韓國IT서비스學會誌
第11卷 第4號
2012年 12月, pp.251-262
http://dx.doi.org/10.9716/KITS.2012.11.4.251

# 원개발자 부재에 따른 원시코드 기반의 단위테스트 노력 분석*

윤 회 진**

# Effort Analysis of Unit Testing Conducted by Non-Developer of Source Code*

Hoijin Yoon**

■ Abstract ■

Unit testing is one of the test levels, which tests an individual unit or a group of related units. Recently, in Agile Development or Safety-critical System Development, the unit testing plays an important role for the qualities. According to the definition of unit testing, it is supposed to be done by the developers of units. That is because test models for the unit testing refers to the structure of units, and others but its original developers hardly can understand the structures.

However, in practice, unit testing is often asked to be done without the original developers. For example, it is when faults are revealed in customer sites and the development team does not exit any more. In this case, instead of original developers, other developers or test engineers take a product and test it. The unit testing done by a non-developer, who is not the original developer, would cause some difficulties or cause more cost.

In this paper, we tests an open source, *JTopas*, as a non-developer, with building test models, implementing test codes, and executing test cases. To fit this experiment to practical testing situations, we designed it based on the practices of unit testing, which were surveyed through SPIN(Software Process Improvement Network). This paper analyzes which part of unit testing done by non-developers needs more effort compared to the unit testing done by original developers. And it concludes that Agile Development contributes on reducing the extra effort caused by non-developers, since it implements test codes first before developing source code. That means all the units have already included their own tests code when they are released.

Keyword : Unit Testing, Agile Development, TDD(Test-Driven Development), JUnit

논문투고일 : 2012년 07월 27일 논문수정완료일 : 2012년 11월 18일 논문게재확정일 : 2012년 11월 21일

* 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임 (2011-0009916).

** 협성대학교 컴퓨터공학과

# 1. 서 론

2011년 여름, 우리나라 초등, 중등, 고등학교 학생들의 학업관련 정보가 담겨있는 NEIS 시스템의 오류가 드러나는 일이 있었다. 동점자 처리 부분에 오류가 있어 학생들의 석차 산출이 잘 못 되는 결과를 낳았다. 이후 특검반이 구성되어 실제 개발 대표 업체를 대상으로 전문가 조사를 실시한 결과, 테스트 특히 단위 테스트(Unit Test)의 부재가 그 원인으로 밝혀졌다. 단위 테스트는 코딩 단위를 대상으로 이루어지는 테스트로서 해당 단위의 개발자가 수행하도록 권고되어지는 테스트 레벨이다[10]. 그러나 소프트웨어 개발 현실에서는 원개발자가 없는 상황에서 단위 테스트를 수행해야하는 경우들이 발생한다. 앞서 언급한 NEIS 시스템의 재테스트의 경우에도, 이미 해체된 개발팀들을 재조직하기 어렵고 설계문서가 모든 단위들에 대하여 작성되어져 있지 않아, 결국 개발 대표 업체의 직원들을 동원하여 제 3자에 의한 단위 테스트가 수행되었다. 이처럼 프로젝트 종료이후에 이루어지는 추가 테스트의 경우가 제 3자에 의한 단위 테스트가 요구되는 상황이다. 또한 이미 테스트가 이루어진 제품이 업그레이드 등의 유지보수로 인한 제품 수정이 발생되면, 해당 제품에 대한 회귀 테스트(Regression Test)를 수행하게 된다. 회귀 테스트는 기존의 제품에 변화가 생겼을 때 수행해야 하는 테스트로서, 개발 단계에서 작성되었던 테스트 케이스들을 활용하는 것을 원칙으로 한다[10]. 그러나 개발문서에 테스트 케이스를 작성하지 않는 경우가 종종 있으며, 특히 단위 테스트를 위한 테스트 케이스는 테스트 엔지니어가 아닌 프로그램 개발자가 생성하게 되므로 테스트 케이스 문서가 남아있기 어려운 현실이다. 이러한 이유로 마이크로소프트사의 경우, 테스트 교육이 이수된 개발자를 SDET(Softwar Development Engineer in Test)이라 하여 우대하고 있다[8].

이렇게 원개발자가 없을 때 단위 테스트를 수행하기 어려운 상황과 더불어, 최근 많은 관심의 대상이 되는 애자일 개발에서의 단위 테스트의 역할은 더욱 커지고 있다. 이러한 문제에 대한 첫 접근 방법으로서, 단위 테스트를 제 3자에 의해 수행하는 경우에 대한 심도있는 분석이 요구되어진다.

본 논문은 규모있는 자바 어플리케이션의 소스코드를 대상으로 제 3자로서의 단위 테스트를 수행하고, 이를 단위 테스트 실제현황(Practice)으로 조사된 항목들[9]에 따라 구분하여 분석한다. 자바 소스코드 이외에 어떤 개발 산출물이 주어지지 않았으며, 화이트박스테스트를 수행하였다. 단위 테스트는 단위 모듈을 기반으로 개발자 중심으로 수행되므로, 코드의 구조를 기반으로 하는 구조 기반 테스트, 즉 화이트박스 테스트가 의미를 갖는다. 이는 블랙박스 테스트에 비하여 테스트 설계 과정이 복잡하고 오래 걸리는 비싼 테스트에 해당한다.

단위 테스트 실제현황으로부터 구성한 단위 테스트 요건과 본 실험을 통해 얻은 결과를 비교 분석하여, 단위 테스트가 원 개발자에 의해 이루어지지 못할 경우 생기는 어려움과 문제가 어디에 있는지를 파악하고, 각 단계 별로 원개발자와 비개발자 입장에서의 난이도를 표현한다. 이를 통하여 단위 테스트가 원개발자에 의해 개발도중 수행할 경우 얻는 잇점을 구체적으로 파악할 수 있다.

