단위 테스트를 위한 테스트 어서션 언어 설계 및 도구 구현

신 우 창*

Test Assertion Language for Unit Test : Design and Implementation

Woochang Shin*

■ Abstract ■

Unit testing which is the first task to perform software testing has a major impact on reducing s/w development cost and improving s/w quality. In order to make unit testing become the formal procedure rather than ad-hoc tasks performed by developer, the language-neutral measures are necessary in the programming which is able to specify the test cases.

This paper presents a test assertion language for the language-neutral specification of the test cases. The suggested language is based on the ISO/IEC 24709-1, but it supports object oriented features and can describe test cases for unit testing. The test cases that are specified by the language can be used for generating test oracle source codes by test oracle generation tools.

Keyword : Unit Test, Test Assertion Language, Software Engineering, Test Oracle, Code Generation Tools

논문투고일 : 2011년 10월 21일 논문수정완료일 : 2011년 12월 08일 논문게재확정일 : 2011년 12월 12일
* 서경대학교 컴퓨터과학과

1. 서 론

소프트웨어 테스트는 정해진 요구사항을 소프트웨어 시스템이 만족하는지 또는 예상과 실제 결과가 어떤 차이가 있는지를 검사하고 평가하는 일련의 과정을 말한다[1, 2].

소프트웨어 테스트는 소프트웨어 개발 단계에 따라서 단위 테스트(unit test), 통합 테스트(integration test), 시스템 테스트, 인수 테스트(acceptance test)로 구분되는데, 이중에서 가장 먼저 테스트가 이루어지는 부분이 단위 테스트이다. 단위 테스트는 프로그램 소스코드의 각 단위별로 요구사항에 맞게 정확하게 기능하는지 여부를 점검하는 것으로, 각 단위는 함수, 모듈, 클래스 등이 될 수 있다.

단위 테스트를 지원하기 위하여 JUnit과 같은 다양한 단위 테스트 프레임워크(unit test framework)들이 개발되어 있다. 자바 프로그램의 단위 테스트를 지원하는 JUnit, C++ 프로그램의 단위 테스트를 지원하는 CppUnit 과 같이 각 프로그래밍 언어들을 지원하는 단위 테스트 프레임워크를 통칭하여 xUnit이라고 부른다[4].

대부분의 경우 단위 테스트는 테스트 대상이 되는 프로그램의 개발자가 비공식적으로 직접 수행한다. 즉, 코드 작성이 완료되고, 문제없이 컴파일된 후에 기능이 제대로 동작하는지 여부를 개발자 본인이 직접 코드 리뷰(review)작업이나 단위 테스트를 수행하여 확인한다. 개발자는 xUnit을 활용하여 좀 더 체계적이고 자동적으로 단위 테스트를 수행할 수 있다.

그러나 이러한 환경 하에서는 코드 리뷰작업이나 테스트 작업이 임시방편적인 것이 되기 쉽다. 개발자에 의해 발견된 결함(defects)은 기록되지 않으며, 테스트 대상이 되는 프로그램 단위의 히스토리(history)에 포함 되지 못하고, 개인적인 경험으로 남을 뿐이다[3].

좋은 테스트가 이루어지기 위해서는 먼저, 테스터(tester)가 개발자 이외의 사람이어야 하고, 둘째로 테스트가 계획적으로 이루어져야 하며, 셋째로 테스트 과정과 결과가 공적(public)으로 기록되어야 한다[3].

개발자에 의한 임시방편적인 단위 테스트를 지양하고, 테스트 전문가에 의하여 공식적인 단위 테스트가 이루어지기 위해서는 프로그래밍 언어에 중립적으로 테스트 케이스(test case)를 명세할 수 있는 수단이 필요하다.

본 논문에서는 프로그래밍 언어에 중립적으로 테스트 케이스를 명세할 수 있는 테스트 어서션 언어(Test Assertion Language)를 제안한다. 제안된 언어로 기술된 테스트 케이스는 테스트 오라클(test oracle) 생성 도구에 의하여, 단위 테스트를 자동적으로 진행하는 테스트 오라클 소스코드 생성에 사용된다.

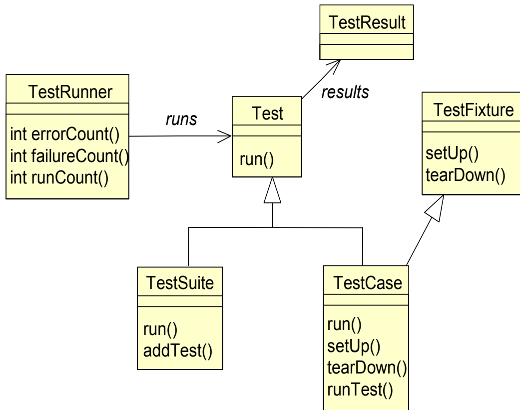
제 2장에서는 본 논문과 관련된 연구들을 소개하고, 제 3장에서는 단위 테스팅을 위한 테스트 어서션 언어를 제안한다. 제 4장에서는 테스트 오라클 생성도구에 대하여 기술한다. 제 5장에서는 제안된 테스트 어서션 언어와 테스트 오라클 생성도구를 실제 프로그램 개발에 적용된 사례를 살펴본다. 마지막 제 6장에서 결론 및 향후 과제에 대하여 살펴보고 마무리를 짓는다.

