韓國 OA學會 論文誌
第3卷 第1號, 1998. 3.

論文 98-3-1-1-6

C++에서 프로그램 의존도 그래프를 이용한 클래스 분해 방법

김영선*, 김홍진**, 손용식***

Class Slicing Method using Program Dependency Graph in C++

Young-Sun, Kim*, Hong-Jin, Kim**, Yong-Sik, Son***

요 약

객체 지향 소프트웨어 개발에서 재사용은 소프트웨어 품질 향상과 소프트웨어 개발자의 생산성 향상을 위하여 연구되고 있다. 그러나 설계 단계에서 부적절한 작성과 유지보수시의 잘못된 변화로 인하여 클래스를 재사용 하는데 문제점이 존재한다. 본 논문에서 제안하는 방법은 C++ 프로그램 의존도 그래프를 이용하여 클래스의 의미론적인 결합도를 분석하고, 결합도에 따라 클래스를 분해하여 클래스의 품질 재평가와 확장성을 지원하도록 하는 방법이다.

따라서 본 논문에서 제안하는 방안은 개발자가 객체 지향 프로그램을 개발시에 원하는 부품만을 선택하여 재사용 시스템 구축을 용이하게 하였다.

Abstract

In object-oriented software development, reuse has been studied for the enhancement of software quality and software developer's productivity. But improper modeling in design phase and uncontrolled change during maintenance activities have a problem to reuse a class. In this paper we analyze a semantic cohesion of classes using C++ program dependency graph, and slice a class with cohesion, so that this method supports the re-estimate of class quality and the extensibility of class.

Therefore this proposed method in this paper increases the easibility of the restructuring in the reusable systems, when the developer selects a requested component for a object-oriented program development.

* 광운대학교 전자계산학과 박사과정
** 경원전문대학 전자계산과 교수
*** 성심외국어대 강사
논문접수 : 98.1.5. 심사완료 : 98.2.25.

Ⅰ. 서 론

객체 지향 소프트웨어 개발은 독립적이고 확장성을 가진 소프트웨어 부품들간의 관련성을 통하여 새로운 소프트웨어 개발을 위한 연구가 추진되어 왔으며, 이를 위하여 소프트웨어의 재사용, 버전제어, 형상관리, CASE, 역공학등 다양한 개발 방법이 대두되고 있는데 이러한 개발방법 중에서 재사용은 소프트웨어 품질과 소프트웨어 개발자의 생산성을 동시에 증대시키기 위한 방안으로 연구되고 있다[4,14].

객체 지향 소프트웨어 개발 방법이 우수한 재 사용성을 제공하는 이유 중의 하나는 클래스 계층 구조를 이용하여 매우 강력한 클래스 재 사용을 지원하기 때문이다. 그러나 클래스의 재사용의 문제점은 재사용 가능한 클래스의 기본 단위인 함수들이 결합도가 너무 높아 재사용을 위해 재사용 가능한 독립적인 부품으로 추출해 내는 것이 용이하지 않으며, 클래스의 부적절한 상속과 클래스 계층에서 추상화가 부족하여 실제 소프트웨어 개발자가 기존 클래스의 수정과 합성을 하는데 어려움이 존재한다[5,10]. 이와 같은 문제를 해결하기 위해, 객체 지향 언어의 하나인 C++ 클래스를 재사용 부품으로 선정하여 클래스의 의존도 관계에 따른 최대 분해 방안을 제시하고자 한다.

본 논문에서 다음과 같은 방법을 제시하게 된다. 첫째, 한 클래스가 외부에 제공하는 인터페이스 부분과 클래스의 내부에서 구현을 위해 사용되는 구현부분을 포함하여 클래스의 의존도 관계를 분석한다. 둘째, 프로그램 의존도 그래프를 이용하여 클래스 상속관계를 분석한다. 셋째, 클래스 재사용성을 증대하기 위한 방안을 제시한다.

본 논문에서 제안한 방법은 클래스 인터페이스에서 상호간에 의존도 관계가 있고 생존 주기가 겹치지 않는 메소드와 인스턴스 변수들을 찾아서 분해함으로써 클래스의 품질 재평가와 클래스의 확장이 용이하도록 한다.

Ⅱ. 관련 연구

2.1 클래스의 재사용

클래스는 객체들의 공통되는 특징과 행위들을 모아 이를 추상적으로 표현한 것으로, 시스템 내에서 클래스들은 상속 관계를 형성하는데 클래스 계층에서 클래스들 사이에는 일반화, 세분화 관계가 형성된다[12, 13, 15, 19].

클래스의 재구성 형태는 새로운 추상화의 발견시에 추상 클래스의 추가와 상속 관계를 통하여 변경할 수 있으며, 기능이 비슷한 클래스를 하나의 클래스로 합치거나 두 개

의 클래스에 공통적인 상위 클래스를 두어 그것들을 하위 클래스로 만드는 과정을 클래스의 일반화(generalization)라 하며, 너무 일반화된 클래스는 특정화할 필요성이 있으며 하나의 클래스를 여러 개의 클래스로 분리하는 과정을 클래스의 추상화(specialization)라고 한다. 또한 공통적인 메소드의 구현을 위해 메소드의 공유부분을 상위 클래스로 이동하는 것을 클래스 프로모션(promotion)이라고 한다[8].

클래스가 외부에 제공하는 메소드의 명세와 외부에서 볼 수 있는 데이터들을 인터페이스라 하며, C++나 Eiffel, Smalltalk 등의 많은 객체지향 프로그래밍 언어들은 인터페이스를 별도로 지원하기보다는 클래스 정의 내부에서 슬롯이나 메소드 선언에 키워드 등을 추가하거나, 메소드만을 인터페이스로 허용하는 방식으로 구분하고 있다[16]. 클래스 프로모션은 인터페이스 제한, 인터페이스 확장, 모듈화의 세가지 방식으로 이루어진다.

2.2 프로그램 절단

프로그램 분할은 프로그램에서 계산에 관련된 스레드를 분리하기 위한 기술이며 분할에 대한 이전의 연구는 분할에 포함되어야 하는 실행 가능한 문장 결정에 대한 질의를 정하는 것이다[2,5,10].

