단위 테스트 자동화를 위한 자바 프로그램 테스트 코드 구축

(Building Test Codes for Unit Test Automation of Java Programs)

윤 회 진‡

(Hoijin Yoon)

요 약 애자일 개발의 XP와 Scrum을 중심으로 단위 테스트 자동화의 중요성이 커지고 있다. 그러나 테스트 결과, 즉 통과 또는 실패를 자동으로 결정하기 위해서는 테스트 실행 결과와 예상 결과를 비교하는 과정이 필요하다. 이 부분의 구현이 자동화의 성패를 좌우한다. 본 연구는 단위 테스트 자동화를 위한 테스트 코드 작성을 소개하고, 테스트 코드 구현에서 고려해야할 사항을 언급한다. 첫째, void 형태의 메소드의 경우 테스트 데이터 실행 결과를 명시적으로 구하기 어려운 문제를 본 연구에서는 Mock 프레임워크를 사용하여 해결하였다. 둘째, void 형태의 메소드의 경우, criteria로 인해 구성된 테스트 경로상의 모든 문장들이 제대로 수행되었는지 하나씩 살펴보아야 하는지, 아니면 최종 문장에 대해서만 보아야 하는지의 문제이다. 본 연구에서는 Mock 프레임워크의 verify 기능을 활용하여 매 순간 제대로 실행되어져야 하는 메소드 호출을 중심으로 명확한 매개변수들을 사용하여 호출이 일어났는지를 확인하고, 그 결과들이 모두 예상 결과와 맞을 때, 해당 테스트 케이스에 대한 테스트를 통과한 것으로 결정하였다.

키워드 단위 테스트, 테스트 스텁, Mock 프레임워크, 테스트 자동화, 테스트 주도 개발

Abstract Agile development is mentioned a lot by developers these days. XP or Scrum is one of the popular development processes, and it says that unit test automation would drive an agile development successful. The success of unit test automation depends on how well to compare an execution result to its own expected result. that is why this paper focuses on the comparison part. This paper introduces how to build test codes for unit testing, and then concludes with mentioning two considerations of unit testing automation. First, test codes for void-typed methods need Mock Framework to monitor their behavior. Second, the comparison of execution results and expected results is hard to implement in case of testing void-typed methods. We check every sentences of a test path to decide if the testing is fail or pass.

Key words Unit testing, Test stub, Mock Framework, Test Automation, Test Driven Development

1. 서 론

포레스터 리서치의 2008년 2월 보고서 (Enterprise Agile Adoption in 2007)[1] 에 따르면, 북미 및 유럽 지역에서 약 1/4 가량의 기업들이 애자일을 이미 도입했다. 그리고 애자일 도입의 속도가 가속화되고 있다. (“Enterprise Agile adoption has accelerated, increasing approximately two and a half times faster between 2006 and 2007 than between 2005 and 2006.”) 이 보고서는 또한 기업의 규모(직원수)가 클수록 애자일 더 많이 하는 경향이 있었다고 분석하였다. 예컨대 2만명

‡ 정 회 원 : 협성대학교 컴퓨터공학과 조교수
hjyoon@uhs.ac.kr
논문접수 : 2010년 11월 2일
심사완료 : 2010년 12월 20일

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. Scum 같은 경우는 프로젝트 관리 방법론에 치중하고 있고, XP의 경우는 개발 기술에 치중하고 있다. 애자일 컨설팅 업체 VersionOne이 2009년 7월부터 12월까지 88개국 총2570명을 대상으로 "어떤 애자일 기법을 적용하고 있습니까"를 조사 결과는 다음과 같다 [2]. 이 조사에 언급된 애자일 기법들은 Scrum과 XP에서 제안하는 기법들이다.

