제

물론 TDD에도 쉽지 않은 문제들이 많이 걸려 있다. 정치적이거나 사회적인 문제를 제외하더라도 기술적으로도 꾸준히 공격받는 사안들이 있다. 우선 '동시성(concurrency)'이 걸려 있는 코드에 대한 테스트 케이스 작성이다. 이런 경우 테스트 자체를 무결하게 유지하기가 매우 어렵다. 상식적으로 파악하기 어려운 불규칙한 문제가 적지 않게 발생하기 때문이다. 현재, 유일한 해결책을 제시하기는 어렵지만 다양한 방식으로 극복되고 있다. 하지만 아쉽게도 필자가 본 케이스들은 말 그대로 '테스트 케이스 작성이 가능하다' 수준이 대부분이었다. '테스트가 가능한'의 수준이 아니라 '테스트를 작성하는 것이 적절한' 수준의 방법이 나와서 알게 되면 어떤 식으로든 공유하도록 하겠다. 이에 대해서는 TDD를 만든 켄트 벡에게도 직접 한 번 물어봤는데, 바로 대답하는 걸 조금 곤란해했다.

■ 접근제한자(private/protected 메소드)

그 다음으로 논의되는 사안은 테스트할 항목의 접근 제한에 대한 논의다. 눈썰미가 좋은 사람은 앞선 예제에서 알아차렸겠지만, private으로 되어 있는 메소드는 일반적인 방법으로는 테스트가 불가능하다. 그러나 그에 앞서, private으로 되어 있어서 접

근이 어려운 메소드의 테스트 필요성 자체에 대한 논의가 먼저 필요할 것 같다. 현재는 '**public으로 되어 있는 메소드만 테스트해도 무방하다**'라는 경향이 대세인 것으로 보인다. 왜냐하면 private 메소드는 public 메소드들이 사용하는 메소드들이고, public 메소드가 테스트될 때, private 메소드들도 함께 테스트가 이뤄진다고 보기 때문이다. 다만, 경우에 따라 우선은 모든 메소드를 public으로 만들어서 테스트가 끝난 다음, 차후에 판단하여 public을 private으로 바꾸는 식의 범위(scope)를 줄이는 형태로 접근하는 것도 한 가지 방법이다. 그럴 때에 기존에 생성한 테스트 케이스는 지우지 말고 주석 형태로 남겨놓을 것을 권장한다.

■ GUI

그 다음은 GUI 애플리케이션에 대한 테스트 작성이다. GUI는 심미적인 부분이 강하고, 사용자의 동작에 따라 화면이 많은 영향을 받기 때문에 단순히 '결과상태예상 → 수행 → 확인' 순으로 테스트하기가 어렵거나 복잡해진다. 현재 웹 애플리케이션이라든가, OS 종속적인 UI(OS Native GUI API)를 이용한 애플리케이션, 그리고 국내에서 유독 활발한 X-Internet 제품들의 GUI 영역에 대한 테스트 케이스 작성은 종종 많은 개발자를 골치 아프게 만든다. 웹 애플리케이션 같은 경우엔 뷰(View)에 해당하는 영역이 TDD를 적용하기 곤란한 GUI 영역이다. HttpUnit이라든가, Selenium이라든가 하는 식의 테스트 지원 툴도 존재하지만, 아직까지는 활발히 사용되고 있진 않다. 노력을 들여 테스트 케이스를 작성하는 경우에도, 대부분은 자동화된 기능 테스트의 의미가 더 강하다. 필자 개인적으로는 GUI 영역 개발에 TDD를 적용하는 것에 대해서는 다소 회의적이다. 오히려 GUI 영역에 대한 자동화된 테스트를 만드는 편이 더 낫다고 생각한다. 그런 상황에서 취할 수 있는 최선의 전략은 'UI 영역에는 비즈니스 로직이 최대한 들어가지 않게 작성한다'이다. OS 종속적인 UI를 갖는 GUI 애플리케이션이나 X-Internet 제품의 경우도 마찬가지다. '화면에 해당하는 영역에서는 최대한 업무 코드를 배제하는 형태로 작성한다'가 기본 원칙이 된다. 아직까지 GUI 영역에 대한 TDD는 여러 가지로 다소 소극적인 형태의 접근이 이뤄지고 있는 수준이다. 사람들의 사고가 심미와 기능을 철저하게 분리할 수 있도록 툴이나 기법이 발전되어 나가면 나아지리라 예상한다. 관련해서는 애플리케이션 영역별 TDD를 설명하는 뒷부분에서 좀 더