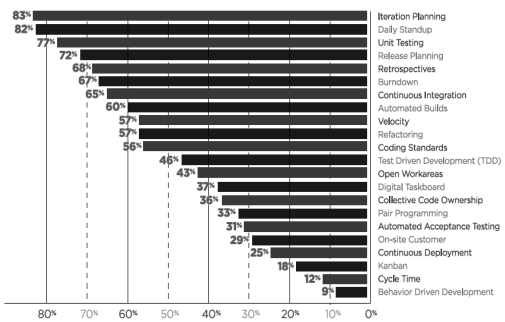
단위 테스트 실제현황 및 애자일 개발에 대하여 제 2장에서 설명하고, 제 3장에서는 본 연구에서 진행한 제 3자로서의 단위 테스트 실험을 보인다. 제 4장은 실험 결과를 단위 테스트 실제현황에 비추어 분석하고, 제 5장에서 본 연구의 결론을 맺는다.

# 2. 관련 연구

## 2.1 애자일 개발에서의 단위 테스트

‘애자일 개발’에 대한 관심이 커지고 있어, 가트너는 2012년까지 소프트웨어 개발 프로젝트의 80%가 애자일 개발 방법을 따를 것으로 예측하였다[5]. 국내 다양한 개발 업체들에서도 애자일 개발

문화로의 변화가 이루어지고 있다. 다음커뮤니케이션, NHN, 야후 코리아 등의 웹 기반 서비스 회사들을 중심으로 애자일에 대한 관심이 있었으나, 최근에는 게임 업계와 임베디드 업계, SI 업계 등에서도 애자일 도입에 많은 관심을 보이고 있다. 게임업계는 근래에 GDC(Game Developer Conference)에서 애자일을 적용해서 게임을 출시한 회사의 성공 사례 공유가 있었고, 그것이 촉매가 되어 국내 게임 회사에 스크럼이란 애자일 방법론이 퍼지게 되었다[6]. 포레스터 리서치의 2008년 2월 보고서(Enterprise Agile Adoption in 2007)[1]에 따르면, 북미 및 유럽 지역에서 약 1/4가량의 기업들이 애자일을 이미 도입했다. 그리고 애자일 도입의 속도가 가속화되고 있다. 이 보고서는 또한 기업의 규모가 클수록 애자일 더 많이 하는 경향이 있었다고 분석하였다. 예컨대 2만 명 이상의 직원을 갖은 기업의 경우 34%가 애자일을 도입하고 있다. 애자일은 프로젝트의 구체적인 개발 방법론이 아니다. 기존의 전통적인 개발 방법론이 가진 한계를 극복하고자 새롭게 나온 개발 방법론을 통칭하여 부르는 개발 프로세스의 이름이다. 애자일 프로세스에는 여러 방법론이 있는데, 각자의 특징이 있다. Scrum 같은 경우는 프로젝트 관리 방법론에 치중하고 있고, XP(eXtreme Programming)의 경우는 개발 기술에 치중하고 있다. 애자일 컨설팅 업체 VersionOne이 2009년 7월부터 12월까지 88개국 총 2570명을 대상으로 “어떤 애자일 기법을 적용하고 있습니까”를 조사 결과는 다음과 같다[12]. 이 조사에 언급된 애자일 기법들은 Scrum과 XP에서 제안하는 기법들이다. 이 가운데 관리 기법을 제외한 XP 중심의 개발 테크닉으로만 순위를 뽑아보면 1위는 단위 테스트이다. 애자일 개발에서의 “단위 테스트”는 개발자 입장에서 반드시 수행해야 한다. 모든 개발 단위들은 그에 대한 테스트 코드를 함께 가지고 있어야 하며, 이 테스트 코드는 요구사항 변화에 따른 추가 테스트에 사용되어 회귀 테스트 비용 절감에 기여할 수 있다. 또한 요구사항 변화에 민첩하게 대응하는 원동력이 된다.



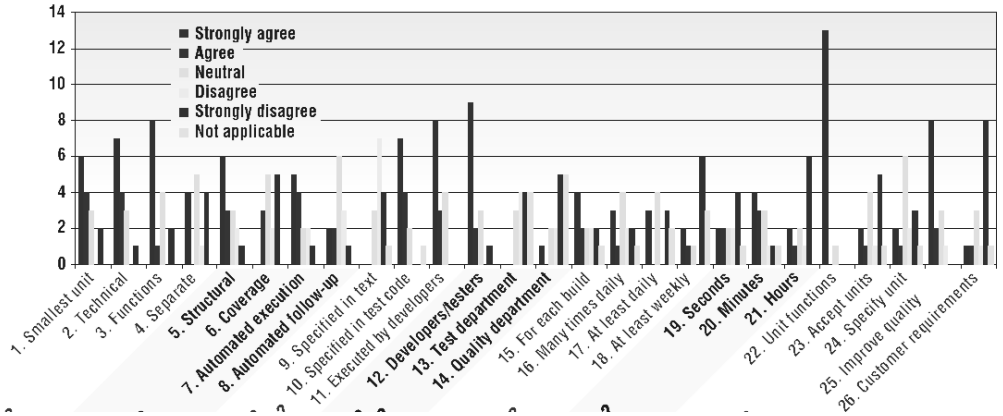

[그림 1] 애자일 기법의 사용 분포도[12]

‘안전중심(safety-critical) 소프트웨어’는 소프트웨어에 오류를 절대 허용할 수 없는 소프트웨어로서, 소프트웨어의 오류가 인간의 생명과 직결될 정도로 오류에 의한 부작용의 범위 및 치명도가 크다. 이는 단위 테스트부터 순차적으로 오류를 모두 찾으며 진행해야 하므로 기타 어플리케이션 개발 분야에 비하여 더욱 단위 테스트의 비중이 클 수 밖에 없다. 최근 안전중심 소프트웨어로 지정되는 것들에 대한 특별한 기술이 요구됨을 인식하여 관련된 품질 보증 기술에 대한 연구가 활발하게 진행되고 있는 현실이다.

이렇듯 단위 테스트에 특별한 관심을 보이는 두 분야, ‘애자일 개발’과 ‘안전중심 소프트웨어 개발’은 현재 소프트웨어 개발의 주요 흐름이다. 그러나 소프트웨어 규모가 커지고 GUI 중심의 개발이 진행되면서, 단위 테스트를 제대로 진행하지 않고 통합 및 시스템 테스트로 개발을 마무리 하는 경우가 많이 발생하고 있다. 개발 종료이후 오류가 발생하여 그 원인을 조사한 결과 단위 테스트가 제대로 시행되지 않았음을 밝혀내기도 했다.

