□특 집□

# 객체 지향 프로그램의 테스팅 방법론

공주교육대학교 실과교육과 한 규 정*
중앙대학교 컴퓨터공학과 이 경 환*
강원대학교 전자계산학과 양 해 술*

목 차



## I. 서 론

1993년 현재 객체 지향 소프트웨어 공학(Object-Oriented Software Engineering, OOSE)[1]이라는 신용어를 가진 책들이 이전에 구조적 분석/설계란 이름하에 여러 책이 출판되었던 것처럼 서서히 등장하고 있다. 더우기 CASE, 소프트웨어 재사용(software reuse) 등과 어울러진 객체 지향 개발 방법론은 90년대 이후의 소프트웨어 개발 방법론의 주역으로 등장하게 될 것임이 틀림없다. 그러나 아직은 Yourdon의 구조적 분석/설계 방법론과 같은 뚜렷한 선두주자격인 객체 지향 분석, 설계 방법론이 나타나고 있지는 않고, 다만 몇 가지 주목할만한 방법론이 대두되고 있다[2, 3, 4]. 또한 이들을 바탕으로 툴들이 제품화되어 있다.

한편 소프트웨어 테스팅 분야에서는 분석, 설계만큼의 많은 수의 논문은 아니더라도 몇 가지 주목할 논문들이 90년대 이후 발표되어지고 있다. 그 주된 내용들은 기존의 프로시듀어(procedure) 지향 프로그램에서의 수 많은 테스팅 방법론 중 몇 가지가 적용되고 있기도 하며, 다른 한편으로는 추상화 자료형(Abstract data types, ADTs), 상속성(Inheritance), 다형화(Polymorphism) 등의 객체 지향 특성을 고려한 새로운 테스팅 방법들이 제시되고 있다. 기존의 프로시듀어 위주의 테스팅 방법으로 객체 지향 프로그램의 테스팅을 수행하기에는 프로그램 구성 특성상 문제가 있다는 것이 차츰 받아들여지고 있다. 따라서 본 논문에서는 프로시듀어 지향 프로그램 테스팅과 객체 지향 프로그램의 테스팅의 차이점, 기존의 연구들, 또 그들이 가지는 문제점, 본 연구실에서의 연구방향들을 소개하려 한다.

## II. 프로시듀어 지향 프로그램 테스팅과 객체 지향 프로그램의 테스팅

객체 지향 프로그램에서나 프로시듀어 프로그램에서의 테스팅의 목적은 같다. 따라서 테스팅 방법론은 소프트웨어가 그 바라는 기능을 수행하는가 뿐만 아니라 결함도 찾아내야 한다. 본 절에서는 비록 그 목적은 같으나 방법론이 달라져야 하는 문제에 대해 논한다.

첫째, 그 대상의 변화에 있다. 기존의 테스팅 방법론에서 일반적으로 알려진 바와 같이 테스팅 대상과 방법에 따라 크게 두 가지로 대별된다. 그 하나는 독립 모듈(individual modules)에 대해 구현된 알고리즘의

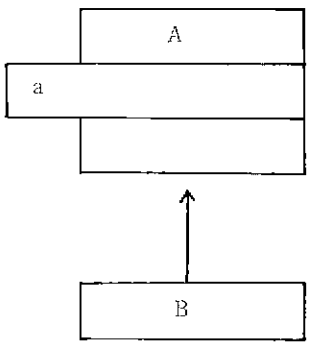
* 종신회원

정확성과 바라는 결과를 리턴하는지에 관한 단위 테스팅(unit testing)이다. 또 다른 하나는 독립 모듈들이 어떻게 잘 조화있게 수행하고, 매개 변수로서 적절한 데이터 유형을 주고 받는가의 통합 테스팅(integrated testing)이다. 이러한 테스팅 대상은 기존의 프로시듀어 프로그램에서 단일 입력(single-entry), 단일 출력(single-exit)의 하나의 모듈을 대상으로의 단위 테스팅, 또한 그 모듈들의 매개 변수의 주고 받음으로 연결된 구조에 대해 통합 테스팅으로 그 대상이 간단화되었다. 그러나 객체지향 프로그램에서의 가장 작은 단위를 클래스(class)라고 할 때 이에 대한 태스팅을 단위 테스팅이라고 하기에는 문제점이 있다. 우선 어디에서나 있을 수 있는 메소드(method)의 호출로 인해 단일 입력이 될 수 없으며, 또 클래스의 메소드들 중 여러 곳에서 다른 클래스의 메소드들을 호출할 수 있어 다중 출력이 될 수 있다.

객체 지향 프로그램에서의 통합 테스팅이라는 것은 결국 클래스 메소드들의 메세지 전달을 대상으로 하게 된다.

둘째, 프로시듀어 프로그램에서는 볼 수 없는 객체 지향 특성, 추상화 자료형, 상속성, 다형화에 대한 테스팅 요인이다. 그 중에서 가령, (그림 1)과 같이 클래스 A가 클래스 B의 상위 클래스이고 A의 메소드 a를 B가 상속받는다고 가정하자. 클래스 A의 메소드 a에 대해 테스팅이 이루어져 이상이 없다하면, 클래스 B에서 그 상속받는 a에 대해서는 테스트가 필요없다고 생각되어지나 Perry[5]는 Weyuker의 적합성 테스팅 공리[6]에 의해 반드시 테스팅이 필요하다고 하였다.

