객체 지향 클래스 계층 구조 재구성 방법

정 계 동† · 최 영 근††

요 약

객체 지향 시스템에서 클래스 추가 및 삭제로 인하여 클래스간의 새로운 관계를 유지할 수 있는 클래스 계층 구조의 변경이 필요하다. 그러나 기존의 방법에서는 클래스 계층 구조 변경시 부모 클래스와 자식 클래스 사이의 의미를 파악하기 어려워 많은 추가적인 분석 비용이 소요된다.

본 논문에서는 클래스간의 유사성을 측정하여 새로운 관계성 분류 방법을 통해 외부적 변화에 따른 수정 방법을 제시한다. 즉, 이 방법은 클래스들의 유사성을 측정하여 관계성을 기준 하여 무관 관계, 부분 관계, 동일 관계, 포함 관계, 부분 집합 관계로 구분하여 클래스 계층 구조를 재구성한다. 본 논문에서 제시하는 방법은 클래스 계층 구조 변경시 클래스간의 의미 오류 가능성을 최소화 할 수 있도록 한다. 또한 다양한 그래픽 및 텍스트 처리를 통하여 사용자에게 재사용의 편리성 및 이해성을 높일 수 있도록 하였다.

Restructuring Method for Object-Oriented Class Hierarchy

Kye Dong Jung† · Young Keun Choi††

ABSTRACT

When the class is added or deleted in object-oriented system, restructuring of class hierarchy is needed which enables new relationship with classes. But existing system requires much additional analysis costs because it is difficult to know the meaning between parent class and child class.

This paper presents the updated method based on semantic modification through new relationship classification method. This method measures the similarity of classes and based on it's relationship, this method restructures class hierarchy by classifying not-equality, part-of, equality, inclusion, subset relation. This method can minimize the probability of meaning error for classes when the class hierarchy is changed. Also this enhances the reusability and understandability through various graphic and text processing.

1. 서 론

소프트웨어 재사용 방법은 기존의 소프트웨어를 사용하여 새로운 소프트웨어를 작성하는 기법으로 소프트웨어 생산성을 높이기 위한 방법이라 할 수 있다 [9,11]. 그러나 기존의 구조적 방법에서는 개발된 소프트웨어 부품을 재사용하고자 하는 [illegible] 변경과가 없었다. 그 [illegible] 프로세스 중심(process oriented)의 방법으로 분석 모델과 [illegible] 의미가 불일치하기 때문이다. 그러나 [illegible] 방법은 분석, 설계, 구현 [illegible] 결과가 재사용될 수 있는 단위로 만들어지므로 재사용을 더욱 용이하게 한다 [2,8,18].

객체 지향 시스템의 소프트웨어 생명주기에서 사용자의 요구사항을 만족시키기 위하여 클래스간의 새로운 관계로 계층 구조의 변경이 요구된다. 즉, 개발자들은

※ 본 연구는 1997년도 광운대학교 연구비 지원에 의해 수행되었음.
† 정 회 원 : 광운대학교 전자계산학과
†† 정 회 원 : 광운대학교 전자계산학과 교수
논문접수 : 1997년 8월 8일, 심사완료 : 1998년 3월 14일

자신이 원하는 기능을 얻기 위하여 새로운 클래스를 작성하기보다는 현존하는 클래스 또는 클래스간의 관계를 변경하는 것으로 보통 변수의 재정의, 멤버 함수의 재구현, 상속 관계 재배열, 인터페이스 변경, 새로운 클래스 추가 및 삭제 등의 작업에 의해서 이루어진다[5,17]. 그러나 기존의 방법에서는 클래스 계층 구조 변경시 부모 클래스와 자식 클래스 사이의 의미를 파악하기 어렵다. 따라서 부적절한 클래스 계층 구조를 이루게 되어 재사용에 문제점을 야기시킨다. 또한 많은 추가적이 분석 비용이 소요된다.