자세히 다룰 예정이다. 하지만 미리 이야기하지만, GUI는 TDD에 있어서 몸에 편한 옷은 아니다.

■ **의존성 모듈 테스트(target = A but A → B)**

테스트하려는 부분에 의존성이 있는 모듈을 사용하고 있는 경우의 테스트도 종종 문제가 된다. 즉, '테스트의 대상이 되는 A가 기타 메소드나 클래스를 참조하고 있을 경우, 그리고 해당 참조 부분에 대한 접근이 쉽지 않을 경우에 B는 어떻게 할 것인가?'라는 문제다. B가 아무런 문제 없이 확실한 모듈이라면 상관없는데, 그렇지 않다면 문제의 소지가 있는 부분이다. 이럴 때는, B를 둘러싼 일종의 프록시 클래스(proxy class)를 하나 만들어서 온전히 B부터 테스트하는 방식으로 접근하는 방식을 취하든가, 아니면 B를 신뢰한다는 가정하에서 A만을 테스트한다. 보통은, 개발자의 성향상 후자를 많이 선택한다. 근래에는 의존성 제거를 위해서 Mock 객체를 많이 사용한다. 마찬가지로 이에 대해서는 Mock 객체를 설명하는 부분에서 다시 다룰 예정이다.

지금 거론한 것들이 가장 흔하고 대표적인 TDD의 한계상황들이다. 일부는 해결책을 찾았고, 일부는 여전히 난점으로 남아 있다. 물론 이 외에도 테스트를 작성하는 데 있어 더 많은 문제상황이 있을 수 있지만, 계속해서 해결책을 찾아가고 있다. 앞으로는 더욱 나아지리라 기대하고 있다.

참고

- Extreme Programming Challenge Fourteen
  http://c2.com/cgi/wiki?ExtremeProgrammingChallengeFourteen
- Testing Private Methods with JUnit and SuiteRunner
  http://www.artima.com/suiterunner/privateP.html
- private 인터페이스를 테스트해야 하는가?
  http://www.xper.org/wiki/xp/TestingPrivateInterfaces?action=print

저자 한마디

TDD의 또 다른 한계

사실 TDD의 한계에는 기술적인 요소만 있는 건 아니다. 『소프트웨어 공학의 사실과 오해(Facts and Fallacies of Software Engineering)』(로버트 L. 글래스)의 31Fact를 보면 "오류 제거는 생명 주기에서 가장 시간이 많이 소모되는 단계다"라는 말이 나온다. 그리고 우리는 흔히 초기에 발견된 오류는 수정비용이 낮고, 나중에 발견된 오류는 수정비용이 크다고 알고 있다. 사실도 그러하다. TDD는 이 두 가지 문제를 줄이는 데 큰 도움을 줄 수 있지만, 다른 한편으로 TDD는 그 자체가 일정한 비용을 필요로 한다. TDD를 적용하면 추가적인 인력과 시간, 즉 **'추가적인 비용'**이 발생한다. 그런데 종종 곤란하게 하는 더 큰 문제는, 당장은 이점을 크게 누리지 못할 수도 있다는 점이다. 그래서 때때로 TDD는 현재의 비용을 사용하는 장기적인 '투자'처럼 보인다. 이런 면은 곧잘 급박하게 돌아가곤 하는 소프트웨어 개발에 있어서 치명적인 단점처럼 느껴질 수도 있다. 고객이 됐든 관리자가 됐든, 그 누구도 자신이 책임을 져야 하는 일정 내에서는 아마 해당 비용을 떠안고 싶지가 않을 것이다. 더군다나, 미래의 이익을 누리지 못하는 위치에 있는 사람(개발만 해놓고 떠나가는 디지털 유목민 같은 개발자들)이라면, 더더욱 그러하다. 수치적으로 비용을 상쇄하는 수준의 품질 향상을 증명하기 어려운 경우, TDD 도입 기피 현상은 TDD가 피할 수 없는 한계 중 하나다. 다행히도 근래에는 TDD에 대한 고객들의 인식도 조금씩 바뀌어가고 있고, TDD를 적용했을 때의 장점과 단점에 대한 연구 결과도 조금씩 발표되고 있다. 그리고 그 결과들의 공통적인 결론을 한 문장으로 요약해서 말하자면 이렇다.