2. 관련 연구

2.1 단위 테스트 프레임워크

1999년에 Kent Beck이 스몰토크(smalltalk) 언어를 위한 단위 테스트 프레임워크인 SUnit를 발표한 이후, Erich Gamma와 Kent Beck이 이를 자바언어에 맞춰 JUnit을 개발하였다. JUnit이 세상에 널리 알려짐에 따라 CppUnit, NUnit, PyUnit, csUnit, PHPUnit 등과 같이 유사한 구조를 가지는 단위 테스트 프레임워크들이 잇따라 개발되었으며, 이들을 통칭하여 xUnit이라고 부른다.

xUnit의 기본 구조는 [그림 1]과 같다.



[그림 1] xUnit 주요 클래스 구조

xUnit 에서 가장 대표적인 프레임워크인 JUnit의 테스트 케이스 명세를 보이기 위해서, 테스트 대상이 되는 Stack 클래스를 [그림 2]와 같이 작성하였으며, 이 클래스를 단위 테스팅하기 위하여 JUnit으로 테스트 케이스를 기술한 예가 [그림 3]이다.

```
public class Stack {
    final int _size;
    int[ ] _data;
    int _top = 0;
    Stack(int size) {
        _size = size;
        _data = new int[_size];
    }
    public void push(int value) {
        if ( !isFull() )
            _data[_top++] = value;
    }
    public int pop() {
        return (!isEmpty())? _data[--_top] : -1;
    }
    public boolean isEmpty() {
        return ( _top <= 0 );
    }
    public boolean isFull() {
        return ( _top >= _size );
    }
}
```