따라서 본 논문에서는 코드 재사용과 같은 구현 인수를 고려하는 재조직으로 클래스간의 유사성을 측정하여 새로운 관계성 분류 방법을 통한 수정 방법을 제시한다. 이 방법은 클래스 유사성 측정 정보를 분석하며, 그 관계성을 기준으로 무관 관계, 부분 관계, 동일 관계, 포함 관계, 부분 집합 관계로 구분하며, 또한 클래스 삭제시 루트(root) 클래스, 중간(middle) 클래스, 리프(leaf) 클래스를 구분함으로써 일관성을 유지할 수 있다. 본 논문에서는 한 모듈의 프로그램을 분석하여 클래스 상속 관계 테이블, 멤버 함수 참조 테이블, 클래스 메트릭스 정의 테이블을 생성하여 인터페이스를 집합화한다. 이러한 테이블은 다음과 같은 장점을 가진다. 프로토콜 일치를 구체화하고 주관적인 매개 변수의 조율(tuning)에 의존하지 않는다. 본 논문에서는 소프트웨어 재사용을 용이하게 하기 위하여 설계 단계에서는 모듈 수준, 클래스 수준, 멤버 함수 수준, 데이터 멤버 수준으로 구분한다. 따라서 각 수준은 상위 수준에서 하위 수준으로의 연계를 가지고 동작한다.

첫째, 모듈 수준에서는 클래스 관계성 형태와 클래스 상속의 깊이를 고려한다.

둘째, 클래스 수준에서는 객체 지향 프로그램의 캡슐화의 정도 및 클래스 개방 형태를 이용하여 데이터 멤버와 멤버 함수의 내용을 저장한다.

셋째, 멤버 함수 수준에서는 함수의 구현 단계로 각 함수별로 내용을 저장하여 관리하고 추출한다.

따라서 본 논문에서 제시하는 방법은 모듈 단위로 프로그램 분석을 기반으로 하여 객체 지향 언어의 특성에 맞는 정보를 추출하고 클래스 계층 구조 변경시 클래스간의 의미 오류 가능성을 최소화 할 수 있도록 한다. 객체지향언어의 가장 큰 특징인 상속성과 다형성은 재사용을 목적으로 하고 있으며 상속에 의한 체계적인 유지가 용이하다는 측면이 있으나, 프로그램 측면으로 보면 다형성문제, 재정의 문제 등이 이해하기가 어렵다. 이러한 정보를 정보 저장소에 저장하고 추출하여 다양한 그래픽 및 텍스트 처리를 통하여 사용자에게 재사용의 편리성 및 이해성을 높일 수 있다.

본 논문의 구성은 다음과 같다. 2장에는 관련 연구를 살펴보고, 3장에서는 객체 지향 시스템에서의 클래스 계층 구조 재구성을 위한 클래스 관계성 분류 방법과 클래스 추가 및 삭제 방법에 따른 알고리즘을 제시하고, 4장에서는 실 시스템의 적용예를 보이며, 끝으로 5장에서는 결론과 향후 연구 방향을 제시한다.

2. 관련 연구

기존의 클래스 계층 구조에서 재구성을 위한 방법, 클래스간의 유사성 측정 방법, 문서 클러스터링 방법, 그리고 클래스 탐색 방법에 관하여 살펴보고자 한다.

클래스 계층 구조는 상속 관계로 구성된 집합사이의 상세화와 일반화 관계로 정의 할 수 있는데 클래스 재사용과 효율적인 시스템 개발을 위해서는 클래스 계층 구조의 변경이 불가피하다[3,4]. 일반적으로 클래스 계층 구조 재구성 형태로는 다음과 같다.

첫째, 새로운 추상화가 발견될 때 클래스의 관계성이 변경될 수 있다.

둘째, 기능이 비슷한 클래스를 하나로 합치어 하나로 만드는 것이다.

셋째, 너무 일반화된 클래스는 세분화가 필요하고 세분화된 클래스를 일반화로 만들면서 하나의 클래스가 여러 개의 클래스로 분리 될 수 있다.

클래스 내부의 변경 형태는 새로운 인스턴스 변수의 추가 및 삭제, 인스턴스 변수의 이름, 타입 변경, 멤버 함수의 추가, 삭제로 인하여 클래스 행위가 변경이 되고 클래스 계층 구조에도 영향을 미친다.[8,19] 그러나 기존의 방법에서 클래스 삽입은 두 클래스 사이에 새로운 클래스를 삽입함으로써 클래스들이 삽입되기 전의 형태를 가지기 어려우며 변경된 계층 구조로 인해 부모 클래스 사이의 관계성의 의미를 파악하여 유지하기가 어렵다[6,7,15,17]. 즉, 부모 클래스와 자식 클래스 사이의 의미를 파악하기 어렵다. 따라서 부적절한 클래스 계층 구조를 이루게 되어 재사용에 문제점을 야기시킨다. 또한 많은 추가적이 분석 비용이 소요되므로 재사용 및 유지 보수성을 할 수 있는 효율적인 방법이 필요하다. 다음은 재사용을 위한 클래스 유사성 측정 방법

