성공을 높이고 싶다면, 두 배로 실패하라.

– 토머스 J. 왓슨, IBM 초대 설립자

CHAPTER 8

TDD에 대한 다양한 시각

8장 체크리스트

- ☐ TDD가 주는 설계상의 이점
- ☐ 유연한 코드
- ☐ 계약에 의한 설계
- ☐ TDD 유의사항
- ☐ TDD와 리팩토링의 관계
- ☐ TDD와 짝 프로그래밍
- ☐ TDD를 어렵게 만드는 요인
- ☐ 행위 주도 개발(Behavior-Driven Development)

8.1 TDD와 소프트웨어 디자인

TDD가 주는 설계상의 이점

소프트웨어를 개발하는 건 쉬운 일이 아니다. 특히, 개발 범위 및 규모가 클수록, 그리고 해당 도메인의 업무가 복잡다단할수록 어지러운 코드가 많이 양산되어 넘어서기 어려운 수준(insurmountable chaos level)의 개발 난이도를 만들어내기 쉽다. 결국 혼란 속에서 자멸하거나 누더기가 돼서는 건드릴 수 없게 변한다. 동작은 하지만 만지

면 곧잘 깨져버리는 유약한 소프트웨어가 된다는 이야기다. 이런 상황에서 TDD는 어떤 설계상의 이점을 줄 수 있을까? TDD로 개발을 하면 TDD 자체가 단위 단위의 작은 설계를 만들어낸다. 입력과 출력, 그리고 해당 모듈이 동작하기 위해 필요한 요소 파악 등이 사전에 확실하게 고려되고 테스트된다. 그렇게 만들어지는 모습을 강형 마이크로 디자인(Robust Micro Design)이라고 한다.[1] 이런 디자인의 특징 중 하나는 다른 모듈에 대한 의존성이 상대적으로 적다는 점이다. 왜냐하면 다른 모듈에 의존성이 많아지면 적절한 테스트 케이스를 작성하기가 점점 어려워지기 때문에, 개발자가 스스로 모듈의 의존성을 줄이려는 노력을 초반부터 해나가게 된다. 또한 각 모듈은 최대한 테스트를 독립적으로 수행할 수 있도록 만들기 때문에 자기 완결성(self completeness)이 높은 모듈이 된다. 결과적으로 신뢰수준을 따져봐야 하는 모듈에 대해, 개발자가 접근해야 하는 영역을 좀 더 추상화된 상위 레벨로 올려줄 수 있기 때문에, 소프트웨어의 복잡도[2]가 낮아지고 개발자가 좀 더 일반화되고 추상화된 형태의 디자인에 더 집중할 수 있게 도와준다.

TDD와 객체 지향 프로그래밍(OOP)

OOP에서는 모듈화가 굉장히 강조되고 있고, 대부분의 객체 지향 프로그래밍 언어는 모듈화를 지원하는 기능을 기본적으로 내장하고 있다. 이를테면 클래스나 패키지, 인터페이스 등이 그런 요소다. OOP가 각광받고 유독 모듈화가 강조되는 이유는 **'안전한 부품'** 그리고 **'재사용성'**을 높여주는 요소를 좀 더 쉽게 만들 수 있기 때문이다. 그리고 그걸 이루기 위해 최대한 따라야 하는 가이드로, OOP의 원칙이라는 것이 있다. 그런 OOP의 근간이 되는 기초 원칙 중에는 **'모듈(module)은 높은 응집도(high cohesion)를 유지하고 낮은 결합도(loose coupling)를 갖도록 만들어야 한다'**는 원칙이 있다. 아주 중요한 OOP의 핵심 원칙 중 하나다. 그런데 아이러니하게도 이 원칙은 너무나도 중요한 원칙

1 나사(NASA)에서 로켓을 만들 때도 이런 방식을 사용한다. 나사의 경우엔 그게 소프트웨어가 아니라 하드웨어라는 차이점은 있지만, 목표 상태를 사전에 정하고, 모듈 스스로 자기 자신을 테스트할 수 있도록 만든 다음, 해당 목표를 맞춘 모듈만이 로켓의 부품으로 쓰게 된다는 점은 TDD의 방식과 매우 유사하다.

2 현대의 소프트웨어 개발은 복잡도와의 싸움이다. '소프트웨어의 복잡도를 얼마나 낮출 수 있는가?' 하는 점이 '얼마나 더 큰 규모의 소프트웨어를 만들 수 있는가?'와 일맥상통한다고 여긴다. 현존하는 대부분의 WAS(웹 애플리케이션 서버)가 Java로 작성되어 있다는 사실을 눈여겨볼 필요가 있다. 그 이유는 C나 C++보다 Java 언어가 좀 더 성능이 좋아서가 아니라, 소프트웨어 개발의 복잡성을 줄여줄 수 있는 요소가 언어적으로 좀 더 많이 포함되어 있기 때문이다.

이라서 많은 사람이 잘 모른다. 이게 무슨 소린가 하면, 이 원칙을 배울 때 항상 말이나 그림으로만 봐왔지, 실제 개발코드를 통해 배울 기회가 없어서, 문구로만 머리에 남고, 실제 설계나 코드는 원칙과는 동떨어진 형태로 나오게 된다는 말이다. (누구 때문이라고 할 순 없지만) 수많은 개발자들이 객체 지향 언어를 사용해서 절차적 프로그래밍을 하고 있다. 또 설사, 위 원칙을 염두에 두고 설계를 하고 코드를 작성했다 하더라도, 정말 잘 따르고 있는 건지, 따랐다면 해당 모듈이 어느 정도 레벨의 응집도와 결합도를 갖고 있는지 판단하기가 쉽지 않다.

어느 회사에서 직원의 연봉을 구하는 프로그램 모듈을 작성한다고 가정해보자. 만일 당신에게 객체 지향 설계를 하라고 한다면 무엇부터 하겠나? 아마 모르긴 몰라도 많은 경우 클래스부터 만들려고 할 것이다. 아쉽지만 그건 별로 좋은 방법이 아니다. 또 그 다음엔 여러 모듈(클래스)을 만들어서 이용할 수 있는 한 최대한 재사용하려고 노력할 것이다. 그러다 보면 다음과 같은 코드도 서슴지 않고 나온다. 일부러 상세 주석을 달지 않았다. 그렇다 하더라도 이해하는 데 어렵지는 않을 것이다.[3] 참고로, 코드가 무슨 내용인지 크게 집중할 필요는 없다. 그냥 코드의 구조만 보자. 그리고 이어지는 두 코드는 서로 관련 없는 각각의 코드다.

예 1

```
public class MeasureCounter {

    public String getTotalMeasure(String name){
        Stock b = new Stock(name); // 객체 생성
        int firstCountRate = b.getFirstCountRate(); // 최초 비율
        int runningCount = b.getRunningCount()/2; // 러닝카운트
        int remainingCount = b.getRemainings(); // 남은 수량
        return name + ": " + (firstCountRate + runningCount +
           remainingCount);
    }
…
```

3 보통 코드를 읽는 데 주석이 필요하다는 생각이 든다면, 그런 상황부터가 나쁜 냄새가 나는 코드라고 본다.

예 2

```
public class PublicItem {

    public int getDualCharge(Sender sender){
        int fare = 0; // 요금
        if ( this.item.getStore().getCustomer(sender.getName()).isVip() ){
            fare = sender.getBox().getFare() * 0.8;
        } else {
            fare = sender.getBox().getFare();
        }
        return fare;
    }
...
```

그나마 깔끔하게 작성한다고 한 코드다. 그리고 잘 동작한다. '아, Java로 짰고, 잘 동작하고, 눈으로 보기에도 별로 복잡하지 않고, 간결한 게 아주 마음에 드네!'라는 생각이 들 수도 있다. 하지만 그러면 안 된다. 만일 뭐가 잘못된 코드인지 잘 모르겠다면 큰 일이다. 위와 같이 작성할 테니 말이다. 고무적인 사실은, 위 두 코드는 TDD로 만들었다면 좀처럼 쉽게 만들어지지 않았을 코드라는 점이다. 우선 첫 번째 코드부터 보자. getTotalMeasure()라는 메소드를 작성하기 위해 TDD로 작성한다고 가정해보자. 테스트 코드 입장에서 미리 준비 가능한 건 name뿐이다. name을 넣으면 해당 이름에 해당하는 Stock의 특정 정보들을 알려주는 기능을 만들면 될 것 같다.

아, 잠깐! 그런데 TotalMeasure를 구하려면 Stock 관련 정보가 있어야 할 것 같은데, 내 클래스 안에는 없잖아! 그럼, Stock 관련 정보는 어디에 있는 거지? 아, Stock 클래스에 있지. 자.. 잠깐! 이거 뭐야? 내가 테스트하고자 하는 기능들은 대부분이 내 자신의 클래스인 MeasureCounter에 있는 게 아니라, Stock에 있잖아! 여기서 이걸 테스트하는 게 맞는 걸까?

TDD로 작성을 하게 되면 이런 식으로 기능과 객체의 관계를 스스로 먼저 고민하게 된다. 그리고는 때때로 현재와 같은 클래스 구조로는 테스트 코드를 미리 작성할 수가 없다는 사실을 스스로 알아차리게 된다. 테스트 케이스 코드는 입력과 출력(in/out)이 명

확하고, 사용되는 재료가 적으면 적을수록 작성하기가 쉽다. 즉, **의존관계가 많은 코드는 테스트 코드 자체를 만들기가 어렵다.** 사람의 본성이란 게, 어려우면 안 하게 된다. 의존관계를 최대한 만들지 않고 기능이 동작하도록 노력을 기울이게 된다. TDD는 자신의 모듈에 필요한 게 무엇인지를 미리미리 고민하게 만든다. 기능 위주로 테스트를 하기 때문에 불필요한 속성/필드가 무엇인지도 초반부터 밝혀진다. 따라서 테스트를 작성하다 보면 getTotalMeasure 메소드는 기능이 전적으로 Stock 클래스에 의존하고 있음을 알게 되고 해당 기능을 Stock 클래스를 테스트하는 걸로 변경해야겠다는 생각이 든다. 그리고 결국에 가서는 MeasureCounter 클래스에서 getTotalMeasure가 존재할 필요가 없음을 깨닫게 된다. 이게 바로 응집도를 높이고 결합도를 낮춘 모습이다. 두 번째 소스도 테스트 케이스를 작성하다 보면, 자신의 모듈이 동작하기 위해 필요한 재료를 상당히 멀리서 가져온다는 사실을 테스트 케이스 작성 시에 발견하게 된다.

```
this.item.getStore().getCustomer(sender.getName()).isVip()
```

진정 자신의 클래스 안에 존재해야 하는 커플링 객체가 item인지 customer인지 다시 한번 고민하도록 유도한다. 왜? 테스트 케이스를 작성하려면 재료가 필요하니까. 그리고 그 재료가 어떻게 쓰이는지 미리 고민해야 하니까. 이렇듯 TDD는 좀 더 나은 설계, 좀 더 나은 모듈 디자인이 될 수 있도록 스스로 생각할 수 있게 유도하고, 결과적으로는 모듈이 더 나은 구조를 가질 수 있게 한다. 이와 관련해서는 10장 '실습 예제'에서 다시 다루도록 한다.