[그림 2] Stack 클래스

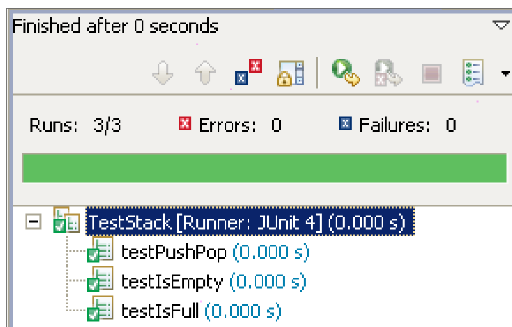
```
import junit.framework.TestCase;

public class TestStack extends TestCase {
    Stack _stack;
    final int _size = 3;

    public TestStack(String methodName) {
        super(methodName);
    }

    public void setUp() throws Exception {
        super.setUp();
        _stack = new Stack(_size);
    }

    public void testPushPop() {
        _stack.push(10);
        _stack.push(20);
        _stack.push(30);
        assertEquals(_stack.pop(), 30);
        assertEquals(_stack.pop(), 20);
        assertEquals(_stack.pop(), 10);
    }

    public void testIsEmpty() {
        assertTrue(_stack.isEmpty());
        _stack.push(10);
        _stack.pop();
        assertTrue(_stack.isEmpty());
    }

    public void testIsFull() {
        assertTrue(!_stack.isFull());
        _stack.push(10);
        _stack.push(20);
        _stack.push(30);
        assertTrue(_stack.isFull());
    }

    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

[그림 3] Stack 클래스를 테스트하기 위한 TestStack 클래스

TestStack 클래스를 이클립스(Eclipse)환경에서나 또는 junit.textui.TestRunner를 이용하여 실행할 수 있다. TestStack 클래스를 JUnit으로 실행하면, 이름이 test로 시작하는 멤버함수들(testPushPop,

testIsEmpty, testIsFull)이 개별적으로 실행되어, Stack 객체가 제대로 기능하는지 여부를 확인한다. 각 test 멤버함수가 수행되기 전에 setUp() 함수가 호출되며, test 멤버함수가 수행된 후에는 tear-Down() 함수가 호출된다.

이클립스 환경에서 TestStack 클래스를 JUnit으로 실행한 결과는 [그림 4]와 같다.



[그림 4] TestStack 실행 결과

xUnit과 같은 단위 테스트 프레임워크를 이용함으로써 다음과 같은 장점을 얻을 수 있다[4, 7].

- 단위 테스트를 자동적으로 수행한다.
- 테스트 결과를 일일이 눈으로 확인할 필요가 없다.
- 테스트가 반복 가능하고, 재사용 가능하다.

본 연구에서는, 제안된 테스트 어서션 언어로 기술된 테스트 케이스를 수행하는데 있어서 JUnit과 같은 단위 테스트 프레임워크를 이용한다.

2.2 BioAPI 표준적합성 테스트 어서션 언어

본 논문에서 제안하는 단위 테스트 명세언어는 국제표준인 ISO/IEC 24709-1에서 정의한 테스트 어서션 언어를 기반으로 하고 있다.

ISO/IEC 24709-1에서는 테스트 어서션(Test Assertion)을 "시험대상이 국제표준에 적합하게 구현되었는지 여부를 확인하기 위한 소프트웨어적인 절차를 기술한 명세서"로 정의하며, 테스트 어서션을 명세할 수 있는 언어의 문법(syntax)과 의미체계(semantics)를 XML 표기법으로 기술한다[5]. 또한 BioAPI 표준적합성 평가를 위하여 BioAPI 각 함수에 대한 테스트 명세서를 ISO/IEC 24709 표준시리즈로 발표하였다[6].

ISO/IEC 24709-1에 정의된 테스트 어서션 언어는 국제표준으로 일관성이 있으며, 한국인터넷진흥원과 같은 국내외 관련 기관에서 평가도구로 이미 활용되어 언어의 유효성이 검증되었다는 점에서 본 연구의 기반으로 선택되었다. 그러나, 이 테스트 어서션 언어는 범용적인 단위 테스트 명세언어로 이용하기에는 다음 몇 가지 면에서 적합하지 않다.

- **변수나 인자(argument)에 대한 자료형 선언을 지원하지 않음** : ISO/IEC 24709-1에 정의된 테스트 어서션 언어로 테스트 되는 대상은 BioAPI 함수로 한정되어 있다. 따라서 호출되는 함수의 인자 자료형이나 반환 값의 자료형이 미리 정해져 있으므로 별도의 자료형 선언이 필요하지 않다. 그러나 범용적인 단위 테스트 명세를 하기 위해서는 함수의 인자, 그리고 결과 값을 받는 변수들에 자료형을 지정할 수 있어야한다.
- **패러미터(parameter)이름으로 함수의 실인자(real parameter)식별** : 테스트 되는 함수를 호출함에 있어서 함수의 실인자(real parame-ter)와 형식인자(formal parameter)를 매칭하는데, 위치정보가 아닌 패러미터 이름을 사용한다. BioAPI의 경우 패러미터 이름이 미리 정해져 있기 때문에 주어진 실인자가 함수의 몇 번째 인자인지 구분할 수 있으나, 범용적인 단위 테스트 명세를 하기 위해서는 패러미터 이름이 아닌 실인자의 위치에 따라 형식인자와의 매칭이 이루어져야한다.
- **객체지향적 프로그래밍 개념을 지원하지 않음** : ISO/IEC 19784-1에서 BioAPI 함수 인터페이스와 관련 자료구조를 C언어로 정의하고 있

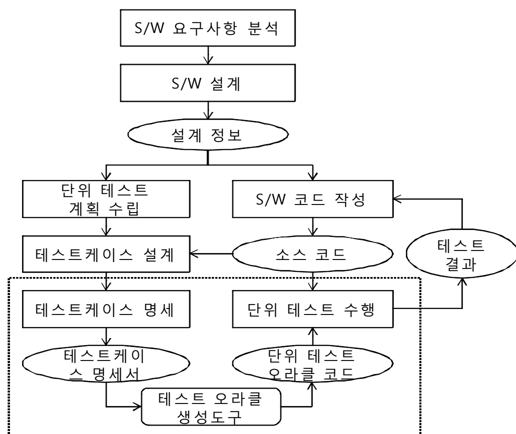
다. 테스트 대상인 BioAPI가 클래스와 같은 객체지향적 개념을 사용하지 않기 때문에, 이를 테스트하기 위한 ISO/IEC 24709-1 테스트 어서션 언어 역시 객체지향적 프로그래밍 개념을 지원하지 않는다. 현재 많이 사용되고 있는 자바나 C++언어와 같은 객체지향적 프로그래밍 언어의 단위테스팅을 지원하기 위해서는 테스트 어서션 언어가 객체지향적 프로그래밍 개념을 지원하여야 한다.

3. 단위 테스팅을 위한 테스트 어서션 언어

ISO/IEC 24709-1에서 정의된 테스트 어서션 언어를 기반으로 하여 단위 테스팅을 위한 테스트 어서션 언어를 제안한다. 제안된 언어는 제 2.2절에서 지적한 문제점을 해결하기 위하여 기존 테스트 어서션 언어의 문법을 일부 수정하거나, 새로운 XML 요소(element)와 속성(attribute)들을 추가하였다.

3.1 단위 테스트 수행 절차

제안된 언어를 이용하여 단위 테스트를 수행하는 절차는 [그림 5]와 같다.



[그림 5] 단위 테스트 수행 절차

S/W 설계 작업을 통해 생성된 설계정보는 S/W 코딩(coding) 작업의 입력이 되는 동시에 단위테스트 계획 수립 작업의 입력이 된다.

테스트 계획이 수립된 후, 단위 테스트를 위한 테스트 케이스 설계가 진행되며, 설계에 따라 테스트 케이스 명세 작업이 이루어진다. 명세 작업의 산출물인 테스트 케이스 명세서는 테스트 오라클 생성도구에 의하여 단위 테스트 작업을 실행하는 테스트 오라클 소스코드로 변환된다. 테스트 오라클 코드는 테스트 대상이 되는 프로그램의 단위 테스팅을 진행하게 되며, 테스트 결과를 생성한다.

본 연구에서는 테스트 케이스 명세서를 작성하는데 사용되는 테스트 어서션 언어를 설계하였으며, 테스트 오라클 생성도구를 개발하였다.

3.2 제안된 테스트 어서션 언어

[그림 3]에 보이는 TestStack 클래스와 동일한 테스트 케이스를, 제안된 테스트 어서션 언어를 이용하여 명세한 것이 [그림 6]이다.

제안된 언어에서 테스트 명세의 기본단위를 표현하는 것은 XML 요소 <assertion>이다. 예에서 Stack 클래스를 테스트하기 위하여 <assertion name = “Stack”> 요소를 정의하고 있다. ISO/IEC 24709 -1에는 <assertion> 요소가 <description>, <input>, <invoke>, <bind> 요소를 포함하는 것으로 정의되어 있으나, 제안된 언어에서는 단위 테스팅에 필요하지 않은 <input>, <bind>요소는 생략하였으며, 단위 테스팅을 지정하기 위한 <test> 요소를 추가하였다.

<test> 요소는 함수나 클래스와 같은 특정 프로그램 단위의 테스트를 지정하는 것으로 예에서는 “TestStack”이라는 이름을 가진 대상물의 테스트 수행을 지정한다.

<unit> 요소는 단위 테스트에서 테스트 실행 단위를 대표한다. 예에서는 “TestStack”이라는 이름의 “class” 단위로 테스트 대상을 정의하고 있다.

<unit> 요소 내에는 내부에서 사용되는 변수를

정의하는 <var> 요소와, 테스트 수행시 실행되는 <activity> 요소가 포함된다. <unit> 요소를 클래스로 비유하자면, <var>와 <activity>는 각각 멤버변수와 멤버함수로 비유할 수 있다.