마크 와이저(Mark Weiser)에 의해 소개된 프로그램 절단 방법은 프로그램의 통합과 자동화된 병렬화, 오류 검사에 유용한 방법이다. 마크 와이저의 방법에서 프로그램 절단은 프로그램 P에서 변수 x의 값에 영향을 주는 모든 문장을 결합하여 처리하는 것으로 소프트웨어 개발자가 복잡한 코드의 이해, 오류 검사에 도움을 주지만, 다음 두 가지의 문제점이 존재한다[1,3,6,9,10].

- 프로그램 P와 변수 x와 관련된 프로그램 절단은 모든 문장과 술어(predicate)를 고려해야 한다.

- 프로그램 P와 변수 x와 관련된 프로그램 절단으로 인한 감소화된 프로그램은 절단 전의 프로그램과 다른 작용을 할 수 있다.

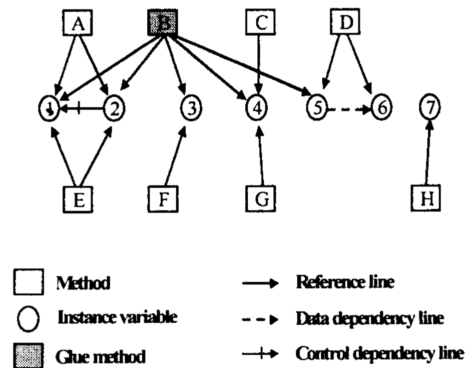


그림 1 클래스 구조 트리
Fig. 1 Class Structure Tree

2.3 클래스 분해

클래스 부품 평가를 위해서는 추상화를 평가하는 모듈 강도와 라이브러리 내에서 클래스들간의 독립성을 평가하는 결합도를 나누어 평가한다. 높은 수준의 자료 추상화를 이룬 클래스는 재사용에서 가장 중요한 장점인 클래스의 확장성을 지원한다 [6,7,12,15].

클래스의 응집력은 메소드와 인스턴스 변수와의 상호 작용 정도에 의해 평가된다. 클래스 내의 응집도를 고려하여 그림 1을 설명한다. 클래스 전체에 대해 다른 메소드와 참조관계가 없는 메소드 H와 인스턴스 변수 7을 제외한 나머지 메소드와 인스턴스 변수를 살펴보면 최대 참조 값을 가진 메소드 B가 존재한다.

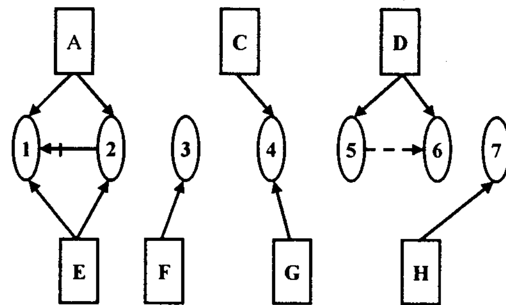


그림 2 분해된 클래스 구조 트리
Fig. 2 Slicing of Class Structure Tree

만약 메소드 B를 제거할 수 있으면 모두

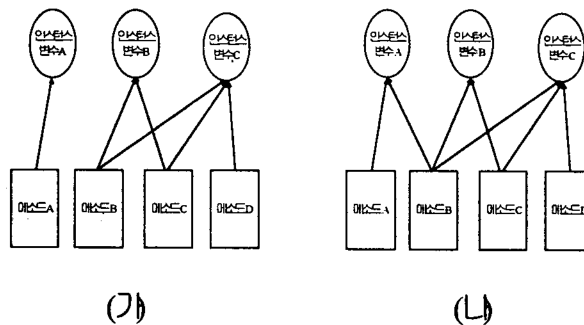


그림 3 클래스의 결합력 비교
Fig. 3 Coupling Comparison between Classes

5개의 구성원으로 나뉘어질 수 있다. 클래스의 참조 그래프를 분해하기 위한 메소드를 해결 메소드라고 하며, 다음 그림 2와 같이 강한 결합력을 가진 메소드들의 집합으로 나뉘게 된다[18].

III. C++ 프로그램 의존도

본 장에서는 객체 지향 시스템에서 클래스 분해를 위한 프로그램 의존도 관계를 서술한다.

3.1 분해 가능한 클래스 집합

클래스, 객체, 상속, 다형성, 동적 바인딩과 같은 객체지향의 특징은 효율적인 객체지향 프로그램의 개발에 영향을 미친다. 위의 2.3절에서 언급한 클래스 중에서 분해 가능한 클래스 집합은 다중 메소드를 유지하고 있는 혼성 클래스와 합성 클래스에서 결합력이 약한 클래스 집합이다.

그림 3에서 클래스 (가)와 클래스 (나)를 비교하면, 클래스 (가)는 클래스 (나)보다 결합력이 약하다. 클래스 (가)는 메소드 A를 가지는 클래스와 메소드 B, C, D를 가지는 클래스로 분해 가능하고, 클래스 (나)는 메소드 A, B, C, D가 각각 서로 연관된 인스턴스 변수를 가지므로 분해가 어렵다.

그러나 클래스 상속관계를 이용하여 분해하면 클래스 (나)는 2.2절에서 언급한 해결 메소드에 의한 분해 방법에 따라 최대 참조선을 가진 해결 메소드 B를 제거하여 분해할 수 있다. 따라서 다중 메소드를 유지하고 있는 클래스는 단일 클래스나 혼합 클래스로 분해가 가능하다. 즉, 혼성 클래스와 합성 클래스는 분해 과정을 통하여 단일 클래스나 혼합 클래스로 분해가 가능하다. 이와 같은 원리를 이용한 클래스의 모델링을 하면 하나의 클래스에 대해서 해결 메소드에 의한 분해를 할 수 있다.

