특집 03

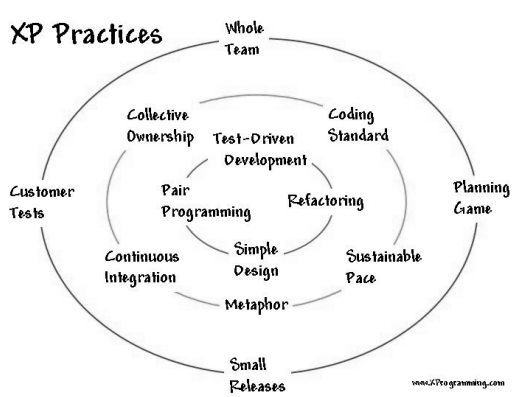
애자일 프로그래밍과 단위 테스트 설계

윤회진 (협성대학교)

목 차 »

1. 서 론

최근 소프트웨어 개발자들이 관심을 갖고 많이 적용하고 있는 방법으로 애자일(Agile)을 들 수 있다. 애자일 방법 적용을 위한 XP(Extreme Programming)의 프랙티스들은 (그림 1)과 같으며, 그 가운데, 특히 지속적 통합(CI:Continuous Integration)과 테스트 주도 개발(TDD:Test Driven Development)은 현실적인 소프트웨어 단위 테스트의 반복적 실행을 요구하고 있다. 테스트 작성을 개발의 시작점으로 하는 TDD와 주기적인 빌드-테스트-통합을 나타내는 CI는 모두 반복적인 단위 테스팅을 요구하며 테스트 자동화 도구 사용을 권장하고 있다. 본 글은 이 두 가지 트랜드가 공통적으로 중요하게 다루는 소프트웨어 단위 테스팅에 대한 재고를 목적으로 하고 있으며, xUnit등의 단위 테스트 도구 사용이 과연 반복적이고 핵심적인 단위 테스트 작업을 완성할 수 있는지에 대해 고찰한다.



(그림 1) XP practices [4]

애자일 프로그래밍 기술인 TDD는 테스팅이 전체 프로세스를 이끌도록 구성되어져 있다. 개발자가 직접 수행하는 단위 테스트 수행을 중심으로 소프트웨어를 개발한다[6]. 또한 CI는 소프트웨어 단위 개발 단계부터 지속적이고 주기적인 통합을 의미하며, 개발자는 자신의 개발 코드를 일정 주기마다 한번씩 -주로 매일- 빌드하고 테스트하여 통합 서버에 로드시켜야 한다. 이는 매 주기마다 개발자 코드에 대하여 수행하는 단위 테스트가 완성되어야 함을 의미한다. 이렇게 TDD

와 CI 모두 개발자가 수행하는 단위 테스트를 중심으로 진행되며, 이러한 단위 테스트는 개발 과정 중에 빈번하게 수행되어야 한다. 그러나 개발자가 테스팅을 수행하는 데에는 한 가지 간과할 수 없는 문제가 제기된다. 만약 개발자가 테스팅에 대한 지식과 경험이 부족하다면, 그들이 수행하는 단위 테스트를 어느 정도 신뢰할 수 있을까? 더군다나 TDD는 그 테스트 결과를 기반으로 개발이 진행되므로 그 문제는 더욱 심각하다고 볼 수 있다. 이는 오히려 소프트웨어 테스트 엔지니어들에게 하나의 기회가 될 수도 있다. 테스트 엔지니어가 구축하는 테스트 케이스가 필요하며, 이는 테스트 설계 전문 기술이 필요하다. 본 글에서는 실제 테스트 품질을 높이기 위한 테스트 설계에 대하여도 언급하고자 한다.