으로, 대표적인 유사성 측정 방법은 inner-product measure, cosine measure, pseudo-cosine measure, Dice measure 방법[12]이 있다. 이러한 방법은 두 문서(i, j)간의 가중치들을 결정하는데 두 문서들의 용어들에 의하여 결정한다. 이 가중치들은 정규화 하는 식을 이용하여 두문서의 유사성을 결정한다. 문서의 유사성을 결정하는 Dice 측정방법에서는 다음 식으로 결정한다.

$$S_{ij} = \frac{2\times C}{A+B}$$

여기에서 C는 i 문서와 j 문서에 공통적으로 존재하는 의미 있는 단어의 수이고, A는 i문서에 존재하는 의미 있는 단어들의 수이고, B는 j문서에 존재하는 의미 있는 단어들의 수이다. 본 논문에서는 C++ 언어의 재사용 가능한 클래스들의 데이터 멤버와 멤버 함수 값들을 참조하여 클래스 삽입 및 삭제로 인한 재구성시 이들의 유사성을 고려한다.

이것을 근거로 클러스터링하는 방법은 Single pass, reallocation, complete link, group average link, Ward방법 등이 있다[10,12]. 그러나 이러한 방법은 객체 지향 언어에서 지원하는 상속 구조를 표현할 수 없다. 객체 지향 언어에서의 유사성은 클래스간의 데이터 멤버와 멤버 함수의 유사성 정도를 나타내며, 이렇게 유사성 정도가 큰 것을 대상으로 클러스터링하게 된다. 클러스터링 하는 이유는 유사한 기능을 수행하는 클래스들을 같은 클래스로 묶어 동일한 장소에 두어 클래스 정보 재사용성이 높으며 인접 기억 장소에 두므로 검색 효율을 높일 수 있다[12,13]. 본 논문에서는 클래스 클래스의 유사성 고려및 삽입과 삭제시, 부모 클래스로부터 자식 클래스간의 상속 문제를 고려한다. 즉, 일반화 및 세분화 개념을 적용하여 클래스를 일관성 있게 확장 할 수 있도록 한다. 일반적으로 재사용을 위한 클래스 탐색 기법은 다음과 같다[10].

첫째, 클래스 이름을 데이터로 탐색하는 경우로 사용자가 클래스 이름을 알고 있을 때 사용할 수 있으며 구현하기가 쉽다.

둘째, 나열식 방법으로 클래스 상속 관계를 트리로 해석해서 찾아낸다.

셋째, 패싯 방법으로 클래스의 특성 값을 이용하여 클래스를 찾아내는 것으로 각 클래스를 정보 저장소에 등록할 때 그 클래스의 특성을 표현할 수 있는 값들을 할당하고 탐색시에 이 값을 이용한다.

넷째, 기존에 알고 있는 클래스를 기준으로 그 클래스의 인접한 관계를 갖는 클래스를 찾아내는 방법이다. 그러나 기존의 부품 탐색 대상은 주로 독립된 소프트웨어를 찾아내는 것이다. 본 논문에서는 모듈은 모듈명으로 탐색하고 모듈내에 있는 클래스는 상속 관계를 트리로 해석할 수 있는 방법을 제시한다.

3. 객체 지향 클래스 계층 구조 재구성 방법

본 장에서는 클래스 라이브러리 계층 구조에 삽입과 삭제로 인해 변경되는 클래스 계층 구조를 재구성하기 위한 기준을 제시하고, 객체 지향의 장점인 클래스간의 관계성을 일관성 있게 유지되도록 한다. 따라서 이 방법은 클래스 계층 구조 변경시 클래스간의 의미 오류 가능성을 최소화할 수 있는 클래스 계층 구조 재구성 방법이라할 수 있다.