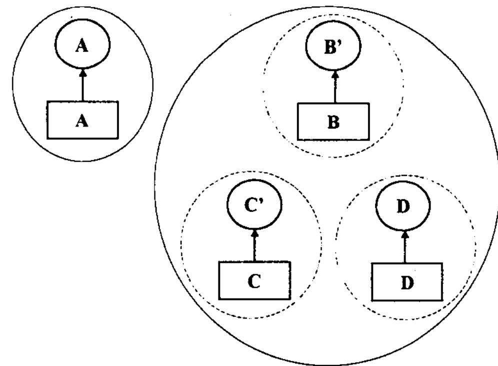


그림 4 클래스의 결합력에 따른 분해
Fig. 4 Slicing by Cohesion of

그림 4를 보면 그림 3의 클래스 (나)에 대해 결합력이 강한 클래스로 분해된 모습을 알 수 있다.

클래스의 품질을 개선하기 위해서 결합력이 약한 클래스를 분해하여 위에서 언급된 단일 클래스나 혼합 클래스를 유지할 수 있다. 본 논문에서 수행되는 과정은 상대적으로 결합력이 약한 클래스를 결합력이 강한 클래스로 분해하는 방법이다.

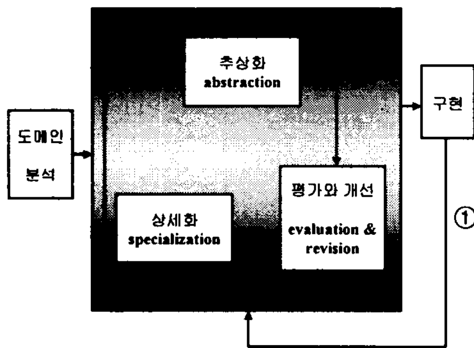


그림 5 객체 지향 소프트웨어 개발
Fig. 5 Object Oriented Software Development

즉, 그림 5와 같이 개발하고자 하는 객체 지향 시스템에 대해서 개발자는 객체 영역을 분석하고 원하는 소프트웨어 부품만을 추출하기 위해 ① 부분을 통하여 음영 부분의 클래스 평가와 개선 과정을 거친 후에 객체 지향 시스템을 구현할 수 있다.

3.2 프로그램 의존도

프로그램의 의존도 그래프(Program dependency graph)는 간선(edge)들에 의해 연결된 정점(vertex)들의 방향성 그래프이다.

C++ 프로그램 상에서 정점은 메소드와 인스턴스 변수들을 의미하며, 두 정점간에 여러 가지 연산에 따른 연결 관계가 있을 경우 두 정점은 간선으로 연결되었다고 하며, 두 정점이 서로 의존도 관계가 있다고 할수 있다. 즉, 여러 개의 정점들이 존재하고 이중에서 두 정점이 선행 관계에 따라 연결된 경우를 보면, 하나의 정점 값이 다른 정점의 연산에 따라 그 값이 변경되는 경우이다. 이러한 의존도 관계는 하나의 같은 정점에 대하여 여러 라인에 걸쳐서 연산들을 수행하는 의존도 관계와 서로 다른 두개의 정점이 조건문이나 연산문에 의해 연결되어 있는 의존도 관계로 나누어질 수 있다.

3.3 C++ 클래스 특성에 따른 프로그램 의존도

클래스의 상속관계에 따른 의존도 관계를 살펴보면, 상위 클래스에서 메소드나 인스턴스 변수를 하위 클래스로 보내는 경우와 하위 클래스에서 상위 클래스의 인스턴스를 상속받는 두 가지 경우로 나눌 수 있다. 전자의 경우를 출력 의존도 관계라 하고, 후자의 경우를 입력 의존도 관계라고 한다.

■ 정의 1 : 클래스 C_A와 클래스 C_B가 is-a

(상속) 관계에 있고, 클래스 C_B가 클래스 C_A에서 상속을 받을 경우에 ·클래스 C_A에서 정의된 속성은 클래스 C_A의 하위 클래스인 클래스 C_B에서도 유지되며 클래스 C_B에서 새롭게 정의된 속성은 클래스 C_B에서 유지되는 것을 입력 의존도(input dependency) 관계가 있다고 한다.

■ 정의 2 : 클래스 C_A와 클래스 C_B가 is-a 관계에 있으며, 클래스 C_B에서 임의의 결과 값을 가지는 경우가 존재하거나 클래스 C_B에 할당 문이 존재하여 결과 값을 가지는 인스턴스가 존재하면 이를 출력 의존도(output dependency)가 있다고 하고, 인스턴스를 v라 하면 $v_1 \rightarrow {}_o v_2$로 표시한다.

하나의 클래스에 대한 의존도 관계는 메소드들간의 관계에 따른 의존도 관계와 메소드와 인스턴스 간에 의존도 관계로 나누어진다. 메소드간의 의존도 관계를 살펴보면 다음과 같다.

■ 정의 3 : 클래스 CA에서 메소드가 재귀적 호출에 의하여 다른 메소드를 호출하는 경우를 의미하며, 하나의 메소드 v1에서 다른 메소드 v2를 호출하는 경우와 하나의 메소드 v1에서 인자로서 다른 메소드 v2를 호출하는 경우에 멤버 함수 참조(member function reference) 관계가 있다고 하며 이를 v1 → rv2로 표시한다. 메소드와 인스턴스 변수간의 의존도 관계를 살펴보면 제어 의존도와 데이터 의존도 관계로 나누어 지며, 데이터 의존도는 다시 흐름 의존도와 정의순서 의존도 관계로 나눌 수 있다.

■ 정의 4 : 제어 의존도는 프로그램(P)의 중첩을 반영할 수 있으며, 만약 인스턴스 변수 v1이 조건문에 존재하고 조건에 따라 인스턴스 변수 v2로 분기가 일어남에 따라 두 인스턴스 변수간에 의존도 관계가 있으면, 두 인스턴스 변수간에 제어의존도(control dependency) 관계가 있으며, v1 → cv2로 표시한다. 인스턴스 변수 v1으로부터 인스턴스 변수 v2로의 데이터 의존도 간선은 프로그램 연산이 두 인스턴스 변수 v_1과 v_2에 의해 표현된 구성 요소의 연관 순서가 반대일 경우 변화될 수 있음을 의미한다. 프로그램 의존도 그래프는 흐름 의존도(flow dependency)와 정의-순서의존도(deforder depend-

ency)로 표현된 데이터 의존도 간선을 포함한다. 프로그램의 데이터 의존도 간선은 데이터 흐름의 분석을 통하여 정해지며, 다음을 만족한다.