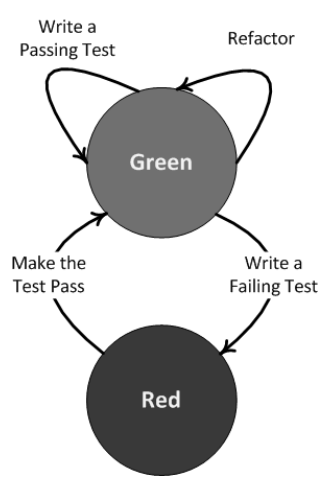
2. TDD와 CI에서의 단위테스트

2.1 TDD, 테스트 주도 개발

테스트 주도 개발이란 테스트할 코드를 먼저 작성하고 코드를 개발하는 개발 방법이다. 지금까지 일반적인 개발 절차는 '설계', '개발,' 그리고 '테스트'의 순서를 밟았다. 먼저 어떻게 개발할지를 설계하고, 실제 운영될 코드를 작성하고, 마지막에 테스트 스크립트를 만들어 테스트하는 방식으로 진행되어 왔다. 그러나 TDD를 사용하면 이 절차는 다음과 같은 순서로 진행된다.

1. TDD는 코드를 먼저 만드는 것이 아니라, 테스트 스크립트를 먼저 만든다.
2. 그 다음에 실제 서버에서 수행되는 코드를 작성하는데, 먼저 만든 테스트 스크립트를 수행하면 통과(pass)될 수 있도록 코딩한다.
3. 코드 작성을 마치고, 그 코드의 가독성, 유지보수성 등을 높이기 위하여 리팩토링을 하면 TDD 작업은 끝난다.

(그림 2)에서와 같이 먼저 작성한 테스트 스크립트가 'Green'이 되도록 코드 작성 및 수정을 반복하는 과정을 거쳐서 일단 테스트가 'pass'되도록 한다. 그 이후 리팩토링 과정을 거쳐 코드의 품질을 향상시킨다. 앞서 언급한대로 대부분 위의 3번 과정을 소홀히 하여 TDD로 작성된 코드의 유지보수성을 떨어뜨리는 결과를 갖게 된다. TDD를 제대로 하려면 '테스트'에 대한 지식과 더불어 마지막 단계인 '리팩토링'에 대한 기술도 적용해야 한다. 리팩토링은 코드가 원래 하는 일이 변경되지 않으면서 내부적인 코드의 구조를 재구성하는 규율에 따른 기술이다(A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.)[2]. 리팩토링에 대하여 아는 개발자가 자신의 코드를 이해하기 쉽고 유지보수성이 높도록 만들어야 한다.



(그림 2) TDD의 테스트 수행

2.2 CI, 지속적 통합

지속적인 통합이라는 용어는 디자인 패턴에서 마틴 파울러가 처음 사용했다. 마틴 파울러는 많은 부분의 개발 공수가 이 통합지점에 들어가게 되며, 동시에 이 지점에서 테스팅이 이루어진다고 지적하면서, 이 통합 작업은 언제 끝나게 될지 모르는 긴 여정으로 표현할 정도였다. 통합은 오래 걸리고 예측하기 어려운 프로세스라는 소프트웨어 프로젝트의 공통된 문제에 대해 인식하며, 그는 이 용어를 XP의 12가지 프랙티스 중 하나에서 가져왔다고 한다. 이 CI를 Thoughtwork 회사에 적용하면서, 통합이 더 이상 힘들고 복잡한 작업이 아니게 변화하는 것을 확인하였다고 한다[3]. 기존의 통합은 개발된 프로그램 단위들을 모두 모아 통합 지점에 동시에 통합 및 조정을 수행하는 고정을 의미했다면, CI는 초반에 그리고 빈번하게 통합 과정을 진행함으로써 통합을 항상하는 가벼운 작업으로 전화시킨다. 물론 이 작업이 '가볍게'되기 위해서는 CI플랫폼 구성이 선행되어져야 한다.

