회귀 시험을 위한 Java 소프트웨어의 수정영향 분석

정혜령○ 김상운 권용래
한국과학기술원 전자전산학과
{hlzheng, swkim, kwon}@salmosa.kaist.ac.kr

Change Impact Analysis to Java Software for Regression Testing

Huiling Zheng○, Sang-Woon Kim, Yong Rae Kwon
Dept. of Electronic Engineering and Computer Science, KAIST

요 약

객체지향 소프트웨어에서의 회귀 시험은 정확성과 효율성의 측면에서 그 수정영향 분석을 할 때, 시험의 기본 단위를 다양하게 하여 접근한다. 클래스, 메소드 및 문장 단위의 수정영향 분석 기법을 살펴 볼 때 메소드를 하나의 시험 단위로 보는 기법이 비교적 효율적이고 효과적이라고 판단된다. 본 논문에서는 Java 소프트웨어의 메소드 단위에서의 회귀 시험을 위한 효율적인 수정영향 분석 기법을 제안하기 위해 수정의 의존관계를 분석하고 발생 가능한 모든 수정을 시험을 유발하는 수정과 그렇지 않은 수정으로 구분한다. 시험을 유발하는 수정에 대한 수정영향 분석만을 통해 중복 영향 분석을 피해 효율성을 높인다. 또한 본 논문에서는 수정영향 분석의 자동화를 위한 설계 및 프로토타입을 제안한다.

1. 서론

소프트웨어에 수정이 가해지면 수정된 부분이 요구사항을 만족하는지 확인하기 위해 회귀 시험을 필요로 한다. 회귀 시험은 이전 시험 단계에서 사용되었던 테스트 케이스를 재사용할 수 있다. 효율적인 테스트 케이스의 재사용을 위해 소프트웨어의 수정된 부분과 그로 인해 영향을 받는 부분을 찾고 검출된 부분들을 수행하는 테스트 케이스를 선택하는 선택적 회귀시험(regression testing selection) 방법이 흔히 사용된다[1].

선택적 회귀시험의 설계는 수정영향 분석(change impact analysis)에서의 정확성과 효율성 관점에서 평가될 수 있다. 정확성은 시험에 필요한 테스트 케이스만을 얼마나 잘 선택하는지에 대한 평가기준이다. 효율성은 수정영향 분석에 필요한 시간에 대한 평가기준이다[2]. 정확성이 높고 효율성이 좋을수록 선택적 회귀 시험의 설계가 좋다고 할 수 있다.

정확성과 효율성이라는 두 가지 측면에서 회귀 시험에 대한 기법을 살펴보면 그 시험 단위를 다양하게 구분하는 것으로 접근 가능한데 객체지향 프로그램에서는 크게 클래스 단위, 메소드 단위 그리고 문장단위를 기본 시험단위로 수정영향 분석을 한다. 클래스를 기본 시험단위로 하는 기법[3, 4, 5]은 수정이 발생한 클래스와 그 클래스와 상속, 집합, 연관관계에 있는 클래스를 분석하는 것을 통해 수정영향 분석을 완성한다. 이런 기법은 클래스라는 큰 단위에서 분석을 함으로써 효율적인 이점을 갖고 있지만, 클래스 내에 수정이 발생한 부분이 매우 작을 때에도 클래스 전체를 시험하는 테스트 케이스를 선택하므로 정확성이 떨어진다. 문장 단위의 수정영향 분석[6, 7]은 소프트웨어를 구현하는 가장 작은 단위에서 수정영향 분석을 진행하므로 정확성은 좋지만, 모든 문장에 대해서 비교분석을 거쳐야 하므로 수정된 소프트웨어의 사이즈가 클 때, 효율성이 떨어지는 단점이 있다. 이런 기법들의 정확성과 효율성 비교에서 볼 때 클래스 단위와 문장 단위의 중간 사이즈인 메소드 단위에서의 수정영향 분석이 비교적 정확하고 효율적이라고 판단된다.