셋째, 테스팅 노력에 차이가 있다. 단위 테스팅, 통합 테스팅 중 어느것이 더 중요하다고는 할 수가




**(그림 1) 클래스에의 메소드 a를 상속받는 B 클래스**

없다. 그러나 클래스에 대한 테스팅은 기존의 프로시듀어에 대해 수행되는 단위 테스트보다는 상당히 복잡해 질 수 있다. 반면에 통합 테스팅 측면에서 객체 지향 프로그램에서의 테스팅은 프로시듀어 프로그램에서의 그것보다 노력은 더욱 필요할 지는 모르지만 동시에 용이해질 수가 있다. 그 이유로는 각 클래스의 메소드를 통해 주고 받는 경로의 수는 매우 많아지지만 경로상의 매개 변수가, 프로시듀어 프로그램에서 주고 받는 매개 변수보다 저 결합도 (low coupling)의 변수를 주고 받는데 있다. 상속성과 다형화도 통합 테스팅의 노력을 복잡화할 수 있다.

## III. 객체 지향 프로그램의 테스팅 방법론

현재 객체 지향 프로그램의 테스팅 방법론으로 알려진 주요 연구는 다음과 같다. Weyuker[6]의 적합성 공리를 객체 지향 프로그램에 적용한 Perry의 테스팅 방법[5], Harrold의 객체지향 클래스 구조의 점진적 테스팅 방법[7], Smith의 FOOT[8] 등이 대표적이다.

### 3.1 Perry의 테스팅 방법

#### 3.1.1 Weyuker의 적합성 테스팅 공리

Weyuker는 기존의 테스팅 기준들이 가지고 있는 약점과 강점을 분석하여 새로운 적합한 테스팅 전략의 제안을 유도할 수 있는 공리(axioms)를 제시했다. 이런 적합성 테스팅 기준은 많은 테스터들이 공감하는 프로그램의 테스팅에 있어서 직관적이고, 경험적인 것들이 비형식적으로 수행했던 사항들을 형식화한 공리들로 정리하여 충분한 테스팅을 지향하고 있다. 한편 일반적으로 적합하게 테스트된 프로그램이 정확하다(correct)는 것은 잘못이다. 테스팅의 목적은 에러를 발견하는 것이지 정확성을 보장하는 것은 아니기 때문이다. 그러나 "좋은(good)" 적합성 기준이 선택되었고, 테스트 집합이 프로그램 P에 대한 기준은 만족시킨다면, P가 정확성에 "가깝다(close to)"고 한다.

(1) 적합성 테스팅 공리

다음은 프로그램에 적합한 태스팅을 위해 필요한 성질들의 공리이다.

공리 1) 적용성(applicability)

| 모든 프로그램에는, 적합한 테스트 집합이 있다. |
|---|

이 성질들은 다음과 같이 정련될 수 있다. 즉 모든 프로그램에는 유한한(finite) 테스트 집합이 있다는 것으로 비록 요구되는 테스트 데이타가 많다 하더라도 테스트 집합은 유한다는 것이다.

공리 2) 비철저한 적용성(non-exhaustive applicability)

> 프로그램 P가 테스트 집합 T에 의해 적합하게 테스트되었을 때. T는 철저한 테스트 집합은 아니다.

공리 3) 단조성(monotonicity)

> 만약 테스트 집합 T가 프로그램 P에 적합하고 T T'이면 T'는 P에 적합하다.

이 성질은 만약 프로그램이 적절히 테스트되었다면 일부 더 이상의 테스트(불필요한 테스트)는 부적합하게 테스트되지는 않는다. 즉 적합한 테스트 적합 T의 수퍼 집합도 적합하다. 가령 어떤 프로그램에서 적절한 테스트 집합 T={0,1,2,3}가 존재한다고 하자.

만약 T T'인 T'={0,1,2,3,4,5}가 존재하더라도 T'는 P에 적합하다.

공리 4) 부적합한 공집합(inadequate empty set)

> 공 집합은 프로그램에 대해 적합한 테스트 데이타가 아니다.

즉 모든 프로그램은 적어도 하나의 입력변수를 가지고 있다. 그렇지 않다면 공집합이 입력없는 프로그램에 적합한 테스트 집합이다.

공리 5) 재명명성(renaming)

> 프로그램 P가 Q가 재명명이라면, 테스트 집합 T가 Q에 적합할 때만 T는 P에 적합하다.

가령, 프로그램 B에 적합한 테스트 집합 {0,1,2,3}은 프로그램 B를 A라고 재명명하면. 프로그램 A에 테스트 집합 {0,1,2,3}은 적합하다.

공리 6) 복잡도(complexity)

> 모든 n에 대해 프로그램 P가 존재하고, P가 n개의테스트 집합에 의해 적합하게 테스트되었다면, n-1의 테스트 집합으로는 적합하게 테스트되었다고 하지 않는다.

공리 7) 문장 커버리지(statement coverage)

> 만약 테스트 집합 T가 P에 대해 적합하다면 T는 P에서 모든 수행가능한 문장을 가진다.

공리 8) 반 확장성(anti-extentionality)

> P≡Q인 두개의 프로그램이 있을 때, 테스트 집합 T가 P에 적합하나, T가 Q에 적합하지 않다.