```
<?xml version = "1.0"?>
<package
    name = "04a01118-0cf9-2085-96d4-
           0002a5d5fd2e">
    <assertion name = "Stack"
              model = "unitTesting">
        <test unit = "TestStack"/>
    </assertion>
    <unit name = "TestStack" type = "class">
        <var name = "_stack" type = "Stack"/>
        <var name = "_size" type = "int"
            value = "3"/>
    <activity name = "setUp" type = "startup">
        <create type = "Stack" args = "_size"
               return = "_stack"/>
    </activity>

    <activity name = "testPushPop" type = "test">
        <var name = "t" type = "int"/>
        <invoke function = "_stack.push"
                args = "10"/>
        <invoke function = "_stack.push"
                args = "20"/>
        <invoke function = "_stack.push"
                args = "30"/>
        <invoke function = "_stack.pop"
                return = "t"/>
        <assert_condition break_if_false = "true" >
            <equal_to var1 = "t" value2 = "30">
        </assert_condition>
        <invoke function = "_stack.pop"
                return = "t"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "t" value2 = "20">
        </assert_condition>
        <invoke function = "_stack.pop"
                return = "t"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "t" value2 = "10">
        </assert_condition>
    </activity>

    <activity name = "testIsEmpty" type = "test">
        <var name = "b" type = "boolean"/>
        <invoke function = "_stack.isEmpty"
                return = "b"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "b" value2 = "true"/>
        </assert_condition>
        <invoke function = "_stack.push"
                args = "10"/>
        <invoke function = "_stack.pop"/>
        <invoke function = "_stack.isEmpty"
                return = "b"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "b" value2 = "true"/>
        </assert_condition>
    </activity>

    <activity name = "testIsFull" type = "test">
        <var name = "b" type = "boolean"/>
        <invoke function = "_stack.isFull"
                return = "b"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "b" value2 = "false"/>
        </assert_condition>
        <invoke function = "_stack.push"
                args = "10"/>
        <invoke function = "_stack.push"
                args = "20"/>
        <invoke function = "_stack.push"
                args = "30"/>
        <invoke function = "_stack.isFull"
                return = "b"/>
        <assert_condition break_if_false = "true">
            <equal_to var1 = "b" value2 = "true"/>
        </assert_condition>
    </activity>
  </unit>
</package>
```

[그림 6] Stack 클래스 단위 테스팅을 위한 테스트 어서션 언어 표기

<activity>의 type 속성의 값이 "test"이면, 상위 <unit>에 대한 테스트가 수행될 때, 해당 <activity>가 실행됨을 의미하며, type 속성의 값이 "startup"이면, 테스트 대상이 되는 <activity>가 실행되기 전에 환경설정을 위해 먼저 실행됨을 나타낸다.

<invoke> 요소는 다른 <activity>나 함수의 호출을 표현한다. ISO/IEC 24709-1의 테스트 어서션 언어는 함수를 호출할 때 <input> 요소를 사용하여 인자를 표현한다. 즉, 함수 add(10,20) 호출을 ISO/IEC 24709-1의 언어로 표현하면 다음과 같다.

```
<invoke function = "add">
    <input name = "arg1" value = "10"/>
    <input name = "arg2" value = "20"/>
</invoke>
```

본 논문에서 제안된 테스트 어서션 언어는 실인자의 위치에 따라 형식인자와의 매칭을 이루기 위하여 <input> 요소 내에 name 속성을 없애고, <input>요소의 표기 순서가 실인자의 위치가 되도록 하였다. 제안된 언어로 위 예를 기술하면 다음과 같다.

```
<invoke function = “add”>
    <input value = “10”/>
    <input value = “20”/>
</invoke>
```

XML 표기의 길이를 단축하기 위하여 제안된 언어에서는 args 속성을 정의하여 다음과 같이 축약된 형태로 표현이 가능하도록 하였다.

