클래스 노드 분석에 의한 객체 지향 소프트웨어 회귀 테스팅

권 영 희[†] · 이 인 혁[††] · 구 연 설[†††]

요 약

본 논문에서는 메소드를 기본 단위로 하는 객체 지향 소프트웨어의 개선된 선택적 회귀 테스팅 방법을 제안한다. 테스팅 방법은 세 단계로 구성하였다. UML 표기법을 이용하여 변경 전 프로그램과 변경 후 프로그램의 객체 관계 그래프와 클래스 의존 그래프(Class Dependency Graph)를 정의한 후, 의존 그래프의 노드 비교를 통하여 변경된 노드와 변경에 의하여 영향을 받는 노드들을 찾는다. 변경 전 테스트케이스 테이블에서 변경된 노드와 변경에 의해 영향을 받는 노드들을 통과하는 테스트케이스를 선택하여 회귀 테스팅을 위한 테스트케이스 집합을 구성한다. 제안한 테스팅 방법을 사용함으로써 테스트해야할 테스트케이스 수가 줄어들고, 변경 전 프로그램의 테스트케이스를 재사용 함으로써 테스팅의 시간과 비용을 절감할 수 있다.

Object-Oriented Software Regression Testing by Class Node Analysis

Young-Hee Kwon[†] · Len-Ge Li[††] · Yeon-Seol Koo[†††]

ABSTRACT

In this paper, we propose an improved regression testing method, which use method as the basic unit of changing. The testing method consists of three steps. We represent the relationship of classes using the notation of UML(Unified Modeling Language), find the nodes of the modified methods and affected methods by node analysis, and then select changed test cases from the original test cases. The proposed object-oriented regression testing method can reduce the number of test cases, testing time and cost through reuse of test cases.

1. 서 론

객체지향 소프트웨어 테스팅은 클래스에 정의된 데이터 속성과 메소드들을 효과적으로 검증하고, 클래스, 상속, 동적 결합, 다형성 등의 특성을 갖는 객체 지향 소프트웨어의 신뢰도를 향상시키기 위한 것이다[1, 2, 3].

회귀 테스팅은 재검증 테스팅 방법으로, 프로그램에 대한 변경 작업이 원래의 프로그램 기능에 어떠한 영향을 미치는가를 검사하는 것이다[4, 5, 6]. 대부분의 객체지향 회귀 테스팅 방법은 클래스를 변경이 발생하는 기본 단위로 설정하여 변경된 클래스와 이 클래스의 영향을 받는 클래스들에 대해 재테스트를 고려해야 하기 때문에, 작은 부분이 변경되어도 많은 부분에 대한 재테스팅이 요구된다. 이에 비해, 회귀 테스팅의 선택적 재테스트 기법은 기존의 테스트 스위트에 있는 테스트 데이터를 선택적으로 재사용하고, 프로그램의 변경된 부분과 변경에 의하여 영향받는 부분만을 선택하

† 정 회 원 : 대덕대학 경상정보계열 교수
†† 준 회 원 : 충북대학교 대학원 전자계산학과
††† 정 회 원 : 충북대학교 컴퓨터과학과 교수
논문접수 : 1999년 4월 16일, 심사완료 : 1999년 10월 25일

여 재테스트함으로써, 테스트스위트에 있는 모든 테스트 데이터를 수행시키는 기존의 재테스트 기법에 비하여 테스팅 비용을 감소시킬 수 있다.

본 논문에서는 테스트케이스를 재사용하고 변경이 미치는 최소한의 범위를 찾기 위해 메소드를 테스팅의 기본 단위로 정의하고, 객체지향 소프트웨어의 클래스간의 관계를 UML 표기법을 사용하여 그래프로 표현하고, 클래스에 속한 메소드 프로시저를 클래스 의존 그래프로 표현한다. 또한, 변경된 메소드와 변경에 의해 영향받는 메소드를 찾아 변경 전 테스트 케이스테이블에서 선택하는 개선된 회귀 테스팅 방법을 제안한다. 논문에서 제안하는 회귀 테스팅 방법은 변경 전 프로그램의 테스트가 완료되고 객체 지향 프로그램의 노드에 변경이 발생할 경우를 고려한 객체지향 회귀 테스팅 기법이다.