## 2.2 단위 테스트 실제현황

2006년 SPIN(Software Process Improvement Network)에 속한 50여 개의 기업들을 대상으로 단위 테스트 실제현황 조사가 이루어졌다[9]. 참여한 기업들은 모두 소프트웨어를 주요 비즈니스 파트로 갖는 기업들로서, 1인 기업으로 컨설팅을 하는




[그림 2] 단위 테스트 실제현황 조사 결과[9]

작은 기업부터 수백 명의 개발자를 거느린 다국적 기업까지 그 규모가 다양하다. 연구 조사는 두가지 활동으로 이루어졌다. 하나는 '포커스 그룹 미팅'이고, 다른 하나는 '질문지를 이용한 인터뷰'이다. 포커스 그룹은 각 조직의 테스트 엔지니어와 품질 보증 전문가들로 구성하였습니다. 포커스 그룹 미팅은 각 주제에 대한 심도있는 토론의 장으로서 실제 개발 또는 테스트 관련 엔지니어들이 자신들의 경험을 바탕으로 토론을 진행하고 의견을 모으는 미팅으로 매달 3시간의 연속된 시간에 집중적으로 이루어졌다. 모아진 의견들을 검증하기 위하여, 포커스 그룹 미팅을 통해 도출된 의견들을 기반으로 만들어진 질문지를 다수의 SPIN 멤버들에게 공개하고 객관식 질문에 대한 답을 하도록 하였다. 이것이 두 번째 방법인 '질문지를 이용한 인터뷰'이다.

연구 조사의 내용은 크게 3가지로 구분되어 진행되었다. 첫째, 단위 테스트는 무엇인가? 둘째, 단위 테스트 관점에서 조사 대상자, 즉 나의 강점은 무엇인가? 즉, 무엇을 잘 하고 있는가? 셋째, 단위 테스트 관점에서 조사 대상자, 즉 나의 약점을 무엇인가? 즉 무엇을 못하고 있는가? 이 가운데 단위 테스트란 무엇인가에 대한 조사 결과를 중심으로 우리가 진행할 단위 테스트 실험을 구성하였다. 다음은 우리 실험을 구성하는데 적용된 조사 결과이다.

단위 테스트는 실험실에서 개발자에 의해 수행되는 테스트이며, 이는 프로그램이 설계 단계에서 정의된 요구사항을 만족시킴을 보여야 한다[7]. 포커스 그룹 미팅에서 이 정의에 반대하는 의견은 없었으며, 한 가지 이견이 있었던 부분은 단위 테스트 케이스를 개발자가 직접 명세해야 하는지의 문제이다. 다양한 도메인마다 서로 다른 의견을 제시하였다. 단위 테스트에 대한 이해를 조사하기 위한 질문지 인터뷰를 통하여 아래 몇 가지 특징적인 결과를 얻었다.

첫째, 단위 테스트는 프로그램 구조에 기반하여 이루어진다. 즉, 화이트박스 테스팅을 단위 테스트 기술로 적용한다. 따라서 테스트 케이스를 설계하기 위하여 테스트 선정기준을 사용한다.

둘째, 단위 테스트 케이스들이 반복되어져 사용되길 원하며, 이를 위한 테스트 자동화에 많은 관심을 갖고 있다. 따라서 테스트 케이스를 관리하며 실행을 도와주는 자동화 도구를 필요로 한다.

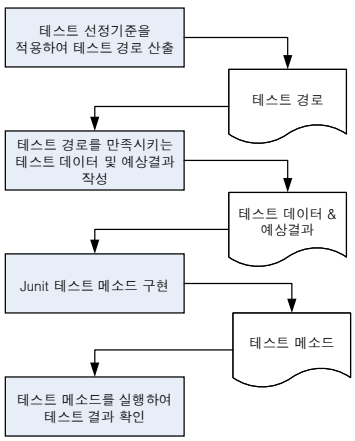
셋째, 단위 테스트 케이스가 테스트 코드로 작성되어지는 것을 원한다. 텍스트 문서에 작성되어진 테스트 케이스는 단위 테스트 실행을 반복하고 자동화하는데 걸림돌이 된다. 따라서 테스트 케이스가 테스트 코드로 작성되어질 것을 요구하고 있다.

위의 세 가지 특징을 고려하여 우리의 단위 테

스트 실험을 설계하였다.

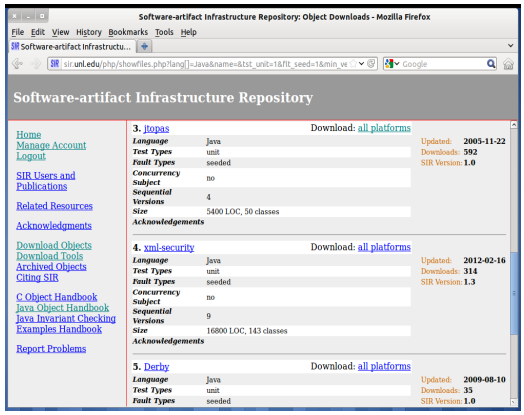
## 3. 원개발자 부재의 단위 테스트 수행

앞서 살펴본 연구 조사에 따르면, 단위 테스트는 개발자 이슈라는 것에 모든 기업이 동의하였다. 그러나 원개발자 없이 이루어지는 단위 테스트 현장이 존재할 수 있으며, 이때 원개발자가 아닌 개발자, 즉 테스트 대상이 되는 단위 코드에 대한 이해가 부족한 자를 본 논문에서는 '비개발자'로 명명하도록 하겠다. [그림 3]은 본장의 단위 테스트가 진행되는 흐름도이다. 다양한 수준의 테스트 선정기준을 적용하여 테스트 경로를 산출하고, 테스트 경로를 흘러갈 수 있도록 하는 테스트 데이터를 추출한다. 동시에 해당 테스트 데이터를 수행할 때 발생되는 올바른 예상 결과를 계산하여, 이를 JUnit 테스트 메소드로 구현한다. JUnit 테스트 메소드들은 JUnit TestRunner를 통해 실행되어 그 결과가 예상결과와 다르게 나올 경우 실패 메시지를, 또는 예상결과와 동일할 경우 성공 메시지를 내보낸다.