CI 플랫폼은 통상 자동빌드와 테스팅 같은 작업에 대해 특별하게 설정이 된 통합 서버 상에서 일정 간격으로 자동 빌드와 어플리케이션에 대한 테스팅을 보장하는 기법과 도구로 이루어진다. 단위 테스트 기법과 자동 빌드 도구와 같이 엮여진 도구의 융합은 CI를 오늘날 소프트웨어 프로젝트를 수행하는데 있어서 반드시 필요한 항목 중의 하나로 만든다. '초반에 그리고 빈번하게 빌드하라(build early and often)' 라는 말에는 빌드뿐 아니라 개발 팀 내의 커뮤니케이션과 공동 작업을 향상시키는 기능 또한 추가되어 있다. CI 구현에서 가장 눈에 띄는 부분은 빌드이며, 이는 통상적으로 자동화된다. 수동 빌드를 할 수 있는 능력이 있어도 빌드는 밤에 시작하거나 소스코드 변경으로 인해 수행될 수도 있습니다. 일단 빌드가 시작되면, 소스코드의 최종 버전은 저장소로부터 가지고 오고, CI 도구가 프로젝트의 빌드를 시도하고 그 다음에 그것을 테스트를 한다. 마지막으로는 빌드 프로세스에 대한 상세한 결과를 통지하도록 한다. 이러한 통지는 이메일이나 인스턴트 메시지를 포함한 다양한 형태로 보내질 수 있다.

CI를 하기 위하여 먼저 개발자는 지정된 시기 -일반적으로 매일 일정 순간-에 자신의 코드를 '빌드'하여 저장소에 등록시켜야 한다. 이때 '빌드'의 내용에는 테스트가 포함되어져 있으며, 주어진 테스트 코드 실행에 오류가 발생하면 '빌드'가 실패한 것으로 판단되며, 그 결과 개발자 자신의 코드는 저장소에 등록될 수 있게 된다. 앞서 설명한 TDD와 마찬가지로 개발자는 테스트가 통과될 수 있도록 코드를 수정하여 재빌드를 수행하고, 테스트 통과까지 마친 빌드된 개발자 코드를 저장소에 등록시키고, 저장소 관리자는 전체 등록된 개발자 코드들을 통합하기를 반복한다. 이때도 물론 테스트 코드가 실행되며, 테스트가 실패된 부분에 대한 통지를 해당 개발자 또는 관련 개발자들에게 보내게 된다. 해당 부분에 대한 오류 수정이 통지를 받는 개발자가 수행해야 할 첫 번째 액션이 된다.

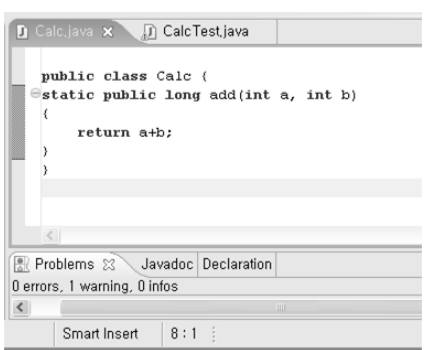
살펴본 것처럼, CI 역시 테스트 자동화를 필요로 하는 기술이다. 특히 각 개발자들이 수행하는 단위 테스트가 빈번하게 진행되어 단위 테스트 자동화가 필요하다.

2.3 xUnit기반 단위 테스트

xUnit은 단위 테스트 실행 도구의 프레임워크로서, 가장 많이 언급되는 것이 JUnit이다. xUnit 기반의 테스트 도구는 테스트 실행의 자동화를

지원한다. 단위 테스트를 수행하려면 테스트 드라이버가 필요하다. xUnit을 사용하면 테스트 드라이버 구현을 고민할 필요 없이, 테스트 데이터 값 설정만을 통하여 테스트 스크립트를 작성할 수 있다. 이 스크립트의 실행도 xUnit내에서 진행되며, 그 결과도 확인할 수 있다.