본 논문에서는 Java 소프트웨어의 회귀 시험을 위한 메소드 단위에서의 효율적인 수정영향 분석 기법을 제안한다. Java 소프트웨어에서 발생 가능한 수정들을 조합 가능한 작은 단위의 수정으로 쪼개고, 각 수정에 대한 동반 수정들을 살펴본다. 이런 수정 중에서 행위의 변경이 일어나 시험이 필요한 수정 그룹을 찾아낸다. 시험이 필요한 수정에 대한 영향분석을 하여 수정이 발생한 메소드와 영향을 받는 메소드를 찾아낸다. 이런 메소드들에 대해서는 단위시험이 필요한 메소드와 통합 시험이 필요한 메소드로 재 구분한다. 수정영향을 받는 모든 수정에 대해서 분석을 하는 것이 아니라, 시험을 필요로 하는 수정을 찾아내고, 이것에 대한 분석만을 함으로써 중복 분석을 피하기 때문에 효율적이다. 또한 수정영향 분석을 자동화 시키기 위한 자동화 도구의 프로토타입을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 객체지향 소프트웨어의 메소드 단위에서의 수정영향 분석에 대한 기존 연구에 대해 설명한다. 3장에서는 Java 소프트웨어의 유지보수단계에서 발생 가능한 수정을 조합 가능한 수정과 동반 가능한 수정으로 분석하고, 또한 이런 수정에 대해서 시험을 유발하는 수정을 찾아내고, 시험을 유발하는 수정들에 대해서는 상속관계, 다형성, 동적 결합에 의한 수정영향 분석을 완성한다. 4장에서는 우리가 제안한 기법에 대한 자동화 도구를 위한 프로토타입에 대해 설명한다. 그리고 5장에서는 결론 및 향후 연구방향에 대해 제시한다.

2. 관련연구

Jang[8]은 C++ 소프트웨어의 메소드를 단위시험 대상으로 하는 회귀 시험 방법을 제안하였다. C++ 소프트웨어에서 발생할 수 있는 수정의 종류를 조사하고, 그 중 상속성과 동적 결합의

영향으로 인해 메소드 사이의 관계에 변경을 일으킬 수 있는 수정을 기본 수정으로 정의하였다. 또한 클래스 단계의 호출 그래프를 이용하여 기본 수정에 의해 영향 받는 메소드 단위의 수정을 순차적으로 분석한다. 분석에 의해 수정영향 받는 메소드들에 대해서는 단위시험을 필요로 하는 메소드와 통합시험을 필요로 하는 메소드, 두 가지 부류로 구분하였다. 그러나 발생 가능한 수정들은 모두 독립적이지 않고, 몇몇의 수정들 간에는 의존성이 나타난다. 즉 하나의 수정이 발생했을 때 여러 가지 수정들이 동반 될 수 있다. 따라서 이 기법의 경우, 발생 가능한 모든 수정에 대해 각각 수정 분석을 수행하므로, 한 수정과 동반해서 발생하는 수정에 대해서도 중복되는 수정영향 분석이 수행됨으로 효율성이 떨어지는 단점이 있다. 또한 Java 소프트웨어의 수정영향 분석에 대해서는 적용 가능성이 증명되지 않았다.

Ryder, Tip[9, 10, 11, 12] 등은 Java 소프트웨어를 대상으로 결함 위치결정(fault localization)을 위한 메소드 단위에서의 수정영향 분석 기법을 제안하였다. Java 소프트웨어에서 발생 가능한 구문(Syntax) 수정을 몇 개의 기본 단위로 분류하였고 이 기본 단위 수정들의 발생순서를 살펴보았다. 또한 테스트 케이스가 주어졌을 때, 발생한 수정과 테스트 케이스 사이의 관계를 정의했다. 하나의 수정이 발생하였을 시, 수정된 부분을 수행할 수 있는 테스트 케이스가 의존관계를 가지는 다른 수정에도 기능적 변화를 준다고 정의를 했다. 즉 하나의 테스트 케이스에 의해 프로그램의 기능적 에러가 검출되었을 경우, 테스트 케이스로 인해 수정된 부분이 수행됨으로 에러가 발견 가능했거나, 의존관계에 있는 다른 수정이 수행되었기 때문에 기능적 에러가 검출될 수도 있다. 하지만 이 기법은 결함 위치결정을 위한 수정영향 분석 기법이므로 수정이 발생하지 않았지만, 그 수정으로 영향을 받을 수 있는 다른 메소드에 대해서는 언급을 하지 않고 있다. 또한 의미(semantic) 수정에 대해서는 설명되어 있지 않다.

Abdi[5]는 Java 소프트웨어에서 발생 가능한 모든 수정을 클래스, 인스턴스 변수, 메소드 레벨에서 자세히 분류하였다. 구문, 의미 수정에 관해서 모두 52개의 작은 분류로 다루었으며 클래스 단위의 수정영향 분석을 하였다. 즉 하나의 수정이 발생하였을 때, 그 수정이 발생한 클래스와 상속, 집합, 연관관계에 있는 클래스 중에서 시험이 필요한 클래스들을 분류하였다. 이 기법은 시험의 기본단위를 클래스로 하여 정확성이 떨어지는 단점이 있다. 본 논문에서는 이들의 분류를 기반으로 향상된 수정 분류를 하고 분류된 수정들의 의존관계를 분석하며 메소드를 시험단위로 하여 수정영향 분석을 하여 정확성과 효율성을 높인다.