2. 회귀 테스팅 방법에 대한 비교 분석

그 동안 연구된 객체지향 소프트웨어 테스팅 방법으로는 계층구조의 점진적 테스팅 기법(Hierarchical Incremental Testing Technique), 절차적 회귀 테스팅 기법, 클래스 방화벽(Class Fire Wall)기법, 메소드 방화벽(Method Fire Wall)등이 있다[7, 8, 9, 10].

계층구조의 점진적 테스팅 기법은 상속 관계를 갖는 클래스를 테스트하기 위해 상위 클래스의 테스트 정보를 이용하여 하위 클래스가 상위 클래스로부터 속성을 상속받을 때 테스트 히스토리도 함께 상속받아 새로운 속성, 변경된 속성 또는 영향을 받는 속성을 찾아 재테스트한다. 절차적 회귀 테스팅 기법은 프로그램 P와 변경된 프로그램 P'에 대한 제어 흐름 그래프(Control Flow Graph)에 의한 자료 흐름 분석을 통해 변경된 부분을 식별하고, P의 테스팅에 사용했던 테스트 케이스 집합에서 변경된 부분을 검사할 수 있는 새로운 테스트 케이스를 구성한다. 클래스 방화벽 기법은 변경된 클래스로부터 영향을 받는 클래스를 클래스 방화벽으로 정의하고, 영향을 받는 클래스를 재테스트한다. 클래스 방화벽을 찾기 위해 클래스를 노드로 하고 클래스 사이의 간선을 상속, 집합, 연관 관계를 나타내는 객체 관계 그래프(Object Relation Graph)를 사용한다. 메소드 방화벽 기법은 클래스 방화벽 기법에 의하여 클래스 방화벽을 계산하고 방화벽에 포함된 클래스의 관계에 따라 메소드 방화벽을 구한다.

계층 구조의 점진적 테스팅 방법은 상위 클래스의 테스트 정보를 이용하여 하위 클래스에서 변경된 부분만을 선택하여 테스트함으로써 테스트 시간과 노력은 절감할 수는 있으나, 상위 클래스에서 하위 클래스로 상속되는 잘 설계된 상속 계층구조에 적합하다. 절차적 회귀 테스팅 기법은 클래스들간의 상속성, 연관성, 집합성을 고려하지 않았다. 클래스 방화벽 기법은 클래스를 변경이 발생하는 기본 단위로 하므로 적은 부분의 클래스 변경이 발생하더라도 대부분의 클래스들이 재테스트 되어야 하기 때문에 비효과적일 뿐만 아니라 클래스간의 관계가 변경되는 경우는 고려하지 않았다. 메소드 방화벽 기법은 클래스 방화벽 기법에 비해 재테스트하는 부분이 적어지므로 효율적이지만, 메소드 사이의 관계가 수정되는 경우와 데이터 속성이 영향받는 경우는 고려하지 않았다.

3. 개선된 회귀 테스팅 방법

2절에서 제기된 회귀 테스팅에서의 문제점들을 개선할 목적으로 메소드를 테스팅의 기본 단위로 정의하고, 클래스간의 객체 관계 의존도를 UML 표기법을 사용한 그래프로 표현하고, 변경이 발생하는 노드와 변경에 의해 영향받는 노드를 그래프에서 찾아 변경 전 프로그램과 변경 후 프로그램을 비교하여 선택적으로 테스트 케이스를 구성하는 개선된 회귀 테스팅 방법을 제안한다.