[그림 3] 단위 테스트 진행 흐름

본 실험은 규모있는 자바 오픈 소스인 JTopas를 대상으로 이루어졌다. 이는 SIR(Software-artifact Infrastructure Repository), 즉 네브라스카대학교에서 운영하는 사이트로서 많이 사용되고 규모있는 소프트웨어들의 소스코드를 관리하여 제공하는 사이트로부터 받은 소스코드들을 실험대상으로 선정하여 사용하였다. *JTopas*는 4개의 패키지, 5400 LOC(Lines Of Code)로 158개 메소드가 50개의 클래스에 구현되어져 있다. JTopas는 SIR에 등록되어져 있어, 테스트 실험에 주로 활용되는 소스 코드이다.



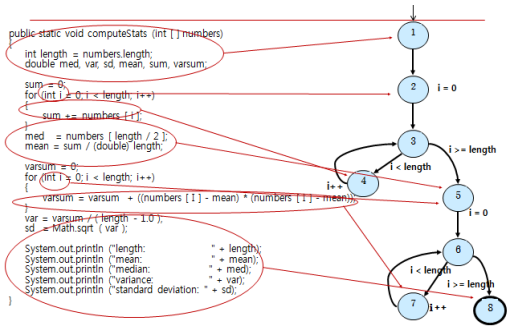

[그림 4] SIR의 JTopas

앞 장에서 도출한 단위 테스트의 특성 세 가지를 본 실험에 적용하였다. 첫째, 테스트 선정기준, 특히 구조기반 테스트 선정기준을 활용한 테스트 케이스 선정을 위하여, 본 실험에서는 *prime-path, branch, edge-pair,* 그리고 *all-use*를 적용하여 다양한 테스트 설계기준에 따른 테스트 케이스를 선정하고, 이를 실험의 테스트 케이스로 활용하였다. 둘째, 테스트 케이스를 텍스트 문서가 아닌 테스트 코드로 작성하기 위하여, 본 실험에서는 JUnit 라이브러리를 활용한 테스트 코드를 작성한다. 셋째, 테스트 실행의 자동화를 위하여, 본 실험에서는 JUnit 테스트 드라이버와 리눅스 쉘 스크립트를 활용하였다.

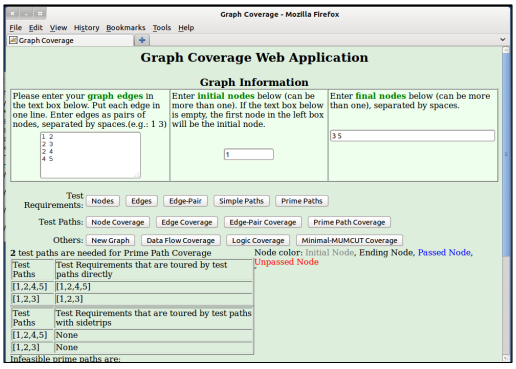
### 3.1 테스트 선정기준 적용

단위 테스트는 구조기반 테스트 기술을 활용한다. 소스 코드의 구조를 표현하기 위하여 제어흐

름도(CFG : Control Flow Graph)를 그린다. CFG는 하나의 문장을 하나의 노드로 표현하지 보다는 하나의 기본블럭(Basic Block)을 하나의 노드로 표현하여 그리는 그래프이다. “기본블럭”은 블록 내의 하나의 문장이 수행될 때 해당 블록의 모든 문장이 수행되는 특징을 갖는 최대 크기의 일련의 문장들, *a maximum sequence of program statements,* 을 의미한다[3]. 따라서 하나의 기본블럭은 하나의 진입점과 하나의 출구점을 갖는다. [그림 5]는 소스코드의 기본블럭으로 그린 흐름도의 예이다. 이는 제어구조를 표현하므로 제어흐름도이다.




[그림 5] 소스코드로부터 CFG 그리기[3]




[그림 6] 테스트 경로 생성 도구[4]

그려진 CFG를 노드 번호들로 표현하여 입력한 후, 원하는 선정기준을 표현하는 버튼을 클릭하면, 테스트 경로를 내어준다. 이때 입력된 CFG의 구조 자체에 오류가 있는지와 검증 작업도 함께 이루어지고, 만일 문제없는 CFG인 경우, 그래프를 그림으로 그려내고 테스트 경로를 산출해준다. 아래 도구를 통하여 유사하지만 분명 차이가 나는 선정기준의 적용 결과에 대한 오류를 최소화할 수 있으며, 시간 비용도 절약할 수 있었다.

위 도구를 통하여 얻은 테스트 경로를 지나가도록 테스트 케이스를 구성한다. 해당 경로를 지나갈 수 있도록 조건을 설정하고, 입력 값과 예상 출력을 찾아내어, 이를 테스트 케이스로 사용한다.

### 3.2 테스트 케이스를 테스트 코드로 구현

단위 테스트에 대한 연구 조사[9]에 따르면, 테스트 케이스는 텍스트 문서가 아닌 테스트 코드로 작성되어져야 한다. 본 실험에서는 테스트 코드의 실행을 염두에 두어 JUnit 코드로 작성하였다. 코드 작성을 위하여, 앞서 작성된 테스트 경로를 만족시키는 조건값과 입력값 그리고 예상출력을 찾아내는 일이 우선되어져야 한다.

```
public void test_setParseFlags_P_1245678() throws Throwable {
  Child_AbstractTokenizer Abs =new Child_AbstractTokenizer();
  Abs.setParseFlags(71);
  assertTrue((Abs.getParseFlags()==67)&&(Abs._lineNumber==0)
  &&(Abs._columnNumber==0));
}
```

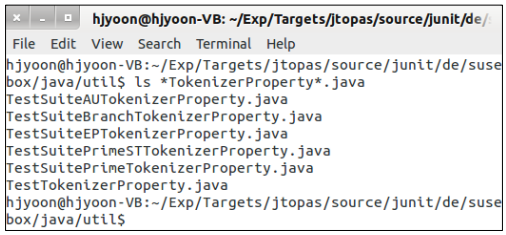
[그림 7] 테스트 코드의 예

예를 들어 setParseFlags() 메소드의 prime-path 선정기준에 의한 테스트 경로가 <1, 2, 4, 5, 6, 7, 8> 일때, 이를 지나가도록 하는 매개변수 flags의 값은 71이어야 하며, setParseFlags(71)의 실행 결과, _flags == 67, _lineNumber == 0, columnNumber == 0이 되어야 한다. 이를 테스트 코드로 작성하면 다음과 같다.