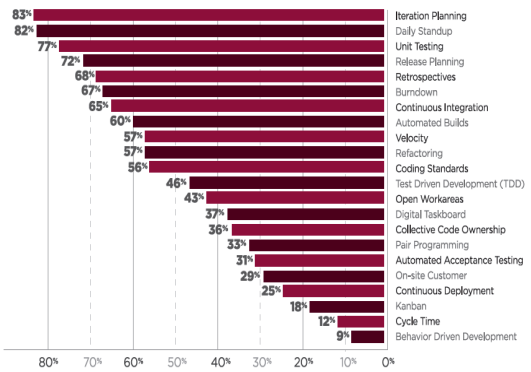

그림 1. 애자일 기법의 사용 분포도 [2]

이 가운데 관리 기법을 제외한 XP중심의 개발 테크닉으로만 순위를 뽑아보면 다음과 같다.

1위: 단위 테스트 (Unit testing)
2위: 지속적 통합 (Continuous Integration)
3위: 자동화된 빌드 (Automated Builds)
4위: 테스트 주도 개발 (TDD)

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. 위의 4위에 해당하는 TDD에서는 특히 테스트 코드를 우선 개발하여야 한다. 이는 테스트를 실행하기 위한 드라이버와 스텁이 모두 갖추어진 테스트 코드를 의미한다. 따라서 테스트 스텁 구현이 필요하다.

본 연구에서는 애자일 등의 최근 개발 방법들의 기법들이 기반으로 하는 단위 테스트 자동화를 위한 테스트 스텁을 구현하고, 이때 실제 개발 현장에서 많이 사용되는 Mock 프레임 워크와 JUnit을 활용한다. 2장에서는 테스트 스텁과 Mock 프레임 워크에 대하여 살펴보고, 3장에서는 실제 JMemorize 프로그램에 대한 테스트 코드를 구현한다. 4장에서는 테스트 자동화를 위한 코드 작성에서 고려해야 할 부분에 대하여 분석하고 5장에서 본 연구에 대한 결론을 언급한다.

2. 관련연구

2.1 테스트 드라이버와 테스트 스텁 (Test Stub)

테스트 드라이버는 테스트할 모듈을 제어하고 동작시키는데 사용된다. 드라이버의 가장 단순한 예들 중 하나는 순차적으로 실행되는 프로그램이나 명령들의 목록인 배치파일이다. 과거의 테스터들은 테스트 모듈의 이름을 포함하는 배치 파일을 만들고 배치 실행을 시작한 다음, 퇴근했다. 오늘날은 xUnit 프레임워크의 등장으로 테스트 드라이버 작성에 큰 도움이 되고 있다. xUnit 프레임워크는 표 1과 같이 다양한 프로그래밍 환경을 도메인으로 맞춤되어져 개발되어져 있다.

표 1. xUnit 프레임워크 기반의 테스트 도구들 [3]

xUnit	프로그래밍언어	관련사이트
CUnit	C	http://cunit.sourceforge.net
CppUnit	C++	http://sourceforge.net/projects/cppunit
PHPUnit	PHP	http://www.phpunit.net
PyUnit	Python	http://pyunit.sourceforge.net
DbUnit	DataBase	http://dbunit.sourceforge.net
utPLSQL	PL/SQL	http://utplsql.sourceforge.net
...	...	...

테스트 드라이버 개념이 xUnit 프레임워크에 포함되어져 있다. xUnit은 테스트 모듈들을 일련의 규칙에 따라 실행하는 코드를 자동 생성시켜줌으로써, 테스트 드라이버 코드 개발을 자동화하고 있다. 테스트 드라이버와 반대 개념의 테스트 스텁은 개발자가 테스트 코드로 개발해주어야 한다. 특히 TDD의 경우 테스트 코드를 미리 작성해야 하므로 테스트 스텁 구현이 필수적이다. 특히 아직 다른 모듈과의 통합이 이루어지지 않은 시기의 단위 테스트에서는 테스트 스텁 코드 구현을 위해 더미(dummy) 모듈을 구현하여 테스트 스텁으로 사용하기도 한다.