"TDD를 적용하면 품질이 아주 높아집니다. 시간지연을 상쇄하고도 남을 만큼 말이죠!"

전문가 인터뷰

변신철 (주)비앤디(BnD) 책임연구원

Q. 안녕하세요? 신철님! 본인 소개와 하시는 일, 혹은 관심 기울이고 있으신 일에 대해 말씀해주시겠습니까?

A. 대구에 있는 (주)비앤디라는 회사에서 12년째 일하고 있습니다. 처음 직장이었고 여전히 저의 직장이지요. 모바일 관련 일을 많이 했었고 현재는 임베디드 리눅스 환경에서 Firmware Application 제작을 주로 하고 있습니다. 대구시에서 운용 중인 버스 관리 시스템에 사용되는 단말기 작업을 통해 TDD와 애자일을 처음 만났습니다.

Q. 현재 테스트 주도 개발(이하 TDD)을 업무에 적용하고 계신가요?

A. QA 쪽으로 Release Binary를 발행하기 전에 CI 쪽에서 TestCase가 자동으로 돌아가도록 하고 있습니다. 최근에 많이 게을러져서 Test Coverage를 측정하고 있지는 않지만 여전히 노력하고 있습니다.

Q. TDD에 대한 경험담이 있으시면 소개해주실 수 있는지요? 좋은 기억이 아니어도 무방합니다.

A. 아쉽게도 좋은 기억뿐입니다.

(경험 1) 데이터 통신과 관련된 다양한 스레드가 사용하는 Queue Base의 저장소를 긴급히 수정해야 하는 경우가 발생한 적이 있습니다. 주고받는 데이터 자체뿐만이 아니라 저장소의 구조를 변경해야만 해서 다양한 처리들을 모두 바꾸는 것이 제법 부담스러웠습니다. 평소의 경우 side effect 등에 대한 부담감으로 상위 레벨에서 수정을 많이 했었지만 이 경우는 근본적으로 수정하지 않는 경우 전체 구조가 많이 나빠질 우려가 있어서 팀원 전체를 소집해 미팅을 가졌습니다. 다행이 팀원들도 모두 수정하는 것에 동의를 했고, 무엇보다 꽤 많은 수의 테스트 케이스가 있어서 용기를 가지고 진행했습니다. 결국 반나절 정도의 시간으로 모든 테스트 케이스를 Pass할 수 있었고 실제 단말에 적용한 결과 아무런 side effect 없이 지금까지 잘 동작하고 있습니다.