유연한 코드

유연한 코드를 만든다는 건 무엇을 의미할까? 흔히 변화에 쉽게 적응할 수 있는 코드(기능 변경 요구에도 구조가 변경되지 않는 코드)를 유연한 코드라 부른다. 또한, 한편으론 요구사항을 받아들이는 데 개발자의 거부감이 적은 코드를 뜻하기도 한다. 개발자의 거부감? 그럼, 개발자의 거부감이 적은 코드란 어떤 걸 의미할까? 대부분의 경우 개발자의 거부감은 개발자가 다루는 소프트웨어 소스의 복잡도에 기인한다. 코드가 복잡하면, 신경 쓸 게 많아지고 민감하게 동작하는 코드가 많아진다. 그래서 복잡한 소스를 수정할 경우 개발자에겐 거부감이 생기게 된다. '아! 싫다!' 이런 느낌.

이 거부감은 따지고 보면 결국 소프트웨어 실패(failure)에 대한 두려움이다. 그런데 사실 이 부분에선 모순이 존재한다. 소프트웨어의 변경이 쉬워지려면 코드가 유연해야 하는데, 소프트웨어 공학에서 추구하는 유연함(flexibility)이란 때로 '개발비용 증가'의 또 다른 요소가 되기도 한다. 이 비용은 온전히 개발자가 떠안게 되는 비용이다. 지나치게 다양한 선택지가 존재하는 환경은 최적의 선택지를 고를 확률을 떨어뜨린다. 그래서 유연한 코드를 만들어서 변경을 용이하게 만든다는 건, 무수히 많은 레고 조각을 제공해놓고는 "어떠한 모양의 자동차도 만들 수 있다!"라고 말하는 것과 유사하다. 그럼 어떻게 해야 할까? 최대한으로 적절한(reasonable) 선택을 할 수밖에 없다. 모호하게 들리겠지만 조금만 노력을 기울이면 좀 더 명확하게 만들 수 있다. **현재 필요한 기능에만 최대한 집중**하는 게 여기서 말하는 조금의 노력이고 적절한 선택이다.

개발자는 일반적으로 다음과 같은 감정을 갖는다.

- **한 클래스의 메소드 숫자가 많을수록 복잡하다고 느낀다.**
- **한 시스템의 클래스 개수가 많을수록 복잡하다고 느낀다.**

따라서 코드 검사(inspection) 툴이나 동료검토(peer-review)를 통해 스스로 작성한 코드가 적절히 유연한 코드인지 측정해볼 필요가 있다. 이때, 다음과 같은 측정은 제대로 된 측정이라 할 수 없다.

- **작성자 자신이 사용한 디자인 패턴들의 목록 감별 시간**
- **작성자 자신이 코드에 변경을 가하는 데 소요되는 시간**

이때 걸리는 노력과 시간으로 코드의 적절한 유연성 여부를 스스로 판단하는 건, 작성자의 지식 수준과 코드 소유욕에 기반한 측정 오류의 흔한 예가 된다. 그것보다는 아래에 해당하는 시간을 측정해보자.

- **시스템에 익숙하지 않은 사람이 새로운 요구사항 반영을 위해 소프트웨어에 변경을 가하는 데 소요되는 시간**

이 시간과 노력을 측정하는 것이 적절히 유연한 코드인지를 판단하는 기준으로 좀 더 유용하다. 그리고 이 원칙은 테스트 케이스를 작성할 때도 마찬가지다.

계약에 의한 설계

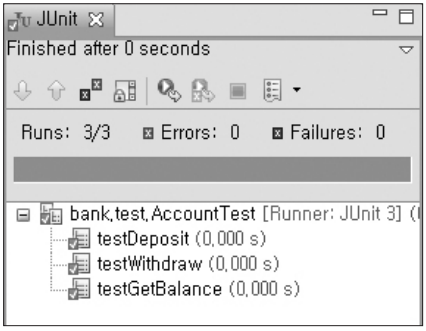
계약에 의한 프로그래밍(Programming by Contract) 혹은 계약에 의한 설계(Design by Contract, DbC)[4]라는 개념이 있다. 여기서의 계약이란 개념은 로직의 **선/후행 조건** 같은 단순한 개념에서부터, 개발에 필요한 **업무규칙**, 고객의 **요구사항**에 이르기까지의 다양한 경우를 지칭한다. 이를테면, 메소드 진행 전에 인자로 받은 객체의 null 여부를 확인해야 하는 선행조건이라든가, 은행에서 통장을 개설할 때는 최소한 100원 이상을 입금해야 한다는 식의 업무규칙, 아니면 모든 회원가입 시 반드시 유효한 이메일 주소를 입력해야 한다든가 하는 식의 고객 요구사항 등으로 말이다. 언뜻 들으면 방어 프로그래밍(Defensive Programming, DP)이나 유효성 검사(validation check)와 무슨 차이가 있나 생각할 수 있다. DbC를 **'선언적인 형태로 프로그램의 상태를 구분해놓고, 이를 통해 작성된 로직의 간결성을 유지하면서도 작성자의 의도를 명확히 전달한다'**라고 한다면, 후자는 **'발생 가능한 문제 상황에 대비해 점검하는 부분을 추가한다'**고 보면 된다. 그래서 DbC의 의도를 **'신뢰에 의한 프로그래밍'**이라고 한다면, DP는 **"의심하라! 점검하라! 추적하라! 그리고 또 의심하라!"**가 되겠다.

이렇게 말을 해도 사실, 무슨 이야기인가 할 수 있는데, 앞서의 Account 예제를 다시 살펴보자. 만일 업무규칙상 계좌를 생성할 때 잔고가 마이너스인 상태로 계좌를 생성하는 것은 허용하지 않는다고 생각해보자. 이를테면 아래와 같은 테스트 케이스를 만들면 업무규칙상으로는 오류다.

```
Account account;
Account wrongAccount;

protected void setUp() throws Exception {
    account = new Account(10000);
    wrongAccount = new Account(-10000);
}
```

4 『실용주의 프로그래머(The Pragmatic Programmer)』에서는 Eiffel 언어와 contract라는 라이브러리로 DbC 구현을 설명하고 있다. DbC에 대한 좀 더 자세한 내용은 http://c2.com/cgi/wiki?DesignByContract나 위키피디아(http://en.wikipedia.org/wiki/Design_by_contract)를 참고하길 권한다.



하지만 현재 상태에서는 테스트가 성공한다. 보는 바와 같이 일반적인 상황에서 단순한 테스트 케이스로는 선/후 조건에 해당하는 계약위반(contract violation)을 막아주지 못한다(물론 약간의 과장과 비약이 있는 건 인정한다). 이런 다소 논란의 여지가 있는 상태를 모면하기 위해 테스트 케이스를 추가할 수는 있다.

문제 해결을 위한 테스트 케이스 1

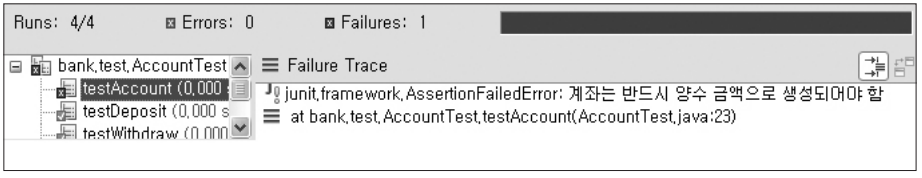
```
public void testAccount(){
    wrongAccount = new Account(-10000);
    assertTrue("계좌는 반드시 양수 금액으로 생성되어야 함",
            wrongAccount.getBalance() > 0);
}
```

위 테스트 케이스는 음수로 계좌가 생성되면 테스트 케이스가 실패하게 만들어봤다. 테스트 케이스를 만들었으니, 이제 이 테스트를 해결하는 로직을 작성하면 문제가 해결될 것처럼 보인다. 하지만 가만히 들여다보면, 이와 같은 테스트 케이스는 Account 클래스의 생성자 Account(int)를 테스트해주는 것도 아니고, 그렇다고 getBalance() 메소드를 테스트하는 것도 아니다. 단지, 계약(업무규칙)을 설명해줄 뿐이다. 이런 식의 테스트 케이스는 테스트를 통과시키려면 어느 부분을 어떻게 수정해야 하는지 쉽게 알 수가 없다. 즉, 디자인 가이드를 개발자에게 명확히 전달해주고 있지 않다.

문제 해결을 위한 테스트 케이스 2

```
public void testAccount(){
    try {
        wrongAccount = new Account(-10000);
        assertTrue("계좌는 반드시 양수 금액으로 생성되어야 함", false);
    } catch (IllegalArgumentException e){
        assertTrue(true);
    }
}
```

오히려 이런 경우는 Account 클래스를 생성할 때 마이너스 금액으로 만들면 예외상황(IllegalArgumentException)을 일으키는 것을 설계의 하나로 정해놓은 경우다. 앞의 경우보다는 조금 낫다.