(그림 3)은 간단한 JUnit의 사용 예를 보인다. 많은 자료에서 JUnit에 대한 설명을 하고 있으므로, (그림 3)에서는 JUnit의 기본흐름만을 소개하고 있다. 이클립스에 플러그인하여 사용하는 JUnit을 사용하는 경우, (그림 3)의 위 그림처럼 하나의 작업 환경에 테스트할 소스코드와 테스트 코드를 두며 테스트를 실행할 수 있다. 왼쪽에 보여지는 코드는 간단한 덧셈코드 Calc.java이다. 이에 대한 JUnit코드는 (그림 3)의 아래 코드이며, 이는 CalcTest.java로 명명되어진다. CalcTest.java 코드에서 *assertEquals((long)5, new Calc().add(2,3));* 가 2와 3을 Calc.java에 입력하면 5가 되는지를




```
// CalcTest.java

import org.junit.Test
import static org.junit.Assert.*;
public class CalcTest
{
    @Test public void testAdd()
    {  // Calc().add() returns long,
    // so we must cast 5
 assertEquals((long)5, new Calc().add(2,3));
    }
}
```

(그림 3) JUnit 사용의 간단한 예제

확인한다. 물론 JUnit은 이 함수 이외에도 단위 테스트에서 필요할 수 있는 다양한 관련 함수들을 제공한다.

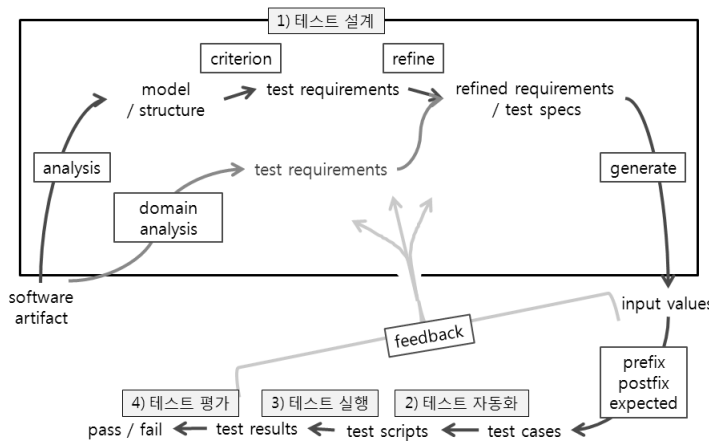
3. 테스트 설계

(그림 3)에서 보았듯이, 자바 코드에 대한 JUnit코드는 "주어진" 테스트 데이터를 사용하여 테스트 대상 코드를 실행시켜 결과값을 확인해주는 작업을 수행한다. 이렇게 JUnit은 테스트 활동 중 단순하지만 반복되어야 하는 부분을 자동으로 수행해 줌으로써 테스트 작업의 효율성을 높여준다. 이런 이유로 테스트의 자동화 도구로서의 JUnit의 가치가 있는 것이다. 그렇다면 JUnit 코드에 사용되는 "주어진" 테스트 데이터는 어디서 오는 걸까? 개발자의 머리에 떠오르는 대로 숫자를 넣어본다면, 테스트 수행 결과에 대한 신뢰가 어느 정도일까? 다시 말하면 어떤 테스트 데이터를 사용하여 테스트하였는지에 따라 어느 정도 테스트가 코드를 확인하였는지를 결정할 수 있다. 이를 "테스트 커버리지"라고 한다.

JUnit등의 자동화 도구의 사용도 물론 중요하지만, 그 이전에 어떤 테스트 데이터를 사용할지를 결정하는 일이 테스트를 통한 코드 품질 보장 면에서 더 큰 의미를 갖는다. 일반적으로 테스트 자동화 도구란 테스트 실행을 자동으로 수행해 주는 도구를 뜻한다. 어떤 테스트 데이터를 사용할지는 온전히 테스트 코드 작성자의 몫인 것이다.

3.1 테스트 활동에서의 테스트 설계

테스트 활동은 다음의 4가지로 분류되어진다. 1) 테스트 설계(Design), 2) 테스트 자동화(Automation), 3) 테스트 실행(Execution), 4) 테스트 평가(Evaluation). (그림 4)는 테스트 활동의 4



(그림 4) 테스트 설계, 자동화, 실행, 평가의 흐름 및 모델 기반 설계 흐름 [5]