(경험 2) IPC 관련 Class를 만들 때 간단한 예제소스를 이용해 작업하지 않고 TDD를 이용해 학습을 하면서 동시에 작업한 적이 있습니다. Unix Domain Socket을 사용한 사례였는데 평소에 잘 사용하지 않던 실험적인 도전이라 예제만으로는 불안한 마음이 있어 작은 단위로 테스트 케이스를 만들어 작업을 진행했습니다. 덕분에 어떤 구조로 사용해야 간단한 코드를 생성할 수 있는지 쉽게 알 수 있게 되었고 실무에서 큰 문제 없이 잘 사용할 수 있었습니다. 이후 얼마간의 시간이 지난 뒤 다른 업체의 개발자가 IPC 관련 문제로 고생을 하고 있다는 이야기를 듣고 제품 클래스와 테스트 클래스를 함께 첨부했더니 테스트 케이스 자체가 훌륭한 예제가 되었다며 고맙다는 회신을 받을 수 있었습니다.

(경험 3) 자동차의 충돌 혹은 전복을 측정하는 단말기를 제작할 때 판단 모듈을 TDD로 작업한 사례입니다. 센서로부터 발생하는 RAW Data를 텍스트 파일로 만들고 간단한 mockup sensor를 이용해 테스트를 진행했습니다. 명시적으로 충돌상황과 전복 상황을 잘 알고 있었기 때문에 작업이 한결 간단하게 진행되어 예상보다 빨리 진행이 되었습니다. 하지만 센서의 민감도 문제로 전복이 아닌 상황에 대한 테스트 케이스가 부족하여 Release 이후에 차량이 둔턱 등을 강하게 넘을 때 전복으로 인식하는 문제가 있었습니다. 보통의 경우 각종 장치들을 차량에 장착하여 현장(운행 중인 차량 내)에서 디버깅하기 마련이지만 몇 개의 전복 관련 테스트 케이스를 추가하고 RAW Data만 QA 팀으로부터 전달받아 얼마 걸리지 않아 가볍게 해결할 수 있었습니다.

(경험 4) 대구에서 사용 중인 BMS에 사용된 경우입니다. 대구가 유난히 종점과 관련된 복잡한 로직을 갖고 있어 막차의 경우 종점 안내가 상당히 어려운 문제가 되었습니다. 이에 관련된 가상의 운행이력을 바탕으로 시간대별 테스트 케이스를 만들어 정상적인 종점을 잘 찾아내는지 확인을 했습니다. 하지만 제 PC에서 모두 Pass하던 테스트 케이스가 일부 개발자의 PC에서 Fail나는 것을 발견하고 검토를 해본 결과 테스트 케이스가 호출 순서와 관련된 의존성이 있다는 사실을 알게 되었습니다. 이후 CI툴에서는 테스트 케이스가 실행되는 순서를 다르게 하는 스크립트를 사용하여 좀 더 안전하게 Release되도록 했습니다.

Q. TDD를 바라보는 본인의 생각, 혹은 'TDD는 무엇이다!'라고 이야기한다면?

A. 테스트 케이스를 작성하는 가장 기본적인 과정인 함수를 생성하기 전에 그 함수가 무엇을 수행하고 무엇을 처리해야 하는지에 대해 생각해보는 것은 여행을 떠나기 전에 지도를 한번 살펴보는 것과 크게 다르지 않다고 봅니다. 내가 도착해야 할 곳이 어디인지를 마음속에 선언하는 것은 내가 다른 유혹에 빠지지 않도록 해주고 본래의 의도 또한 좀 더 명확하게 하는 데 큰 도움을 줄 수가 있습니다. 또한 이런 믿음과 확신을 바탕으로 테스트 케이스를 동료와 공유함으로써 개발자의 인식 상태를 '나'의 코드가 얼마나 컴퓨터에서 잘 동작할까 하고 고민하는 맹목적 '효율성'으로부터 '우리'의 코드가 얼마나 쉽게 기억해낼 수 있고, 잘 이해할 수 있는지와 같은 '인간성'에 대한 실천으로 옮겨놓을 수 있습니다. 간혹 개발자로서 아키텍처 설계와 같은 구조적 혹은 기술적으로 아름다운 상위 레벨의 어떤 것들을 동경하게 됩니다. 하지만 작은 피드백에 좀 더 신경을 쓰고 자신의 표현을 좀 더 인간의 어떤 것에 가깝게 하는 것은 작고 초라해 보일 수 있을지 몰라도 우리를 가장 안전하고 행복하게 하는 '책임감' 있는 움직임이라 생각합니다. 많은 분들이 TDD가 어렵다고 말씀하십니다. 저와 함께 작업하는 팀원들도 아직까지 TDD가 어렵다 합니다. 그리고 저 또한 여전히 TDD가 어렵습니다. 어쩌면 사람답게 사는 것(누군가와 소통을 하고, 사회에 어떤 기여를 하고, 동료를 믿고, 때로 실패하는 것) 자체가 쉽지 않을 수 있습니다. 늘 문제 해결사로만 살아온 우리는 동료가 때로는 인간 존재로서가 아닌 무엇의 원인은 아닌가 하고 물질적으로 바라보기도 합니다. 켄트 벡은 구현 패턴에서 "Programming, then, is a human task done by