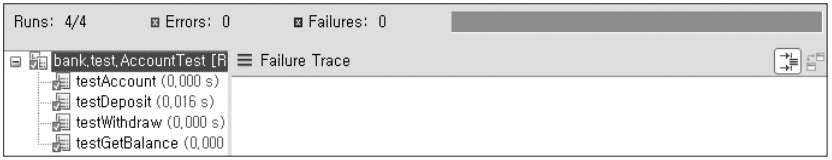


시스템에 질문(ASK)

우선, Account(int money) 클래스를 수정해서, 실패하고 있는 위 테스트 케이스를 통과시켜 보자.

통과하는 코드 작성, 즉 응답(Respond)

```
public Account(int money) {
  if (money < 0) {
      throw new IllegalArgumentException("계좌 생성 시 money는 양수여야 함, 현재:"
              + money);
  }
  this.balance = money;
}
```



그리고 통과

즉, 이런 형태의 방어형 코딩을 유도해줄 수도 있다. "으흠! 그렇군!" 하면서 뭔가 잘 되어가는 기분이 드는가? 그럼 좀 더 상황을 추가해보자. 계좌 생성 시에는 단순히 마이너스 금액으로 계좌를 생성하는 것을 금지할 뿐만 아니라, 다음과 같은 조건이 있다고 생각해보자.[5]

- 21억 이상은 초기 금액으로 설정 불가
- 최소 입력 단위는 10원(10001원 같은 경우로는 생성하지 않음)

이제 위 규칙을 적용해서 테스트 케이스를 만들고, 로직을 추가할 수 있다. 그런데 만일 시스템에서 결코 위와 같은 상황은 발생하지 않으리라는 보장이 있다면 어떨까? 개발 시에는 확실히 하기 위해 체크가 필요하지만 실제로는 신뢰 모듈(trusted system)에서 값을 넘겨받기 때문에, 비용이 많이 드는 검사과정이 필요 없다고 하면 어떻게 해야 할까? 여기서 바로 앞에서 말한 '신뢰에 의한 프로그래밍'이라는 개념이 나온다. 개발을 진행할 때나 특정 검증 작업을 할 때는 이런저런 검사가 필요하지만, 실제 시스템이 동작할 때는 비용이나 시간이 드는 특정 검사를 하지 않아도 될 때, 그럴 때 필요한 개념이 바로 DbC이다. 자바는 JDK 1.4 버전부터 미약하게나마 assert라는 키워드를 통해 해당 개념을 지원해주고 있다. assert 문은 특정 상황의 상태(status)를 확증해주는 데 사용한다.

```
public Account(int money) {
    assert money > -1 : "계정 생성 시 money는 양수여야 함, 현재: " + money;
    this.balance = money;
}
```

5 물론 실제의 경우 int를 사용하진 않는다. 큰 숫자와 높은 수준의 정밀도를 위해 보통은 BigDecimal을 사용한다.
참조: http://java.sun.com/mailers/techtips/corejava/2007/tt0707.html#2

```
public static void main(String[] args) {
    Account account = new Account(-10000);
    account.deposit(10000);
    System.out.println(account.getBalance());
}
```

위의 코드는 assert 문을 이용해서 선행조건에 대한 의도를 명시해놓은 모습이다.

일반적으로 의도와 로직, 제한조건이 명확히 분리될수록 좋은 프로그램에 가까워진다. 의도는 보통 잘 작성된 이름에서 시작하고, 로직은 간결하게 작성하며, 제한조건은 로직과 분리해서 작성하는 걸 원칙으로 삼는다.

저자 한마디

"자바에서 int는 21억이 한계라고? 이봐! C/C++는 32bit 환경이라는 가정하에서 두 배인 42억까지 가능하다고!"

Java의 int 타입, 그리고 C/C++ 같은 C 계열의 long 타입은 32bit(4Byte)의 크기를 갖는다. 그래서 일반적으로 Java의 int나 C 계열의 long 타입에 저장 가능한 숫자는 2의 32승 −2,147,483,648에서 2,147,483,647까지다. 하지만 Java는 C 계열 언어와 달리 unsigned 타입을 지원하지 않는다. 그래서 C 계열에서는 unsigned라는 키워드를 사용해 음수나 양수, 한쪽만 사용하게 될 경우 두 배 크기 영역까지 데이터 저장에 사용할 수 있는 반면 Java는 그렇게 할 수가 없다. 초창기 Java로 넘어온 사람들 중 일부는 이에 대해 불만을 토로했다. 특히 기존 레거시 시스템과 연동해야 하거나, 이미 unsigned 타입들을 사용하고 있는 네트워크 프로토콜을 사용할 때, 동일한 32bit 타입임에도 long이 int로 바로 변환되지 않았기 때문에, 적지 않은 불편함을 줬다. 왜 C와 C++의 많은 부분을 흡수한 포스트 C++인 자바는 unsigned 타입을 지원하지 않는 걸까?

공식적인 이유는 찾기가 어려운데, 적지 않은 개발자들이 signed와 unsigned 연산으로 인한 다양한 문제(버그)에 시달렸기에 간결한 언어를 지향하는 자바에서는 signed, unsigned로 구분해서 사용하는 방법을 아예 제외해버린 걸로 추측된다. 정답은 Java를 만든 사람에게 직접 물어보는 수밖에.

그런데 마침 「자바 리포트(Java Report)」*라는 잡지의 2000년 7월호에 실린 인터뷰 기사에서 이와 관련된 참고할 만한 이야기가 나온다. 'C 언어 패밀리 인터뷰(The C Family of Languages)'라고 제목 붙은 이 인터뷰에는 데니스 리치(Dennis Ritchie), 뱐 스트라우스트럽(Bjarne Stroustrup), 제임스 고슬링(James Gosling), 이렇게 셋이 참석하는데, 각각 C, C++, Java의 아버지라 불리는 사람들이다.

가히 기념비적이라 할 수 있는 이 인터뷰에는 매우 흥미로운 이야기가 다수 등장하는데, 그중 '간결한 언어'와 관련된 질문에 대한 대답의 일부분으로, 고슬링은 다음과 같이 이야기한다. "C 개발자들에게 unsigned에 대해 한번 물어보세요. unsigned가 어떻게 되고, 연산이 어떻게 이뤄지는지 제대로 이해하고 있는 개발자가 거의 없다는 사실을 바로 알 수 있을 겁니다." 그러면서 그는, 자바는 다른 언어들이 봉착하게 되는 다양한 경계조건(edge case)과, 제대로 이해하는 사람이 거의 없는 것들은 포함시키지 않았다고 말했다. 그리고 실제로 C 계열 최신 언어인 C#에서는 컴파일러 차원에서 unsigned와 signed 타입들 간의 연산 자체를 상당히 제한하는 형태로 언어의 간결함을 추구하고 있다. '간결함이냐, 확장성이냐?'라는 오래된 주제는 여기서 논의할 내용은 아닌지라 이쯤에서 정리하기로 한다. 'The C Family of Languages' 인터뷰 전문은 현재 http://www.gotw.ca/publications/c_family_interview.htm에서 찾아볼 수 있다.

* 「자바 리포트」는 2001년 10월호를 마지막으로 폐간됐다.

참고

- MSDN: C# vs JAVA Data Type
- 방준영님 블로그: http://bangjunyoung.blogspot.com/2009/05/cc-signedunsigned.html

8.2 TDD 유의사항

■ 테스트 케이스는 이름이 중요하다

TDD에 익숙하지 않고 학습 단계에 있을 때 흔히 간과하는 것 중 하나는, 작성된 테스트를 수행하거나 읽는 사람이 자신 혼자라고 은연중에 가정한다는 점이다. 그러다 보니 무엇을 테스트하는 것인지 테스트 메소드 이름만으로는 알기 어려운 경우가 종종 발생한다. 자신만 알아보기 쉽게 만들었다든가, 아니면 오해의 소지가 있는 단어를 썼기 때문이다. 보통 테스트가 실패하면 실패한 메소드의 이름이 테스트의 기준이 된다. 따라서 테스트의 이름이 잘 작성되어 있어야 실패에 대한 대응을 빠르게 할 수 있다. 또 가끔은, testDeposit_minusCase01, testDeposit_minusCase02 같은 식의 일련번호를 이름에 사용하는 모습을 보곤 한다. 이런 경우도 권장하지 않는다. 테스트 메소드의 이름이 좀 더 길어지더라도 해당 테스트가 무엇이고 실패한 세부 항목이 무엇인지 곧바로 알 수 있어야 한다. 번호 대신에 의미 있는 이름을 붙여주자. 그리고 경우에 따라서는 테스트 메소드의 이름에 한글을 사용하는 것도 도움이 된다.

testWithdraw_마이너스통장_대출한계이상으로_인출요구시( )

testWithdraw_마이너스통장_연체이자미납자_인출요구시( )

그런데 이렇게 테스트 메소드의 이름을 설명적으로 풀어쓰는 식으로 작성하다 보면 자칫 테스트 메소드의 이름이 길어질 수 있다. 그렇다고 해서 굳이 불편해한다든가, 약어를 써서 줄이려고 한다든가 할 필요는 전혀 없다. 로직을 살펴보지 않고 메소드 이름만으로도 충분히 의미를 전달할 수 있을 정도로 작성하는 것이 중요하다.

■ **더 이상 제대로 동작하지 않는 테스트 케이스는 제거한다**

자동화된 테스트 메소드는 소스의 품질을 좌우하는 중요한 자산이다. 하지만 그렇다고 해서 많으면 많을수록 무조건 좋은 건 절대 아니다. 중복된 테스트 케이스나 업무 변경 등의 이유로 더 이상 제대로 동작하지 않는 테스트 케이스는 과감하게 제거한다. 소스의 품질과 테스트 케이스의 숫자는 항상 일치하는 게 아니기 때문이다.