■ 정의 5 : 인스턴스 변수 v1이 변수 x로 정의된 정점이고, v2는 x를 사용하는 정점이면, x의 중간 정의 없이 수행 경로를 통해 정점 v1 후에 정점 v2로 도달할 수 있다. 이는 v1의 x 정의에 의해 v2의 x를 사용하는 프로그램에서 기본적인 제어 흐름 그래프 안의 경로이다. 정점 v1부터 정점 v2까지 존재하는 흐름 의존도(flow dependency)는 v1 → fv2로 표시한다. 만약 흐름 의존도가 루프(loop)에 의해 수행된다면 v1 → lv2로 표현되며, 루프 안에서 역간선(backed-ge)를 만족하는 경로가 존재한다.

■ 정의 6 : 흐름 의존도 관계인 메소드나 인스턴스 변수가 각각 독립적인 루프(또는 라인)에서 사용되며, 동시에 상위에 존재하는 v1과 하위에 존재하는 v2의 역 간선이 존재하지 않는다면 정의순서 의존도(def-order dependency)관계가 있으며, 정의순서 의존도 관계는 v1 → dov2로 표현한다. 그리고, 정점들간에 이행관계(transit-ive)가 성립할 수 있으며, v1 → fv3와 v2 → fv3와 같은 인스턴스 변수 v3가 존재할 수 있다.

그림 6은 카운트를 1에서 10까지 증가시키는 예제 프로그램으로 클래스의 인스턴스에 대한 의존도 관계를 표현한 것이다.

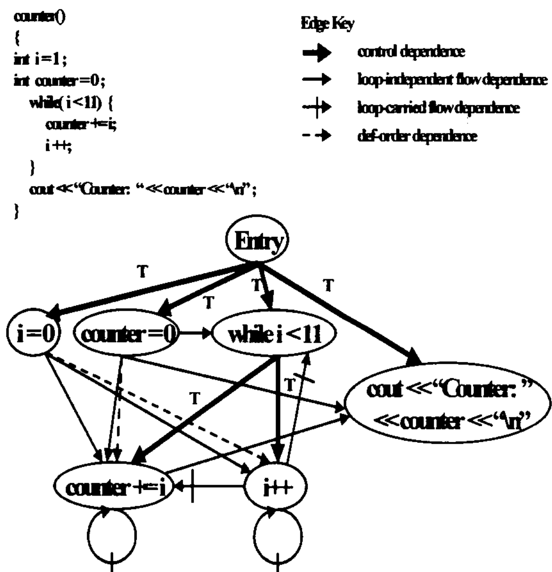


그림 6 프로그램 의존도 그래프의 예
Fig. 6 An example of Program Dependency Graph

IV. 클래스 재평가와 적용

본 장에서는 C++ 프로그램을 재사용하기

위해 프로그램 단위인 클래스를 분해하고 클래스간의 관계를 유지하는 방법을 제안한다.

4.1 클래스 분해

객체 지향 시스템은 하나의 값만을 리턴하는 메소드 호출과 관련된 인스턴스 변수가 존재하며, 이러한 메소드들을 분해하기 위해서 3.4 절에서 정의한 의존도 관계를 이용해 분석가는 클래스를 토큰별로 분석하여 분해 영역으로 나누어야 한다.

알고리즘 1. 의존도 추출 알고리즘

1. 분해할 클래스에 대하여 다음을 수행한다.

```
    class MyClass {
      protected:
e1:       int a;
e2:       void mycall();
e3:       int z;
e4:       MyClass() { z = 0;}
    };

e5: void MyClass::mycall(int a) {
e6:   if (a != 0 &&z == 0)
e7:       z = a;
e8:   z += 1;
    };

     e1 : entry1
```

```
     MyClass::~mycall( ) { }

     class MySubClass : public MyClass {
       private:
e9:        int x ;
e10:       void mysubcall() ;
       public:
e11:       MySubClass( ) { x = 1;}
e12:       void mycall(int a) {
e13:         if(a != 0 &&z == 0)
e14:             z = a + 1;
e15:         z += x;
e16:         return z; }
e17:       void mysubcall( ) {
r18:         return mycall( );
           }
     };
```

그림 7 MyClass와 MySubClass
Fig. 7 MyClass and MySubClass

2. 클래스를 토큰별로 스캔하면서 모든 토큰에 대해 정의 1에서 정의 6의 의존도 관계성이 존재하면 3~6 단계를 반복 처리한다.

3. 토큰이 처음으로 정의되고 정의 1에 의해 $_{conc}C_{parent}$에 존재하거나 현재 클래스에 새롭게 정의되면 입력 의존도 테이블에 정의됨을 표시하여 저장한다.

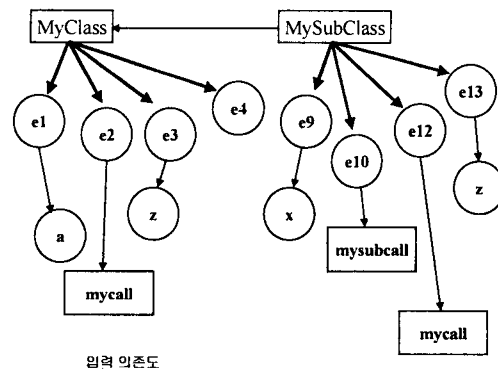


그림 8 클래스의 입력 의존도
Fig. 8 Input Dependency of a Class

그림 7에서 메소드 A, B, C, D와 인스턴스 변수 a, b, c, d는 입력 의존도 테이블에 저장되고 부모 클래스에서 메소드 A와 인스턴스 변수 a가 현재 클래스에서 재정의되었다.

4. 토큰이 정의 2에 의하여 $v_1 \rightarrow {}_{o}v_{token}$인 v_1이 존재하면 v_{token}를 출력 의존도 테이블에 저장한다.