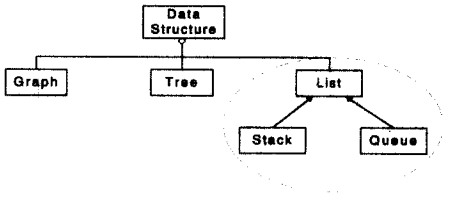
3.1 클래스 관계성 표현

소프트웨어 테스팅에는 제어 흐름 그래프(Control Flow Graph)와 같은 그래픽 표현을 이용하는데, 객체지향 소프트웨어는 전통적인 소프트웨어처럼 프로시저나 프로그램에 대한 표현과 함께 클래스 사이의 집합성, 연관성, 상속성에 대해서도 표현해야 하기 때문에, 이런 전통적인 소프트웨어의 그래픽 표현을 객체 지향 소프트웨어에 그대로 적용할 수가 없다. 따라서, 본 논문에서는 객체지향 개발방법을 위한 통합 모델링 기법인 UML 표기법을 사용하여 클래스 객체 사이의 상속, 집합, 연관 관계를 객체 관계 그래프로 표현하고, 클래스를 구성하고 있는 메소드 사이의 제어와 데이터 의존성을 클래스 의존 그래프로 표현한다[11]. UML 표기법을 그대로 적용하므로 기존의 분석 툴이나 기법의 재사용이 용이하고, 수정에 의해 영향을 받는 부분만

을 찾아낼 수 있어 재테스팅의 시간과 비용을 줄일 수 있다.

3.2 클래스간의 관계성 표현을 위한 그래프 사례

(그림 1)은 Data Structure를 UML 표기법을 이용하여 표현한 관계 그래프로, 여기에서는 클래스 List, Stack, Queue만을 제한적으로 고려한다.



(그림 1) Data Structure에 대한 객체 관계 그래프

```
      const int MAXLIST=10;
      class list
      {
          int *list;
          int numentries;
          int maxentries;
      public
M1        list(int n=MAXLIST)
S2            {list=new int[n];
S3             maxentries=n;
S4             numentries=0;};
M5        ~list()
S6            {delete list;};
M7        int getnumentries()
S8            {return numentries;};
M9        int getmaxentries()
S10           {return maxentries;};
          int serach();
          void print();
          int putitem(int, int);
          int getitem (int&, int);
M11       void setnum(int n)
S12           {numentries=n;};
M13       void incnum();
P14           {if(numentries<maxentries)
S15             ++numentries;};
      }
```

```
M16   int list::putitem(int item, int loc)
      {
P17       if(0<=loc && loc<maxentries)
S18         { list[loc]=item;
S19           return 0; }
          else
S20           return -1;
      }
```

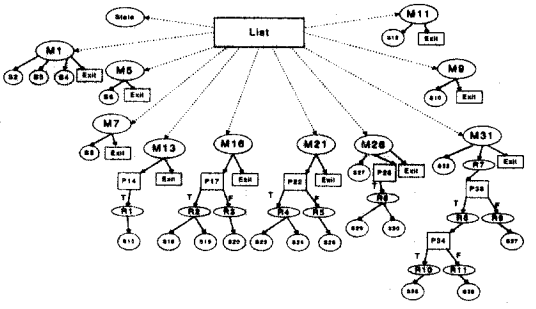
```
M21   int list::getitem(int& item, int loc)
      {
P22       if (0<=loc<maxentries)
S23         { item=list[loc];
S24           return 0;  }
          else
S25         return -1;
      }
```

```
M26   void list::print()
      {
S27       int i=0;
P28       while(i<numentries)
S29         {count<<list[i]<<"\n";
S30          ++i; }
      }
```

```
M31   int list::serach(int& loc, int item)
      {
S32       loc=0;
P33       while(loc<numentries)
P34           { if(list[loc]==item
S35             return 0;
                else
S36             ++loc;  }
S37       return -1;
      }
```

(그림 2) List 프로시저

List는 상위 클래스이며, 여러 데이터 속성과 list(), ~list(), search(), print(), getmaxentries(), getnumentries(), putitem(), getitem(), incnum(), setnum()과 같은 10개의 메소드로 구성된다. (그림 2)와 (그림 3)은 List에 대한 프로시저 클래스 의존 그래프이다. 그래프에서 S_i는 문장을 나타내고, P_i는 조건을 나타낸다. 또한, 원은 치환문(assignment), 입/출력문, 변수 선언문, 호출문과 같은 문장을 표현하는 노드를, 타원은 제어 종속 조건을 간략하게 나타내는 영역 노드(region node)를, 정사각형은 두 개의 간선을 나타내는 조건 노드(predicate node)를, 직사각형은 출구(exit)를 나타낸다.