Gerard Meszaros의 xUnit Test Pattern[4]에는 "Test Double"이라는 것을 설명하고 있다. 이는 테스트를 수행하기 위해서 실제 모듈 역할을 대체하는 기능을 가진 객체나 컴포넌트를 말한다. 대역의 의미를 갖는 Double이라는 용어를 사용한다. 그림 2의 Test double의 분류에서, Mock Object는 Dummy, Stub, Spy등을 통합해 놓은 것으로 볼 수 있다. Mock은 테스트 스텁의 기능에 검증(Assertion)기능을 추가한 형태로서, 테스트 실행 입장에서 더 의미가 있다.

본 논문에서는 Mock Object를 활용하여 테스트 스텁을 구성하고 동시에 검증하는 환경을 구현하여, 테스트 드라이버 기능을 제공하는 JUnit에서의 활용 방법을 보인다.

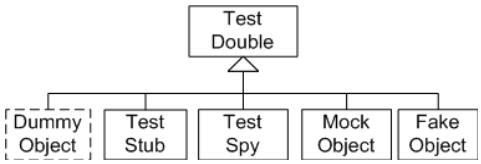

그림 2. Test Double의 분류[4]

2.2 Mock 프레임워크

테스트 더블의 기능을 구현하기 위한 도구로 Mock 프레임워크를 사용할 수 있다. 자바 기반의 Mock 중에서 많이 사용되고, 업데이트가 잘 이루어지고 있는 것에는 EasyMock, JMock, Mockito가 있다. 세계적인 랭킹을 보면, EasyMock과 jMock이 1,2등을 다툰다. 특히 EasyMock의 경우 만들어진지가 오래됐고, 많은 프레임워크에서 EasyMock을 사용했었다. 그리고 Mockito는 역사는 오래되지 않았지만 간편함으로 인해 다크호스로 떠오르고 있다.

Mokito를 개발한 수제빵 파베르(Szczepan Faber)가 설명하는 Mockito의 차별점은 다음과 같다. 첫째, 테스트 그 자체에 집중한다. 둘째, 테스트 스텁을 만드는 것과 검증을 분리시켰다. 셋째, Mock만드는 방법을 단일화했다. 넷째, 테스트 스텁을 만들기 쉽다. 다섯째, API가 간단하다. 여섯째, 프레임워크가 지원해주지 않으면 안되는 코드를 최대한 배제했다. 마지막으로 일곱째, 실패시에 발생하는 에러추적이 깔끔하다. 쉽게 말하면, Mockito는 EasyMock과 jMock의 단점을 보완하기 위해 나온 Mock 프레임워크라고 생각하면 되겠다. 본 연구에서는 기존 기술의 단점을 보완하여 개발된 Mockito를 이용한다.

3. Mockito를 활용한 테스트 스텁 구현

본 절에서는 Opensource로 공개된 JMemorize 소스코드를 단위 테스트한다. 이때 테스트 드라이버의 기능을 하는 jUnit 4와 테스트 스텁으로 Mockito 1.8.2를 이용한다.

3.1 JMemorize

본 연구에서는 자바 오픈소스 프로그램인 JMemorize를 단위 테스트 하기 위한 자동화 환경을 JUnit과 Mokito를 이용하여 구현한다. JMemorize는 플래쉬 카드 방식의 암기 프로그램으로, 라이트너 학습 암기 방식으로 만들어져 유명하다. 인터페이스는 영어와 일어, 에스페란토 등을 지원하고 있으며, 그림 3과 같이 외워야 하는 내용과 그에 대한 암기 기록을 보여주고 있다. 이는 자바 언어로 작성되어져 있다.

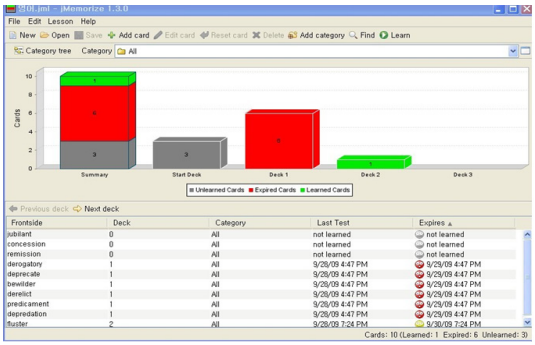

그림 3. JMemorize

이 프로그램은 125개의 클래스로 이루어져 있으며, 각 클래스에 30개 전후의 메소드가 구현되어져 있다. 이에 대한 단위 테스트 환경을 구성한다.