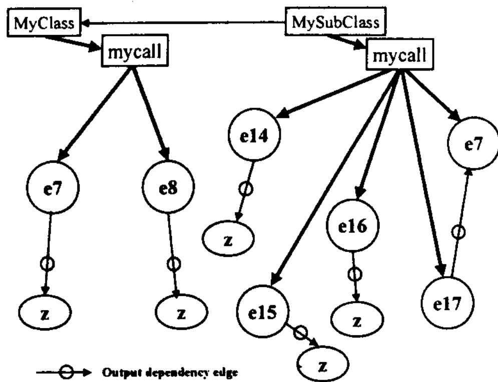


그림 9 클래스의 출력 의존도
Fig. 9 Output Dependency of a Class

5. 토큰이 정의 5에 의하여 $v_1 \rightarrow {}_{f}v_{token}$인 v_1과 v_{token}이 존재하면 각각을 데이터 의존도 테이블에 저장한다.

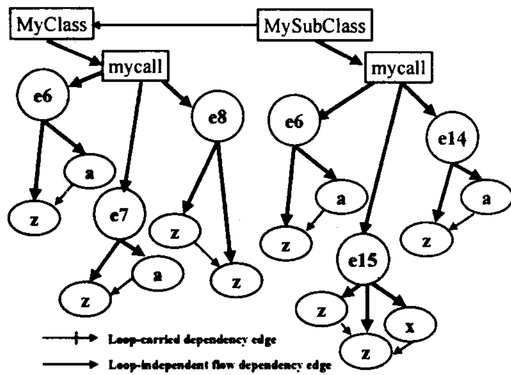


그림 10 클래스의 흐름 의존도
Fig. 10 Flow Dependency of a Class

6. 토큰이 정의 4에 의하여 $v_1 \rightarrow {}_{c}v_{token}$인 v_1과 v_{token}이 존재하면 각각을 제어 의존도 테이블에 저장한다.

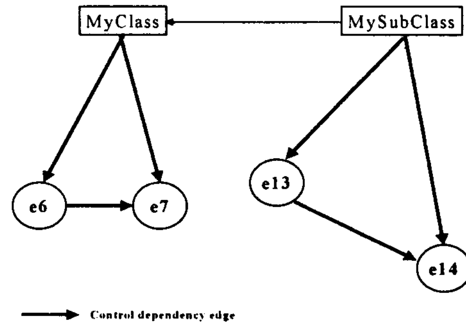


그림 11 클래스의 제어 의존도
Fig. 11 Control Dependency of a Class

7. 토큰이 정의 6에 의하여 $v_1 \rightarrow {}_{do}v_2$인 인스턴스 변수 v_1과 인스턴스 변수 v_2가 존재하면 정의 순서 의존도 테이블에 저장한다.

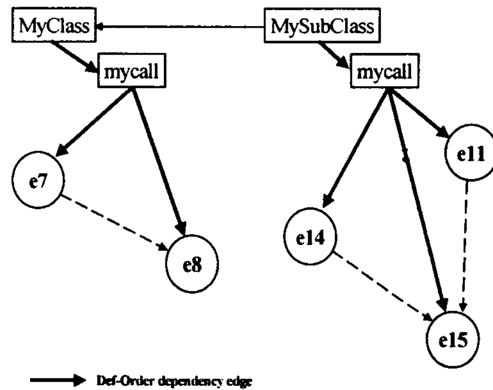


그림 12 클래스의 정의 순서 의존도
Fig. 12 Def-Order Dependency of a Class

8. 토큰이 정의 3에 의하여 $v_1 \rightarrow {}_r v_2$인 v_1과 v_2이 존재하면 멤버 함수 참조 관계 테이블에 저장한다.

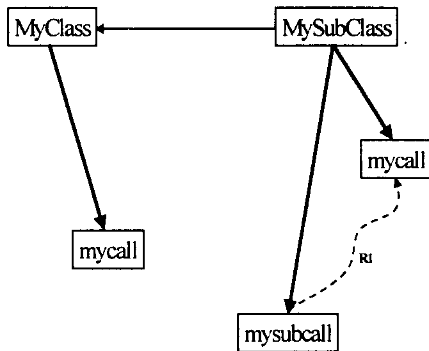


그림 13 클래스의 멤버 함수 참조
Fig. 13 Member Function Reference of a Class

9. 저장된 테이블을 이용하여 참조관계 테이블을 생성한다.

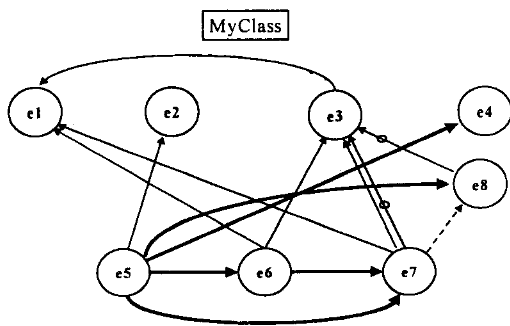


그림 14 MyClass의 인스턴스 참조 관계
Fig. 14 Instance Reference Relationship of a Class

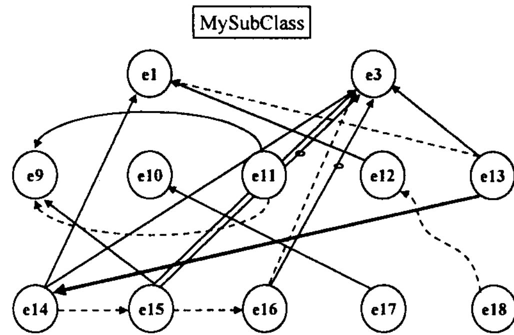


그림 4.9 MySubClass의 인스턴스 참조 관계
Fig. 4.9 Instance Reference Relationship of a Class

클래스 저장소에서 분해하고자 하는 클래스를 선택하여 그림 16에서 보이는 바와 같이 각각의 의존도에 관하여 클래스의 메소드와 인스턴스 변수를 토큰별로 검사하여 해당되는 의존도 테이블에 저장하고 모든 클래스 내의 메소드와 인스턴스 변수의 관계를 정의하여 참조관계 테이블에 저장한다.