(그림 3) List에 대한 클래스 의존 그래프

Stack은 List의 하위 클래스로, List의 일부 메소드인 incnum(), getnumentries(), search(), putitem(), getitem(), setnum()를 상속받고, 메소드 print()를 재정의하였으며, 메소드 pop(), push()를 새로 추가하였다. (그림 4)는 Stack에 대한 프로시저이고, Stack과 List에 대한 클래스 의존그래프는 (그림 5)와 같다.

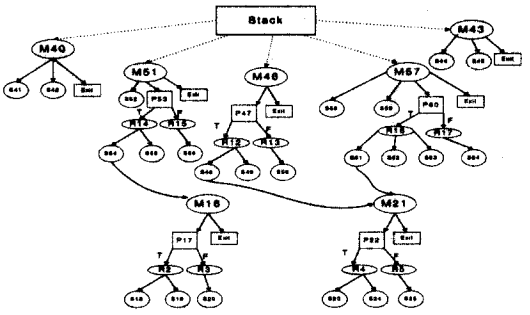
```
      class stack: public list
      {
      int top;
      public
M40       stack(int n);
S41       list[n];
S42       {top=0;};
M43       ~stack();
S44       ~list();
S45       {top=0;};
          int push (int item);
          int pop(int& item);
          void print();
      }
```

```
M46   int stack::pop(int&, item)
      {
P47   { if(top>0)
S48       {getitem(item, --top);
S49         return 0;  }
          else
S50         return -1;
          }
```

```
M57   void stack::print()
      {
S58       int i=top-1;
S59       int item;
P60       while (i>=0)
S61         { getitem(item,i);
S62           count<<item<<"\n";
S63           --i;  }
          else
S64           return -1;
      }
```

```
M51   int stack::push(int item)
      {
S52       int max=getmaxentries();
S53       if(top<max)
S54         { putitem(item, top++);
S55           return 0; }
          else
S56           return -1;
      }
```

(그림 4) Stack 프로시저



(그림 5) Stack과 List에 대한 클래스 의존 그래프

Queue는 List의 하위 클래스로, List의 일부 메소드인 incnum(), getnumentries(), search(), putitem(), getitem(), setnum()를 상속받고, 메소드 print()를 재정의하였으며, 메소드 addq(), deleteq()를 새로 추가하였다. (그림 6)은 Queue에 대한 프로시저이고, Queue와 List에 대한 클래스 의존 그래프는 (그림 7)과 같다.

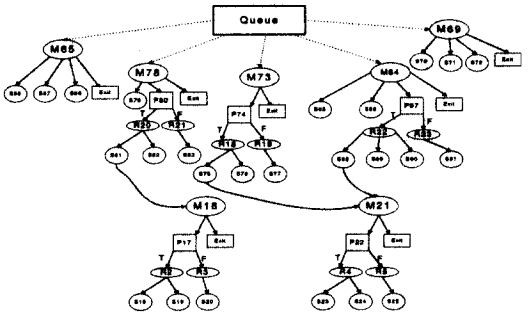
```
        class queue : public list
        {
          int front, rear;
        public
M65       queue(int n);
S66         list[n];
S67         {front=0;
S68          rear=0;}
M69       ~queue();
S70         ~list();
S71          {front=0;
S72           rear=0;};
          int addq (int item);
          int deleteq(int& item);
            void print();
          }

M73   int queue::deleteq(int&, item)
      {
P74     if (front != rear)
S75       {getitem(item, front++);
S76        return 0;}
        else
S77       return -1;
        }
```

```
M84   void queue::print()
      {
S85   int i=rear-front;
S86   int item;
P87   while (i>0)
S88     {getitem(item,i);
S89      count<<item<<"n";
S90      --i; }
        else
S91       return -1;
      }

M78   int queue::addq(int item)
      {
S79     int max=getmaxentries();v
P80     if((rear-front)<max)
S81       { putitem(item, rear++);
S82         return 0;}
        else
S83       return -1;
      }
```

(그림 6) Queue 프로시저



(그림 7) Queue와 List에 대한 클래스 의존 그래프

3.3 회귀 테스팅 알고리즘

변경된 프로그램에서 변경된 노드와 변경에 의하여 영향받는 노드들을 찾아내고 테스트 케이스를 선택하기 위해 변경 전 프로그램의 클래스 C와 변경 후 프로그램의 클래스 C′의 객체 관계 그래프, 클래스 의존 그래프를 구성하는 각 메소드 노드들에 대한 테스트 케이스 테이블 T를 구성한다.