3. 수정영향 분석

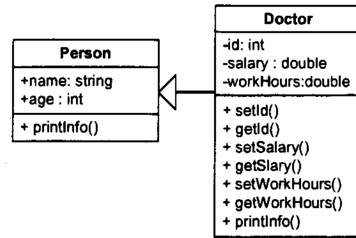
3.1 수정의 의존관계

객체지향 프로그램의 수정은 하나의 수정을 여러 수정의 조합으로 볼 수 있거나 또는 일련의 수정이 동반되는 특성을 갖고 있다.

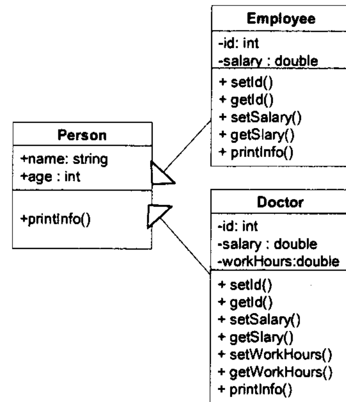
1) 수정의 조합 가능성

<그림 1>은 병원 직원관리 프로그램에 대한 클래스 다이어그램의 일부분을 나타낸다.

이 프로그램에 Employee 클래스를 추가하여 Person 클래스로부터 상속 관계를 가지도록 1차 수정이 발생한다고 가정한다. 이 수정은 Employee 클래스의 추가와 Person 클래스에 대한 상속관계의 추가를 포함한다. 완료된 수정은 변경된 클래스 다이어그램 <그림 2> 에서 보여주고 있다.



<그림 1. 병원 직원관리 클래스 다이어그램 >



<그림 2. 1차 수정 후의 클래스 다이어그램>

Employee 클래스의 추가는 본 클래스에서 사용되는 인스턴스 변수의 추가와 메소드 추가의 조합으로 볼 수 있다. 이 예제에서 가해진 수정과 그를 조합하는 수정을 <표 1>에 나열하였다. 추가(add)된 *C*는 클래스, *m*은 메소드, *v*는 인스턴스 변수를 의미한다.

<표 1. 예제 프로그램의 1차 수정과 조합되는 수정>

수정	조합되는 수정
add(*C*)	add(*v*) add(*m*)
add(*I*)	-

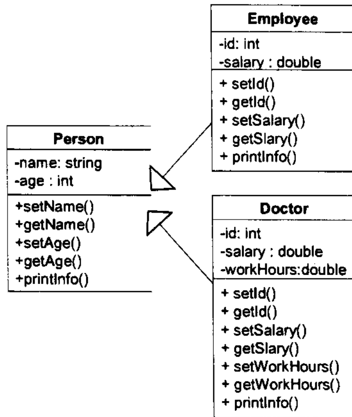
수정의 조합 가능성은 하나의 비교적 큰 단위의 수정이 일어났을 때, 이를 작은 단위의 수정들의 조합으로 보고, 각 수정의 수정영향 분석을 하는 것을 통해 전체 분석을 완성할 수 있음을 의미한다.

2) 수정의 동반 가능성

앞선 예제에서 Person 클래스에서 사용하는 인스턴스 변수 name과 age는 public으로 선언되어, 자식 클래스인 Employee와 Doctor, 그리고 외부에서 직접 접근 가능하다. 본 상태에 Person 클래스의 두 인스턴스 변수를 모두 private으로 수정하여, 외부 클래스에서의 직접 접근 권한을 제한한다고 가정하자.

이 수정에는 다음과 같은 일련의 수정이 동반된다. 우선 Person 클래스에는 private 변수의 정의와 사용을 위한 get/set 메소드가 추가되어, 수정 전과 동일한 접근 성을 제공해야 한다. 상속받은 Doctor 클래스와 Employee 클래스에서는 새로 정의된 get/set 메소드를 호출하여 접근하도록 변경된다. 동시에 이전, 외부 클래스에서 단순 인스턴스 변수에 대

한 정의 및 사용을 제거하는 수정이 발생한다. 가능한 동반 수정들까지 완료된 후 변경된 클래스 다이어그램은 <그림 3> 에서 보여주고 있다.