3.2 테스트 코드를 위한 테스트 설계

테스트 코드를 작성하려면 테스트 경로와 그를 트리거하는 입력값, 그리고 그에 대한 예상 결과값을 사용해야 한다. 본 연구에서는 향후 테스트 criteria의 효율성을 분석하기 위하여 화이트박스 테스트 criteria가운데 5개의 criteria를 적용하였다. All-Use, Prime Path, Branch coverage, Edge Pair, Prime Path with sidetrip 등이 그것들이다. 이 가운데 Branch Coverage를 제외한 나머지 4종류의 Criteria에 따른 테스트 경로는 Jeff Offutt이 제공하는 웹도구를 사용하여 도출해 내었다[5]. 즉, 하나의 메소드에 대한 테스트 코드를 5가지 존재하며 각각은 해당 criteria를 통하여 정의된 테스트 경로이다.

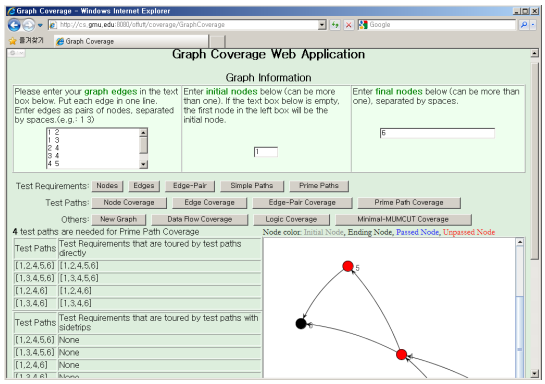

그림 4. Criteria에 따fms 테스트 경로 산출 도구[5]

그림 4에서 보듯이 테스트 대상 메소드의 CFG의 에지들과 시작노드번호, 그리고 종결노드 번호를 입력하고 원하는 criteria를 클릭하면 그에 대한 테스트 경로를 내보내준다. '테스트 코드'는 바로 이 경로를 실행시키는 기능을 한다.

3.3 테스트 코드를 위한 Mokito 활용

Mokito를 Eclipse에 패키지로 추가한후 Eclipse 환경에서 JUnit과 함께 Mokito를 사용한다. Mokito는 앞서 설명한대로, 테스트 스텁을 구현하는 도구이다. 실제 다른 모듈과의 통합 이전의 단위 테스트에 필요한 도구이며, 동시에 테스트 코드를 실행하여 테스트의 자동화를 구현하는데 필수적이다. Mock프레임워크를 어떻게 활용하는지 그림 5가 그 한 예이다.

```
@Test
  public void card_setLearnedAmount_PP_13456()
  { testCard=new Card("Front","Back");
  Category mockedCategory=mock(Category.class);
  testCard.m_category=mockedCategory;
  testCard.setLearnedAmount(false, 100);
  verify(mockedCategory).fireCardEvent(mockedCategor
  y.DECK_EVENT,testCard, testCard.getCategory(),
  testCard.m_level);
  assertEquals(100,testCard.m_backHitsCorrect);
  }
```

그림 5. Mokito 사용 테스트 코드 예

위 테스트 코드는 Card 클래스의 setLearned Amount메소드의 CFG에 Prime Path criteria를 적용한 테스트 경로들 가운데 “1,3,4,5,6”을 실행하는 코드이다. 이 경로를 수행하는 중에 Category 클래스를 호출하여 값을 활용한다. 이 부분을 Mock으로 구현하기 위하여, 다음 코드를 이용한다. 이는 Category 클래스를 흉내내는 Mock 클래스 즉, mockedCategory를 생성하여 테스트 코드에서 이 클래스가 일을 대신하게 하는 것이다.

```
Category
mockedCategory=mock(Category.class);
```