즉 만약 두 프로그램이 같은 기능을 계산시(의미적으로 같음), 한 프로그램에 테스트 집합이 적합한 것이 다른 프로그램에도 적합한 것은 아니다. 가령 (그림 2)와 (그림 3)이 동등일지라도, (그림 2)에서의 적합한 테스트 집합 {0,1,2,3}은 (그림 3)에 적합하지 않다. 왜냐하면 (그림 3)에서의 두 부분의 if 문에서 참인 값을 운행하는 테스트 데이타값은 존재하지 않는다. 그러나 이런 경로는 어떤 테스트 데이타로도 수행 불가능한 경로이다.

명세-기반 테스팅에서는 적합한 테스팅은 명세를 커버하므로, 두개의 동등 프로그램은 정의에 의해 같은 명세를 갖고, 한 프로그램에 대해 적합한 테스트 집합은 다른 프로그램에 대해서도 반드시 적합하다. 그러나 주 연구에서 대상으로하는 프로그램-기반 테스팅에서의 적합한 테스팅은 소스코드를 커버하므로 두개의 동등프로그램이 약간의 다른 구현을 가질 수 있다. 즉 같은 기능을 갖고 있더라도 적합한 테스팅을 위해서 같은 테스트 데이타를 요구하는 것은 아니다. 즉 (그림 2)와 (그림 3)에서처럼 프로그램-기반 테스팅은 전적으로 구현사항에 의존하기 때문이다.

공리 9) 일반적 다중 변경(general multiple change)

```
input x
if x < 11    then   x <- 0
             else   x <- 1
             end
output x
```

(그림 2)

```
input x
if x= 0    then  x <- 0
           else  x <- 1
           end
output x
```

(그림 3)

P≡Q인 두개의 프로그램이 있을 때, 테스트 집합 T가 P에 적합하나, T가 Q에 적합하지 않다.

반 확장성 공리는 의미적 밀접성을 가진 두 프로그램이 같은 테스트 집합으로 테스트되지는 않았다. 일반적 다중 변경공리도 두 프로그램이 비록 구문적 밀접성을 갖고 있더라도 같은 테스트 집합을 요구하지는 않는다. 구문적 밀접성은 프로그램을 그래프로써 표현시, 같은 구조를 가질 수 있으나 테스트 집합에 의해서 운행되는 경로와 프로그램들의 기능이 같다고 할 수는 없다.

공리 10) 반 분해성(anti-decomposition)

프로그램 P와 그의 부속 컴포넌트 Q가 존재하고 테스트 집합 T가 P에 적합하다고 하면, 테스트 집합 T'는 T의 일부 t에 대해서 컴포넌트 Q의 입력 변수에 대한 값의 벡터 집합일 때 T'는 Q에 적합하지 않다.

이 성질은 비록 프로그램이 적합히 테스트되었다고 하더라도 프로그램의 각 부속 컴포넌트들이 적합히 테스트되었다고 보장할 수는 없다는 것이다. 가령 Q가 P에서 수행불가능하다면 테스트 집합 T'는 공집합이 된다. 그러므로 만약 P에 적합한 테스팅이 Q에도 적합한 테스팅이라고 한다면 공집합이 Q를 테스팅하는데 적합하다고 하여야 한다. 그러면 적합성 기준은 공리 4) 부적합한 공집합을 만족하지 않게 된다.

공리 11) 반 결합성(anti-composition)

프로그램 P와 Q, 그리고 테스트 집합 T가 존재시, T가 P에 적합하고 T에서 입력에 대해 Q의 입구에 들어오는 변수의 모든 값의 벡터 집합이 Q에 대해 적합하더라도 T는 P;Q에 적합하지 않다. P;Q는 P의 출력이 Q의 입력과 동일하여 결합한다.

이 성질은 각각 프로그램을 테스팅하는 것이 두 프로그램의 결합된 전체 프로그램을 모두 적합하게 테스팅하지는 않는다는 것을 의미한다. 가령 (그림 2)와 (그림 3)의 예를 살펴보자. 테스트 집합 T={10,11}이 (그림 2)에 적합하다. 그러면 (그림 2)의 출력 {0,1}은 (그림 3)에 적합하다. 그러나 반 결합성 공리에 의해서 (그림 2);(그림3)에 테스트 집합 T={10,11}은 적합하지 않다. 즉 (그림 2)의 if문의 참값과 (그림 3)의 if문의 거짓을 동시에 운행할 데이타는 없다.

### 3.1.2 Perry의 객체 지향 프로그램의 적합성 테스팅

(1) 클래스상의 구현변경

캡슐화 메카니즘으로 정보은익을 지원하는 클래스는 ADT로써의 전체 구현내용을 응용 프로그램에서 분리가능하며, 그 메소드구현에 대한 접근은 메소드를 구현한 루틴의 헤딩인 메소드 명세(Method Specification)에 국한한다. 그러므로 구현내의 설계를 감출 수 있고, 메소드 명세를 통해 다른 컴포넌트(클래스, 응용 프로그램 등)들간의 상호의존성이 이루어진다. 이러한 캡슐화의 장점은 메소드 구현의 변경은 다른 컴포넌트에 영향을 주지 않는다. 따라서 직관적으로 명세위주 테스팅관점에서 생각하여 단지 변경된 부분만 테스팅을 수행하면 될것 같으나, 반드시 반 결합성 공리로 인해 그 컴포넌트에 관련된 다른 컴포넌트에도 테스팅이 필요하다.