<그림 3. 2차 수정 후의 클래스 다이어그램 >

2차 변경에서 가해진 수정과 동반되는 수정들은 <표 2>에 나열하였다. <표 1>에서 사용되는 기호를 사용하며 추가적으로 추가 및 삭제(del)된 ***vd***는 인스턴스 변수의 정의, ***vu***는 인스턴스 변수 사용, ***call***은 호출 문장을 의미한다.

<표 2. 예제 프로그램의 2차 수정과 동반되는 수정>

수정	동반되는 수정
Scope change (public->private)	add(***m***) del(***vd***) del(***vu***)
add(***m***)	add(***call***)
add(***call***)	-
del(***vd***)	del(***vu***)
del(***vu***)	-

<표 2>의 수정에서, 원래 **Person** 클래스의 인스턴스 변수 name과 age의 범위를 public에서 private으로 바꾸는 수정 자체는 행위에 대해 아무런 영향을 일으키지 않지만, 본 수정으로 인해 수정된 변수를 참조하는 메소드가 추가*(add(m))*, 인스턴스 변수의 정의의 제거*(del(vd))*와 인스턴스 변수의 사용이 제거되는 경우*(del(vu))*가 발생할 수 있고, 또다시 이들로 인해 메소드 *m*에 대한 호출이 추가될 수도 있다.

수정의 동반 가능성은 하나의 수정이 발생하였을 때, 원인 수정과 이에 동반하는 모든 수정을 하나의 변경으로 생각하고 수정영향 분석을 하는 것이 아니라 이런 수정들을 모두 별도의 수정으로 생각하고 각 수정에 개별분석을 하는 것을 통해 중복되는 수정 영향분석을 피할 수 있게 한다. 즉 위의 표에서, 메소드의 추가((add(m))와 이에 대한 호출(add(call))의 추가를 하나의 수정 영향으로 분석 하는 것이 아니라, 두 개의 추가를 전혀 다른 수정으로 보고 분석하여 전체 분석을 완성함을 의미한다.

<표 1>과 <표 2>와 같은 방식으로 원인이 되는 수정과 해당 수정에 의해 동반되거나 조합되는 모든 의존관계를 수집, 분석하였다. 본 연구는 Abdi [5]가 분류한 Java에서 발생 가능한 모든 수정을 기반으로 위의 단계와 같이 의존관계를 분석하여 [부록 1]에 수록하였다. Abdi의 분류된 수정 중, 공통 특성을 갖는 수정에 대해서는 통합하고, 메소드 구현의 수정에 대해서는 호출관계의 추가 및 삭제로 구분하였다. 또한 수정의 동반 가능성과 조합 가능성에 의해 이루어지는 수정은 간단히 상관관계가 있는 수정(related change)으로 분류했다.

3.2 수정 그룹의 정의

조합되는 수정과 동반되는 수정들은 다음과 같은 두 가지 유형의 수정으로 구분할 수 있다.

1) 시험을 유발하는 수정(testable Change)

시험을 유발하는 수정이란 그 수정 하나만 발생하더라도 소프트웨어의 요구사항의 만족여부를 판단하기 위해 단위시험 혹은 통합시험을 필요로 하는 수정을 말한다. 예를 들면 메소드 추가, 호출관계 문장 추가 등이다.

2) 시험을 유발하지 않는 수정

시험을 유발하지 않는 수정이란 그 수정 자체만으로는 단위시험이나 통합시험을 필요로 하지 않는 것을 말한다. 앞선 예제에서 **Person** 클래스의 인스턴스 변수 name과 age의 범위를 public에서 private으로 바꾸는 수정은 이 그룹에 해당한다.

본 논문에서는 소프트웨어에 수정이 발생하였을 때 시험을 유발하는 수정에 대해서만 수정영향 분석을 하는 것을 통해 전체 분석을 완성한다.

3.3 시험을 유발하는 수정과 그 수정영향 분석

본 논문에서는 행위의 변경이 발생하거나 그 영향을 받는 메소드에 대해서는 단위시험을 수행하고, 또한 클래스 사이의 상속관계, 메소드 사이의 호출관계, 인스턴스 변수의 정의 및 사용에 대한 분석을 통해 통합시험의 필요 여부를 결정한다. 분석된 각 수정과 동반되는 수정[부록 1]들은 메소드 추가 혹은 삭제, 호출관계의 추가 혹은 삭제를 동반한다. 이런 동반되는 수정에서 단위시험이나 통합시험이 유발되는 수정에 대해 구분하고, 각각의 유발되는 시험에 대해 다음과 같이 시험한다.