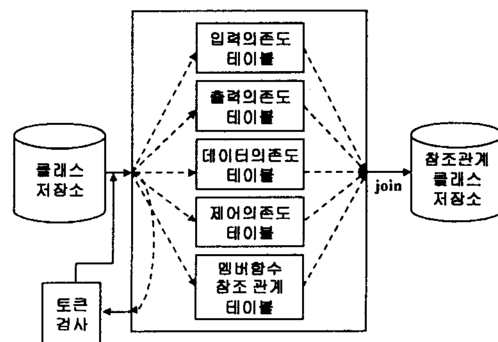


그림 16 분해 알고리즘
Fig. 16 Slicing Algorithm

클래스의 의미론적인 접근을 통하여 새로운 추상 클래스를 생성하며 생성된 클래스의 올바른 상속은 부모 클래스로부터 모든 형태를 상속하는 부 클래스를 강화시킨다. 클래스의 분해 시에 인터페이스를 포함함으로써 상속으로 인한 부적절한 사용을 제한하고 상위 클래스의 모든 행위가 하위 클래스에서 유지되며, 메소드나 인스턴스 변수의 행위 위치 정보를 저장하여 오버라이드나 오버라이딩과 같은 재 정의하다 발견하도록 한다. 또한 이행성(transitive) 관계를 처리하여 x가 y에 대해 데이터/제어 의존하고, y는 z에 대해 데이터/제어 의존한다면 x 역시 z에 대해 데이터/제어 의존함을 보여주는 테이블을 생성할 수 있다.

4.2 의존도 관계에 따른 클래스의 분해

의존도 관계성을 이용하여 분해된 클래스는 정적인 분석을 제공하므로 객체 지향 프로그램에서 다형성에 따른 선택의 폭을 줄일 수 있다. 본 절에서는 4.1절에서 생성된 테이블을 이용하여 분해된 클래스의 관계성 유지를 위한 방안을 제안한다.

알고리즘 2. 클래스 분해 알고리즘

1. 의존도 추출 알고리즘에 의해 생성된 테이블을 이용하여 클래스를 다음 과정에 의해 재구성한다.

2. 추상 메소드와 simple 메소드를 최상위 클래스로 만든다.

3. 최대 멤버 함수 참조 관계가 있는 메소드를 추출하여 위 2에서 만들어진 추상 클래스에 삽입한다.

4. 나머지 클래스에 대해 멤버 함수 참조 값이 1일 때까지 반복적으로 추출하여 클래스를 생성한다.

5. 메소드에 대해 더 이상 분해되지 않을 경우 나머지 클래스를 가지고 추상 클래스를 생성한다.

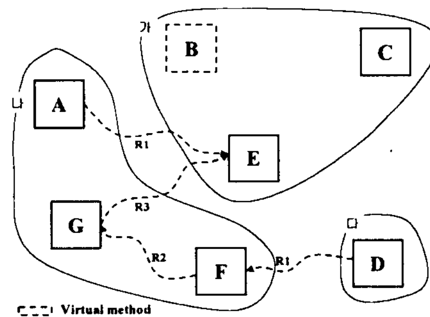


그림 17 멤버 함수 참조 값에 의한 분해
Fig. 17 Slicing by Member Function Reference Value

최상위 클래스에 추상 메소드와 simple 메소드만을 유지하고, 그림 13에서 나온 참조 관계를 이용하여 클래스를 분해한다. 추상 메소드 B와 simple 메소드, 최대 참조되는

메소드 E를 최상위 클래스로 만들고(가 부분), 다시 메소드 A의 참조 관계가 0이 되고 최대 참조되는 메소드가 G 메소드가 되므로 나 부분이 되며 다시 나머지 다 부분으로 나뉠 수 있다.

6. **재구성된 추상 클래스의 메소드에 대해 인스턴스 변수를 기준으로 다시 다음의 과**정에 의해 재구성한다.

6단계에서 8단계는 위의 5단계까지 분해**된 클래스에 대해 메소드와 인스턴스 변수**에 대해 분해하는 과정으로 제어 의존도와 흐름 의존도가 있는 메소드들을 합치면 그림 18에서 가, 나와 다 부분의 두 부분으로 나뉠 수 있다.

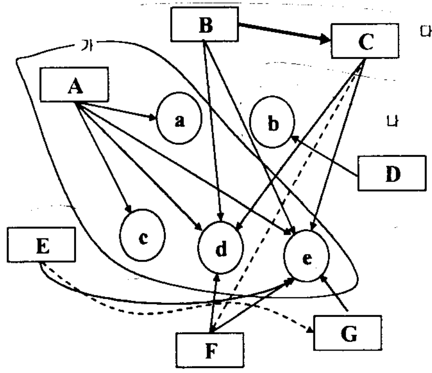


그림 18 클래스의 인스턴스 분해
Fig. 18 Instance Slicing of a Class

7. 재구성된 추상 클래스중에 최대 인스턴스 변수 참조 값을 가진 메소드를 기준으로 분해한다.

8. 분해된 클래스를 상위 클래스로 만들고 나머지 메소드들에 대해 single 메소드가 될 때까지 분해하여 클래스를 생성한다.

위의 그림 18에서 가와 나 부분으로 다시 나뉘어 져서 최종적으로 위 그림 18과 같이 가, 나, 다 부분으로 나뉠 수 있다. 의존도 관계 테이블을 이용하여 클래스의 메소드와 인스턴스 변수의 결합도를 측정할 수 있으며, 메소드간의 결합도를 이용 클래스를 분해하며 메소드에 대해서 인스턴스와의 의존도를 측정하여 다시 분해한다. 결합도는 3,4절에서 언급한 정의를 이용하여 분석할 수 있다.

최상위 계층은 메소드로부터 유도된 가상 클래스를 포함한다. 가상 클래스는 하위 클래스에서 사용됨을 선언하는 것으로 본 절에서 제공하는 알고리즘을 적용하기 위해서는 새로운 추상화 클래스를 만들고 나서 생성된 클래스들을 재가공(예 : protected 선언)하는 작업이 필요하다. 이는 클래스를 생성하고 이름을 새롭게 이름을 붙이거나 클래스를 상속에 의해 사용 가능하도록 타입 선언을 변경하는 작업등이 필요하다.