임의의 클래스 C1에서 같은 메소드 명세에 있어서 그 메소드 m1이 적합하게 테스트되었다고 하자. 만약 m1이 명세의 입력이 동등한 m1'로써 변경이 일어난다 할지라도, m1에 적합한 테스트 집합 T는 m1'에 적합하지 않다. 그러므로 m1'에 새로운 테스트 집합이 필요하고, 반 결합성 공리에 의해, 이 메소드를 사용하는 모든 클래스와 응용 프로그램은 재 테스팅되어야 한다.

즉 이와같은 테스팅 속성은 프로그램을 구성하는 각각의 컴포넌트가 적합히 테스트되었다고 하더라도 그 컴포넌트의 결합은 적합히 테스트 되었다고 보장할 수 없음을 기반으로 하는 반 결합성 공리에 의존한다.

(2) 메소드의 상속

일반적으로 잘 사용되고, 테스트된 클래스들은 상위 클래스로써 재사용되고 상위 클래스의 각 메소드들은 이들을 상속받는 클래스에서는 테스팅이 필요가 없을 것이라는 생각을 한다. 그러나 이러한 상위 클래스의 메소드를 상속받는 클래스 상의 메소드에 대해서도 반 분해성 공리에 의해 테스팅이 필요하다.

상위 클래스 C1가 메소드 m1을 가지고 있고, C1에 대해 m1을 적합하게 테스트하였다고 하자. 만약 이 C1을 상속받는 클래스 C2를 생성하였고 메소드 m1을 상속받았을 때는 반 분해성 공리에 의해 클래스 C2에서의 메소드 m1을 재 테스팅하여야 한다.

(3) 메소드의 재정의

모든 객체 지향 프로그래밍언어는 상속받는 클래스가 그 상위 클래스의 같은 이름을 갖고 지역적으로 정의된 메소드로 대치가 가능하도록 한다. 이런 메

소드의 재정의의 경우에 테스팅을 수행할 것인가에 대해 명확하지가 않다. 그러나 비록 두 메소드들이 의미적으로 같은 기능을 가지고 있더라도 반확장성 공리에 의해 반드시 테스팅이 필요하다.

즉 클래스 C1이 메소드 m1을 가지고 있고 C1에 대해 m1을 적합히 테스트하였다고 하자. C1을 상속받는 클래스 C2에서 C1의 메소드를 재정의한 메소드 m1'도 반 확장성 공리로 인해 재 테스팅을 해야 한다.

(4) 다중 상속에서의 메소드 변경

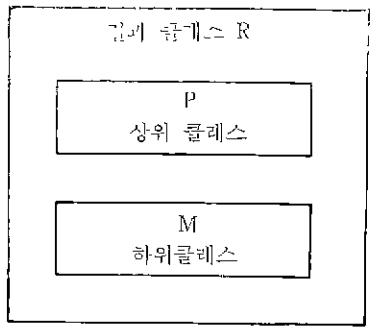
모든 객체 지향 언어에서는 아니지만 일부 객체 지향 프로그래밍언어에서는 다중상속을 지원한다. 즉 각 클래스는 여러개의 상위 클래스를 가질 수 있다. 이런 다중상속에는 여러 상위 클래스에서 같은 메소드를 상속받으므로써 일어나는 "다중 상속" 문제가 있다. 보통 이런 상속의 문제는 "우선 순위"로써 문제를 해결한다. 이런 상속을 받는 메소드의 변경에 대해서도 일반적 다중 변경공리에 의해 테스팅이 요구된다.

즉 상위 클래스 C1과 C2에는 같은 모양의 메소드 m1와 m1'이 있고 클래스 C3가 C1과 C2로부터 상속을 받을 때, 가령 우선순위로써 C1의 메소드 m1을 상속받아서 적합하게 테스트하였다고 가정하자. 만약 클래스 C1가 C2의 m1'으로 상속을 변경시, 일반적 다중 변경공리에 의해 클래스 C3에 대한 메소드 m1'에 대한 재 테스팅이 필요하다.

### 3.2 Harrold의 객체지향 클래스 구조의 점진적 테스팅 방법

#### 3.2.1 점진적 변경 기법(incremental modification technique)

상속성은 현존하는 클래스를 기반으로 새로운 클래스를 개발을 지원한다. 또한 클래스 명세와 코드 공유를 지원하는 메카니즘임이 널리 알려져 있는 사실이다. 하위클래스의 정의에서 변경자(modifier)는 상위 클래스의 속성을 변경하거나 다른 속성을 정의하는 것이다. 따라서 하위클래스는 상위 클래스와 변경자의 합으로서 표현될 수 있다. Wegner[9]는 이러한 상속성 특성을 상위 클래스 P에 변경자(modifier) M을 합하여 결과 클래스 R로 만드는 점진적 변경 기법으로 표현하였다. (그림 4)는 이러한 관계를 나타낸 것이다.




(그림 4) 점진적 수정 기법

#### 3.2.2 객체 지향 클래스 구조의 점진적 테스팅 방법

Harrold 등은 Wegner의 변경자 요인 등을 확장하여 그 요인을 대상으로 테스팅 방법을 제안하였다.

즉, 상속성 구조를 가지는 특성에 따라 상위 클래스의 테스팅 정보를 재사용하여 하위클래스의 테스팅을 유도하게 되는데 그 과정은 다음과 같다.