1) 인스턴스 변수 정의에 대한 변경

인스턴스 변수 정의에 대한 변경(추가 혹은 삭제)은 변경된 메소드에 대해 단위시험을 해야 하며 또한 이 인스턴스 변수를 정의 또는 사용하는 모든 메소드와의 통합시험을 하여 인스턴스 변수의 정의와 사용에서 충돌이 없도록 한다[15].

2) 인스턴스 변수 사용에 대한 변경

인스턴스 변수 사용에 대한 변경(추가 혹은 삭제)은 변경이 발생한 메소드에 대해 단위시험을 해야 하며 또한 인스턴스 변수 사용의 추가는 이 인스턴스 변수를 정의하는 모든 메소드와의 통합시험을 하여 인스턴스 변수의 정의와 사용에 충돌이 없도록 한다.

3) 일반(non-overriding) 메소드 추가

메소드의 추가는 새 메소드의 기능확인을 위해 단위시험을 한다.

4) 오버라이딩(overriding) 메소드 추가

부모 클래스에 있는 메소드를 재정의하는 경우, 상속관계에 의한 동적 결합이 발생하여, 부모 클래스의 메소드에 대한 호출이 새로 추가된 자식 메소드를 호출할 수 있다. 따라서 상속된 메소드는 해당 메소드를 호출하는 메소드와 통합시험을 한다.

5) 오버라이딩 메소드 삭제

일반 메소드의 삭제는 시험을 필요하지 않는다. 그러나 삭제된 메소드가 오버라이딩 메소드일 경우는 해당 메소드에 대한 호출이 부모 클래스의 오버라이딩 했던 메소드와 호출관계를 형성한다. 본 호출관계에 해당하는 메소드 간의 통합시험

이 필요하다.

6) 호출관계 추가

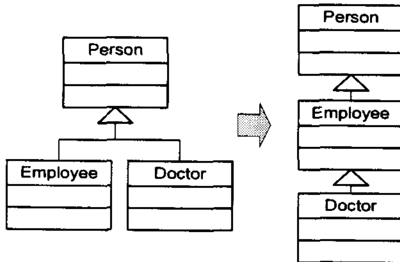
호출관계 추가는 메소드 내용의 변경이므로 우선 변경된 메소드의 기능확인을 위한 단위시험이 필요하다. 또한 호출대상 메소드와의 통합적 기능의 정확여부를 검증하기 위한 통합시험을 한다. 이때 정적 결합과 동적 결합관계를 가지는 모든 메소드와 통합시험을 해야 한다.

7) 호출관계 삭제

호출관계 삭제도 메소드 내용의 변경이므로 우선 변경된 메소드의 기능확인을 위한 단위시험이 필요하다. 또한 오버라이딩 된 메소드에 대한 호출문장의 삭제는 그 메소드가 오버라이딩한 부모클래스의 메소드와 함께 통합시험을 한다.

8) 상속관계의 변경

<그림 4>의 상속관계에 대한 변경은 시험이 필요하다. **Person** 클래스로부터 상속받던 **Doctor** 클래스가 클래스 내부의 변화 없이 **Person** 클래스의 자식 클래스인 **Employee** 클래스와 새로운 상속관계를 형성하는 것을 보여준다. 상속관계에 의한 동적 결합 변화로 **Doctor** 클래스의 메소드는 자신을 호출하는 메소드와 통합시험을 하여 정확한 기능의 수행여부를 확인해야 한다.



<그림 4. 상속관계 수정>

각각의 세분화된 8가지 수정 유형에서 언급하는 단위 시험과 통합 시험의 대상에 대해 일차적으로 분석한 결과를 <표 3>에 정리했다. 표에서 사용하는 기호 중 *m*은 변경된 메소드, *caller(m)*은 변경된 메소드 m을 호출하는 메소드, *callee(m)*은 변경된 메소드 m이 호출하는 메소드, *m'*는 메소드 m의 부모 클래스에 있으면서 *m*이 오버로딩한 메소드, *def(v)*는 인스턴스 변수 *v*를 정의한 메소드, *use(v)*는 인스턴스 변수 v를 사용한 메소드를 가리킨다. 단위 시험은 메소드 하나를 시험의 대상으로 수집하지만, 통합시험은 한 쌍의 메소드를 대상으로 한다.