■ **TDD는 자동화된 테스트를 만드는 것이 최종 목표가 아니다**

TDD는 개발의 목표 지점을 미리 정하기 위해 단위 테스트 케이스를 만들고, 목표 상태 도달 여부를 빨리 확인하기 위해 단위 테스트 케이스를 자동화시킨다. 자동화된 단위 테스트 케이스들은 TDD의 부산물이다. 개발과 설계를 위한 보조 도구이지 목적은 아니다. 자동화된 테스트는 말 그대로 자동적으로 수행되는 테스트를 의미하고, 소프트웨어의 현재 상태의 정상 여부 판단이 목표이다. 향후 소프트웨어 내부가 변경됐을 때 문제가 없는지를 판단하는 데도 물론 사용할 수 있다. 미묘한 차이점인데, 확실히 구분해놓아야 하는 개념이다.

■ **모든 상황에 대한 테스트 케이스를 만들 필요는 없다**

요구사항에 맞는 현재 필요한 기능에 대한 테스트만 만든다. 개발자가 개발을 하다 보면 종종 지나치게 테스트에 집착하는 경향을 보이다가 어느 순간엔가 지쳐버리는 모습을 보곤 한다. 'TDD는 힘들고 어렵고 복잡하구나'라고 말이다.

■ **여러 개의 실패하는 테스트 케이스를 한 번에 만들지 않는다**

개발 중에는, 작성하고 있는 하나의 클래스에 대해서는 하나의 실패하는 테스트만 유지한다. 해당 실패를 성공시킨 다음에 다음 실패하는 케이스를 작성한다.

■ **하나의 테스트 케이스는 하나만 테스트하도록 작성한다**

때로는 하나의 테스트 메소드에서 하나 이상의 항목을 한꺼번에 테스트하고자 하는 욕망에 빠질 수 있다. "귀찮게 테스트 메소드를 뭘 또 만들어! getter 테스트했으니까, setter도 이참에 한꺼번에 테스트해버리자구!" 같은 식으로 말이다. 부디 그렇게 하지 말자. 하나의 테스트 메소드 내에서는 한 가지만 테스트한다는 이 원칙을 강조하는 어떤 사람은 하나의 테스트 메소드 내에서는 assert 계열의 단정문도 딱 한 번씩만 쓰자고 말한다. 그렇게까진 못하더라도 테스트 메소드 이름을 배신하듯 여러 항목을 테스트하는 건 최대한 자제해야 한다.

■ **전통적인 테스트 기법을 배워두자**

각종 테스트 기법을 잘 알면 좀 더 효율적인 테스트 클래스 작성이 가능해진다. 그리고 어떤 부분이 테스트가 필요한지도 더 잘 파악할 수 있게 된다. 테스트 마스터가 될 필요는 없지만, 기초 정도는 배워놓으면 TDD를 잘하는 데에 좀 더 도움이 된다.

■ **테스트 케이스는 최대한 고립시킨다**

각각의 단위 테스트는 다른 모듈이나 시스템에 최대한 독립적이고 고립된 형태로 작성될수록 단단한 테스트 케이스가 될 수 있다. 그래서 가급적이면 다음과 같은 것들이 테스트에 들어가지 않도록, 혹은 직접적으로 영향을 미치지 않도록 작성하면 더 좋다.

- 테스트 케이스가 작성되어 있지 않은 다른 모듈
- 데이터베이스 연동
- 외부 시스템
- 콘솔 출력(System.out)
- 네트워크

물론 위에 나열한 것 자체를 테스트한다면 달라지겠지만, 대부분의 경우 현재 기능을 위해 외부에 의존하는 것들이다. 이를테면 데이터베이스가 없다면 테스트 케이스를 수행할 수 없다든가, 네트워크가 끊겨 있거나 불안정한 상태에서는 관련된 기능 전체를 다 테스트를 수행할 수 없게 된다면 곤란하다. 결국 최대한 의존관계를 줄이고, 경우에 따라서는 Mock 객체를 이용해서라도 독립성을 보장해줘야 특정 테스트 실패에 따른

실패 전파현상(failure propagation)을 최소한으로 줄일 수 있다. 참고로, 의존성을 분리시키지 못해서 코드를 조금만 수정해도 많은 테스트가 실패하게 되는 테스트를 깨지기 쉬운 테스트(fragile test)라고 부른다.

8.3 TDD와 리팩토링

효과적인 TDD를 진행하는 데 있어 알아야 할 사항이 많지만, 그중에서 TDD 자체를 제외하고 딱 하나만 우선적으로 알아야 한다면, 그건 아마 리팩토링(refactoring)일 것이다. 그만큼 TDD에서 리팩토링은 중요한 부분을 차지한다. TDD는 잘 동작하는 깔끔한 코드를 목표로 한다. 이를 통해 간결한 설계와 개발을 지향한다. 리팩토링은 설계를 개선하고 유지하는 것이 목표다. 이를 위한 기본 조건은 잘 동작하는 깔끔한 코드일 수밖에 없다. 둘은 서로 함께했을 때, 각자 더 높은 가치를 갖게 된다. 그리고 자동화된 테스트 케이스 없이 리팩토링을 하는 건 절대 금지하는 항목 중 하나다. 그런데 자동화된 테스트 케이스는 소스가 작성된 다음에 작성하려면, 심리적인 요인(아, 귀찮아! 이미 잘 돌고 있다구!!)에다가 테스트 작성에 소요되는 시간과 육체적인 상황(피곤해! 피곤해! 그만하고 쉴래!)으로 인해 쉽지가 않다. 따라서 소스가 작성되는 시점에 함께 만들 수 있으면 금상첨화일 것 같다. 어떻게 하면 되려나 생각해봤더니, TDD를 하면 코드가 작성됨과 동시에 자동화된 테스트 케이스가 나오게 된다고 한다. 'TDD를 적용해야겠다'는 생각이 든다. TDD를 할 때 하나의 단계가 끝날 때마다 리팩토링을 하면, 대상 코드뿐 아니라 자동화된 테스트 케이스 자체도 사용하기 좋은 코드로 정련이 된다. 서로가 서로를 맞물고 있는 형태다. 혹시라도, 앞에서 필자로부터 리팩토링을 강조하는 느낌을 받지 못했다면, 그건 필자의 잘못이다. **TDD에 있어서 리팩토링은 매우 중요하다.**

리팩토링의 궁극적인 목적은 **'이해하기 쉽고, 수정하기 쉬운 코드로 만들자'**이다. 그리고 여기서의 **이해**는 컴퓨터가 아니라 사람이 주어다. 우리는 종종 다른 사람을 배려하지 않는 코드를 작성하곤 한다. 나타내고자 하는 의미를 좀 더 명확하게 하는 것, 그것이 리

팩토링의 기본이다. 리팩토링은 복잡하거나 어려운 그 무엇, 이를테면 디자인 패턴 같은 부류가 절대 아니다. 쉽게 배우고 크게 효과를 볼 수 있는 몇 안 되는 기술 중 하나다. 만일 아직 '리팩토링'에 대해 학습해본 적이 없다면, 당장 가서 관련 서적을 보길 권장한다.[6] 리팩토링이 지향하는 소스에 대한 접근 방식은 부록 A.3절 '코드 리뷰'에서도 볼 수 있다.

8.4 TDD와 짝 프로그래밍[7]

사람은 혼자 배울 수는 없다. 책이든 사람이든 스승이 필요하다. 책은 일방적으로 전달하는 매체이고 사람은 상호작용할 수 있다. 책은 내용이 고정되어 있고, 사람은 1~2분 사이에도 이전보다 발전한 상태가 될 수 있다. 사람 사이의 상호작용과 서로의 경험과 지식이 혼합되는 형태의 학습은 단순히 책을 보고 따라 할 때와는 상당히 다르다. 애자일 진영에서 쉽게 많이 사용되고, 그 효과에 대해 널리 이야기되는 실천법으로 짝 프로그래밍(pair programming, 페어 프로그래밍)이 있다. 오랜 기간 동안 함께 일해온 동료들끼리도, 옆 동료의 일하는 방식을 자세히 살펴본 경험이 거의 없는 경우가 많다. TDD는 설계를 위한 기법이고 작업을 해나가려면 때때로 많은 전략과 사고가 필요한 기법이다. TDD와 짝 프로그래밍은 잘 어울린다. 한 사람보다는 두 사람의 머리가 더 나은 답을 찾을 확률이 높다. 보통 사용자 스토리를 완성하기 위한 세부 ToDo 리스트가 만들어진 다음에, 짝 프로그래밍으로 TDD를 적용해보면 효과가 더 높다. TDD에 빨리 적응하는 데도 도움이 되고, 도메인 업무에 대한 이해를 높이는 데도 크게 도움이 된다. 특히 프로젝트 초반일수록 얻게 되는 이점이 더욱더 크다. 그 외에도 짝 프로그래밍의 장점 몇 가지를 꼽아보자면 다음과 같다.

6 그래 봐야 우리나라에 책이 몇 권 없다. 그중 한 권은 바로 보기에는 다소 부담스럽고(『패턴을 활용한 리팩토링(Pattern to Refactoring)』, 인사이트), 한 권은 선수과목이 있는 실습책(『리팩토링 워크북(Refactoring Workbook)』, 인사이트)에 가깝다. 마틴 파울러의 『리팩토링』(대청)이 정답이 되겠다.

7 필자는 짝 프로그래밍이라는 단어를 좋아하는데 어떤 이들은 '페어 프로그래밍'이라는 단어보다 속칭 '없어보인다'라고 가끔씩 내게 이야기 한다. 아.. 그래도 난 짝 프로그래밍이란 표현이 더 좋다. 도화지 한 장에 그림을 함께 그렸던 옛날 짝꿍은 지금 어디서 무얼 할까? :)

- 팀원 간의 좀 더 많은 대화로 목표 시스템에 대한 이해 증가
- 제품에 대한 공동설계와 공동소유
- 개발에 효율을 높일 수 있는 작업 방식의 공유
- 개발 스킬 향상

물론 이 외에도 많은 것을 얻을 수 있다. 하지만 많은 사람들이 짝 프로그래밍을 '같이 앉아서 함께 코딩하는 것' 정도로만 이해하고 작업하기 때문에 실패하는 경우가 많다. 짝 프로그래밍을 하기 전에는 주의사항을 반드시 확인하고 시작하지 않으면, 자칫 오해의 골이 깊어져서 서로 가문의 원수가 되는 수도 있다. '짝 프로그래밍 팁과 트릭'이라는 필자의 글을 읽어보면 좀 더 도움이 될 것이다.[8]

짝 프로그래밍과 TDD에 대해 정리하자면 이렇다. TDD의 스킬을 빨리 높이고 싶다면, 힘들더라도 짝 프로그래밍을 도전해보기 바란다. 고생한 만큼 얻는 게 클 것이다. 비록 실패하더라도 다음에 동일하게 실패하지 않을 수 있는 배움은 남게 되어 있다. 그리고, 필요하다면 TDD가 어느 정도 익숙해질 때까지 팀 내에서 짝 프로그래밍을 의무화하는 것도 좋은 방법이 될 수 있다. 하다 보면 느니까 포기하지 말고 꼭 해보자.