첫째, 베이스 테스팅(Base testing)으로 상위 클래스가 없는 클래스의 각 멤버 함수에 대한 적합한 테스트를 수행한 후에, 멤버 함수 상호간의 테스트를 수행한다.

둘째, 하위클래스 테스팅(Subclass testing)으로 상위 클래스로부터 수정된 속성(attributes), 상속 받은 속성, 하위클래스에서 새로 정의된 속성들 등의 변경자 요인에 대해 테스팅을 수행하나 가능하면 상위 클래스의 테스팅 정보를 점진적으로(incrementally) 재사용한다. 또한 상속받은 속성들은 하위클래스에서 새로 정의된 속성과의 상호작용을 테스팅함으로서 수행된다.

(1) 변경자 요인

(가) 새로운 속성(New attribute)

A가 상위 클래스 P에 없으며 변경자 M에 정의된 속성이거나, A가 M과 P에 있는 멤버 함수이나 A의 인수 리스트가 M과 P에서 다른 경우이다. 이 경우 A는 결과 클래스 R에 지역적으로 정의되었지 P에 있는 것으로 생각하면 안된다.

(나) 반복적 속성(Recursive attribute)

A가 상위 클래스 P에는 정의되어 있으나 M에는 없음. A는 P에서 지역적으로 정의된 것으로 R에서도 사용 가능하다.

(다) 재정의 속성(Rdefined attribute)

A가 P와 M 모든 부분에 정의되어 있고 A의 인수가 M과 P에서 모두 같다. 이 경우 A는 M에서 지역적

```
class P {
  private:
      int i;
      int j;
  public:
     P(){}
     void A(int a, int b)
       { i=a; j=a+2*b}

     virtual int B()
         {return i;}
     int C()
         {return j;}
};
```

**(그림 5) 상위 클래스 P**

```
class R:public P{
  private
    float i;
  public :
    R(){}

    void A(int a)
       { P::A(a, 0);}
    virtual int B()
       {return 3*P::B();}
    int C()
       {return 2*P::C();}
};
```

**(그림 6) 하위 클래스 R**

으로 정의된 속성이다.

(라) 가상의 새로운 속성(Virtual-new attribute)

A는 M에 명세되어 있으나 그의 구현이 M에서 나타나지 않고 후에 구현되거나 혹은, A는 M과 P에 명세되어 있으나 P에서 구현되지 않고 나중에 구현된다. 그러나 A의 인수 리스트는 M과 P에서 다르다. 이 경우 A는 P가 아니라 R에서 지역적으로 정의된 속성이다.

(마) 가상의 반복적 속성(Virturaq-recursive attribute)

A는 P에 명세되어 있으나 P에서 구현되어 있지 않고 나중에 구현 그리고 A는 M에 정의되어 있지 않다. 이 경우 A는 P에서 지역적으로 정의되었으나 R에서도 사용 가능하다.

(바) 가상의 재정의 속성(Virtual-redifine attribute)

A는 P에 명세되어 있으나 그의 구현은 나중에 이루어지고 A는 M에 정의되어 있는데 P에서의 A와

```
R의 변경자 속성

private:
 float i;    // 새로운 속성

public

 void A(int a,  int b)     // 반복적인 속성
   {i=a;j=a+2*b;}
 void A(int a)             // 새로운 속성
   {P::A(a, 0);}
 virtual int B()           // 가상의 재정의 속성
   {return 3*P::B();}
 int C()                   // 재정의 속성

 hidden
   int i;
   int j;
```

**(그림 7) 하위클래스 R에서의 변경자 속성**

같은 인수 리스트를 가진다. 이 경우 A는 M에서 지역적으로 정의된 속성이다.

이러한 변경자 요인에 대한 예제 프로그램 분석 내용이 (그림 5), (그림 6), (그림 7)에 나타나 있다. (그림 5)는 상위 클래스 P를 나타낸 것이고, (그림 6)은 그것을 상속받는 하위클래스 R을 나타낸다. (그림 7)은 하위클래스 R에서의 변경자 속성에 대한 사항이다.

(2) 계층적 구조를 가진 점진적 클래스 테스팅

(가) 베이스 클래스 테스팅(base class testing)

Perry의 적합성 테스팅 관점중 반 분해성(anti-decomposition) 공리를 기반으로 각 맴버 함수에 대해서 테스팅을 수행한다. 즉, 반 분해성 공리는 한 클래스의 테스팅이 적합하였다고 하더라도 그 구성 요인에 대한 적합한 테스팅이 필요함을 의미한다. 각 멤버 함수는 명세 위주(spec-based) 테스팅과 프로그램 위주(program-based) 테스팅을 나누어 수행한다. 테스팅 정보(tesing history)는 다음과 같이 그 구성된다.

$\{m_i, (TS_i, test?), (TP_i, test?)\}$

$m_i$는 멤버 함수이고, $TS_i$는 명세 위주 테스트 집합이며, $TP_i$는 프로그램 위주 테스트 집합이다. 또한 test?는 테스트 집합을 수행할 것인지? 수행 안할 것인지?를 나타낸다.

베이스 클래스 테스팅에서는 또한 반 결합성(anti-composition) 관점에서 다음과 같이 나뉘어 테스팅을 수행한다.

첫째, 내부 클래스(intra-class) 테스팅 관점에서 한 클래스에서 각 멤버 함수간의 상호 작용을 테스팅한

다.