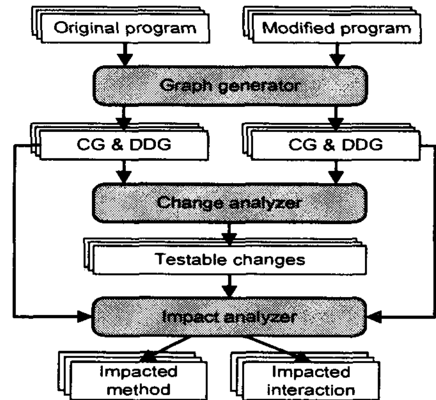
<표 3. 시험을 유발하는 수정과 그 수정영향 분석>

수정 유형		단위시험	통합시험
인스턴스 변수 *v* 정의		*m*	*(m, def(v))* *(m, use(v))*
인스턴스 변수 *v* 사용		*m*	*(def(v), m)*
메소드 *m* 추가	일반	*m*	-
	오버라이딩	*m, caller(m')*	*(m', caller(m'))*
메소드 *m* 삭제	오버라이딩	*caller(m)*	*(m', caller(m))*
호출관계 추가		*m*	*(callee(m), m)*
호출관계 삭제		*m*	-
상속관계의 변경		*caller(all m')*	*(m, caller(m'))*

분석된 결과는 재차 반복적으로 간접적 영향을 받는 메소드에 대해 수정영향 분야를 확장하게 된다. 즉, 단위 시험 대상으로 분석된 메소드의 집합을 구하고, 이들을 호출하는 또 다른 메소드들이 존재하는 경우, 단위시험의 대상에 추가한다. 그리고 호출관계를 갖는 이러한 메소드들은 새로운 통합시험의 시험 대상으로 추가된다. 최후 더 이상 추가할 메소드, 또는 메소드 들간의 관계가 존재하지 않을 때까지 반복하여 수정에 영향 받는 모든 경우를 분석 대상으로 한다.

4. 프로토타입

이 장에서는 본 논문에서 제안한 기법을 자동화한 프로토타입을 제안한다. <그림 5>와 같이 호출 그래프 및 데이터 의존성 그래프 생성기(Graph generator), 수정 분석기(Change analyzer) 그리고 수정영향 분석기(Impact analyzer), 세 부분으로 구성된다. 수정 전과 후의 프로그램의 소스코드를 입력받아 시험을 필요로 하는 수정과 그 수정에 영향 받는 메소드들, 그리고 메소드들의 단위시험 및 통합시험의 여부를 결정한다. 각 부분의 기능은 다음과 같다.



<그림 5. 프로토타입의 구조도>

1) 호출 그래프 및 데이터 의존성 그래프 생성기

수정 전후의 프로그램의 소스 코드를 입력 받아, 두 프로그램의 호출 그래프와 데이터 의존성 그래프를 출력하여 그 결과를 수정 분석기와 영향 분석기에서 입력으로 사용할 수 있도록 한다.

2) 수정 분석기

수정 분석기는 그래프 생성기에서 얻은 두 프로그램의 메소드 호출관계와 데이터의 의존성 관계를 비교하여 유지 보수 단계에서 발생한 수정을 살펴본다. 발생한 수정에 대해서는 시험을 유발하는 수정과 그렇지 않은 수정으로 구분한다.

3) 수정영향 분석기

시험을 유발하는 수정과 생성된 그래프를 이용하여 수정이 발생한 메소드와 수정에 의해 영향을 받는 메소드의 집합을 출력 한다. 집합은 단위시험과 통합시험을 필요로 하는 메소드의 집합으로 재 구분된다. 또한 이 메소드집합의 직, 간접 호출관계에 있는 메소드도 단위시험과 통합시험이 필요한 집합으로 구분하여 시험 대상에 포함시킨다.

본 논문에서 제안한 프로토타입의 정확한 동작여부를 검증하기 위해 예제 프로그램을 수정하고, 분석하였다. 메소드의 추가 및 삭제, 호출관계의 추가 및 삭제에 의한 수정이 적용되었다. <그림 6>은 수정 분석기를 통해 얻은 결과의 일부이며, 그 중 메소드의 추가 및 삭제 에 의해 영향 받는 메소드

와 통합시험이 필요한 메소드 집합을 <그림 7>에 나타난다.

```
<Overriding Method Addition>
void Doctor.printHospitalCharge()

<Non Overriding Method Deletion>
void Hospital.printDoctorByIncome()

<Overriding Method Deletion>
void Patient.printInfo()
```