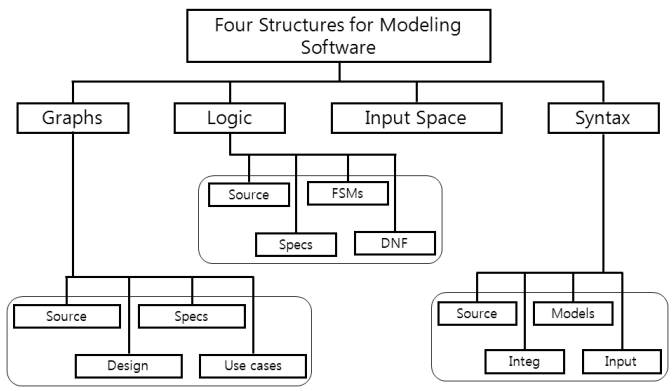
가지 종류들 사이의 흐름을 나타내고 있다.

4가지 활동들은 서로 다른 기술과 개경 지식과 교육 수준을 갖는다. 이 가운데 '테스트 설계' 활동은 테스트 케이스와 값을 설계하는 활동으로서 이때 커버리지 선정기준(Criteria)등의 기술들을 적용한다. 따라서 테스트 설계가 소프트웨어 테스팅에서 가장 기술적이 작업이 된다[Jeff]. 이는 소프트웨어 개발에서의 아키텍트의 역할에 해당되는 수준의 중요성을 갖고 있으며, 테스팅 전문 지식이 풍부해야 한다. 그럼에도 불구하고 많은 현장에서는 테스트 설계 작업의 중요성을 인식하지 못하여 코드를 잘 이해한다고 생각하는 개발자의 직관 또는 경험에 의존하여 테스트 설계 작업을 수행하고 있다. 위의 2)와 3)의 테스트 자동화와 테스트 실행은 JUnit등의 도구를 사용하거나 Unix의 Shell 스크립트 등을 통해서도 구현할 수 있다. 이때 필요한 것이 테스트 설계를 통해 나오는 테스트 데이터들이다. 테스트 설계는 테스트 데이터를 결정하는 단계이고, 어떤 테스트 데이터를 선정하느냐가 테스트 품질을 결정짓게 되므로 매우 중요한 위치를 차지한다. 테스트 설계에서 요구되는 기술들에 대한 여러 연구 결과들이 있다. 본 글에서는 그 가운데 모델 기반 테스트 설계를 소개하고자 한다.

3.2 모델 기반 테스트 설계

소프트웨어 테스팅을 언급할 때 많은 사람들이 쉽게 구분하는 용어는 "블랙박스테스팅"과 "화이트박스테스팅"이라는 말일 것이다. 테스트하고자 하는 프로그램의 소스코드를 보지 않고 소스코드 이외의 명세문서등을 테스팅에 이용하는 것을 "블랙박스테스팅"이라하고, 그 반대로 소스코드 자체를 테스팅에 이용하는 것을 "화이트박스테스팅"이라고 한다. 이 두 경우 모두 결국 테스트 데이터를 추출하는데 목적을 두고 있으며 소스코드의 사용 여부에 따라 다른 분류를 갖고 있다. 최근에는 이러한 '블랙박스 화이트박스 분류'와는 다른 관점의 테스트 설계 방법이 제시된다. 그것이 모델 기반 테스트이다[1].

모델 기반 테스트는 1) 테스트 대상을 표현하는 구조(structures)와 2) 적용할 커버리지 criteria 중심으로 테스트를 바라본다. 이는 테스트 대상의 무엇을 사용할지에 대한 관점 보다는 어떤



(그림 5) 소프트웨어 모델링 구조 [5]

criteria를 적용할지에 초점을 두고 있다. 이렇게 함으로써, 테스트 작업은 테스트 대상 소프트웨어를 구조에 따라 모델링하고 그것을 criteria로 커버하는 것으로 간단해 진다. (그림 5)는 모델 기반 테스트 설계의 개요이다.