3.1 클래스 계층 구조 형성 방법

클래스 계층 구조 형성 방법은 정의된 클래스들의 정의를 조사하여 그들의 공통점을 이용하여 부모 클래스를 만들어 가는 과정을 나타낸 것이다. 이러한 점에서 볼 때 클래스 계층 구조를 통해 얻을 수 있는 장점은 수정이 쉽고 여러 클래스가 공통된 클래스 정의를 공유하기 때문에 정의 중복을 방지하여 일관성을 유지할 수 있다. 그러나 이러한 방법의 클래스 계층 구조의 장점은 하향식으로 [illegible] 때 나타난다. 즉, 기존의 클래스 계층 구조 내에 [illegible] 새로운 클래스를 자식 클래스로 생성하면 새로이 생성된 클래스는 자신의 부모 클래스가 갖고 있던 정의와 부모 클래스가 다른 클래스로부터 상속받는 정의를 모두 상속받기 때문이다. 그러나 구문적 접근 방법을 사용하여 고품질의 설계 결과를 이끌어 낸다는 것은 어려운 일이므로 여러 번 재수행으로 [illegible] 만들어진다. 따라서 아직까지 정형적(formal)인 [illegible] 존재하고 있지 않다. 이와 같은 클래스 계층 [illegible] 방법에는 상속 및 복합 계층 방법이 있는데, [illegible] 상속 계층을 위한 정의 및 조건은 다음과 같다.

[기본 정의]

도메인(domain) : 동일한 클래스내의 데이터 멤버와 멤버 함수

[정의 1] 상속 계층 관계에는 일반화와 세분화 관계로

나뉘어 진다.

〈조건 1〉 일반화는 부모 클래스 C_i의 도메인(domain)이 자식 클래스 C_j의 부분 집합이라면 C_i는 C_j의 일반화라 한다. 즉, $(C_i \supset C_j)$의 조건을 만족한다.

〈조건 2〉 세분화는 일반화의 역으로 $(C_j \supset C_i)$가 성립하면 자식 클래스 C_j는 클래스 C_i의 세분화라고 한다. 즉, $(C_j \supset C_i)$의 조건을 만족한다.

〔정의 2〕 클래스 상속 계층 구조 관계는 다음의 조건을 만족해야 한다.

〈조건 1〉 역으로 클래스 C_i와 C_j가 임의의 클래스 C_k의 상속 계층 구조를 가지려면 임의의 클래스 C_i의 도메인과 클래스 C_j의 도메인은 상호 독립이 아니다. 즉, $(C_i(d) \cap C_j(d)) \neq \varnothing$ 이다.

〈조건 2〉 클래스 C_i와 C_j가 상속 계층 구조라면 부모 클래스 C_i와 자식 클래스 C_j는, $C_i \supset C_j$이다.

〈조건 3〉 (3)의 조건을 만족하고 새로운 클래스 C_k가 $C_k \subset C_j$를 만족하면 클래스 C_i는 상위 클래스이고, 클래스 C_j는 중간 클래스이며, 클래스 C_k는 하위 클래스이다.

즉, $(C_i \supset C_j \supset C_k)$의 조건을 만족한다.

〈조건 4〉 클래스 $\{(C_k \subset C_i) \wedge (C_k \subset C_j)\}$를 만족하면 클래스 C_k는 클래스 C_i와 C_j로 부터 다중 상속(multiple inheritance) 관계이다.

새로운 클래스를 생성할 때 기능적인 면에서 유사한 클래스를 찾고 새로운 클래스를 기존의 클래스에 자식 클래스로 만들고 원하는 기능을 얻기 위해 부모 클래스를 추상화한다. 따라서 본 논문에서 제안하는 클래스 재구성시 기존의 클래스와 새로운 클래스와의 유사성을 검사하여 공통성을 포함하는 새로운 클래스를 만들고 그 클래스에서 상속을 받게 한다.

객체 지향 프로그래밍은 상속의 많은 이점을 제공하고 있지만 상속성의 사용과 관련된 비용을 생각하지 않을 수 없다. 상속을 남용하게 되면 메시지 전달에 따른 오버헤드가 발생하게 되어 지나치게 복잡도를 증가시킨다. 일반적으로 잘 개발된 클래스 상속 계층 구조는 상속 트리의 깊이가 깊고 너비가 좁아야 한다고 한다. 그러나 폭이 깊을수록 상속되는 함수가 많아지게 되므로 테스트 및 재사용을 어렵게 하여 심리적 복잡도를 더욱 증가시킨다. 이러한 경우 클래스 계층을 재구성하게 되면 지나치게 중첩된 상속 계층 구조의 깊이를 감축시킬 수 있다. 그 밖에 클래스의 상속 계층 구조의 깊이와 너비는 소프트웨어 복잡도 문제에 대한 하나의 해결책으로 임의의 상속 메카니즘보다는 복합 객체를 구성하는 객체를 구성 객체(sub-object)로 구성하는 것이 바람직하다.