```
<invoke function = “add” args = “10,20”/>
```

<invoke> 요소 안에 <return> 요소를 선언하여 호출된 함수의 반환 값을 받는 변수를 지정하지만, 이 역시 <invoke> 요소 내에 return 속성을 선언하여 반환 값을 받는 변수를 지정할 수 있다.

<assert_condition> 요소는 단위 테스팅에 있어서 코드 실행 결과를 검증하는 기능을 명세 한다. <assert_condition>은 하위에 <and>, <or>, <not>, <equal_to>, <greater_than>, <exsiting> 등과 같은 요소들로 구성된 불린(boolean) 수식을 포함하고 있으며, 테스트 수행 시 이 수식 값에 따라 단위 테스트의 통과(pass)와 실패(fail)가 결정된다 [5]. 기본적으로 수식 값이 참이며, 그 다음 테스트를 진행하지만, 수식 값이 거짓이면 해당 단위 테스트 결과 값이 실패(fail)가 된다. break_if_false 속성 값이 “true”일 때, 내부 수식 값이 거짓이면, 남은 테스트 과정을 중단하게 된다.

변수를 선언하는 <var> 요소에서 자료형은 type 속성으로 지정한다. type 속성 값으로는 기본 자료형(primitive type), 배열(array), 클래스(class), 그리고 특정 프로그래밍 의존적 자료형이 있다. 기본 자료형으로는 boolean, byte, char, short, int, long, float, double, string(문자열도 기본 자료형으로 간주)이 있으며, 배열은 기본 자료형에 “[ ]” 기호를 붙여 표시한다. 또한 외부에서 정의한 클래스 명을 type 속성 값으로 사용할 수 있다.

제안된 어서션 언어가 특정 프로그래밍 언어에 의존적이지 않고 범용적으로 사용되는 것을 목표로 하지만, 특정 프로그래밍에 의존적인 자료형이 사용되는 경우 이를 지원할 필요성이 있다. 예로 C++언어의 포인터를 인자로 사용하는 함수를 테스트 할 경우, 이를 호출하기 위해서는 포인터 자료형을 명세할 수 있어야한다.

제안된 테스트 어서션 언어에서는 포인터 및 특정 프로그래밍 언어 의존적인 자료형의 경우 이를 어서션 언어 내에서 미리 정의하지 않고, type속성 값으로 기술한 다음, 이에 대한 해석을 해당 언어의 테스트 오라클 생성도구에 일임하는 방식으로 처리한다.

즉, <var name = “a” type = “int×[10]”>이라고 선언하면, C++ 테스트 오라클 생성도구에서 이 자료형을 “array(with size 10) of pointer to integer”로 해석하여 코드를 생성한다.

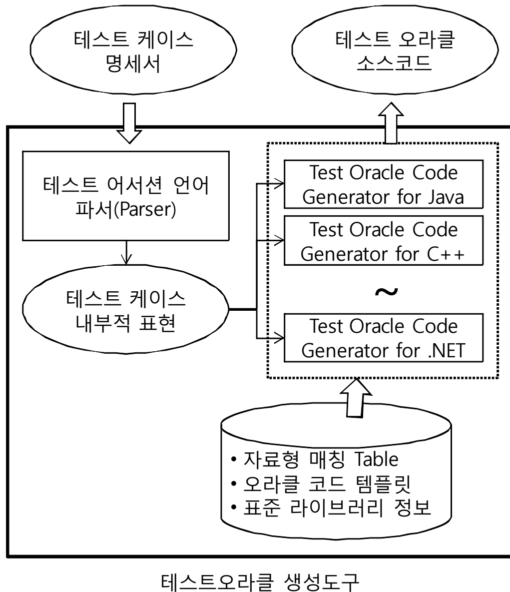
[그림 6]에서는 한 클래스에 대한 테스트 케이스를 작성한 모습을 보였으나, 제안된 언어의 <package>내에 여러 <unit>들을 선언하고, 각 <unit>에 대한 테스트를 <assertion> 내에서 복수개의 <test> 요소로 선언함으로써 테스트 스위트(test suite) 지정도 가능하다.

제안된 테스트 어서션 언어를 정의하는 XML 스키마 중에서 ISO/IEC 24709-1에 정의된 테스트 어서션 언어와 중복되는 부분은 제외하고 차이나는 부분만 <부록 A>에 기술하였다.

4. 테스트 오라클 생성도구 구현

테스트 오라클 생성도구는 테스트 어서션 언어로 기술된 테스트 케이스 명세를 입력으로 하여,

해당 단위 테스트를 자동적으로 수행하는 테스트 오라클 소스코드를 생성한다. 테스트 오라클 생성도구의 구조는 [그림 7]과 같다.



[그림 7] 테스트 오라클 생성도구 구조

테스트 어서션 명세서는 오라클 생성도구 내에 있는 테스트 어서션 언어 파서 모듈에 의하여 테스트 케이스의 내부적 표현으로 변환된다.

내부적으로 표현된 테스트 케이스는 각 언어별(또는 각 단위 테스트 프레임워크별) 오라클 코드 생성도구에 의하여 해당 프로그래밍 언어와 단위 테스트 프레임워크에서 실행될 수 있는 소스코드를 생성하는데 이용된다.

테스트 어서션 명세서에 기술된 자료구조와 각 언어가 사용하는 자료형이 다를 경우, 이를 매칭(matching)하거나 보정해 줄 수 있는 정보를 DB화한다. 또한 각 단위 테스트 프레임워크 별로 기본적인 코드 양식(template)이 필요한 데, 이 역시 오라클 코드 템플릿 정보로서 DB에 저장하여 코드 생성에 사용한다.

테스트 어서션 명세서에서 언어별 표준 라이브러리 함수나 객체를 사용하였을 경우, 코드 생성기에서 이를 지원하는 코드(예로 import 문장이나 include 문장)를 생성하기 위하여 각 언어별로 표준 라이브러리 정보를 DB에 저장한다.

본 연구에서는 테스트 어서션 언어 파서(parser)와 함께 Java 언어를 위한 테스트 오라클 코드 생성도구를 구현하였다.

Java 언어를 위한 테스트 오라클 코드는 JUnit 프레임워크에서 수행된다. 어서션 명세서에 기술된 테스트 케이스가 한 개의 클래스형 테스트 유닛으로 구성되어 있다면, [그림 8]과 같은 코드 템플릿을 이용하여 테스트 오라클 코드가 생성된다.

코드 템플릿에서 “%{CLASS_NAME}%”에 테스트 단위 이름이 들어가며, “%{VARIABLES}%”에 <unit> 요소 내에 선언된 <var>의 정보가 삽입된다.