테스팅의 기본 단위를 메소드로 하여 클래스 C와 C′의 클래스 의존 그래프에 의한 실행 기록에 의해 각 메소드의 루트 노드부터 시작하여 각 레벨의 노드를 서로 비교한다. 새로운 노드의 추가, 변경, 삭제 등을 발견하면 변경된 노드 N′을 선택하고 변경 전 단계의 테스트 케이스 테이블에서 변경된 노드 N을 통과하는 테스트 케이스를 선택하여 테스트 집합 T′에 추가한다. 그리고 변경된 노드에 의해 영향받는 노드를 찾는다. 영향받는 노드 N′′이 선택되면 변경 전 테스트 케이스 테이블에서 영향받는 노드 N′′을 통과하는 테스트 케이스를 선택하여 테스트 집합 T′에 추가한다. 영향받는 노드들이 더 이상 없다면 다음 레이블의 후임자를 계속하여 비교한다.

본 논문에서 제안한 알고리즘은 (그림 8)과 같다.

```
algorithm  SelectClassTests(C, C', L, L',T): T'
input    C, C': a class and its modified version
         L, L': label in C and C'
         T : a test set used previously to test C
         PubM, PubM': list of public methods in C, and C'

output  T' : the subset of T selected for use in regression testing C'

begin SelectClassTests
    G=Construct CDG(C, PubM)
    G'=Construct CDG(C', PubM')
    for each node n in G and G' do mark n "not-visited"
    for each new or modified or deleted node n in state do
        mark nodes containing used reached by that node "affected"
    T'=∅
    let GROOT and GROOT' be the root nodes of G and G', respectively
    for each successor node N of G do
        if a successor node N' of G' is not equivalent to N then
            T' = T' ∪ N.history
            if any node N'' is affected by the N' then
                T' = T' ∪ N''.history
        else
            T' = T' ∪ Compare(N, N')
end  SelectClassTests
```

(그림 8) 회귀 테스팅 알고리즘

4. 실험 및 평가

4.1 테스트 케이스 테이블 구성

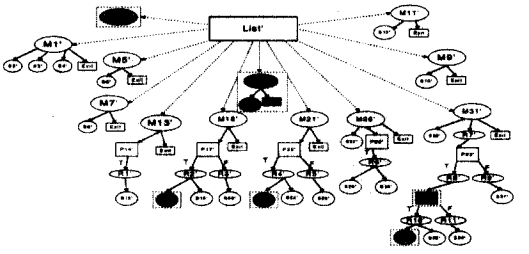
본 절에서는 3.2절에서 그래프 사례로 제시한 Data Structure의 클래스 List에서 변경이·발생하는 경우에 대하여 제안한 회귀 테스팅 방법을 적용하고, 유용성을 검증한다.

〈표 1〉 List, Stack, Queue에 대한 테스트 케이스 테이블

클래스	메소드	테스트케이스	Region Trace
List	M16	T1	M16, P17, R3
		T2	M16, P17, R2
	M21	T3	M21, P22, R5
		T4	M21, P22, R4
	M26	T5	M26, P28, R6
	M31	T6	M31, R7, P33, R9
		T7	M31, R7, P33, R8, P34, R11, R10
		T8	M31, R7, P33, R8, P34, R10, R9
		T9	M31, R7, P33, R8, P34, R10
Stack	M46	T10	M46, P47, R13
		T11	M46, P47, R12
	M51	T12	M51, P53, R14
		T13	M51, P53, R15
	M57	T14	M57, P60, R15
		T15	M57, P60, R16
	M31	T16	M31, R7, P33, R9
		T17	M31, R7, P33, R8, P34, R11, R10
		T18	M31, R7, P33, R8, P34, R10, R9
		T19	M31, R7, P33, R8, P34, R10
Queue	M73	T20	M73, P74, R18
		T21	M73, P74, P19
	M78	T22	M78, P80, R20
		T23	M78, P80, R21
	M84	T24	M84, P87, R22
		T25	M84, P87, R23
	M31	T26	M31, R7, P33, R9
		T27	M31, R7, P33, R8, P34, R11, R10
		T28	M31, R7, P33, R8, P34, R10, R9
		T29	M31, R7, P33, R8, P34, R10

(그림 9)는 변경된 클래스 List의 의존 그래프이다.