또한 상속 관계인 is-a관계만이 아니라 클래스가 part-of 관계인 경우에는 포인터

개념을 도입하여 클래스간의 행위를 원활하도록 유지하는 조작이 필요하다.

4.3 클래스 분해의 예

본 논문에서 적용하는 예제는 [11]에 나오는 프로그램을 이용하여 적용 해본 것이다. 클래스 구성도를 살펴보면 그림 19와 같다.

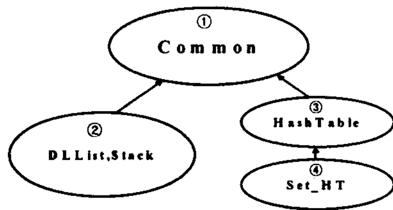


그림 19 분해전 클래스 구조
Fig. 19 Class Structure before Slicing

의존도 추출 알고리즘을 적용해보면 다음과 같다.

① 분해할 클래스에 대하여 다음 과정을 수행한다.

위 그림에서 ②부분을 임의로 결합한 후에 이에 대한 처리를 하여본다.

② 클래스를 토큰별로 스캔하면서 모든 토큰에 대해 3~6을 반복 처리한다. 클래스 List_Stack에 대해 차례대로 스캔하면서 모든 클래스내의 메소드와 인터페이스를 처리한다.

③ 토큰이 처음으로 정의되었고 정의 5에 의해 ${}_{conc}C_{parent}$에 존재하면 입력 의존도 테이블에 재 정의됨을 표시하여 저장한다.

표 1 입력 의존도 테이블
Table. 1 Input Dependency Table

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
list_stack	list_stack	insert()		is-a
list_stack	list_stack	prepend		is-a
list_stack	list_stack	remove()		is-a

④ 토큰이 정의 6에 의하여 $v_1 \rightarrow {}_{o}v_{token}$인 v_1이 존재하면 v_{token}를 출력 의존도 테이블에 저장한다.

표 2 출력 의존도 테이블
Table. 2 Ouput Dependency Table

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
list_stack	list_stack	isEmpty()	head->next	is-a
list_stack	list_stack	first()	head->next	is-a
list_stack	list_stack	end()	head	is-a
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

⑤ 토큰이 정의 2에 의하여 $v_1 \rightarrow {}_{f}v_{token}$인 v_1과 v_{token}이 존재하면 각각을 데이터 의존도 테이블에 저장한다.

표 3 흐름 의존도 테이블
Table. 3 Flow Dependency Table

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
list_stack	insert()	before->prev	newNode->prev	alignment
list_stack	isEmpty()	head	head->next	alignment
list_stack	insert()	item	newNode->next	alignment

⑥ 토큰이 정의 1,3에 의하여 $v_1 \rightarrow {}_{c}v_{token}$인 v_1과 v_{token}이 존재하면 각각을 제어 의존도 테이블에 저장한다.

표 4 제어 의존도 테이블
Table. 4 Control Dependency Table

class_name	method	internal_node	target_node	[illegible]
[illegible]	forEach	node	node	loop
list_stack	forEach	node	node->next	loop
list_stack	firstThat	node	node	loop
list_stack	firstThat	node->item	node	condition

⑦ 토큰이 정의 4에 의하여 $v_1 \rightarrow {}_{do(v3)}v_2$인 v_1과 v_2이 존재하면 정의 순서 의존도 테이블에 저장한다.

표 5 정의 순서 의존도 테이블
Table. 5 Def-Order Dependency Table

class_name	method	internal_line	target_line	appro_method
list_stack	insert()	PRECONDITION(node != 0)	return node->item	sequence

⑧ 토큰이 정의 7에 의하여 $v_1 \rightarrow {}_{r}v_2$인 v_1과 v_2이 존재하면 멤버 함수 참조 관계 테이블에 저장한다.

표 6 멤버 함수 참조 테이블
Table. 6 Member Function Reference Table

class_name	method	[illegible]	[illegible]	[illegible]	reference_num
list_stack	insert()			☑	2
list_stack	append()	insert()	~end()	☐	1
list_stack	prepend()	insert()	~end()	☐	1
list_stack	remove()	[illegible]	[illegible]	☑	[illegible]

⑨ 저장된 테이블을 이용하여 참조관계 테이블을 생성한다.

표 7 데이터 참조 관계 테이블
Table. 7 Data Reference Relation Table

class_name	method	instance_variable
[illegible]	insert()	item,before
list_stack	remove()	next,prev

4.4 결과 분석

적용한 예제 프로그램은 리스트와 스택을 합성하여 분해 과정을 보여주는 예이다. 그러나 적용 예제 프로그램이 메소드간의 결합에 의해서만 분해가 가능하였고, 인스턴스 변수는 구조체 선언으로 인하여 공통부분만을 처리하는 바 결합력이 강하여 분해가 안 되었다. 의존도 관계에 따른 결과는 다음과 같다.

그러나, 그림 10에서 보면 정규 경로 수에 비해 고립 노드의 증가로 인하여 실제 메모리 상에 적용될 범위의 축소를 가져올 수 있으며 이로 인한 메모리 힙 사이즈를 줄여 줄 수 있고, 원하는 부품만을 추출할 수 있다. 따라서 분석된 추상클래스를 적절히 재구성을 하면 클래스의 확장성을 지원할 수 있다.