이때 클래스 단위로 테스트 코드를 관리하였고, 이들을 선정기준별로 JUnit의 *TestSuite*로 묶어서 개발하였다. 하나의 클래스에 속하는 메소드 별로 수행해야 하는 테스트 경로들을 해결하는 테스트

메소드들을 구현하였다. 만일 하나의 클래스에 m개의 메소드가 있고, 각 메소드마다 n개의 테스트 경로가 존재한다면, 클래스를 위한 JUnit 테스트 소스파일에는 m×n개의 테스트 메소드가 존재한다. 이때 각 선정기준 별로 수행해야 하는 테스트 경로가 다르므로, 선정기준 단위로 테스트 그룹을 구성하여 *TestSuite*로 구현하는 소스코드를 작성하였다. 따라서 하나의 클래스를 위한 테스트 소스 파일 하나와 우리의 5개의 선정기준들을 위한 각각의 Test Suite 파일 5개가 구현되어진다.

[그림 7]은 JTopas의 de.susebox.java.util 패키지의 TokenizerProperty 클래스를 위한 JUnit 테스트 소스파일 구성을 보여준다. 가장 아래줄의 TestTokenizerProperty.java에 메소드들의 테스트 경로를 실행하는 JUnit 메소드들을 구현하고, 이들 테스트 메소들을 각 선정기준마다 *TestSuite*로 구현한 테스트 코드 파일이 존재한다. 예를 들어, Branch 선정기준을 위한 *TestSuite*는 TestSuiteBranchTokenizerProperty.java 파일 내에 구현되며, 이는 *TestSuite* 하나를 생성하고 TestTokenizerProperty.java에 구현된 테스트 메소드들 가운데 Branch 선정기준으로 선정된 테스트 경로를 실행하는 메소드들을 테스트 그룹, 즉 *TestSuite*에 추가시킨다. 따라서 만일 Branch 선정기준을 적용한 결과를 알고 싶다면, 우리는 TestSuiteBranchTokenizerProperty 클래스를 실행시키게 된다.




[그림 8] 테스트 코드 소스파일 구성

### 3.3 JUnit과 쉘스크립트 기반의 테스트 자동화

본 실험은 쉘스크립트를 활용하기 위하여 리눅스(Ubuntu 10.11) 시스템에서 이루어졌다. 이는 테스트 코드가 작성되어지면, *TestSuite* 테스트 코드를 호출하여 각 단위 메소드를 대상으로 테스트를 실행한다. 이때 *TestRunner*를 *TextUI*로 지정하여 테스트 결과를 텍스트 파일로 저장시켜둔다. 이때 테스트 코드 실행 결과는 *outputs.alt* 폴더에 모으고, 정상적인 예상 결과값은 *outputs*폴더에 저장시킨다. [그림 9]는 해당 부분에 대한 쉘프로그램이다.

```
56 echo ">>>>>>>>Branch"
57 java junit.textui.TestRunner $1.$TS_B  > /home/hjyoon/Exp/Targets/jtopas/
   outputs.alt/branch 2>&1
58 /home/hjyoon/Exp/Targets/jtopas/testplans.alt/testscripts/RemoveTime.sh
   outputs.alt branch
59
60 echo ">>>>>>>>PrimePath"
61 java junit.textui.TestRunner $1.$TS_P  > /home/hjyoon/Exp/Targets/jtopas/
   outputs.alt/prime 2>&1
62 /home/hjyoon/Exp/Targets/jtopas/testplans.alt/testscripts/RemoveTime.sh
   outputs.alt prime
63
64 echo ">>>>>>>>PrimePath with SideTrip"
65 java junit.textui.TestRunner $1.$TS_Ps > /home/hjyoon/Exp/Targets/jtopas/
   outputs.alt/primeST 2>&1
66 /home/hjyoon/Exp/Targets/jtopas/testplans.alt/testscripts/RemoveTime.sh
   outputs.alt primeST
67
68 echo ">>>>>>>>EdgePair"
69 java junit.textui.TestRunner $1.$TS_EP  > /home/hjyoon/Exp/Targets/
   jtopas/outputs.alt/EP 2>&1
70 /home/hjyoon/Exp/Targets/jtopas/testplans.alt/testscripts/RemoveTime.sh
   outputs.alt EP
71
72 echo ">>>>>>>>AllUse"
73 java junit.textui.TestRunner $1.$TS_AU > /home/hjyoon/Exp/Targets/jtopas/
   outputs.alt/AU 2>&1
74 /home/hjyoon/Exp/Targets/jtopas/testplans.alt/testscripts/RemoveTime.sh
   outputs.alt AU
```