(그림 9)에서 표시한 바와 같이 List의 메소드 M16의 S18, M21의 S23, M31의 P34와 S35 노드에서 변경이 발생한다고 가정하면, 클래스 List의 테스트케이스 테이블에서 변경된 노드 S18, S23, P34를 통과하는 테스트케이스{T2, T4, T7, T8, T9}가 선택되어 테스트 집합에 추가된다. 변경에 의하여 영향받는 클래스의 테스트에서 영향받는 노드를 통과하는 테스트케이스를 선택하면, Stack의 S54, S49, S61노드에서 List의 메소드 M16과 M21을 호출하므로 Stack 테스트케이스 테이블에서 노드 S54, S49, S61을 통과하는 테스트케이스{T11, T12, T15} 3개가 선택되어 테스트 집합에 추가된다. 또한, Queue의 노드 S81, S75, S88에서 List의 메소드 M16, M21을 호출하므로 Queue의 테스트케이스 테이블에서 노드 S81, S75, S88을 통과하는 테스트케이스{T20, T22, T24}가 선택되어 테스트 집합에 추가된다.



(그림 9) 변경된 클래스 List의 의존 그래프

4.2 평가

List에서 변경이 발생할 때 클래스 방화벽, 계층구조의 점진적 테스팅 기법, 메소드 방화벽 기법과 본 논문에서 제안한 기법에 의해 선택된 테스트케이스 수를 비교하여 테이블로 표현하면 <표 2>와 같다.

〈표 2〉 테스트케이스 수의 비교 테이블

방 법 / 클래스	계층구조의 점진적 기법	메소드 방화벽	클래스 방화벽	제안한 기법
List	9	9	9	**5**
Stack	10	6	10	**3**
Queue	10	6	10	**3**
Number of Test Case	29	21	29	**11**

본 논문에서 제안한 회귀 테스팅 기법이 테스트 케이스의 양적인 면에서 다른 테스팅 기법에 비해 효율적임을 알 수 있다. 즉, 회귀 테스팅에서 메소드를 테스팅의 기본 단위로 하여 변경된 노드와 변경에 의하여 영향받는 노드를 통과하는 테스트 케이스만을 선택하여 테스트함으로써 테스트 케이스 수를 현저히 줄이고, 변경 전 프로그램의 테스트 케이스를 재사용 함으로써 회귀 테스팅에 소요되는 노력을 절감한다. 또한, 변경된 부분과 변경에 의하여 영향받는 부분을 선택하

여 테스트함으로써, 변경된 프로그램의 새로운 요구의 만족 여부를 검증하여 객체 지향 프로그램의 신뢰도를 향상시킬 수 있다.

5. 결 론

본 논문에서는 객체 지향 프로그램의 신뢰도를 향상시키고 회귀 테스팅의 비용 절감을 위해 객체 지향 소프트웨어를 구성하고 있는 클래스에서 변경이 발생할 때, 메소드를 테스팅의 기본 단위로 정의하고, 변경된 클래스의 노드와 변경에 의하여 영향받는 클래스의 노드들을 찾아내어 변경 전 테스트케이스에서 변경과 변경에 의하여 영향받는 부분만을 테스트하기 위한 개선된 회귀 테스팅 방법에 대해 기술하였다.

회귀 테스팅에서 테스트 케이스를 재사용하고 변경의 영향이 미치는 최소한의 범위를 찾아내는 것은 테스트의 비용과 테스트에 소요되는 시간의 문제와 직결되기 때문에 매우 중요하다. 이를 위해 본 논문에서 제안한 방법은 변경된 노드와 변경에 의하여 영향받는 노드만을 통과하는 테스트 케이스를 선택하여 사용함으로써 테스트해야 할 테스트케이스의 수가 현저히 줄어들고, 변경 전 프로그램의 테스트 케이스를 재사용함으로써 회귀 테스팅 시간과 비용을 절감할 수 있다.