humans for humans(프로그래밍은 '사람'을 위해 '사람'이 행하는 인간적인 작업이다)"라고 말합니다. 저는 이 글이 TDD가 가지는 무엇인가에 '반응'하고 그것에 '책임'을 진다는 인간 정신과 관련된 내용이라고 믿습니다.

Q. TDD를 적용하는 데 있어 중요하다고 생각하는 부분, 혹은 유의점은 무엇이라고 생각합니까?

A. TDD에서 가장 중요한 사항은 무엇인가를 해결하는 데만 집중하는 것이 아니라 내가 하고자 하는 게 무엇인지에 정신이 머무르게 하는 것입니다. 저의 경우 임베디드 분야에서 TDD를 공부하고 실무에서 적용을 해보니 예상치 못했던 문제들도 많이 만났고, 테스트 케이스의 '함정'에 빠져 한참을 힘들어했던 적도 있습니다. 단편적인 예로 다양한 스레드 문제와 하드웨어 관련된 문제의 경우 동작원리를 제대로 이해하지 못한 상태에서 만들어내는 mockup들은 재앙에 가까운 문제를 만들기도 합니다. 이런 경우 가능한 실제 데이터를 Scan할 수 있는 환경 구축이 우선이겠지만 상당히 귀찮은 일이 되기도 해서 대부분의 경우 '테스트 케이스를 좀 더 추가해야 하겠지만 일단은 확인했으니 괜찮을 거야'라며 적당히 타협하기 마련입니다. 또는 테스트 케이스들이 너무나 세밀해지거나 단편화되면서 전체 맥락을 잃어버려 모든 테스트 케이스는 Pass하지만 결국 자신의 코드는 엉뚱한 결과를 만들어 낼 수도 있습니다. 하지만 잘 살펴보면 이런 경우 대부분 공통점이 있습니다. 내가 처음부터 테스트하려 했던 게 무엇인지보다는 테스트 성공이 주는 문제 해결의 쾌락에 빠진다는 것입니다. 좀 더 많은 테스트 케이스를 만들어 팀원에게 자랑하고 매일 늘어가는 테스트 보고서를 바라보면서 프로젝트가 성공을 향해 간다고 믿게 되는 것! 저는 이것에 '초록 막대 증후군'이라 이름을 붙여봤습니다. TDD는 외재적 보상을 위한 것이 아닙니다. 테스트를 통과시키는 것은 나에게 주어진 '책임'에 정중히 '응답'하는 것입니다.

Q. 마지막으로, TDD를 주저하는 후배에게 해주고 싶은 말이 있으시다면?

A. 어쩌면 우리는 애자일이라는 단어 뒤에 숨어서 더욱더 비겁해지고 있는 건 아닌지 모르겠습니다. 이런 때에 TDD는 우리를 움직이게 하고 실천하게 합니다. 이것만으로도 더할 나위 없는 훌륭한 이유라 생각합니다.

바쁘신 와중에도 귀한 시간을 내어 인터뷰에 응해주셔서 감사합니다.