```
%{PACKAGE_NAME}%

import junit.framework.TestCase;
%{IMPORTS}%

public class %{CLASS_NAME}% extends
TestCase {
    %{VARIABLES}%

    public void setUp() throws Exception {
        super.setUp();
        %{STARTUP_CODE}%
    }

    %{TEST_METHODS}%

    public void tearDown() throws Exception {
        %{FINAL_CODE}%
        super.tearDown();
    }
}
```

[그림 8] JUnit용 테스트 오라클 코드 템플릿

JUnit 프레임워크에서는 테스트 수행에 앞서 실행되는 메소드가 setUp()이며, 또 테스트 수행 후 실행되는 메소드가 tearDown() 으로 정해져 있다. 이에 따라 <unit> 요소 내에 선언된 <activity>

들 중에서 type 속성값이 “startup”인 <activity>의 수행 코드는 “%{STARTUP_CODE}%” 부분에, type 속성값이 “final”인 <activity>의 수행 코드는 ”%{FINAL_CODE}%” 부분에 변환되어 삽입된다. type 속성값이 “test”인 각 <activity>들은 %{TEST_METHODS}% 부분에 함수 형태로 생성된다. JUnit 프레임워크의 관례상 테스트 대상 함수의 이름이 “test”로 시작하지 않으면, 자동적으로 함수 이름 앞에 “test” 단어를 붙인다.

[그림 6]에 기술된 테스트 어서션 명세서를 Java 언어를 위한 테스트 오라클 코드 생성도구에 입력하여 생성한 오라클 코드가 [그림 9]이다.

자동 생성된 오라클 코드와 손으로 작성된 [그림 3] 코드를 비교하면 손으로 작성된 코드가 좀 더 간결하다. 이는 테스트 어서션 언어에서 결과를 테스트하는 <assert_condition> 요소가 함수를 직접 호출하도록 설계되어 있지 않기 때문이다. 이러한 차이는 테스트 기능에 전혀 영향을 주지 않으며, 또한 생성도구 내에서 코드 최적화 과정을 통하여 개선 할 수 있다.

```
import junit.framework.TestCase;

public class TestStack extends TestCase {
    Stack _stack;
    int _size = 3;

    public void setUp() throws Exception {
        super.setUp();
        _stack = new Stack(_size);
    }

    public void testPushPop() {
        int t;
        _stack.push(10);
        _stack.push(20);
        _stack.push(30);
        t = _stack.pop();
        assertTrue(t == (30));
        t = _stack.pop();
        assertTrue(t == (20));
        t = _stack.pop();
        assertTrue(t == (10));
    }

    public void testIsEmpty() {
        boolean b;
        b = _stack.isEmpty();
        assertTrue(b);
        _stack.push(10);
        _stack.pop();
        b = _stack.isEmpty();
        assertTrue(b);
    }

    public void testIsFull() {
        boolean b;
        b = _stack.isFull();
        assertTrue(b == (false));
        _stack.push(10);
        _stack.push(20);
        _stack.push(30);
        b = _stack.isFull();
        assertTrue(b == (true));
    }

    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

[그림 9] 테스트 어서션 명세서로부터 자동 생성된 테스트 오라클 코드

5. 적용 사례

제안된 테스트 어서션 언어와 테스트 오라클 생성도구의 효용성을 검증하기 위하여 상업용 모바일 앱 프로그램 구축과제에 적용해 보았다.

상업용 모바일 앱 프로그램은 이클립스(Eclipse) 개발 환경에서 Android 플랫폼 v2.1 API 레벨 7을 사용하여 개발되었다. 개발된 프로그램은 라이센싱 관련 모듈을 제외하고 22개의 주요 클래스로 구성되어 있다. 이중에서 사용자 인터페이스를 담당하는 Activity 클래스는 11개이며, 이를 제외한 나머지 11개 클래스에 대한 단위 테스트를 수행하였다.

테스트 대상이 되는 각 클래스마다 테스트 어서션 언어를 이용하여 테스트 케이스 명세서를 작성하였으며, 테스트 오라클 생성도구를 사용하여 단위 테스트를 자동 진행하는 테스트 오라클 코드를

생성하였다.

안드로이드 환경에서의 단위 테스트 수행을 위하여 Android JUnit을 사용하였으며, 이를 위하여 테스트 오라클 생성 과정에서 [그림 10]에 보이는 테스트 오라클 코드 템플릿을 사용하였다.

```
%{PACKAGE_NAME}%

import android.content.Context;
import android.test.AndroidTestCase;
%{IMPORTS}%

public class %{CLASS_NAME}% extends
AndroidTestCase {
    %{VARIABLES}%

    public void setUp() throws Exception {
        super.setUp();
        %{STARTUP_CODE}%
    }

     %{TEST_METHODS}%

     public void tearDown() throws Exception {
        %{FINAL_CODE}%
        super.tearDown();
    }
}
```

[그림 10] JUnit용 테스트 오라클 코드 템플릿

테스트 케이스 명세서 작성자는 XML과 프로그래밍 언어에 대한 기초지식이 있는 자로서, 한 두 시간의 테스트 어서션 언어에 대한 교육을 받은 후 별 어려움 없이 명세서를 작성할 수 있었다.

11개의 단위 테스트 대상 클래스에 대하여 총 120여개의 테스트 함수가 작성되었으며, 이 테스트 함수들은 프로그램 개발 진행 중에 반복적으로 자동 테스팅 되었다.

명세서를 작성한 후 오라클 코드를 생성하고 이를 수행하는 방식이, 기존의 직접 JUnit 코드를 작성하여 수행하는 방식에 비하여 단위 테스트 수행 기능의 차이는 보이지 않았으나, 시간이 좀 더 소요되었다. 그러나, 제안된 테스트 어서션 언어를 사용함으로써 테스트 명세서 작성자가 특정 프로그래밍 언어와 단위 테스팅 프레임워크에 관한 지식이 없어도 되며, 둘째로 테스트 케이스 자체에 더욱 집중할 수 있어 일관성 있는 테스트를 진행할 수 있었다.