[그림 9] 테스트 코드 실행 스크립트

일단 테스트 코드 실행 결과가 *outputs.alt* 폴더에 저장되면, 그것이 예상결과와 일치하는지를 판단하여, 만일 일치한다면, 해당 테스트 코드는 오류를 감지하지 못한 것이고, 만일 일치하지 않는다면, 대상 단위 메소드에 오류가 존재하는 것으로 판단할 수 있다. 따라서 테스트 코드 실행 결과에 대한 비교 과정이 요구된다. JUnit의 *TextUI*에 의한 테스트 러너는 테스트 코드 실행 결과를 각 메소드 단위로 점(.) 또는 E, F로 표시한다. 앞의 스크립트에 의해 이 내용이 *output.alt* 폴더의 파일에 저장되어져 있으므로, 이 파일의 내용과 미리 작성된 예상 결과를 갖는 *output* 폴더의 파일들을 “diff” 명령어로 비교하여 해당 테스트 코드가 오류를 감지하는지를 판단한다. 이 과정에 대한 쉘프로그램은 [그림 10]과 같다.

```
76 echo "3. Comparing... two kinds of outputs"
77 diff --brief ${experiment_root}/jtopas/outputs/branch ${experiment_root}/
   jtopas/outputs.alt/branch
78 if ((S?))
79 then let branch+=1
80 fi
81 diff --brief ${experiment_root}/jtopas/outputs/prime ${experiment_root}/
   jtopas/outputs.alt/prime
82 if ((S?))
83 then let prime+=1
84 fi
85 diff --brief ${experiment_root}/jtopas/outputs/primeST ${experiment_root}/
   jtopas/outputs.alt/primeST
86 if ((S?))
87 then let primeST+=1
88 fi
89 diff --brief ${experiment_root}/jtopas/outputs/EP ${experiment_root}/
   jtopas/outputs.alt/EP
90 if ((S?))
91 then let ep+=1
92 fi
93 diff --brief ${experiment_root}/jtopas/outputs/AU ${experiment_root}/
   jtopas/outputs.alt/AU
94 if ((S?))
95 then let au+=1
96 fi
97
98 let i+=1
99 done
```

[그림 10] 테스트 실행 결과 비교 스크립트

쉘스크립트를 실행하여 JTopas에 대한 테스트를 실행한 결과, 각 선정기준별로 다음과 같은 결과를 얻을 수 있었다.

| 클래스 | 오류를 갖는 프로그램 버전의 수 | 테스트 코드에 의해 발견한 오류의 수 | | | | |
|---|---|---|---|---|---|---|
| | | Branch | PrimePath | PrimePath with Side Trip | EdgePair | AllUse |
| ExtIOException | 24 | 16 | 16 | 16 | 16 | 13 |
| ExtIndexOutOfBoundsException | 27 | 16 | 16 | 16 | 16 | 13 |
| ExtRuntimeException | 25 | 16 | 16 | 16 | 16 | 13 |
| InputStreamTokenizer | 11 | 8 | 8 | 8 | 8 | 5 |
| Token | 47 | 37 | 37 | 37 | 37 | 33 |
| TokenizerException | 26 | 17 | 17 | 17 | 17 | 15 |
| TokenizerProperty | 14 | 0 | 0 | 0 | 0 | 0 |
| PluginTokenizer | 72 | 5 | 47 | 10 | 11 | 9 |
| InputStreamSource | 12 | 10 | 10 | 10 | 10 | 6 |

[그림 11] JTopas 단위 테스트 실행 결과

## 4. 단위 테스트 수행 과정 분석

본 실험은 원개발자가 아닌 제 3차, 즉 해당 단위 코드에 대한 이해가 없는 상태의 테스트 참여자, 가 규모 있는 자바 오픈 소스를 대상으로 단위 테스트를 진행하는 실험이다. 이때 전제하였던 환경으로 단위 테스트의 실제현황에 대한 연구 조사에서 도출한 세 가지 조건을 설정하였으며, 각 조건에 따라 제 3자에 의한 단위 테스트를 진행한 내용을 제 3장에 기술하였다. <표 1>은 단위 테스트 실제현황에서 도출된 세 가지 조건과 그에 적용되는 본 논문의 단위 테스트 구성 내용을 나타낸다.

<표 1>과 같이 구성된 제 3장의 단위 테스트 경험을 가지고, 두 가지 관점, 즉, 각 항목을 수행할 때 소요되었던 인력 및 시간, 그리고 원개발자가 아닌 제 3자로서 수행하였기에 생겼던 추가노력 여부를 구분하여 <표 2>로 분석하였다.

〈표 1〉 단위 테스트 실제현황 특성에의 적용

| 항목 | 단위 테스트 실제현황 | JTopas 단위 테스트 |
|---|---|---|
| 1 | 테스트 코드 작성 | JUnit 코드 작성 |
| 2 | 테스트 선정기준에 의한 테스트 케이스 명세 | CFG 기반 선정기준 적용 |
| | 구조기반 테스트 설계 | |
| 3 | 테스트 실행 자동화 | JUnit 테스트 드라이버 활용 |
| | | 테스트 실행 쉘스크립트 작성 |

〈표 2〉 원개발자 부재의 단위 테스트 노력

| 항목 | ManMonth | 추가노력 여부 |
|---|---|---|
| 1 | 2.8mm | 있음 |
| 2 | 0.8mm | 없음 |
| 3 | 0.2mm | 없음 |

<표 2>의 *ManMonth*는 하루 8시간 시간, 한달 20일을 기준으로 산출하였다. 항목 번호는 <표 1>에 근거를 두고 있다. 항목 1은 한명의 연구원이 주 2일 작업을 하고 7개월의 시간이 걸려 진행되었다. 따라서 테스트 코드 작성에 소요한 작업 일수 56일에 대한 *ManMonth*, 즉 56÷20 = 2.8이 산출되었다. 항목 2 또한 동일한 한명의 연구원이 진행하였고 같은 조건에서 2개월의 시간이 소요되었다. 따라서 16÷20 = 0.8로 산출되었다. 항목 3은 작업이 복잡하고 어렵지 않아 4일의 작업시간이 소요되었고, 이를 *ManMonth*로 산출하면 4÷20 = 0.2가 된다. 수치로 확인되는 바대로, 테스트 코드 작성에 가장 많은 노력이 소요됨을 알 수 있다.