8.5 TDD와 심리학

이제 TDD를 실천할 준비가 된 것 같다. 하지만 어쩌면 이제 당신에게는 지금까지보다 더 어려운 문제가 남아 있을 수 있다. 바로 팀 전체에 TDD를 도입하는 것이다. 혼자서 작성하는 코드에 TDD를 적용해 소프트웨어의 품질을 높이는 것도 좋지만, 그것보다 TDD의 결과로 만들어진 테스트 케이스들이 빛을 발하는 곳은 바로 팀 제품을 만들 때다. 자, 그럼 어떻게 팀원 모두와 함께 TDD를 할 것인가? 물론 TDD에 대한 기본 교육은 받았다는 가정하에서 정할 수 있는 몇 가지 방법이 있다.

8 해당 글은 http://blog.doortts.com/79에서 찾을 수 있다.

방법 1. '앞으로 TDD는 필수다!'라고 강제력 있는 발언을 한다.

방법 2. TDD로 개발하라고 말한 다음 자율에 맡긴다.

방법 3. 테스트 커버리지를 강제화한다. 매일 체크해서 관리지표로 삼는다.

우리가 흔히 취하기 쉬운 방법은 첫 번째다. 가끔은 TDD에 심취한 나머지, 팀원들에게 TDD를 하도록 강제하는 경우가 있는데, 이럴 경우 즐거워야 할 개발 자체를 부담스러운 과정으로 만들 위험이 있다. 좀 더 긍정적이면서도, 개발과 별개가 아닌 일부라는 인식을 가질 수 있도록 유도할 필요가 있다. 이를테면, "테스트 케이스를 함께 만들어야 개발 완료로 인정하겠다"는 식으로 말이다. 별 차이가 아닌 것처럼 느낄 수도 있지만, "기능 개발만이 아니라 테스트 케이스도 만들어야 합니다"와 "테스트 케이스와 개발 모듈의 합을 개발완료 모듈로 인정합니다"는 받아들이는 측면에서는 좀 다를 수 있다. 신입사원이라면 원래 당연한 것처럼 받아들이도록 슬쩍 속일 수도 있다(혹, 비열하다 생각되는가? 절대 그렇지 않다. 후배들은 언제가 자신이 자기도 모르는 사이에 굉장히 좋은 습관을 갖고 있다는 걸 알게 될 것이다).

두 번째로 자율에 맡기는 방법을 취할 수도 있다. 하지만 해당 팀이 자기 개발과 자기 업무에 강한 욕망을 보이는 의욕집단('willing to' team)이 아니라면, 십중팔구 흐지부지해진다. 심지어 의욕집단에서조차 곧잘 흐지부지해지기 쉬운데, 몇 가지 이유는 다음과 같다.

- TDD을 실제로 적용하려니 잘 안 되거나 어려워서
- TDD를 안 해도 잘 만들고 있었기 때문에
- '해보니 별거 아니네. 귀찮기만 하고 지겨운데…'라고 느껴서. 즉, 효과를 잘 못 느껴서

각각의 경우 팀의 상태를 보고 적절한 대응이 필요하다.

세 번째로, 테스트 커버리지의 지표를 정하고 강제화하는 방법이다. 어감과는 다르게 꽤 괜찮은 효과를 볼 수 있다. 무언가 숫자로 지표가 나오고, 그것이 개인과 결부되는 것은 일부분 스트레스적인 요소를 갖기도 하지만, 반대로 의욕을 갖게 만들어주는 요

소이기도 하다. 다만, 팀의 목표 커버리지가 적절하지 못하다면 마찬가지로 제대로 된 효과를 보기 어렵다. 목표 테스트 커버리지가 너무 낮으면, 대충 짠다. 너무 높으면, 좌절하거나 속이려는 시도가 발생한다. 일정 시점 동안 팀을 살펴본 다음 적절한 커버리지 비율을 찾는 작업이 선행돼야 한다. 물론 당근을 준비해놓으면 더 좋다(주의! 채찍은 안 된다).

저자 한마디

이익의 충돌과 세 가지 관점

TDD를 바라보는 세 가지 관점이 있다. 기본적으로 개발자, 관리자, 고객의 관점이다.

[개발자]

TDD는 쉽게 시작할 수는 있지만 실제 자연스런 응용에는 적지 않은 노력이 필요하다는 걸 알게 된다. 그리고 그 노력은 때때로 많은 생각을 필요로 하고 결과적으로 고통스러울 수 있다. 하지만 쉽게 배우는 지식은 그만큼 가치가 적을 수 있다. TDD는 배우는 데 시간이 다소 걸릴 수 있지만, 한번 익숙해지기 시작하면 오히려 벗어나기 어려운 매력이 있다. 인내의 열매는 달다고 하지 않는가? 힘들더라도 포기하지 말자.

[관리자]

말을 물가에 데려갈 순 있지만 물을 먹이긴 어렵듯이, 개발자들에게 TDD식 개발을 강제화하는 데는 한계가 있다. 당근, 채찍, 의무사용, 사용하지 않을 때의 불이익 등 다양한 방식을 시도할 수 있지만, 무엇보다 개발자의 사기를 높이고 성장감을 제공해줄 수 없는 상황이라면, TDD를 팀 내에 전파하기가 어렵다. 환경적으로 준비가 되어 있지 않음에도 강제화한다면, 진상이라는 단어와 동의어로 쓰일 관리자의 이미지만이 남을 뿐이다. TDD를 하기 위해 야근이 늘어난다든가, 인사평가와의 결부 같은 심적 압박은 선순환을 목표로 하는 TDD 개발 방식에 오히려 해가 될 수 있다. 부디, 그런 방법을 사용하진 말자. 그럼, 개발자로 하여금 어떻게 자의적으로 TDD를 수행하게 만들 수 있을까? 트레이드 오프가 하나의 열쇠다. '등가교환의 법칙'이라고 어느 애니메이션에서도 표현했던 걸로 기억한다. 더 힘든 만큼 더 많은 걸 얻을 수 있도록 도와야 한다. 관리자인 당신이 원하는 게 있다면, 줄 수 있는 건 무언지도 함께 생각해봐야 한다. 그렇다고 월급을 더 주긴 사실 어렵다. 다행인 점은, 생각보다 많은 개발자가 '돈'보다는 다른 걸 더 가치우위에 놓는다는 사실이다. 그건 바로 **'인정해주는 것'**, **'성장감'**, 그리고 **'신뢰'**다. 능력 있는 팀원일수록 이 세 가지를 더 우선시하는 경향이 있다. 병 안에 갇혀 날개가 접힌 채 굳어가는 팀원들이 크게 날개를 펼 수 있는 기회를 주자. 팀원을 진심으로 대하고, 최선을 다해 믿고, 신뢰하고, 그리고 매 순간을 인정해주자. 칭찬은 고래도 춤추게 한다지 않는가?

[고객]

TDD를 적용하면 품질이 높아질 수 있지만, 개발에 들어가는 자원(effort)도 함께 더 늘어난다는 걸 알아야 한다. 대부분의 고객은 짧은 기간 내에 높은 품질의 제품을 원한다. 하지만 두 마리 토끼를 잡는 건 몇 배는 더 어려운 법이다. TDD를 적용하면 품질이 좀 더 높아질 수 있다고 고객에게 말하라. 시스템적이고 절차적인 기법이며, 이를 통해 결함을 줄일 수 있다고 말이다. 하지만 단점도 함께 이야기해줘야 한다. 필요하다면, 연구 결과도 보여주자.[9] 그리고 고객에게 최대한의 정보를 제공했다면, 그 다음은 고객이 결정하게 하라. 더 품질을 높일 수 있는 방법을 사전에 제시했음을 고객에게 인지시켜라. 언젠가 고객이 동일한 주제로 당신을 찾게 될 것이다.

8.6 TDD를 어렵게 만드는 요인

환경적 요인

필자가 참여했던 프로젝트 중 하나는 대기업 주도 SI(System Integration, 시스템 통합 프로젝트)이면서도 드물게 애자일을 적용한 프로젝트였다.[10] 사용 가능한 모든 기법을 사용해도 무방하다는 일종의 '자유이용권(free pass ticket)'을 받았기 때문에, 단순히 '프로젝트에 애자일 방식을 1~2개 정도 도입해보자'가 아니라, 애자일 실천 기법(Agile Practice)[11]을 풀 스펙(Full Spec)에 가까운 형태로 적용해볼 수 있었다. 해당 프로젝트는 애자일 방식 중 스크럼(Scrum)의 관리 방식과 XP의 개발 기법을 혼용한 형태로 진행됐다. 고객과 함께 스토리를 선정, 그 후 해당 스토리에 대한 스토리 포인트 추정을 위한 플래닝 포커게임(Planning Poker Game)을 했으며, 실제 개발 시에 TDD를 사용했고 짝 프로그래밍까지 적용해 작업했다. 그뿐 아니라 지속적인 통합 서버인 허드슨(Hudson)을 두어서, 개발자가 매번 소스코드를 커밋(commit)할 때마다 빌드가 자동으로 실행될 수 있게 했다. 허드슨은 해당 빌드를 정상적으로 마치면 소스를 다시 빌드

9 부록 'TDD에 대한 연구 보고서' 참조

10 2010년 현재는 드물지 않다. 삼성 SDS, LG CNS를 비롯한 SI 대기업들에서도 점차 확산되고 있다.