6. 결 론

소프트웨어 테스트 중에서 가장 먼저 수행되는 단위테스트는 소프트웨어 개발 비용을 줄이고, 품질을 향상하는데 큰 영향을 미친다.

단위 테스트가 개발자에 의한 임시방편적인 작업이 아니라, 테스트 전문가에 의한 공식적인 작업이 되기 위해서는 프로그래밍 언어에 중립적으로 테스트 케이스를 표현할 수 있는 수단이 필요하다.

본 논문에서는 프로그래밍 언어에 중립적인 테스트 어서션 언어를 제안하였으며, 제안된 언어로 기술된 테스트 케이스 명세서로부터 테스트 오라클 소스코드를 자동으로 생성하는 도구의 구현에 대하여 설명하였다.

제안된 언어로 테스트 케이스를 기술할 때 장점은 다음과 같다.

- DOM(Document Object Model) 표준과 같이 프로그래밍 언어에 중립적인 인터페이스에 대한 테스트 케이스를 기술할 때 적합하다.
- 시스템이 여러 프로그래밍 언어로 개발될 때, 프로그래밍 언어에 상관없이, 테스트 케이스를 통일된 형태로 기술할 수 있다.
- 단위 테스트 작업이 개발자에 의한 임시방편적인 작업이 아니라, 테스트 전문가에 의한 공식적인 작업이 되도록 한다.
- 테스트 케이스 보고서 생성과 같은 자동화된 도구를 이용할 수 있다.

향후 과제로는 테스트 어서션 언어를 활용하여 통합 테스트와 시스템 테스트를 위한 테스트 시나리오를 명세하고, 이를 자동적으로 수행하는 테스

트 오라클 코드를 생성하는 연구가 있다.

참 고 문 헌

[1] *An American National Standard/IEEE Standard for Software Unit Testing*, ANSI/IEEE Std 1008~1987, 1986.

[2] *IEEE Standard Glossary of Software Engineering Terms*, IEEE Society Press, Addison-Wesley, 1989.

[3] Ilene Burnstein, *Practical Software Testing,* Springer-Verlag, 2003.

[4] Paul Hamill, *Unit Test Frameworks,* O'Reilly Media, 2003.

[5] ISO/IEC 24709-1 : 2007(E) *Information technology-Conformance testing for the biometric application programming interface (Bio API)-Part 1 : Methods and procedures*, ISO/IEC, 2007.

[6] ISO/IEC 24709-2 : 2007(E) *Information technology-Conformance testing for the biometric application programming interface (Bio API)-Part 2 : Test assertions for biometric service providers,* ISO/IEC, 2007.

[7] Tahchiev, P., F. Leme, V. Massol, and G. Gregory, *JUnit in Action 2nd. Ed.,* Manning Publications Co., 2010.

[8] Nebut, C., F. Fleurey, Y. L. Traon, and J. M. Jezequel, "Automatic Test Generation : A Use Case Driven Approach", *IEEE Transactions on Software Engineering*, Vol.32, No.3(2006).