<그림 6. 메소드 추가 및 삭제에 관한 수정>

```
==================Overriding Method Deletion============
<Unit Test List>
void Hospital.printPatientByTreatday()
void TestHospital.main(String)

<Integration Test List>
void TestHospital.main(String) -> void Hospital.printPatientByTreatday()
void Hospital.printPatientByTreatday() -> void Person.printInfo()
void TestHospital.main(String) -> void Person.printInfo()

==================Overriding Method Addition============
<Unit Test List>
void Hospital.showHospitalCharge()
void TestHospital.main(String)

<Integration Test List>
void TestHospital.main(String) -> void Hospital.showHospitalCharge()
void Hospital.showHospitalCharge() -> void Person.printHospitalCharge()
```

<그림 7. 메소드 추가 및 삭제에 관한 수정영향 분석>

출력된 수정에 대한 결과는 본 논문에서 프로토타입의 정확여부를 판단하기 위해 의도적으로 변화를 주었던 수정과 일치했으며, 수정영향 분석기를 통해 얻은 결과는 Jang[8]이 제안한 방법을 통해 얻을 수 있는 메소드의 집합과 같았다.

5. 결론 및 향후 연구

본 논문에서는 Java 소프트웨어의 회귀 시험을 위해 메소드 단위에서 발생 가능한 수정의 종류를 구분하여 각각의 상관 관계를 분석하였다. 분석된 의존도로부터 구분된 발생 가능한 수정들을 시험을 유발하는 수정 그룹과 그렇지 않은 수정 그룹, 두 가지로 구분하였다. 시험을 유발하는 그룹으로 분석된 8개의 수정 항목들에 한해서 본 논문에서는 해당 수정의 영향을 받는 메소드를 일차적으로 찾아낸다. 또한 메소드들로 인해 간접적으로 영향 받는 메소드들에 대해 재차 수정영향 분석을 완성한다. 수정영향을 받는 것으로 분석된 메소드들에 한해 단위시험이 필요한 메소드들과 통합시험이 필요한 메소드로 분류하여 차후 발생 가능한 시험 관련 작업 시에 필요한 수정영향 정보를 제공한다. 따라서 기존의 기법에서 가능한 모든 수정에 대해 검색하는 방식에 비해 보다 나은 효율성을 보일 수 있을 것으로 예상한다.

제안된 수정영향 분석을 지원하기 위한 프로토타입도 설계하였다. 현재, 수정 분석기 부분은 기본적으로 구현되었고, 수정영향 분석기에서 메소드의 추가 및 삭제, 호출관계의 추가 및 삭제에 대한 수정영향 분석은 자동화 가능하다. 보다 나은 자동화를 지원하기 위해 수정영향 분석이 이루어지지 않는 나머지 수정에 대해서도 점차 구현될 것이다. 구현된 프로토타입을 이용한 실험에서는 본 수정영향 분석 방법이 효율적이고, 기존의 효과를 보여줄 수 있음을 알 수 있었다.

기존의 객체지향 기반의 수정영향 분석 방법 중 메소드 단계의 연구기법을 기반하여 연구방법을 제안하였으나, 다른 클래스 단계와 문장 단계, 두 가지 단계의 기법들의 효율성과 정확성과의 비교가 필요하다. 이에 앞으로 실험을 통해서 메소드 기반 기법의 우수성을 보여주고자 한다. 또한 본 논문에서는 수정에 영향 받은 메소드 정보의 생성까지만을 고려하고, 이를 이용하여 주어진 테스트 케이스들로부터 수정영향 범주에 속하는 테스트 케이스를 추출하는 다음 단계의 연구에 대해서는 언급 정도 수준이다. 회귀 시험의 전체적인 모습을 위해 향후 본 연구 기법의 정보를 이용하는 테스트 케이스의 선택 기법에 대해서도 관심 가지고 있다.

참고문헌

[1] G. Rothermel and M.J. Harrold, “A safe, efficient regression test selection technique”, ACM Trans. on Software Engineering and Methodology, vol.6, no.2, pp.173-210, 1997.

[2] G. Rothermel and M.J. Harrold, "Analyzing Reqression Test Selection Techniques," *IEEE Trans. Software Engineering.*, vol. 22, no. 8, pp. 529-551, 1996.

[3] D.C. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima."Class firewall, test order, and regression testing of object-oriented programs", *Journal of Object-Oriented Programming*, Vol. 8, No. 2, pp. 51-65, May 1995.

[4] M.A. Chaumun, H. Kabaili, R.K. Keller and F. Lustman, “A change Impact Model for Changeability Assessment in Object Oriented Software Systems,” *Proc. Of 2nd Euromicro Conf. Software Maintenance and Reengineering*, 1999.

[5] M.K. Abdi, H. Lounis and H. Sahraoui, "Analyzing Change Impact in Object-Oriented Systems," *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 310-319, 2006.