본 논문은 원개발자가 없는 상황에서의 단위 테스트를 분석하고 있다. 따라서 각 항목들에 대하여 원개발자 부재로 인한 추가 노력을 분석할 필요가 있다. 항목 1, 즉 'JUnit 코드 작성'은 (1)주어진 테스트 경로가 실행될 수 있는 테스트 데이터

를 찾아 코드로 작성하고, (2) 해당 경로 실행 결과가 예상한 결과와 같은지를 판단하는 문장을 작성하여야 한다. 진행한 단위 테스트는 소스코드의 구조를 기반으로 하는 선정기준들을 적용하여 테스트 경로를 생성하므로 소스코드 구조에 익숙하지 않은 경우, 주어진 경로를 지나가도록 하기 위하여 어떤 입력값을 갖도록 해야 하는지 판단하기 어렵게 된다. 따라서 (1)의 입력이 되는 테스트 데이터를 찾기 위하여 대상이 되는 단위메소드는 물론 논리적으로 연결되어진 다른 메소드들의 구조를 모두 파악해야 하며, 이 때 많은 시간이 소요되었다. 예를 들어 AbstractTokenizer 클래스의 nextToken() 메소드의 경우, *prime-path* 선정기준의 경우 122개의 테스트 경로가 산출된다. 경로의 수도 많지만, 경로의 복잡도도 크다. 선정된 경로 가운데 하나인 <1, 2, 8, 9, 11, 12, 14, 15, 16, 1, 2, 3, 4, 5, 11, 14, 15, 16, 17>의 경우, 분기와 반복 구조를 동시에 갖고 있어 이 경로를 지나도록 하는 입력환경을 구축하기 쉽지 않다. 이는 만일 원개발자였다면 본인이 개발한 코드의 구조는 물론 어느 변수가 어디에 영향을 미치는지에 대한 체계를 인식하고 있으므로 이 부분을 무리없이 진행할 수 있다. (2)의 예상 결과를 산출하는 부분이 제 3자로서의 단위 테스트에서 가장 어려웠던 부분이다. (1)의 테스트 입력 값은 입력 매개변수들을 중심으로 찾게 되면 경우의 수가 제한적이다. 그러나 (2)의 경우, 예상출력을 알아야 하는데, 많은 단위메소드의 경우 명확한 반환값이 아닌 “void”형의 반환값으로 정의되어져 있다. [그림 12]의 setParseFlags() 메소드가 여기에 해당한다. 즉, 메소드의 결과가 하나의 값으로 표현되지 않는 경우이다. 이러한 void형 단위 메소드의 경우에는 주어진 테스트 경로가 수행하면서 영향을 미치는 전역변수의 값을 분석하여 수행결과로 설정하고 이를 실제 실행결과와 비교하는 테스트 코드를 작성하여야 한다. (2) 과정 또한 원개발자의 경우 보다 쉽게 접근할 수 있는 부분이 된다. 이처럼, 소스코드의 전체 구조와 변수들의 의미를 파악하고 있을 때, (1)과 (2)의 작업이 가능해지며, 따라서 단위 테스트를 개발자가 수행해야 하는 테스트 레벨로 정의하는 이유가 여기에 있다.

```
public void setParseFlags(int flags) {
    _flags = flags;
    
    // we would like to test flags on F_KEYWORDS_CASE
    if ((_flags & (Tokenizer.F_KEYWORDS_NO_CASE | Tokenizer.F_NO_CASE)) == 0) {
      _flags |= Tokenizer.F_KEYWORDS_CASE;
    } else if ((_flags & Tokenizer.F_KEYWORDS_CASE) != 0) {
      _flags &= ~Tokenizer.F_KEYWORDS_NO_CASE;
    }
    
    // when counting lines initialize the current line and column position
    if ((_flags & Tokenizer.F_COUNT_LINES) != 0) {
      _lineNumber   = 0;
      _columnNumber = 0;
    }
  }
```

[그림 12] void형 단위메소드의 예

<표 2>의 항목 2와 항목 3에서는 원개발자가 아닌 제 3자로서 요구되는 추가노력이 없다. 항목 2의 경우 소스코드에 대한 CFG를 그려야 하는데, CFG와 같은 테스트 모델은 개발과정의 설계 모델과는 독립적으로 다시 그려져야 한다[11]. 이는 소스코드 정보만을 가지고 모델을 구축하는 것으로서, 소프트웨어 재공학(Reengineering)과 관련되어진다. 일반적인 재공학은 설계 모델의 부재로 인해 이루어지므로 만일 개발 산출물로서 설계모델이 존재하는 경우라면 재공학의 절차가 필요하지 않을 수 있다. 그러나 본 연구 내용인 테스트를 진행하기 위해서는 개발 산출물로서의 모델과는 반드시 독립적인 테스트 모델이 요구된다. 개발과정의 모델을 테스트 모델로 활용할 경우, 개발 과정으로 오류를 테스트를 통하여 감지하기 어려워지기 때문이다. 테스트 모델의 독립성[11]은 개발과정에서의 오류를 감지해야 하는 테스트의 목적을 고려할 때 반드시 보장되어져야 하는 부분이다. 따라서 본 실험에서의 항목 2는 원개발자와 비개발자 입장에서 모두 테스트 모델을 작성해야하므로, 비개발자로서 수행하는 테스트에 추가노력이 요구되는 부분이 아니다. 그러므로, 원개발자인 경우에도, 소스코드를 제 2장에서 설명한대로 기본블럭

으로 나누어 CFG를 그리는 작업이 요구된다. 일단 CFG가 그려지면, 제 3장에서 소개한 도구 등에 입력하여 테스트 경로를 생성하게 된다. 따라서 항목 2에 대한 추가노력은 없음으로 표현할 수 있다.

항목 3은 테스트 실행을 지원하는 테스트 실행 명령어들의 스크립트를 작성하는 과정이다. 자동화 부분은 테스트 과정에 대한 이해와 리눅스 쉘스크립트 개발에 대한 정보가 있는 경우, 소스 코드에 대한 이해없이 진행할 수 있었다. 이는 소스코드의 구조에 대한 정보가 필요하지 않음으로, 제 3자로서의 추가노력이 없는 부분이다.

본 실험의 한계점은 수행한 연구원의 역량의 영향을 배제하지 못한 부분이다. 참여한 연구원은 컴퓨터공학을 전공하고 테스트 관련 자격증인 CSTS (Certified Software Test Specialist)를 소지하고 있으며, 본 실험의 제 3자로서의 단위 테스트를 진행하였다. 따라서 향후 테스트수행 인력의 능력에 따른 영향력 분석도 추가로 요구된다.