11 애자일로 진행되는 프로젝트들에서 사용했더니 그 효과가 높더라고 판명난 기법들을 애자일 실천 기법(Agile Practice)이라고 한다. 아침 스탠드업 미팅, TDD, 일일 빌드, 지속적인 통합 서버 운영, 짝 프로그래밍, 번 다운 차트, 회고 등이 대표적인 애자일 실천 기법이다.

해서 통합 테스트 서버이면서 동시에 운영 서버가 될 예정인 서버에 바로 반영했다. 그로 인해 항상 실 운영환경의 모습을 바로 확인할 수 있었다. 또 그 서버를 기준으로 분석 설계자는 테스트를 수행했다. 빌드를 진행함과 동시에 각종 소스 정적 분석 도구들(PMD, CPD, Findbugs)은 각종 유용한 결과 보고서를 만들어냈고, JUnit 테스트 보고서와 코버추라 커버리지 보고서(Cobertura Coverage Report)가 만들어주는 결과 보고서들로 자동화된 테스트의 상태를 항상 확인했다. 이런 기술적인 시스템과 함께, 매일 아침에는 팀원 전체가(심지어는 디자이너까지 함께!) 스크럼 미팅이라 불리는 스탠드업 미팅[12]을 진행했고 매주 말에는 한 주를 마감하는 마감 회고, 스프린트[13] 종료 시에는 시간을 따로 잡아 팀원이 해당 스프린트의 교훈을 얻어내는 스프린트 회고까지 수행했다. 내용만 본다면 대기업 주도 SI에서 행해진 유례없는 애자일 프로젝트라고 해도 무방할 것 같다. 자, 해당 프로젝트는 어떻게 됐을까? 특히, TDD나 짝 프로그래밍은 결과가 어떻게 나왔을까?

내용대로라면, 대단히 성공했어야 할 프로젝트(Jack Pot Project)이나, 결과적으로는 평이하게 프로젝트가 마감됐다. 어째서일까? 성격 나쁜 개발자가 있어서? 아니다. 성격은 다 좋았다. 팀원들이 시키는 대로 잘 따르지 않아서? 음. 글쎄. 다들 열의가 있었더랬다. 그럼 대체 문제가 뭐였을까? 원인이자 문제는, 가이드를 착실히 따를 만한 여

12 하루 중 일정한 시간을 잡아서 팀원들이 한 장소에 모여 자신이 한 일과 해야 할 일, 그리고 문제점에 대해 이야기하는 10여 분 내외의 짧은 미팅을 말한다. 보통 서서 회의를 진행하는 방식으로 회의가 길어지는 것을 막는다.

13 애자일 개발에서는 개발의 특정 주기로 잘라서 진행하는데, 그중 스크럼(SCRUM)이라는 기법에서는 그 하나의 개발 주기를 스프린트(sprint)라고 부른다. sprint는 단거리를 빠르게 뛰는 걸 의미하는데, 열심히 달리는 기간이라서 그렇게 붙인 것 같다. 하나의 스프린트는 '준비 → 스프린트 → 회고' 이런 순서로 진행된다.

유가 없었다는 점이다. 새로운 것을 배우면, 적응돼서 효과를 볼 때까지 소요되는 학습 곡선의 마이너스(minus) 영역을 가질 여유가 도저히 없었다. 개발자들도 마찬가지로 느꼈다.

짝 프로그래밍을 하게 되면 정말 금새 지친다. 옆에 사람이 있기 때문에, 평소보다 신경도 더 많이 쓰이고 집중도도 높아지기 때문이다. 거기다 TDD로 개발을 하려면 단순히 자판을 누르면서 진행해나가는 시행착오 반복(Try & Repeat) 식의 개발보다는 초반에 고민해야 하는 요소가 더 많았다. 개발하면서 계속 생각을 해야 한다는 건 누군가에겐 즐거운 도전일 수 있지만, 누군가에겐 에너지 소비가 큰 부담이자 고통일 수 있다. TDD나 짝 프로그래밍이 익숙해지려면 일정 시간 학습시간과 적응기간이 필요한데, 이게 쉽지가 않더라는 거다. 하루 업무를 9-to-6로 정확히 일한다면 가능했을 텐데, 그렇지가 않았다. 피로는 매일 누적되고, 그 와중에도 일정은 기다려주지 않았다. 미래를 위해 투자하기엔 당장 눈앞의 배고픔으로 인해, 먹을 것 없는 작은 고기라도 우선 굽고 보는 게 더 매혹적으로 느껴졌었기 때문에 그러했다. 더군다나 개발자에게 있어 TDD는 짝 프로그래밍보다 훨씬 익숙지가 않은 기법이었다. 게다가 당장 효과가 느껴지는 것 같지도 않았기 때문에 피 말리는 일정을 핑계로 자꾸 건너뛰게 됐다. 필자도 차마, 현실에 허덕이는 개발자에게 강제로 "No Pain, No Gain!!"만을 외칠 수 없었다. 개발자들은 이미 한참 전부터 "No More Pain!!"을 외치는 상태였기 때문이다.

하지만 그럼에도 TDD와 짝 프로그래밍은 효과가 있는 기법이라는 데는 참여했던 개발자들이 전부 동의했었다. 특히 큰 효과를 발휘할 때는, 생각이 많이 필요한 영역을 개발할 때였다. 혼자서 하루 종일 끙끙대며 고민하던 내용을 짝 프로그래밍과 TDD를 통해, 테스트 케이스를 포함한 소스까지 만들어내는 데 2~3시간이면 충분했던 적이 종종 있었다. 물론 상황 자체가 두 개발자의 실력이 비슷비슷한 경우였고, 정말 누가 했어도 혼자 했을 때 하루 종일 걸렸을 분량인지는 불명확하지만, 어쨌든 이미 혼자서 하루를 고민했어도 제대로 진도를 나가지 못했던 내용을 처음부터 다시 작성해서 둘이 했을 때 2시간여 만에 끝냈다면 TDD, 짝 프로그래밍의 효과에 대해서는 어느 정도 가늠 가능할 것이다. 그리고 개발하는 데 있어 지연(delay)이 발생하는 많은 경우는, 작성해야 하는 소스의 타이핑 분량이 많아서가 아니라, 작성하게 되는 업무로직의 난이

도가 높고, 그에 대한 개발자의 이해도가 부족한 경우였다. 이런 경우 두 사람의 협력은 한 사람보다 확실히 두 배 이상의 효과를 거두기 쉬웠다.

참고로, 해당 프로젝트를 진행했던 모습은 doortts.textcube.com에서 좀 더 생생하게 살펴볼 수 있다.

저자 한마디

사람이 중요하다?

사람이 중요하다. 우리는 비정한 0101의 세계에서 각종 기호와 매일 전쟁을 치루고 있지만 정작 현실은, 서버실에 들어 있는 장비들이나 배포(deploy)할 jar 파일들보다도 못한 처우를 받을 때가 있다. 방법론이란 이름으로 사람들을 옭매여서 컨베이어 벨트 라인 앞의 작업자처럼 개발을 진행하려 하고 개발자를 무감각한 시스템의 일부처럼 만들려고 하는 경우가 적지 않다. 프로세스(혹은 방법론, 절차, 스텝, 체크리스트 같은 네모 박스로 표시할 수 있는 것들)는 개발을 지원하기 위해 쓰여야지, 개발자를 지배하기 위해 사용돼선 안 된다고 생각한다. 하지만 어느 순간엔가는 프로세스의 의미를 종종 망각하는 것 같다. 예전에, 필자가 속했던 한 프로젝트에서의 에피소드다. 개발자가 개발을 끝내면 해당 내용을 서버에 반영하는 업무를 팀에서 맡고 있었다. 그런데 종종 반영 작업을 하다가 실수를 해서 장애가 발생하곤 했다. 장애가 생기면 고객이나 우리 팀이나 모두 다 별로 아름답지 않은 상황과 대화에 들어가야 했는데, 꽤 힘들었다. 그래서 대책으로 변경사항을 운영 장비에 반영하기 전에 반드시 동료검토(peer-review)라는 단계를 먼저 수행하고 작업하기로 정했다. 그런데 동료검토 시간은 언제부턴가 작업 담당자를 정하는 시간이 되거나, 때때로 흠을 찾아내 경고하고, 오류를 만들면 안 된다는 압박주입의 시간이 되곤 했다. 그리고 결정적으로 동료검토라는 프로세스를 도입했음에도 오류는 여전히 발생했다. 안타깝지만, 어찌 보면 당연한 일이다. 어쨌든, 그러자 보완책으로 체크리스트를 작성하는 단계가 추가됐다. 소스 반영에, 테스트에, 그리고 체크리스트 작성까지 동원됐지만, 그래도 오류가 사라지진 않았다. 그래서 이번엔 check-twice라는 형태로 야근하고 퇴근하기 전에, 작업 내용을 두 번씩 검사한 다음 퇴근하도록 프로세스를 변경했다. 자, 결과는 어떻게 되었을까? 그래서 오류가 줄어들었을까? 안타깝지만 큰 차이를 만들진 못했다. 오류가 줄기보다는, 작업자들의 건강이 더 많이 지속적으로 줄어들었던 것 같다. 수십 대의 장비에 각종 작업 모듈을 반영하기도 밤이 부족한데, 부가업무까지 전후로 있으니 할 말 다했지 싶다. 그 당시 반영 실수의 큰 원인 하나에는 작업자의 만성피로도 포함되어 있었다. 일주일에 사나흘을 새벽에 들어가고, 새벽에 나오는데, 그리고 그런 사람이 작업을 하는데 프로세스나 체크리스트가 해결책이 될 리 만무하다. 생각해보면, 그 당시 정말 도입해야 했던 것은 최대한의 작업 자동화와 작업자의 충분한 휴식이 아니었나 싶다.