둘째, 상호 클래스(inter-class) 테스팅 관점에서 클래스 간의 각 멤버 함수간에 있을 수 있는 상호 작용을 테스팅한다.

셋째, 내부 클래스 통합(intra-class integration) 테스팅 관점으로 각 클래스 그래프들을 통합하여 그들의 인터페이스에 대한 테스트 케이스를 개발, 테스팅을 수행한다. 여기서 클래스 그래프란 노드는 멤버 함수나 데이타 멤버를 나타내고, 에지는 메세지를 나타내는 그래프이다.

통합 테스팅 관점에서의 테스팅 정보는 다음과 같이 표현된다.

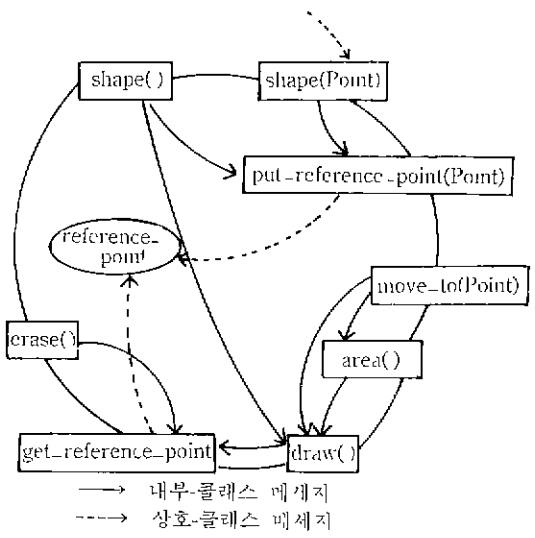
{$m_i$, ($TIS_i$, test ? ), ($TIP_i$, test ? )}

$m_i$: 클래스 그래프 서브그래프의 루트 노드.

$TIS_i$: 명세 위주의 통합테스트 집합

```
class Shape {
  private:
          Point reference_point;
  public:
          void put_refernece-point(Point);
          point get_refernce_point();
          void move_to(Point);
          void erase();
  virtual void draw()=0;
  virtual float area();
          shape(Point);
          shape();
}
```

**(그림 8) 클래스 shape의 정의**




**(그림 9) shape에 대한 클래스 그래프**

| Shape에 대한 테스팅 정보 | | |
|---|---|---|
| 속성 | 명세 위주의 테스트 집합 | 프로그램 위주의 테스트 집합 |
| 독립적 멤버 함수 | | |
| put_refernce_point | ($TS_1$, Y) | ($TP_1$ Y) |
| get_refernce_point | ($TS_2$ Y) | ($TP_2$ Y) |
| move to | ($TS_3$, Y) | ($TP_3$, Y) |
| erase | ($TS_4$ Y) | ($TP_4$, Y) |
| draw | ($TS_5$ Y) | ( ) |
| area | ($TS_6$ Y) | ($TP_6$, Y) |
| shape | ($TS_7$ Y) | ($TP_7$, Y) |
| shape | ($TS_8$, Y) | ($TP_8$, Y) |
| 멤버 함수의 상호 작용 | | |
| move to | ($TIS_9$ Y) | ($TIP_9$ , Y) |
| erase | ($TIS_{10}$ Y) | ($TIP_{10}$ Y) |

**(그림 10) 클래스 shape에 대한 테스팅 정보**

```
class Triangle  public Shape {
      private
            Point vertex2
            Point vertex3.
      public
            Point get_vertex1()        //새로운 속성
            Point get vertex2().       //새로운 속성
            Point get vertex3():       //새로운 속성
            void set vertex1(Point).   //새로운 속성
            void set_vertex2(Point).   //새로운 속성
            void set vertex3(Point).   //새로운 속성
            void draw();    //기상의 재정의된 속성
            float area().   //기상의 재정의된 속성
            triangle().                    //새로운 속성
            triangle(Point Point, Point).  //새로운 속성
```

**(그림 11) 하위 클래스 triangle**

$TIP_i$: 프로그램 위주의 통합테스트 집합

test ? : 테스트 집합이 수행할 것인가. 부분적으로 수행할 것인가를 나타낸다.

이러한 예를 (그림 8)의 클래스에 대해 (그림 9)은 클래스 그래프를 나타낸다.

(그림 10)은 Class Shape에 대한 테스팅 정보이다.

(나) 하위클래스의 테스팅

상위 클래스 P의 테스팅 정보를 하위클래스 결과 R의 테스팅 정보를 전환하게 된다. 즉, 변경자의 속성에 따라 상위 테스팅에서의 테스팅 정보를 하위 테스팅에서 그대로 재사용할지, 아니면 전혀 새로운 테스트 집합이 요구되는 지를 점진적으로 결정하여 테스팅을 실시한다. 다음 (그림 11)은 (그림 8)의 클래스 shape를 상속받는 클래스 triangle을 나타내고 (그림 12)는 triangle에 대한 새로 작성한 테스팅 정보이다. (그림 12)에서 'Y'나 'P'로 표현된 것은 하위클래스에서 테스트 집합이 재사용되는 것이고, 'N'