〈부록 A〉 제안된 테스트 어서션 언어를 정의하는 XML 스키마

```
<?xml version = '1.0' encoding = "utf-8"?>
<xs : schema xmlns : xs
      = "http://www.w3.org/2001/XMLSchema">

   <xs : element name = "package"
                type = "package"/>

   <xs : complexType name = "package">
      <xs : sequence>
         <xs : element name = "author"
               minOccurs = "0"
               type = "xs : string"/>
         <xs : element name = "description"
               minOccurs = "0"
               type = "xs : string"/>
         <xs : element name = "assertion"
               type = "assertion" minOccurs = "0"
               maxOccurs = "unbounded"/>
         <xs : choice minOccurs = "0"
               maxOccurs = "unbounded">
            <xs : element name = "activity"
                  type = "activity"/>
            <xs : element name = "unit"
                  type = "unit"/>
         </xs : choice>
      </xs : sequence>
      <xs : attribute name = "name" type = "uuid"
            use = "required"/>
   </xs : complexType>

   <xs : complexType name = "assertion">
      <xs : sequence>
         <xs : element name = "description"
               type = "xs : string"
               minOccurs = "0"/>
         <xs : choice minOccurs = "0"
               maxOccurs = "unbounded">
            <xs : element name = "invoke"
                  type = "assertionInvoke"/>
            <xs : element name = "test"
                  type = "test"/>
         </xs : choice>
      </xs : sequence>
      <xs : attribute name = "name"
            type = "xs : NCName"
            use = "required"/>
   <xs : attribute name = "model" type = "model"
         use = "required"/>
   </xs : complexType>
   <xs : complexType name = "unit">
      <xs : sequence>
         <xs : element name = "var" type = "var"
               minOccurs = "0"
               maxOccurs = "unbounded"/>
         <xs : element name = "activity"
               type = "activity"
               minOccurs = "0"
               maxOccurs = "unbounded"/>
      </xs : sequence>
      <xs : attribute name = "name"
            type = "xs : NCName"
            use = "required"/>
      <xs : attribute name = "type"
            type = "unitType" use = "required"/>
   </xs : complexType>

   <xs : complexType name = "activity">
      <xs : sequence>
         <xs : element name = "input"
               type = "activityInputOutput"
               minOccurs = "0"
               maxOccurs = "unbounded"/>
         <xs : element name = "output"
               type = "activityInputOutput"
               minOccurs = "0"
               maxOccurs = "unbounded"/>
         <xs : element name = "var" type = "var"
               minOccurs = "0"
               maxOccurs = "unbounded"/>
         <xs : choice minOccurs = "0"
               maxOccurs = "unbounded">
            <xs : element name = "set"
                  type = "set"/>
            <xs : element name = "add"
                  type = "arithmetic"/>
            <xs : element name = "subtract"
                  type = "arithmetic"/>
            <xs : element name = "wait_until"
                  type = "wait_until"/>
            <xs : element name = "assert_condition"
                  type = "assert_condition"/>
            <xs : element name = "invoke"
                  type = "activityInvoke"/>
            <xs : element name = "create"
                  type = "activityCreate"/>
         </xs : choice>
      </xs : sequence>
      <xs : attribute name = "name"
            type = "xs : NCName"
            use = "required"/>
      <xs : attribute name = "atomic"
            type = "xs : boolean"
            default = "false"/>
      <xs : attribute name = "type"
            type = "activityType"/>
   </xs : complexType>
   <xs : simpleType name = "model">
      <xs : restriction base = "xs : token">
         <xs : enumeration value = "unitTesting"/>
      </xs : restriction>
   </xs : simpleType>

   <xs : simpleType name = "unitType">
      <xs : restriction base = "xs : token">
         <xs : enumeration value = "class"/>
```

```
      <xs : enumeration value = “module”/>
    </xs : restriction>
  </xs : simpleType>

  <xs : simpleType name = “activityType”>
    <xs : restriction base = “xs : token”>
      <xs : enumeration value = “startup”/>
      <xs : enumeration value = “test”/>
      <xs : enumeration value = “final”/>
    </xs : restriction>
  </xs : simpleType>

  <xs : simpleType name = “primitiveDataType”>
    <xs : restriction base = “xs : token”>
      <xs : enumeration value = “boolean”/>
      <xs : enumeration value = “byte”/>
      <xs : enumeration value = “char”/>
      <xs : enumeration value = “short”/>
      <xs : enumeration value = “int”/>
      <xs : enumeration value = “long”/>
      <xs : enumeration value = “float”/>
      <xs : enumeration value = “double”/>
      <xs : enumeration value = “string”/>
    </xs : restriction>
  </xs : simpleType>

  <xs : simpeType name = “dataType”>
    <xs : union memberTypes =
          “xs : string primitiveDataType”/>
  </xs : simpleType>

  <xs : complexType name = “test” >
    <xs : attribute name = “unit”
          type = “xs : NCName”
          use = “required”/>
  </xs : complexType>

  <xs : complexType name = “var” >
    <xs : attribute name = “name”
          type = “xs : NCName”
          use = “required”/>
    <xs : attribute name = “type”
          type = “dataType”
          use = “required”/>
    <xs : attribute name = “value”
          type = “xs : string”/>
  </xs : complexType>
    <xs : complexType name =
      “activityInputOutput”>
      <xs : attribute name = “name”
            type = “xs : NCName”
            use = “required”/>
      <xs : attribute name = “type”
            type = “dataType”
            use = “required”/>
    </xs : complexType>

  <xs : complexType name = “activityInvoke”>
    <xs : sequence>
      <xs : element name = “only_if”
            type = “only_if”
            minOccurs = “0”/>
      <xs : element name = “input”
            type = “invokeInput”
            minOccurs = “0”
            maxOccurs = “unbounded”/>
      <xs : element name = “output”
            type = “invokeOutput”
            minOccurs = “0”
            maxOccurs = “unbounded”/>
      <xs : element name = “return”
            type = “invokeReturn”
            minOccurs = “0”/>
    </xs : sequence>
    <xs : attribute name = “activity”
          type = “xs : NCName”/>
    <xs : attribute name = “package”
          type = “uuid”/>
    <xs : attribute name = “function”
          type = “xs : NCName”/>
    <xs : attribute name = “args”
          type = “xs : string”/>
    <xs : attribute name = “return”
          type = “xs : NCName”/>
  </xs : complexType>
    <xs : complexType name = “invokeInput”>
      <xs : attribute name = “value”
            type = “xs : string”/>
      <xs : attribute name = “var”
            type = “xs : NCName”/>
    </xs : complexType>

    <xs : complexType name = “invokeOutput”>
      <xs : attribute name = “var”
            type = “xs : NCName”/>
    </xs : complexType>

    <xs : complexType name = “invokeReturn”>
      <xs : attribute name = “setvar”
            type = “xs : NCName”
            use = “required”/>
    </xs : complexType>
    <xs : complexType name = “activityCreate”>
      <xs : sequence>
        <xs : element name = “only_if”
              type = “only_if”
              minOccurs = “0”/>
        <xs : element name = “input”
              type = “invokeInput”
              minOccurs = “0”
              maxOccurs = “unbounded”/>
        <xs : element name = “return”
              type = “invokeReturn”
              minOccurs = “0”/>
      </xs : sequence>
      <xs : attribute name = “type”
            type = “dataType”/>
      <xs : attribute name = “args”
            type = “xs : string”/>
      <xs : attribute name = “return”
            type = “xs : NCName”/>
    </xs : complexType>
</xs : schema>
```

◈ 저 자 소 개 ◈

신 우 창 (wcshin@skuniv.ac.kr)

서울대학교에서 전산학 학사·석사를 하고 동 대학에서 컴퓨터공학 박사를 받았다. 현재 서경대학교 컴퓨터과학과 교수로 재직 중이며 주요 연구관심 분야는 소프트웨어 평가, 설계패턴, 컴포넌트기반 개발, 소프트웨어 재구조화, 소프트웨어 아키텍처 등이다.