## 5. 결 론

단위 테스트에 대하여 모든 개발자가 동의하는 명확한 특성은, “단위 테스트는 개발자가 직접 진행해야 한다.”는 것이다. 그러나 서론에서 언급한 여러 정황상, 소스코드의 원개발자가 단위 테스트를 진행할 수 없는 경우가 존재하게 된다. 본 논문은 원개발자 부재 상황에서의 단위 테스트 노력을 분석하였다. 이는 기존의 테스트 노력 산정[2]과는 다른 관점을 갖는다. 테스트 노력 산정 기술은 물리적인 테스트 대상 코드의 복잡도를 기반으로 어느 정도의 노력이 들지를 예측산정하는 기술이다. 여기에는 누가 테스트를 하느냐에 대한 요소가 배제되어진다. 원개발자없이 진행되는 단위 테스트의 문제점을 파악하기 위하여, 4개의 패키지에 5400 LOC를 갖으며, 158개 메소드가 50개의 클래스에 구현되어져 있는 자바 오픈소스를 대상으로 직접 비개발자 입장의 단위 테스트를 수행하였다. 대상이 된 JTopas는 자바언어의 다양한 기술들이 적용된 코드로서 여러 테스트 관련 논문들에서 실험 대상으로 이용하는 어플리케이션이다.

실험은 이미 연구된 단위 테스트 실제현황의 각 항목을 만족하도록 구성하였으며, 그 결과 테스트 코드를 작성하여 테스트 실행 자동화를 구현하는 항목에 대한 노력이 가장 크게 산출되었으며, 그 부분이 또한 원개발자 부재로 인한 추가노력으로 분석되었다. 이러한 결과는 다음의 결론으로 정리할 수 있다.

첫째, 원개발자는 단위 테스트 관련 문서를 작성할 필요가 있다. 개발 문서 가운데 테스트 계획서, 설계서, 결과보고서 등의 산출물이 존재하게 된다. 그러나 이는 대부분 테스트 엔지니어에 의해 작성되어지므로 통합테스트와 시스템테스트를 대상으로 하는 문서이다. 앞서 서론에서 언급한 NEIS 개발문서의 경우에도 통합테스트와 시스템 테스트에 대한 문서는 정확하게 존재하였다. 그러나 유지보수 관점에서, 회귀테스트 또는 원개발자 부재를 대비한 단위 테스트 설계 문서가 요구된다. 테스트 설계 문서에는 일반적으로 테스트 케이스를 기록한다. 따라서 테스트 케이스를 다시 작성하는 수고를 덜어주며, 회귀테스트 수행의 경우, 변경된 부분의 테스트 케이스만을 골라 사용할 수 있는 편의를 제공한다.

둘째, 애자일 개발의 경우 <표 2>의 항목 1과 같이, 원개발자 부재로 인한 추가노력이 요구되는 문제를 해결할 수 있다. 애자일 개발을 구현하는 XP의 기술 가운데 TDD(Test Driven Development)를 적용하면, 모든 단위 메소드의 구현 전에 반드시 그를 지원하는 테스트 코드를 우선 작성하게 된다. 따라서 단위 메소드마다 해당되는 테스트 코드가 구현되며, 테스트 코드는 테스트 케이스를 담고 있다. 테스트 설계 문서의 테스트 케이스를 구현하는 테스트 코드 작성이 단위 메소드 구현의 전제조건이 된다.

단위 테스트의 경우 이와 같이 개발자의 역할이 크게 작용되므로, 개발자들이 테스트 설계 기술을 반드시 습득하여 문서화 또는 테스트 코드 작성을

할 필요가 있다. 앞서 언급한 애자일의 경우 개발자가 테스트 코드를 작성해야 하므로 더욱 그러하다. 특히 구조기반 테스트, 즉 화이트박스테스트를 진행해야 하는 단위 테스트를 테스트 케이스 설계 과정 없이 진행하기에는 무리가 있다.

결국, 애자일 개발과 같이, 테스트 코드 작성이 개발자에 의하여 개발단계에 이루어진다면, 원개발자없이 진행됨으로써 추가노력이 발생하는 '테스트 코드 작성'이 해결되므로, 원개발자 부재로 인한 테스트 노력 비용 증가를 완화하는데 기여할 수 있다.

## 참 고 문 헌


[1] 포레스터 리서치, Enterprise Agile Adoption in 2007, 2008.

[2] Abhishek, C., V. P. Kumar, et al., "Test Effort Estimation Using Neural Network", *Journal of Software Engineering and Applications*, Vol.3, No.4(2010).

[3] Ammann, P. and J. Offutt, *Introduction to Software Testing*, 2008.

[4] Companion Software, Graph Coverage Web Application, "http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage", 2007.

[5] Gartner Group, Predicts 2010 : Agile and Cloud Impact Application Development Directions. http://www.gartner.com/DisplayDocument?id = 1244514.

[6] Kieth, C., "Agile Game Development", GDC proceeding, 2007.

[7] Koomen, T. and M. Pol, *Test Process Improvement-A Practical Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999.

[8] Page, A. and K. Johnston, *How We Test Software at Microsoft*, Microsoft Press, 2008.

[9] Runeson, P., "A Survey of Unit Testing Practices", *IEEE Software*, Vol.23, No.4(2006), pp.22-29

[10] Sommerville, *Software Engineering* 8th edition, Addison-Wesley, 2007.

[11] Utting, M., A. Pretchner, and B. Legeard, "A taxonomy of model-based testing approaches", *Software Testing, Verification and Reliablility*, Vol.22, No.5(2012), pp.297-312.

[12] VersionOne, 3rd Annual Survey : State of Agile Development Survey, http://pm.versionone.com/whitepaper_AgileSurvey2008.html, 2009.

## ◈ 저 자 소 개 ◈

**윤 회 진 (hjyoon@uhs.ac.kr)**

이화여자대학교 전자계산학과를 졸업하고, 동대학교 대학원에서 컴퓨터학과에서 이학석사 및 공학박사를 취득하였다. 이후 Georgia Institute of Technology에 방문연구원으로 연구를 하였으며, 이화여자대학교 컴퓨터학과 전임강사로 일하였다. 현재 협성대학교 컴퓨터공학과 조교수로 재직 중이다. 관심분야는 소프트웨어테스트, 뮤테이션테스팅, 애자일개발 등이다.