생성된 mock클래스인 mockedCategory는 관찰이 가능해진다. 즉 어떤 테스트 경로상에 mocked Category의한 메소드가 실행되었는지를 확인할 수 있다. 이는 메소드 실행 결과가 특정 값이 아닌 void의 형태로 반환되어 실행되었던 메소드의 실행 여부가 결과가 되는 경우에 매우 유용하다. 그림 5의 테스트 대상 메소드인 setLearnedAmount도 LearnedAmount라는 변수에 특정 값을 설정하는 일을 하며, 그에 대한 어떤 반환값도 없어서, 제대로 수행되는지를 판단하려면 테스트 경로에 따라 잘 진행되었는지를 확인할 필요가 있다. 이때 Mock 프레임워크가 제공하는 verify 메소드를 활용한다. 그림 5에서는 다음과 같이 활용하였다.

```
“verify(mockedCategory).fireCardEvent(mocked
Category.DECK_EVENT,testCard, testCard.getCategory(),
testCard.m_level);”
```

이는 앞서 만든 Mock 클래스인 mocked Category를 사용하여, 테스트 경로상의 메소드인 Category.fireCardEvent 주어진 매개변수들로 실행되었는지를 확인해준다. 이처럼 Mock 프레임워크를 활용하면, 단위 테스트 대상인 메소드를 다른 메소드와의 실제 연결없이 해당 메소드만을 테스트할 수 있다. 이처럼 Mock 클래스를 생성하여 연결하고자 하는 다른 클래스 역할을 대신하게 하고, 또한 그의 행위 전체가 관찰될 수 있게 하여 테스트 경로 상의 작업들이 정확히 잘 수행되었는지를 확인할 수 있는 부가효과가 있다.

3.4 테스트 코드 실행

그림 6과 같이 테스트 코드는 각 criteria별로 패키지를 달리하고 있게 구현하였다. src패키지에는 JMemorize 소스코드들이, Branch_test에는 Branch coverage에 따라 구성된 테스트 경로가 실행될 수 있는 테스트 코드가 테스트 대상 클래스 단위로 하나의 파일로 구성되어져 있다. 예를 들어 Branch_CardTest.java파일은 Card 클래스의 메소드들을 위한 테스트 코드들이다. EdgePair_test, AllUse_test, SideTrip_test, PP_test는 각각 Edge Pair, All Use, Prime Path with SideTrip, 그리고 Prime Path Criteria를 구현하고 있다.

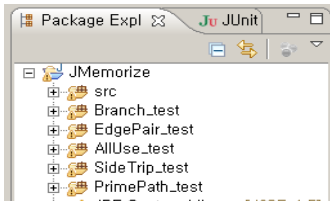

그림 6. Criteria 단위의 패키지

이 가운데 Prime Path를 적용한 테스트 코드를 실행시키보면 그림7의 결과로 보인다. Card 클래스에 속한 메소드들의 테스트 경로가 JUnit의 Run 버튼 하나로 실행되며, 그 결과 하나의 실패, 즉 예상출력과 다른 결과가 나온 테스트 케이스가 발생했음을 보이고 있다.

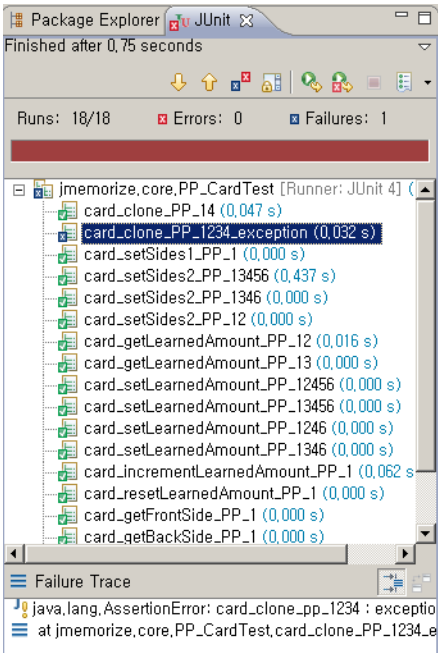

그림 7. 테스트 코드 실행 예

4. 단위 테스트 자동화

본 연구에서는 단위 테스트 자동화 환경을 구축하였다. 자동화를 하기 위하여 테스트 코들 구현해야 하며, 그 코드는 실행 가능해야 한다. 이때 테스트 드라이버와 테스트 스텁이 필요해지는데, 본 연구에서는 JUnit을 테스트 드라이버로, Mock 프레임워크를 테스트 스텁으로 활용하였다.