복합 계층 구성방법을 살펴보면, 복합 객체는 관련있는 객체들이 모여서 구성된 집합화의 예로서 하나의 객체를 구성하는 구성원 역시 객체가 된다는 성질을 말한다. 여기에서 복합 객체를 구성하는 객체를 구성 객체(sub-object)라 하며 이는 재귀적(recursive)인 의미를 갖는다. 여기에서 구성 객체가 한 객체에만 속할 경우에는 배타적(exclusive) 성격이 되며, 동시에 여러 객체에 속할 수 있을 경우에는 공유적(shared) 성격이 된다. 이런 성격에 따라 복합 객체에서 발생하는 참조 관계는 배타적 의존, 배타적 독립, 공유적 의존, 공유적 독립 등의 관계를 가진다. 따라서 복합 객체는 구성 객체와의 관계를 기술함으로써 정의되며, 이들 관계는 복합 객체 계층 구성도로 표현된다.

3.2 클래스 유사성 형식 정의

이와 관련된 클래스 객체들의 관계성을 위한 형식 정의(formal definition)에 관한 연구는 상대적으로 미흡하다. 그 이유는 객체 지향 기법의 발전이 수학적 기반보다는 경험적 원리를 기반으로 하고 있기 때문이다. 형식 정의는 Wand에 의하여 객체 지향 시스템에 적합하도록 정의되었다[14]. 그리고 클래스들의 유사성(similarity)이란 클래스 사이에서 동일한 도메인(데이터 멤버 + 멤버 함수)이 많을수록 클래스간의 유사성 값은 증가하고 반면에 적을수록 감소한다. 따라서 이러한 값의 결과로 설계에서 상속 계층 구조 설정과 구현시 코드의 재사용 측면을 결정하게 된다[1,15,16].

본 논문에서는 기존의 연구 결과를 기반으로 정의한다[15].

〔기본 정의〕

- dm : 데이터 멤버
- dmf : 멤버 함수

〔정의 3〕 클래스들간의 유사성(SIM)에서는 다음을 만족하여야 한다.

$SIMC_{ij}$ = 임의의 클래스들간의 유사성을 측정하는 척도로 즉, 임의의 클래스 C_i와 C_j 사이의 함수 $SIM(C_{ij}) = f(C_i, C_j)$로 정의하며, 다음의 조건을 만족하여야 한다.

〈조건 1〉 $0 \le SIM(C_{ij}) \le 1$,

〈조건 2〉 $SIM(C_{ij}) = SIM(C_{ji})$

〈조건 3〉 $C_i = C_j \Rightarrow SIM(C_{ij}) = 1$

〔정의 4〕 클래스의 데이터 멤버 유사성

$$SIM_{dm}(A, B) = \frac{dm(A \cap B))^2}{dm(A) \times dm(B)}$$

〔정의 4〕에서 $dm(A)$는 클래스 A를 구성하는 데이터 멤버의 개수이고, $dm(B)$는 클래스 B를 구성하는 데이터 멤버의 개수를, 그리고 $dm(A \cap B)$는 두 클래스 내에서 공통적인 기능을 갖는 데이터 멤버의 개수를 나타낸다

〔정의 5〕 클래스의 멤버 함수 유사성

$$SIM_{dmf}(A, B) = \frac{dmf(A \cap B))^2}{dmf(A) \times dmf(B)}$$

〔정의 5〕 에서도 〔정의 4〕 같은 개념이지만 여기에서는 멤버 함수(dmf)를 나타낸다. 그러나 두 개의 클래스 멤버 함수가 같은 기능을 수행한다는 것을 판별하는 것은 어렵기 때문에 본 논문에서는 다음의 기준을 따른다[15,16].

첫째, 두 함수의 이름과 함수가 반환하는 값의 타입(type)이 같아야 한다.

둘째, 함수의 모든 인자들의 타입이 같아야 한다.