[6] G. Rothermel, M.J. Harrold, and J. Dedhia. “Regression test selection for C++ software”, *Journal of Software Testing, Verification, and Reliability*, vol.10, no.6, pp.77- 109, 2000.

[7] M.J. Harrold, J. A. Jones , T. Li , D. Liang , A. Orso, M. Pennings, S. Sinha, S.A. Spoon and A. Gujarathi, “Regression test selection for Java software”, *Proc. of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp.312-326, 2001.

[8] Y.K. Jang, M. Munro, and Y.R. Kwon. An improved method of regression tests for C++ programs. *Journal of Software Maintenance and Evolution: Research and Practice*, vol.13, no.5, pp.331 – 350, 2001.

[9] B.G. Ryder and F. Tip, “Change Impact for Object Oriented Programs,” *Proc. ACM SIGPLAN/SIGSOFT Workshop Program Analysis and Software Testing*, pp.46-53, 2001.

[10] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, and J. Dolby, “Chianti: A Prototype Change Impact Analysis Tool for Java,” *Technical Report DCS-TR-533, Rutgers Univ., Dept. of Computer Science*, 2003.

[11] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, “Chianti: A Tool for Change Impact Analysis of Java Programs,” *Proc. ACM SIGPLAN Conf. Object Oriented Programming Languages and Systems*, pp. 432-448, 2004.

[12] X. Ren, O.C. Chesley, and B.G. Ryder, “Identifying failure causes in java programs: An application of change impact analysis”. *IEEE Trans. on Software Engineering*, vol.32, no.9, pp.718-732, 2006.

[13] D.E. Perry, G.E. Kaiser. “Adequate testing and object-oriented programming”. *Journal of Object-Oriented Programming*, vol.2, no.5, pp13–19, 1990.

<부록 1>

Change ID	Change description	Related change
v.1.1	Variable value change	*del(vu), add(vu)*
v.1.2	Variable type change	*del(vd), add(vu)*
v.1.3	Variable addition	*add(vd), add(vu)*
v.1.4	Variable deletion	*del(vd), del(vu)*
v.1.5	Variable scope change	
v.1.5.1 v.1.5.2 v.1.5.3	Public -> Private Public -> Protected Protected -> Private	*del(vd), del(vu)* in local class *add(m), add(call)* in other class
v.1.5.4 v.1.5.5 v.1.5.6	Protected -> Public Private ->Public Private->Protected	*del(m), del(call),* *add(vd), add(vu)*
v.1.6	Variable change (Static/Non-static)	
v.1.6.1	Static ->Non-static	*del(vd), del(vu)* in other method
v.1.6.2	Non-static ->Static	*add(vd), add(vu)* in some methods
m.2.1	Method change (Static/Non-static)	
m.2.1.1	Static ->Non-static	*mod(call)* to use object name
m.2.1.2	Non-static ->Static	*mod(call)* to use class name
m.2.2	Method change (Abstract/Non-abstract)	
m.2.2.1	Abstract ->Non-abstract	*add(m-body)*
m.2.2.2	Non-abstract ->Abstract	c.3.1.1 *del(vd), del(vu),del(call)* *add(m)* in derived class
m.2.3	Method Return type change	*del(vu)* *add(vu)* v1.2
m.2.4	Method implementation change	
m.2.4.1	Variable define	-
m.2.4.2	Variable use change	-
m.2.4.3	Invocation change	-
m.2.5	Method signature change	*add(vd)*
m.2.6	Method Scope change	
m.2.6.1 m.2.6.2 m.2.6.3	Public-> Private Public -> Protected Protected ->Private	*del(call)* in other class
m.2.6.4 m.2.6.5 m.2.6.6	Protected -> Public Private ->Public Private->Protected	*add(call)* in other class
m.2.7	Method addition	
m.2.7.1	Abstract method	*add(call)* *add(m)* in derived class
m.2.7.2	Non-abstract method	*add(call)*
m.2.8	Method deletion	
m.2.8.1	Abstract method	*del(call)*
m.2.8.2	Non-abstract method	*del(call)*
c.3.1	Class change(Abstract/Non-abstract)	
c.3.1.1	Non-abstract->Abstract	m.2.2.2
c.3.1.2	Abstract ->Non-abstract	m.2.2.1
c.3.2	Class addition	*add(v), add(m), add(I)*
c.3.3	Class deletion	*del(v), del(m), del(I)*
c.3.4	Class inheritance	-
c.3.4.1	Inheritance relation addition	-
c.3.4.2	Inheritance relation deletion	-