4.1 "테스트 주도 개발"에서의 단위 테스트

단위 테스트 자동화는 앞서 살펴본 대로 현재의 소프트웨어 개발 문화에서 요구되는 기술이다. 빠른 릴리즈를 목적으로 하는 애자일 접근법으로 소프트웨어를 개발하고자 하는 노력이 커지고 있으며, XP 와 Scrum등의 개발 방법론을 중심으로 그에 따른 구체적인 개발 기술들이 대두되고 있다. 이 기술들의 공통점은 모두 테스트 자동화를 필요로 하고 있다. 특히 테스트 주도 개발 (Test-Driven Development : TDD)의 경우, 모듈 개발 이전에 그를 위한 테스트 코드를 우선 개발하는 방법으로서, 테스트 코드가 다른 모듈과의 실제 연결 없이 단위 모듈에 충실하게 작성될 필요가 있다. 이때 Mock 프레임워크가 필요하다[6].

그림 8은 TDD의 구조를 설명하고 있다. 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.)[7]. 리팩토링에 대하여 아는 개발자가 자신의 코드를 이해하기 쉽고 유지보수성이 높도록 만들어야 한다.

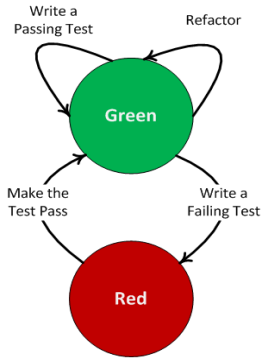

그림 8.TDD 개발 구조

위의 전반적인 과정이 이루어지기 위해서는 해당 단위 모듈마다 테스트 코드 작성이 선행되어져야 한다. 본 연구는 이에 더 나아가 애자일의

다른 개발 기술인 “지속적 통합”과 “자동 빌드” 기술을 지원하기 위하여 단위 테스트 코드 자동화 문제를 고려하였다. 단위 테스트 코드 자동화는 테스트 코드 그 자체가 단독 실행가능하며, 그 결과가 테스트의 통과 및 실패로 명시되어야 한다.

4.2 단위 테스트 자동화의 제약점

본 연구에서 시도했던 단위 테스트 자동화 실험을 통해, 테스트 자동화의 몇가지 제약점을 도출해 내었다. 그동안 단위 테스트 자동화는 테스트 케이스 내의 예상 출력 부분을 도출하는 문제가 극복되어야할 문제도 언급되었었다. 그러나 그 이외에 실제 자동화 코드 구현을 진행하는 동안 고민했던 몇가지 부분들은 다음과 같다.