| triangle에 대한 테스팅 정보 | | |
|---|---|---|
| 속성 | 명세 위주 테스트 집합 | 프로그램 위주 테스트 집합 |
| 독립적 멤버 함수 | | |
| put_refernce_point | ($TS_1$, N) | ($TP_1$, N) |
| get_refernce_point | ($TS_2$ N) | ($TP_2$ N) |
| move_to | ($TS_3$ N) | ($TP_3$ N) |
| erase | ($TS_4$, N) | ($TP_4$ N) |
| draw | ($TS_5$, Y) | ($TP_5$ Y) |
| aren | ($TS_6$ Y) | ($TP'_6$ Y) |
| shape | ($TS_7$, Y) | ($TP_7$ Y) |
| shape | ($TS_8$ Y) | ($TP_8$, Y) |
| get_vertex1 | ($TS_{11}$ Y) | ($TP'_{11}$, Y) |
| get_vertex2 | ($TS'_{12}$ Y) | ($TP'_{12}$ Y) |
| get_vertex3 | ($TS_{13}$ Y) | ($TP'_{13}$ Y) |
| put_vertex1 | ($TS_{14}$ Y) | ($TP'_{14}$ Y) |
| put_vertex2 | ($TS'_{15}$ Y) | ($TP'_{15}$ Y) |
| put_vertex3 | ($TS_{16}$ Y) | ($TP'_{16}$ Y) |
| triangle | ($TS_{17}$ Y) | ($TP'_{17}$, Y) |
| triangle | ($TS'_{18}$ Y) | ($TP'_{18}$ Y) |
| 멤버 함수의 상호 적용 | | |
| move_to | ($TIS'_9$ P) | ($TIP_9$ P) |
| erase | ($TIS_{10}$ P) | ($TIP''_{10}$ P) |
| aren | ($TIS_{19}$ Y) | ($TIP''_{19}$ Y) |
| get_vertex1 | ($TIS_{20}$ Y) | ($TIP''_{20}$ Y) |
| get_vertex2 | ($TIS'_{21}$ Y) | ($TIP_{21}$ Y) |

**(그림 12) triangle에 대한 테스팅 정보**

는 수행되지 않음을 표현한다. 또한 '의 표현은 새로운 테스트 집합이 요구됨을 나타내고, "는 앞으로 새롭게 테스트 집합이 필요할 수도 있음을 표현한다. 가령 put_refernce_point에서 ($TS_1$, N)란 명세위주의 테스트 집합을 다시 사용하여 테스팅을 수행하지 않아도 됨을 보여준다. 또 draw에서 ($TS_5$, Y) 표현은 가상의 재정의 속성이므로 명세 위주의 테스팅에서는 상위 클래스에서의 테스팅 정보를 재사용하여 테스팅을 수행함을 의미하고, ($TP'_5$, Y)의 의미는 새로운 프로그램위주의 테스트 집합으로 다시 테스팅이 필요함을 나타낸다.

### 3.3 기존 연구의 문제점 및 연구 방향

#### 3.3.1 문제점

전술한 방법론 이외의 객체 지향 프로그램 테스팅의 방법론으로 Smith의 FOOT(A Framework for Object-Oriented Testing) 등이 있다. 이러한 방법론들은 객체 지향 패러다임에서의 테스팅이라는 주제로서 신선함을 가져다주고 있음에는 틀림이 없으나 기존의 프로시듀어 프로그램 기반의 놀랍도록 발전된 테스팅 방법론, 툴들을 접해본 사람들은 왠지 나약하다는 느낌을 떨쳐버리지 못한다. 따라서 필자는 다음과 같은 문제점을 지적한다.

첫째, 지금까지 주요 객체 지향 테스팅 방법론의 흐름이 Weyuker의 적합성 테스팅 공리를 대상으로 객체 지향 프로그램 특성에 적용하였으나 실제 자동화하지는 못하고 전략적 차원에서 방법론에 제한되고 있다.

둘째, 단위 테스팅, 통합 테스팅, 프로그램 위주 테스팅, 명세위주 테스팅 등에 대한 명확한 구분과 구현 방법에 대해 미흡하다. Perry의 테스팅 방법은 객체 지향 테스팅을 어떻게 실시하는가에 대해 시사 속성에 대한 구조를 적절한 테스트 집합을 점진적으로 생성하여 테스트 집합의 생성을 줄임으로서 효과있는 테스팅을 수행할 수 있음을 역설하였다. 그러나 그의 방법도 실제로 멤버 함수에 대한 테스트는 데이타 흐름에서 all-use방법에 의한 테스팅 수행으로 테스트를 수행할 수 있고 명세 위주의 테스팅 방법에 대해서는 자세한 언급을 하지않아 기존의 방법을 그냥 도입할 수 밖에 없다.

셋째, 프로시듀어 프로그램에서의 테스팅 완결은 결국 테스트 케이스 생성으로 인한 수행이라고 볼때, 객체 지향 프로그램에서의 테스트 케이스 생성은 수많은 멤버 함수와 메세지 교환으로 기존의 데이타 흐름 분석으로 유도된 테스트 케이스 생성은 더욱 복잡하여지게 될 것이다. 그러나 이러한 연구들이 이루어지지 않고 있다.

#### 3.3.2 연구 방향

Weyuker의 적합성 테스팅 공리가 기존의 테스팅 분야에서 타당하게 받아들여지고 있고 객체 지향 프로그램에서도 물론 그 특성에 따른 변화가 필요하게 되겠지만 결국 테스팅 분야의 주요 기반이 되고 있다. 따라서 본 연구실에서는 Weyuker의 공리를 기반으로 클래스 내부 테스팅, 상호간의 인터페이스 테스팅을 수행할 그래프를 생성, 각 그래프를 대상으로 객체 지향 프로그램 특유의 에러를 쉽게 발견할 테스트 케이스 생성 전략을 연구하고 있다.