산만한 아키텍처

TDD가 점진적 설계를 통해 전체적인 아키텍처를 만들어가는 형태라고 봤을 때, 이미 아키텍처가 구축되어 있는 상황에서는 TDD를 통한 디자인 개선의 여지가 적다. 또한 TDD가 프로그램의 제일 말단에 가까운 단위 메소드에 대한 테스트 케이스 작성으로 개발의 시작을 유도하기 때문에, 'TDD와 단위 테스트 작성'이 동일 모습으로 간주되기 쉽다. 거기까지는 그래도 괜찮은데 그런 방식으로 개발이 되다 보면 미묘한 괴리가 발생한다. 우리가 TDD에서 집중하는 건 단위 기능(메소드)이지만 객체 지향 언어로 작성된 대부분의 소프트웨어는 오브젝트(클래스)들로 구성된다. 오브젝트는 API나 라이브러리와는 또 다르다. 권한, 위임, 책임 등을 갖고 외부와 통신한다. TDD라는 행위 자체, 혹은 TDD를 통해 만들어진 자동화된 단위 테스트는 그 영역을 커버하지 못한다. 그러다 보니 열심히 단위 테스트 케이스를 작성해 나갔음에도 전체적인 아키텍처의 모습이 한 방향으로 치우치거나 산만하게 흩어져 있는 모습이 될 수 있다. 그러다 결국 어느 순간엔가는 소프트웨어의 전체 구조가 혼란에 빠져서는 소위 말하는 아키텍처적인 막다른 골목(architectural dead end)에 다다르기도 한다. 이런 현상은 TDD만이 지나치게 강조됐을 때 발생한다. 해결 방법은, TDD를 진행하기에 앞서 적절한 레벨의 설계를 선행하는 것이다. 적어도 고수준설계(high level design) 정도는 마친 다음에 TDD를 적용해야 이런 문제를 막을 수 있다.

의존성 전파로 인한 연쇄적인 테스트 실패들

아래 예는 테스트 케이스 작성 시 간섭을 없앤 고립 테스트(isolation test)를 지향하지만, 경우에 따라서는 위반할 수밖에 없는 테스트의 예를 보여주고 있다. 아래 코드는 스프링으로 되어 있는 코드라 스프링이 익숙치 않은 개발자에겐 다소 위화감이 들 수도 있지만, 가만히 들여다보면 그다지 복잡한 내용은 아니다. DB 작업이 관련되어 있기 때문에 @Transactional 애노테이션이 사용됐다. 이 애노테이션으로 인해 DML 작업 후에 자동으로 롤백이 된다.

```
@Transactional @Test
    public void testAddAccount() {
        // Id는 1이고 금액은 50
        Account account = new Account("1", 50.00d);
        accountRepository.addAccount(account);  // 저장소에 계좌 추가

        Map<String, Object> result =
            jdbcTemplate.queryForMap("select id, balance from account
            where id=?", account.getId());

        assertEquals("1", result.get("id"));
        assertEquals(50.00d, result.get("balance"));
    }

    @Transactional @Test
    public void testUpdateAccount() {

        // 다른 테스트 메소드를 호출하고 있다.
        testAddAccount();

        // 방금 추가한 account를 업데이트한다.
        Account account = new Account("1", 75.00d);
        accountRepository.updateAccount(account);

        // SQL 문을 통한 검증
        Map<String, Object> result =
            jdbcTemplate.queryForMap("select id, balance from account
            where id=?", account.getId());

        assertEquals("1", result.get("id"));
        assertEquals(75.00d, result.get("balance"));
    }

    @Transactional @Test
    public void testLoadAccount() {
        // 다른 테스트 메소드를 호출하고 있다.
        testAddAccount();
```

```
        Account account = accountRepository.loadAccount("1");

        assertThat(account.getId(), equalTo("1"));
        assertThat(account.getBalance(), equalTo(50.00d));
    }

    // 추가한 적이 없는 계좌를 불러들일 때
    @Transactional @Test(expected=IllegalArgumentException.class)
    public void testLoadNonExistentAccount() {
        accountRepository.loadAccount("1");
    }
```

testAddAcount()가 실패하면 testUpdateAccount()와 testLoadAccount()도 함께 실패하게 되어 있다. 이런 형태는 테스트 케이스를 작성하다 보면 종종 발생하는데, 해당 경우를 완전히 제거하는 것은 매우 어렵다. TDD 형태로 개발을 할 경우 보통 testAddAccount부터 작성해서 테스트 케이스를 성공시킨 다음 진행하기 때문에, 테스트 작성 당시에는 별 영향이 없는데, 나중에 조금은 당황스러운 상황이 발생할 여지가 있다. 이를테면, 어느 날인가 기능 추가나 버그 수정을 위해 프로그램을 조금 수정했는데, 테스트 케이스 수십여 개가 동시에 실패해버리는 경우가 발생한다. 이 현상이 의존성 전파(dependency propagation)로 인한 테스트 실패다.

8.7 행위 주도 개발

TDD는 프로그램의 가장 하위 부분인 단위 메소드의 기능에 집중한다. 물론 테스트 케이스 작성이 추가되고 메소드 호출의 단계가 상위레벨로 진행될수록, 단위 테스트라기보다는 사용자 테스트(acceptance test)에 가까워지긴 한다. 하지만, 어쨌든 시스템의 하위 수준이라고 불릴 만한 밑바닥에서부터 시작하는 것이 일반적이다. 시스템 안쪽에서부터 시작해서 사용자에 가까운 바깥쪽으로 만들어나가는 방식이기 때문에, TDD와 같은 방식을 인사이드-아웃(inside-out) 방식이라고 부른다. 보통 이런 형식으로 진행되는 TDD에는 몇 가지 흔한 어려움이 있다. 바로, "TDD를 어디에서부터 시작할 것

인가?', '테스트를 어느 정도 작성하면 될 것인가?'와 같은 TDD의 시작점과 범위에 대한 문제다. 그리고 때에 따라서는 '어떤 것은 꼭 테스트해야 하고, 어떤 것은 테스트를 안 해도 될까?'에 대한 적절한 판단도 함께 필요하다. 이런 문제를 좀 더 문맥적으로 해결할 수 있게 도와주는 방식으로 행위 주도 개발(Behavior-Driven Development, BDD)이라는 게 있다.

BDD의 정의와 목표

BDD는 **책임관계자(stake holder)[14]의 관점에서 보는 애플리케이션의 행위(동작) 중 가치 있는 기능부터 개발**하는 방식이다. 그렇기 때문에 보통, 애플리케이션의 로직 중에서도 특히 업무규칙과 관련된 부분을 밖으로 노출하는 데 집중한다. 그리고 프로그래머가 아닌 책임관계자도 쉽게 살펴보고 읽을 수 있도록 가급적 자연어를 사용해 기술하려 노력한다. BDD는 이런 식으로 프로젝트에 참여하는 인원들, 이를테면 품질전문가(QA), 업무전문가, 경우에 따라서는 고객에 이르기까지, 참여자들의 협업을 증진시키는 것을 목표로 만들어졌다. BDD라는 이름 자체는 2003년 댄 노스(Dan North)[15]라는 엔지니어가 만들었다.

BDD의 특징

BDD는 세 가지 측면에서 TDD와 차이점이 있다.

첫 번째로, **BDD는 사용자(고객)에 좀 더 가까운 고수준의 기능영역을 우선적으로 다룬다.** 고객은 JUnit이 애노테이션으로 동작하는지, test로 시작하는 메소드로 동작하는지 관심이 없다. 이를테면 TDD로 개발한 시스템에 대해 다음과 같은 대화를 하게 될 수도 있다.

14 책임관계자(stake holder): 애플리케이션 개발에 대해 직접적인 책임이나 연관이 있는 사람을 지칭한다. 보통 고객이나 고객 쪽 담당자, 업무전문가 등이 책임관계자가 된다. 영문 그대로 읽어서 '스테이크 홀더'라고 표현하는 경우가 많다.

15 작성자인 댄 노스는 BDD를 2세대 애자일 개발 방법론이라고 칭한다. 댄 노스는 현재 마틴 파울러와 같은 회사인 소트웍스(ThoughtWorks)에서 일하고 있다.

개발자: "이 시스템은 현재 자동화된 단위 테스트 125개가 정상 동작하고 있습니다. 마지막 실행 결과를 보세요. 오! 멋지지 않습니까? 원하시면, 지금 바로 실행해볼 수도 있습니다!"

고객: "음… 결과 화면 맨 윗줄에 있는 저 테스트는 뭔가요?"

개발자 : "이 테스트 메소드는 deposit() 메소드를 테스트하기 위한 건데요. 성공했다는 녹색 불이 들어와 있죠! 하하!"

고객: …

대부분의 경우, 고객은 위와 같은 대화보다는, "VIP 고객은 잔액이 없는 상태에서도 인출이 가능했으면 좋겠는데요. 작성하신 애플리케이션이 해당 기능이 잘 동작하나요?"에 대한 대답이 더 궁금할 것이다.

테스트 시나리오 #17: VIP 고객의 마이너스 계좌 인출 테스트

사전조건	VIP 고객이 잔액이 없는 상태에서
기능 수행	현금 인출 요청을 했을 때
예상 결과	마이너스 통장 한도금액까지 인출 가능해야 한다.

만일 테스트 케이스의 이름과 형식이 위와 같았고, 수행결과를 바로 눈으로 확인할 수 있었다면, 고객이나 업무 전문가들은 매우 만족스러웠을 테고, 여타 테스트에도 크게 관심을 보였을 것이다. BDD는 이런 스타일을 지향한다.

두 번째로 BDD는 **'무엇을 테스트할 것인가?'**에 좀 더 초점을 맞추고 있다. 흔히 BDD로 개발을 진행할 때, 전체를 관통하는 가장 중요한 질문은 **"현재 시스템에 들어 있지 않은 기능 중 가장 먼저 구현돼야 하는 기능은 무엇인가?"**라고 말한다. 작성하고자 하는 프로그램 기능들의 비즈니스적인 가치를 파악하고 그에 대한 우선순위를 부여한다. 만일 현재 계좌이체 기능(Transaction)이 구현되어 있지 않은 상태이고, 해당 기능이 우선적으로 구현돼야 할 만큼 중요하다고 판단되면, 다음과 같은 테스트 메소드를 작성하는 것이다.

```
class AccountTest {
    …
    public void shouldTransferMoneyToOtherAccount(){
        ..
    }
```

사용자에 가까운 부분에서 안쪽 모듈을 향해 개발을 진행해나가기 때문에 BDD를 아웃사이드-인(outside-in) 방식이라고도 부른다.