첫째, 테스트 실행 결과가 테스트 예상 결과와 같은지를 비교하는 코드 작성이 단순하지 않다. 기존의 JUnit을 이용한 단위 테스트 예제들은 대부분 테스트 대상 메소드의 결과가 int 또는 double 등과 같이 명시적인 반환값을 갖는 경우들이다. 그림 9의 (a)는 string의 반환값을 갖는 getBackSide 메소드의 테스트 코드의 예이고, (b)는 의 경우는 void 메소드인 setLearnedAmount의 테스트 코드이다. (a)의 경우가 일반적인 JUnit 사용법에 등장하는 형태이며, void의 경우는 이 방법을 사용할 수 없다. (b)는 해당 메소드가 테스트 경로를 따라 제대로 실행되고 있는지를 관찰하는 Mock 객체를 생성하여 그 결과를 검증하는 방법을 사용하였다. (a)의 경우보다 (b)의 경우가 구현이 복잡하고, 그림 9의 (a)에서의 getBackSide 형태의 메소드 이상으로 그림9의 (b)에서의 setLearnedAmount의 형태의 메소드가 많이 쓰인다. JMemorize의 Card클래스의 경우, 총 40개의 메소드 가운데, 명시적으로 String 또는 int를 반환하는 메소드는 9개, 다른 클래스의 형태로 반환하는 경우는 10개, 그 나머지 21개의 메소드는 void이다. 즉, 50% 이상의 메소드가 void형태, 즉 명시적으로 반환값이 없는 형태를 가지므로, 반환값이 예상대로 나왔는지를 확인하는 방법으로는 테스트 결과가 실패인지 통과인지 판단할 수 없다. 쉽게 예상되는 장벽이었음에도 단순히 JUnit을 활용하면 단위 테스트가 해결될 거라는 기대감이 있었다. JUnit이 이 문제를 어떻게 처리하고 있는지를 확인하기 위하여 자료들을 찾아보았으나, 그에 대한 참고 자료로서의 문헌은 발견하지 못했으며, 단지 Mock을 사용해야 한다는 것과 행위 기반 테스팅에 대한 정보를 얻었다. void 메소드의 경우처럼 테스트 대상 메소드가 명확한 결과를 내지 않는 경우, 실행 도중의 행위들을 스파이 객체 등을 구현하여 관찰하고 그 관찰 결과가 예상과 같은지를 테스트하는 것이다. 이때 Mock 프레임워크가 사용된다.

```
@Test
public void card_getBackSide_PP_1(){
Card testCard=new Card("Front","Back");
assertEquals(testCard.m_backSide,testCard.getBackSide());
}
```

(a)

```
@Test
public void card_setLearnedAmount_PP_1346(){
testCard=new Card("Front","Back");
Category mockedCategory=mock(Category.class);
testCard.m_category=null;
testCard.setLearnedAmount(false, 100);
verify(mockedCategory,never()).fireCardEvent(mockedCateg
ory.DECK_EVENT, testCard, testCard.getCategory(),
testCard.m_level);
assertEquals(100,testCard.m_backHitsCorrect);
}
```

(b)

그림 9. 테스트 코드의 두가지 경우

둘째, 어디까지 테스트 데이터의 실행 결과로 볼 것인가의 문제이다. 명확한 반환값을 갖지 않는, 앞서 언급한 void 형태의 메소드의 경우, criteria로 인해 구성된 테스트 경로상의 모든 문장들이 제대로 수행되었는지 하나씩 살펴보아야 하는지, 아니면 최종 문장에 대해서만 보아야 하는지의 문제이다. 그림 9의 (a)는 명확한 반환값이 명시되어져 있는 메소드를 테스트하는 경우로서 그 실행 결과도

assertion을 사용하여 반환값이 제대로 나오는지를 검증하여 테스트 결과를 결정한다. 그에 반해 그림 9의 (b)는 Category클래스의 fireCardEvent메소드가 주어진 매개변수들을 가지고 실행된 적이 없는지를 확인한 후 또 Card 클래스의 m_backHitsCorrect의 값이 100과 같은지를 assertion으로 확인한다. (b)의 경우는 두가지 내용에 대하여 확인하여 두 경우 모두 통과된 경우, 메소드가 테스트를 통과하였다고 결정짓는다. 여기에서 (b)의 코드가 충분히 테스트 결과를 확인하는지에 대한 의구심이 들 수 있다. 실제 테스트 코드를 작성하면서 테스트 결과를 테스트 코드에서 어느 정도 구체적으로 또는 자세히 확인해야 테스트 결과를 결정하는데 의미가 있을까에 대한 판단 기준이 필요하였다. 이에 대한 보다 객관적이고 검증된 자료가 요구됨을 알고, 향후 이 문제에 대한 연구를 진행하기로 한다.