첫째, 하나의 클래스를 대상으로 하는 내부 클래스 테스팅(intra-class testing)로서 클래스를 구성하는 비공개 영역의 변수들에 대한 멤버 함수(member function)의 사용 그래프를 구성한다. 여기서 다른 클래스를 호출하는 부분은 수행하지 않는다.

둘째, 클래스간의 테스트로서 상호 클래스 테스팅(inter-class testing)을 수행하는데 여기서는 클래스간의 호출순서, 그래프를 구축하고 그 흐름을 변수

차원에서 추적하는 그레프를 구축한다.

## IV. 결 론

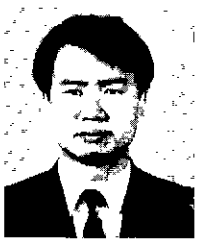
지금까지 객체 지향 프로그램 테스팅의 기존 연구와 문제점을 서술하였다. 1987년 객체 지향 컴퓨팅의 튜토리얼 주관자인 Peterson[10]은 서문에서 객체 지향의 선풍은 분석, 설계, 언어, 그리고 "객체 지향 프로그램 테스팅"이라는 신용어를 만들어 낼 것이라고 예측했었다. 그의 지적대로 분석, 설계만큼의 많은 수의 논문은 아니더라도 몇 가지 주목할 논문들이 90년대 이후 발표되어지고 있다. 현재, 미국에서는 HP, AT & T 등의 연구소에서, 또 대학에서는 좋은 연구 테마로서 활발히 연구가 진행중이며 C++ 프로그램의 활성화로 인한 디버깅 툴과 맞물려서 자동화 툴의 생산이 시도되고 있다. 따라서 앞으로 우리나라도 기존의 테스팅 분야에서의 연구 인력을 충분히 활용, 객체 지향 프로그램 테스팅 분야로의 연구 활성화를 유도하여야 할 것이다.

## 참 고 문 헌


1. I. Jacobspon, Object-Oriented Software Engineering, Addison-Wesly, 1992.
2. G. Booch, Object-Oriented Design with Appications, Benjamin/Cummings, 1991.
3. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1991.
4. P. Coad and E. Yourdon, Object-Oriented Analysis, Yourdon Press, 1991.
5. D. E. Perry and G. E. Kaiser, "Adequate testing and Object-Oriented Programming," Journal of Object-Oriented Programming, Vol. 2, January /February 1990, pp. 13~19.
6. E. J. Weyuker, "Axiomatizing software test data adequacy," IEEE Transactions on SE., Vol. SE-12, No. 12, pp. 1128~1138, December 1986.
7. M. J. Harrold, J. D. McGregor, and K. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", In Proceedings of the 14th International Conference on Software Engineering, 1992, pp. 68~80.
8. M. D. Smith and D. J. Robson, "A Framework for testing object-oriented programs," Journal of Object-Oriented Programming, Vol. 5, No. 3, June 1992, pp. 45~53.
9. P. Wegner and S. B. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn't like," Proceedings of ECOOP '88, Springer-Verlag, 1988, pp. 55~77.
10. G. E. Peterson, "Tutorial: object-oriented computing," IEEE Computer Society press, Vol. 2, 1987, pp. 1~3.



### 한 규 정

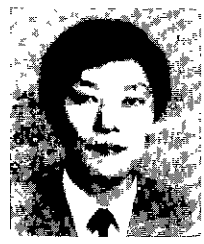


1986 중앙대학교 문리과대학 화학과 졸업(학사)
1988 중앙대학교 대학원 전자계산학과 졸업(석사)
1991 중앙대학교 대학원 전자계산학과 졸업(공학박사)
1987 ~1988 중앙대학교 전자계산학과 연구조교
1991 ~1992 중앙대학교 수피컴 연구소 객원연구원
1990 ~현재 선경 정보교육센터 자문위원
1992~현재 공주교육대학교 전임강사
관심분야 : 소프트웨어 공학(특히, 객체 지향 분석/설계, 객체지향 테스팅, 소프트웨어 메트릭스)

### 이 경 환

1980 중앙대학교 대학원 수학과 응용수학 전공(이학박사)
1982 미국 Auburn대학에서 S/W Engineering에 대하여 연구
1986 서독 Bonn대학 Institute for informatik과 공동연구
1971 ~현재 중앙대학교 컴퓨터공학과 교수
1993 ~현재 중앙대학교 공과대학장
관심 분야 : 소프트웨어 공학(특히, 소프트웨어 테스팅, 소프트웨어 재사용, Objected-Oriented Design)

### 양 해 술



1975 홍익대학교 공과대학 졸업(학사)
1978 성균관대학교 정보처리학과 졸업(석사)
1991 日本 오사카대학 기초공학부 정보공학과 소프트웨어 공학 전공(공학박사)
1975 ~1979 육군 중앙정리단 전자계산실 근무
1984 ~1992 성균관대, 명지대 경영대학원 강사
1986 ~1987 日本 오사카대학 객원 연구원
1980 ~현재 강원대학교 전자계산학과 교수
관심 분야 : 소프트웨어 공학(특히, S/W 품질보증과 평가, SA/SD, OOA/OOD/OOP, CASE), 전문가 시스템