BDD의 세 번째 특징은, **테스트 메소드 작성에 집중할 수 있는 문장적인 템플릿을 제공한다**는 점이다. 다음은 일반적으로 BDD를 적용할 때 사용하는 문장 템플릿이다.

BDD의 기본 템플릿

Given	주어진 상황이나 조건
When	기능 수행
Then	예상 결과

어찌 보면 별것 아닌 템플릿이지만, 많은 BDD 지지자들은 위 템플릿의 강력함에 대해 칭송한다. 그 이유 중 하나로 일반적인 인간의 사고방식과 비슷한 스타일이라는 점을 꼽는다. 사용하는 언어는 생각의 방식에 영향을 크게 미치기 마련이다. 이런 식으로 선조건과 동작, 그리고 결과를 구분지어 생각하도록 유도하는 방식은, 그것도 자연어 문장으로 만들어지는 템플릿은 assertEquals보다는 훨씬 편안함을 준다. 그리고 여기에 덧붙여 메소드 이름에 'Should' 같은 목표 행위를 기술하기 위한 단어를 적절히 사용하면 코드의 목적이 좀 더 분명해지곤 한다. 그리고 이왕이면 책임관계자들이 이해할 수 있는 자연어 문장으로 표현된 테스트 케이스(시나리오)와 결과를 만드는 것을 지향한다.

예

```
public void shouldWithdrawMoney() {
…
}
```

사실 TDD의 테스트 케이스를 잘 살펴보면 test라는 이름으로 시작했기 때문에 흔히 맞는지 틀리는지(true or false)만을 체크하는 식으로 이름을 짓게 된다. 하지만 행위 기준으로 보면 예상하는 행동을 하는지(should do something) 위주로 작성하게 된다. 언뜻 보면 비슷한 듯 보이지만, 사실 잘 살펴보면 지향하는 바가 조금 다르다.

BDD 접근 전략

개발 시에 BDD와 TDD 중 꼭 하나만을 선택해야 하는 건 아니다. 각각 필요한 영역이 다르기 때문이다. 고객은 알 필요가 없지만, 개발자는 반드시 알아야 하고 견뎌내야 하는 로직을 다룰 때는 TDD가 좋다. 그리고 고객 관점에서 이야기하거나 대상 시스템의 최종 형태를 파악하며 개발해야 할 때, BDD를 우선적으로 사용한다.

BDD와 TDD

TDD가 예제에 의한 개발(driven by example)과 단위 테스트 케이스를 지향한다면, BDD는 사용자 시나리오를 통해 사용자 테스트와 회귀 테스트를 지향한다(물론 둘 다 자동화는 기본이다).

TDD로 개발하는 시스템은 모듈 모듈이 자칫 지나치게 작고 얇게 만들어져서 산만하게 흩어진 모습을 갖게 될 수 있다. BDD는 최종 목표를 미리 명확하게 해서, 이렇게 시스템의 모듈이 잘게 산재되는 것을 막고 가장 높은 가치를 제공하는 데 집중해서 만들 수 있도록 유도한다. TDD가 개발자를 위한 방식이라면 BDD는 책임관계자와 함께 하기 위한 방식이다.

BDD 정리

종종 BDD는 단어가 주는 뉘앙스로 인해, TDD에서 유래된 xDD 시리즈의 말장난처럼 들리기도 한다. 하지만 그렇지는 않다. BDD는 TDD에서 사용되는 일종의 개발 전략이고 어떤 면에서는 발전형이라고 볼 수도 있다. 그와 동시에 핵심 업무도메인을 대하는 일종의 대화법이다. 아마, BDD가 크게 발전하진 못하더라도 이런 사상은 TDD에 접목되는 형태로 융화 발전되리라 본다. 사실, BDD와 TDD는 사용하는 어휘와 목적이 조금 다를 뿐, 목표가 같다. 간결한 설계를 만들고, 소프트웨어를 만들기 위한 출발점을 제공한다는 것, 바로 그것이다.

한 가지 덧붙이자면, 개인적으로 흥미롭다고 생각하는 것 중 하나는, DDD(도메인 주도 개발)나 BDD가 추구하는 목표 지점 중에 개발자와 업무전문가가 함께 이야기할 수 있는 언어를 만들기 위한 노력이 존재한다는 점이다. 이를 위한 DDD에서는 DSL(Domain Specific Language)이라는 좀 더 생활언어 친화적인 중간 프로그래밍 언어(half programming language)[16]를 제시하고 있고, BDD에서는 문장으로 이뤄진 시나리오나 스토리를 반자동적으로 프로그래밍화를 만드는 기법을 제시한다. 과연 미래에는 고객이나 업무전문가가 업무 시나리오를 문장으로 기술하고 그것이 바로 테스트 케이스로 만들어지는 이런 방식이 일반화될지, 아니면 계속 개발자와 고객은 각자의 영역에 남게 될지 궁금하다.

현재 Java 언어로 작성 가능한 대표적인 BDD 프레임워크로는 JBehave, JDave가 있고, Java 언어의 Ruby 버전에 해당하는 구루비(Groovy)에 easyB라는 프레임워크가 있다.[17]

마지막으로, BDD는 대략 어떠한 모습으로 만들어지는지 간략한 샘플을 살펴보는 걸로 BDD에 대한 설명을 마치고자 한다.

16 일상어와 프로그래밍 언어의 중간 형태 언어

17 Ruby 언어에서는 RSpec(http://rspec.info)이라는 BDD 프레임워크가 유명했다. 현재는 Cucumber(http://cukes.info)로 통합됐다. 베타리더 한 분은 Java 버전에 해당하는 cuke4duke를 쓰고 있다고 귀띔해줬다.

다음은 JBehave(http://jbehave.org/)라는 BDD 프레임워크로 만들어진 테스트 케이스 예제다.

JBehave로 만들어진 코드 샘플

```
public class LoginSteps extends Steps {
// setup 코드들
…

    @Given("로그인하지 않은 상태라고 가정하고")
    public void logOut() {
        page.click("logout");
    }

    @When("$username과 패스워드 $password로 로그인했을 때")
    public void logIn(String username, String password) {
        page.click("login");
    }

    @Then("다음과 같은 메시지가 보여야 한다, \"$message\"")
    public void checkMessage(String message) {
        ensureThat(page, containsMessage(message));
    }
}
```

다음은 easyb(http://www.easyb.org/)로 작성된 테스트 케이스 예제다.

easyB로 작성된 테스트 코드

```
scenario "계좌에서 인출하기", {
    given "초기 10000원으로 계좌를 생성했다고 가정하고", {
        initialBalance = 10000
        account = new Account()
        account.balance = initialBalance
    }
    when "1000원을 인출하면", {
        withdrawals = 1000
```

```
            account.withdraw(withdrawals)
        }
        then "1000원을 뺀 나머지금액 9000원이 있어야 한다", {
            account.balance.shouldBe initialBalance - withdrawals
        }
    }
```

BDD 프레임워크는 아직까진 TDD의 안정화된 프레임워크보다 좀 더 실험적인 성격이 강하다. 아마도 DDD가 얼마만큼 일반화되느냐, DSL을 얼마만큼 사용하게 되느냐에 따라 활성화가 결정될 것으로 보인다.

참고

- Behaviour-Driven Development
  http://behaviour-driven.org/
- The Truth about BDD
  http://blog.objectmentor.com/articles/2008/11/27/the-truth-about-bdd
- Dan North on Behavior Driven Development
  http://www.infoq.com/interviews/dan-north-bdd

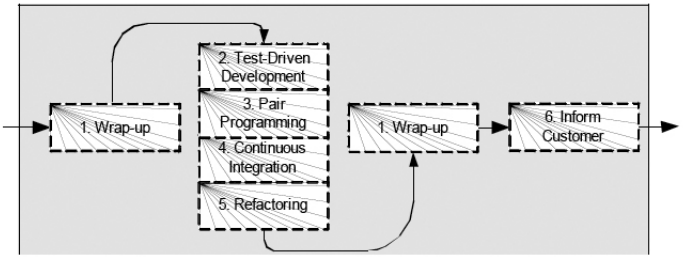
저자 한마디

노키아의 모바일 개발 방법론 MobileD의 개발 프로세스

휴대폰 개발의 핵심 영역 중 하나는 시장 분석에서 판매까지의 사이클을 얼마만큼이나 짧게 가져갈 수 있느냐 하는 부분이라고 한다. 따라서 하드웨어와 함께 소프트웨어도 빠른 시간 내에 개발이 돼야 하고, 또한 짧은 기간에 이뤄진 개발일지라도 품질은 보장돼야 하는 두 마리 토끼 잡기의 노력이 필요해진다. 지금은 LG/삼성도 나름의 방법으로 개발기간을 앞당기고 있지만, 노키아는 짧은 기간 대비 높은 품질로 유명했다. 그렇게 되기 위해 다양한 방법이 사용됐는데, 그중 그들이 취했던 모바일 개발 방법론이 MobileD라는 방법론이다. 다음은 MobileD의 개발 방법론 중 실제 개발이 이뤄지는 개발기간(Working Day)의 프로세스 흐름도 일부다.

Process

The following figure illustrates the process of Working Day:



Tasks

The individual tasks/practices of a Working Day are:

1. **Wrap-up** is an interactive session to communicate progress and problems within the team. Wrap-up is usually conducted as the first

개발이 이뤄지는 Working Day 내에 TDD와 짝 프로그래밍(Pair Programming), 지속적인 통합(Continuous Integration), 그리고 리팩토링(Refactoring)이 핵심으로 이뤄져 있다. 보는 바와 같이 개발범위에 대한 의논을 마친 다음엔 바로 TDD와 짝 프로그래밍으로 개발을 시작하도록 하고 있다. 글로벌 1위 기업인 노키아는 TDD를 하는데, 우리는 못한다는 말은 어떤 의미를 갖는지는 한 번쯤 생각해볼 필요가 있다.