5. 결론

애자일 개발에 대한 관심이 커지면서, XP와 Scrum을 중심으로 하는 개발 기술들이 대두되고 있다. 이들의 대부분은 단위 테스트를 중요하게 고려하고 있으며, 덧붙여 자동화를 필수 요소로 내세우고 있다. 이에 따라 단위 테스트 자동화에 대한 논의가 일고 있으나, 지금까지 테스트 자동화는 테스트 데이터를 적용하는 도구 수준에 머물고 있었다. 그러나 실제 테스트 결과, 즉 통과 또는 실패를 결정하기 위해서는 테스트 실행 결과와 예상 결과를 비교하는 과정이 필요하다. 이 부분의 구현이 자동화의 성패를 좌우한다.

본 연구는 단위 테스트 자동화를 위한 테스트 코드 작성을 과정을 진행하는 과정을 소개하고, 테스트 코드 구현에서 해결해야 하는 문제들을 언급하고 있다. 단위 테스트 도구로서의 JUnit을 사용하면서 반환값이 명시적이지 않은 void형태의 메소드를 테스트 할 때, 실제 테스트 현장에서 만나게 되는 문제로 다음의 두 가지를 도출해 내고, 그에 대한 현재의 해결 방안을 설명했다. 첫째, void 형태의 메소드의 경우 테스트 데이터 실행 결과를 명시적으로 구하기 어려운 문제를 본 연구에서는 Mock 프레임워크를 사용하여 해결하였다. 둘째, void 형태의 메소드의 경우, criteria로 인해 구성된 테스트 경로상의 모든 문장들이 제대로 수행되었는지 하나씩 살펴보아야 하는지, 아니면 최종 문장에 대해서만 보아야 하는지의 문제이다. 이 문제는 본 연구에서는 Mock 프레임워크의 verify 기능을 활용하여 매 순간 제대로 실행되어져야 하는 메소드 호출을 중심으로, 명확한 매개변수들을 사용하여 호출이 일어났는지를 확인하고, 그 결과들이 모두 예상 결과와 맞을 때, 해당 테스트 케이스에 대한 테스트를 통과한 것으로 결정하였다.

결국 단위 테스트 자동화의 문제는 테스트 대상 모듈의 테스트 데이터 적용 이후의 결과를 비교하는데 있다. 명시적인 반환값이 있는 경우는 어렵지 않게 해결할 수 있고 모두 그 경우에 대한 내용으로 단위 테스트를 설명하고 있다. 그러나 실제 명시적 반환값이 없는 테스트 void 형태의 메소드가 많은 부분을 차지하고 있다. 실험에 사용된 JMemorize의 하나의 클래스의 경우, 총 40개의 메소드 가운데, 명시적으로 String 또는 int를 반환하는 메소드는 9개, 다른 클래스의 형태로 반환하는 경우는 10개, 그 나머지 21개의 메소드는 void이다. 즉, 50%이상의 메소드가 void형태이며, 다른 클래스를 반환하는 10개의 메소드들도 하나의 값으로 그 결과를 판단하기 어려운 상황이다.

향후 명시적 반환값을 갖는 메소드와 아닌경우의 분포를 많은 수의 클래스들을 분석하여 통계로 향후 구축할 계획이며, 테스트 결과 분석 대상을 어느 수준으로 해야 하는지에 대한 판단 기준에 대한 연구도 진행하고 있다.

참 고 문 헌

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

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

[3] 이상민, 자바 개발자도 쉽고 즐겁게 배우는 테스팅 이야기, 한빛미디어, 2009

[4] Gerard Meszaros, Xunit Test Patterns: Refactoring Test Code, Addison-Wesley Professional, 2007

[5] Jeff Offutt, Graph Coverage, http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage

[6] 채수원, 테스트 주도 개발 TDD 실천법과 도구, 한빛미디어, 2010

[7] Martin Fowler, Refactoring : Improving the Design of Existing Code (Improving the Design of Existing Code), Addison-Wesley Professional, 1999

저자소개

윤 회 진

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 조교수

〈관심분야〉 소프트웨어 테스팅, 요구사항 검증, 애자일 테스팅