V. 결 론

본 논문에서는 클래스의 재사용을 위한 클래스의 분해 방안을 제시하였다. 이 방법

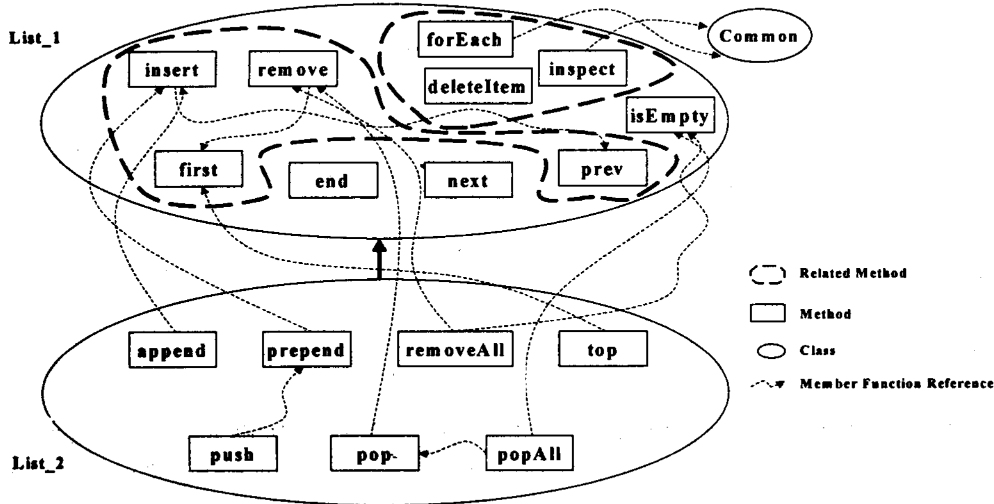


그림 20 의존도 결과 그래프
Fig. 20 Dependency Result Graph

은 기존의 단일 프로그램 상에서 프로그램 의존도 그래프를 이용하여 접근하였던 프로그램 분해 방법을 확장하여 C++ 프로그램의 의존도 관계를 분석하고 상속 관계에 따른 다형성 문제 해결 방안으로 입력 의존도와 출력 의존도를 결합하여 새로운 분해 방안을 제시하였다. 즉, 프로그램 상에서 결합 가능한 요소들 간의 모듈화는 데이터 의존도와 제어 의존도에 따라 결합할 수 있으며 이를 통하여 각 정보를 테이블에 저장하고 분해된 클래스간의 관계성을 추출하기 위한 정보를 유지시켜 새로운 시스템에 적용할 때에 원하는 부품만을 선택하여 사용할 수 있도록 하였다.

따라서 본 논문에서 제시하는 방법은 객체 지향 소프트웨어 개발 시에 원하는 부품만을 추출하여 사용할 수 있도록 하여 힙 사이즈 절감과 프로그램 확장성을 제공한다.

참 고 문 헌

[1] Ferrante J., Ottenstein K., and Warren J., "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems, pp. 319-349, 1987.

[2] Horwitz S., Reps T., and Binkley D., "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and System, Vol. 12, No. 1, pp. 26-60, 1990.

[3] Keith B. Gallagher and James R. Lyle, "Using Program Slicing in Software maintenance," IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 751-761, 1991.

[4] K. W. Jameson, "A Model for the Reuse of Software Design Information", Proceedings of the 11th ICSE, pp. 205-216, 1989.

[5] Kuck D. J., Kuhn R. H., Leasure B., Padua D. A. and Wolfe M., "Dependence graphs and compiler optimizations," The conference record of the 18th ACM Symposium on Principles of Programming Languages, pp. 207-218, 1981.

[6] Loren L. and Mary J. Harrold, "Slicing Object-Oriented Software," Proceeding of the 18th International Conference on Software Engineering, pp. 495-505, 1996.

[7] Reps T., Horwitz S. and Rosay G., "Speeding up Slicing," In Proceeding of Second ACM Conference on Foundations of Software Engineering, pp. 11-20, 1994.

[8] Roger S. Pressman, "Software Engineering: A practitioner's approach 4th edition," pp. 599-600, 1996.

[9] Tomas Reps, "Program Analysis via Graph Reachability," Int. Logic Prog. Symp. University of Wisconsin, 1997.

[10] Weiser M., "Program slicing," IEEE Transaction on Software Engineering, SE-10(4), pp. 352-357, 1984.

[11] URL-http://hem2.passagen.se/mrasm/c.htm

[12] 김갑수, 신영길, "소프트웨어 재사용을 위한 C++ 클래스 계층 구조 변형 방법," 한국 정보과학회 논문지, 제 22권 제 1호, pp. 88-93, 1995.

[13] 김치수, 이경환, "ADT 재사용을 위한 지식구조와 관계정보의 검색," 한국정보과학회 논문지, 제 19권 제 3호, pp. 237-245, 1992.

[14] 이기환, 안장원, 조동섭, 황희융, 전주식, "주어진 클래스를 최적 개수의 노드로 분리하는 2단계 임계 논리망," 한국정보과학회논문지, 제 21권 제 5호, 1994.

[15] 전일수, 이상조, "객체 지향 데이터 모델에서 일반화/세분화 및 집단화 관계를 이용한 집중화 기법," 한국정보과학회 논문지, 제 21권 10호, 1994.

[16] 조은선, 한상영, 김형주, "클래스의 인터페이스와 구현부의 분리를 지원하기 위한 객체 지향 프로그래밍 언어의 확장," 한국정보과학회 논문지, 제 24권 제 2호, 1997.

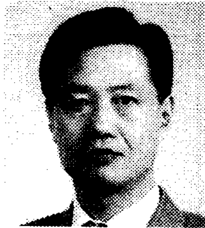
[17] 최기호, 이정옥, 이상정, 임인철, "Matrix를 이용한 마이크로코드의 국소적 최적화 알고리즘," 한국정보과학회 논문지, 제 13권 제 2호, 1986.

[18] 채홍석, "객체지향 시스템에서의 클래스와 계승관계의 재구성," KAIST 대학원 석사논문, 1996.

[19] 허계범, 최영근, "객체지향 소프트웨어 공학," 도서출판 한국 실리콘, 1995.

□ 筆者紹介

김영선
1985년 광운대학교 전자계산기공학과(공학사)
1997년 광운대학교 전자계산학과(이학석사)
1997년~ 현재 광운대학교 전자계산학과 박사과정
1993년 ~ 현재 서울여대 전자계산소 주임



김홍진
1983년 광운대학교 전자계산학과(이학사)
1985년 광운대학교 전자계산학과(이학석사)
1992년 광운대학교 전자계산학과(이학박사)
1984년 ~ 현재 경원전문대학 전자계산과 교수

손용식
1996년 고려대학교 전산과(이학사)
1998년 광운대학교 전자계산학과(이학석사)
1998년 ~ 현재 성심외국어대 강사