본 논문에서 제안한 방법의 제약 사항으로는 클래스 의존 그래프 표현에 많은 시간이 소요되고, 변경된 부분과 변경에 의하여 영향받는 부분을 찾는데 노력이 필요하다. 따라서 이런 문제점 해결을 위해 변경된 부분과 변경에 의하여 영향받는 부분을 찾기 위한 자동화 도구의 개발에 대한 연구가 요구된다.

참 고 문 헌

[1] Shel Siegel, "Object-Oriented Software Testing : A Hierarchical Approach," John Wiley & Sons, 1996.

[2] Object-Oriented Software Testing, Special Issue, CACM, Vol.37, No.9, 1994.

[3] Paul C. Jorgenson and Carl Erikson, "Object-Oriented Integration Testing," Communications of the ACM, Vol.37, No.9, pp.30-38, September 1994.

[4] T. L. Graves et al., "An Empirical study of Regression Test Selection Techniques," In Proceedings of the 14th International Conference on Software Engineering, April, 1998.

[5] M. J. Harrold, and Gregg Rothernel, "Computation of Interprocedural control dependence," ACM International symposium on software Testing and Analysis, pp.11-21, March 1998.

[6] M.J. Harrold, and Gregg Rothernel, "Select Tests and Identifying Test coverage Requirements for Modified Software," ACM International symposium on software Testing and Analysis, pp.169-184, August 1994.

[7] M. J. Harrold, J. D. McGregorand K. J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structure," In Proceedings of the 14th International Conference on Software Engineering, pp.201-208, October, 1991.

[8] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," ACM Transaction on Software Engineering and Methodology, Vol.6, No.2, pp.173-210, April, 1997.

[9] David C. Kung, J. Gao and P. Hisa, "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs," Journal of Object- Oriented Programming, pp.51-65, May 1995.

[10] Jungwon Bang, "Change Analysis for Regression Test of Object-Oriented Software," KAIST, MS Thesis, 1997.

[11] G. Booch and J. Rumbaugh, Unified Modeling Language for Object-Oriented, Documentation Set Version 1.3 Rational Software Corporation, 1997.

권 영 희

e-mail : yhkwon@mail.ddc.ac.kr

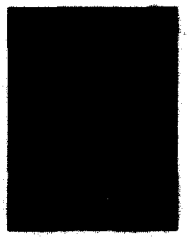
1987년 충남대학교 계산통계학과 졸업(이학사)

1989년 충남대학교대학원 계산통계학과 졸업(이학석사)

1996년 충북대학교대학원 전자계산학과 박사과정수료

1998년~현재 대덕대학 경상정보계열 전임강사

관심분야 : 소프트웨어 재사용, 객체지향 소프트웨어 명세화 기법, 소프트웨어 테스팅 등



이 인 혁

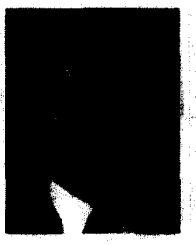
e-mail : leerg@selab.chungbuk.ac.kr

1992년 중국 하얼빈공업대학 자동차설계학과 졸업(학사)

1998년 충북대학교 대학원 전자계산학과 졸업(이학석사)

1999년~현재 충북대학교 대학원 전자계산학과 박사과정

관심분야 : 소프트웨어 테스팅, 소프트웨어 재사용, 정보가시화 시스템 등



구 연 설

e-mail : yskoo@cbucc.chungbuk.ac.kr

1964년 청주대학교 상학과 졸업

1975년 성균관대학교 경영대학원 전자자료처리학과 졸업(경영학석사)

1981년 동국대학교대학원 통계학과 졸업(이학석사)

1988년 광운대학교대학원 전자계산학과 졸업(이학박사)

1979년~현재 충북대학교 컴퓨터과학과 교수

충북대학교 전자계산소장, 한국정보과학회이사, 전산교육연구회위원장, 충청지부장, 부회장 역임

관심분야 : 소프트웨어공학, 정보통신, 알고리즘 등