셋째, 함수가 참조하는 전역 변수의 타입이 같아야 한다.

넷째, 지역 변수들의 타입이 같아야 한다.

다섯째, 함수내의 제어 흐름도가 같아야 한다.

여섯째, 각 변수(함수 인자, 전역 변수, 지역 변수)별로 프로그램 상에서 프로그램 흐름도가 같아야 한다.

〔정의 6〕 클래스들의 유사성

$$SIM(A, B) = P \times SIM_{dm}(A, B) + (1-P) \times SIM_{dmf}(A, B)$$

본 논문에서는 같은 이름을 갖는 멤버 함수인 경우 타입을 비교하여 측정한다. 〔정의 6〕은 클래스들의 유사성으로 〔정의 4〕와 〔정의 5〕의 식을 사용하여 정의한 것이다. 여기에서 P는 클래스 A, B의 데이터 멤버들의 총 개수와 멤버 함수들의 총 개수를 합한 값에서 데이터 멤버들의 총 개수의 비율이다.

3.3 클래스 계층 구조 재구성을 위한 관계성 분류 방법

일반적으로 클래스의 삽입과 삭제로 인하여 클래스의 계층 구조가 변경될 경우 상속 관계가 변경되어 클래스 내부 구현 프로그램의 의미가 달라질 경우가 있다. 따라서 본 논문에서는 클래스 계층 구조 재구성시 의미의 변화를 최소화할 수 있는 방법을 제안한다.

본 논문에서 제안한 방법은 유사성 함수 정의에 따라 데이터 멤버 및 멤버 함수의 동일성을 판별하였다. 클래스 계층 변경을 위한 조건은 삽입과 삭제로 구분하였으며, 클래스를 검색하는 방법은 모듈 단위로 선형 검색을 하고 모듈안의 클래스는 클래스 관계성에 의한 트리 검색 방법을 채택하여 그 트리를 유사성을 비교해 가장 유사성이 높은 경로를 채택하여 계층 변경을 하고자 한다. 클래스 삭제로 인한 계층 변경은 삭제 위치에 따라 고려해야할 점이 서로 다르므로 다음과 같이 세 가지로 분류하였다.

첫째, 루트 클래스 삭제는 상속 계층에서 잘 일어나지 않는 경우이다. 하지만 한 모듈내에서 새로운 클래스를 삽입함으로 인해 계층 구조가 복잡하게 되면 클래스 복잡도를 고려하여 어느 시점에서는 모듈을 분해해야할 경우가 발생하므로 이때 루트 클래스를 삭제하여 모듈을 분해하게 된다. 이러한 경우는 삭제되는 클래스의 참조 데이터 멤버와 참조 멤버 함수를 세분화하고 모듈을 분해하여 서브 모듈을 작성한다. 그리고 상속 관계를 삭제한다.

둘째, 한 모듈 시스템에서 하나의 서브 모듈의 트리를 삭제할 때도 위와 같은 방법에 따라 루트 클래스를 삭제한 후 한 서브 모듈의 클래스를 삭제하면 된다. 이 방법은 리프 클래스 삭제이다. 리프 클래스 자체는 자식 클래스를 갖지 않으므로 상속 관계를 삭제하고 삭제하는 클래스의 참조 데이터 멤버와 참조 멤버 함수를 삭제한다.

셋째, 중간클래스를 삭제하는 경우이다. 중간 클래스를 삭제할 때는 자식 클래스에 상속으로 인한 참조 관계가 있다. 중간 클래스를 삭제할 때를 삭제 범위에 따라 두 가지로 나누어보면 삭제 클래스와 관련하는 클래스의 모든 참조 멤버 함수와 데이터 멤버를 모두 삭제하는 것과, 참조 데이터 멤버와 참조 멤버 함수를 세분화하고 삭제 클래스를 삭제하는 것으로 나눌 수 있다. 중간 클래스 삭제시 세분화로 내려주는 것은 기존의 추상화를 유지하기 위해서이다. 삭제에 의한 분류는 〈표 1〉과 같다.