그래프, 로직, Input space, 그리고 문법이 테스트 설계를 위한 구조이며, 각 구조마다 적용할 수 있는 대상은 소프트웨어의 소스코드, 명세문서, 설계문서, 사용사례, 상태도 등이다. 만일 그래프 구조를 사용하는 경우, 대상 소프트웨어의 소스코드 또는 명세문서 무엇이 되던지 그로부터 그래프를 그려낸다. 그리고 그래프를 지원하는 커버리지 criteria를 적용하여 그를 만족하는 테스트 데이터를 구해낸다. 구체적인 예제와 활용 방법은 [1]을 참조하기 바란다. 구체적인 예제들과 연습문제를 통해 테스트 설계 지식을 습득할 수 있게 구성되어져 있다.

4. 결 론

애자일 방법에 대한 관심으로 개발 플랫폼을 CI기반으로 구축하는 곳이 늘고 있다. CI 플랫폼이 구축되어져 있다며, TDD를 하기 위한 기술적인 지원이 된 것으로 본다. 그러나 실제 여러 소프트웨어 개발 사이트들은 애자일 프로그래밍이 제공할 “속도감(speedup)”을 기대하면서 애자일 방법론을 적용하고자 한다. 혹자는 애자일 방법을 사용하면 품질이 떨어질 것으로 예측하기도 하고, 또 다른 의견으로는 애자일 방법을 “정확하게” 적용한다면 품질도 보장할 수 있다고 한다. 저자가 찾아본 대부분의 개발 회사들은 다음 세 가지 이유로 실패하고 있었다. 첫째, 리팩토링을 하지 않았고, 둘째, 문서화를 하지 않았다, 그리고 셋째, 테스트 데이터의 전문성이 떨어진다. 이들은 모두 소프트웨어의 품질에 직접 영향을 미치는 요소임에도, “시간 절약”을 위해 위의 두 가지를 등안시하는 오류를 범하고 있다. 리팩토링과 문서화가 빠진 상태로 3~4 iteration이 지나고 나면, 개발되는 코드는 유지보수가 불가능한 존재가 됨은 명백하다. 또한 테스트 작업을 수행하는 것에만 집중할 뿐 우수한 테스트 데이터 선정을 고민하지 않은 경우, 테스트 기반으로 수행되는 프로그램에 대한 신뢰성을 보장할 수 없고, 오히려 테스트 작업이 오버헤드로만 느껴질 수 있는 위험이 있다.

본 글에서는 애자일 방법에서의 단위 테스트의 중요성을 살펴보고, 단위 테스트 도구 사용만으로는 해결되지 않는 테스트 데이터 설계에 대하

여 간단히 설명하였다. 보다 전문적인 지식이 요구되는 테스트 설계에 대한 투자가 결국 단위 테스트를 기반으로 하는 애자일 프로그래밍의 품질을 좌우한다. 품질을 중시하는 현대 소프트웨어 시장에서는, 애자일의 빠른 개발 사이클과 변화에 대한 유연한 대응만으로는 충분하지 않다. 진정한 테스팅의 효과를 누릴 수 있는 테스트 설계에 대한 이해와 적용이 요구된다.

참 고 문 헌

[1] Paul Ammann and Jeff Offutt, Introduction to Software Testing, Cambridge University Press, 2008.

[2] Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[3] Continuous Integration, http://martinfowler.com/articles/continuousIntegration.html, 2006.

[4] Ron Jeffries, What is Extreme Programming? http://xprogramming.com/book/whatisxp/, Nov., 2001.

[5] Jeff Offutt, Introduction to SW Testing, Intensive Workshop, May, 2009.

[6] 채수원, 고품질 쾌속개발을 위한 테스트 주도 개발 (TDD 실천법과 도구), 2010.

저 자 약 력

윤 회 진

이메일 : hjyoon@uhs.ac.kr

- 1993년 이화여자대학교 전자계산학과(학사)
- 1998년 이화여자대학교 컴퓨터학과(석사)
- 2004년 이화여자대학교 컴퓨터학과(박사)
- 2004년~2005년 Georgia Institute of Technology (Post Doc.)
- 2005년~2007년 이화여자대학교 컴퓨터학과(전임강사)
- 2007년~현재 협성대학교 컴퓨터공학과 교수
- 관심분야 : 소프트웨어 테스팅, 요구사항 검증, 웹서비스 테스팅