〈표 1〉 삭제에 대한 분류
〈Table 1〉Classification of deletion

삭제 클래스	분류	삭제 방법
루트 클래스	모든 분해시	자식 클래스에 루트의 데이터 멤버와 멤버 함수를 세분화하고 루트 클래스를 삭제
	모든의 서브 클래스를 삭제시	루트에서 부터의 모든 서브 클래스 삭제
리프 클래스	리프 클래스 삭제	서브 클래스가 없으므로 삭제
중간클래스	삭제 클래스를 참조하는 데이타 멤버와 멤버 함수를 모두 삭제	자식 클래스에서 참조하는 데이터 멤버와 멤버 함수를 제거하고 삭제 클래스의 루트 클래스와 자식 클래스와 연결한 후 클래스 삭제
	삭제 클래스를 참조하는 데이타 멤버와 멤버 함수를 보존	자식 클래스에 공통의 멤버 데이터와 멤버 함수를 참조하고 있으면 공통 참조 클래스를 작성하고 부모 노드와 연결후, 각각 참조하는 데이터 멤버와 멤버 함수를 세분화한 후 공통 참조 클래스의 연결, 공통 참조 멤버가 없으면 세분화후 클래스 삭제

클래스를 삽입으로 인한 계층 변경은 데이터 멤버와 멤버 함수의 관계성을 고려한다. 우선 유사성을 비교한 결과에 따라 데이터 멤버 및 멤버 함수를 5가지로 구분하였다.

본 논문에서는 클래스의 관계성을 분석하기 위해 이들의 관계에 대하여 다음과 같이 정의한다. 그리고 클래스 삽입에 의한 분류는 〈표 2〉와 같다.

〔정의 7〕 유사 클래스
"삽입 클래스의 데이터 멤버와 멤버 함수가 한 모듈에 있는 기존의 클래스와 같은 경우가 있는 경로의 클래스들"

〔정의 8〕 인접 클래스
"삽입하기 위한 클래스에 인접된 유사 클래스"

〔정의 9〕 유사성 임시 클래스
"클래스 삽입시 클래스간의 유사성이 높은 경로를 추출하여 추출된 경로의 클래스를 임시 클래스에 세분화한 클래스"를 말하며 여기에서 임시 클래스는 삽입 클래스와 일반화한 후 삭제한다.

〔정의 10〕 가상 삽입 클래스
"삽입 클래스와 유사성 임시 클래스간의 공통인 데이터 멤버와 멤버 함수 및 가상 함수를 일반화한 클래스로 새로운 삽입 클래스가 된다".

〔정의 11〕 무관 관계
① 데이터 멤버 무관 관계 : "삽입할 데이터 멤버와 유사성 임시 클래스의 데이터 멤버 모두가 동일하지 않는 경우"
② 멤버 함수의 무관 관계 : "삽입할 멤버 함수와 유사성 임시 클래스의 멤버 함수 모두가 동일하지 않는 경우"

〔정의 12〕 부분 집합 관계
① 데이터 멤버 부분 집합 관계 : "삽입할 데이터 멤버가 유사성 임시 클래스의 데이터 멤버의 부분 집합일 경우"
② 멤버 함수의 부분 집합 관계 : "삽입할 멤버 함수가 유사성 임시 클래스의 멤버 함수의 부분 집합일 경우"

〔정의 13〕 포함 관계
① 데이터 멤버 포함 관계 : "삽입할 데이터 멤버가 유사성 임시 클래스의 데이터 멤버를 포함할 경우"
② 멤버 함수의 포함 관계 : "삽입할 멤버 함수가 유사성 임시 클래스의 멤버 함수를 포함할 경우"

〔정의 14〕 부분 관계
① 데이터 멤버 부분 관계 : "삽입할 데이터 멤버가 유사성 임시 클래스 데이터 멤버의 일부가 동일하고 일부는 동일하지 않는 경우"
② 멤버 함수의 부분 관계 : "삽입할 멤버 함수가 유사성 임시 클래스 멤버 함수가 일부는 동일하고 일부는 동일하지 않는 경우"

〔정의 15〕 동일 관계
① 데이터 멤버 동일 관계 : "삽입할 데이터 멤버가 유사성 임시 클래스의 데이터 멤버가 동일한 경우"
② 멤버 함수의 동일 관계 : "삽입할 멤버 함수가 유사성 임시 클래스의 멤버 함수와 동일한 경우"

3.4 클래스 계층 구조에서 삽입 방법

일반적으로 자동화되고 안정된 클래스 삽입과 삭제로 인한 계층의 재구성이 가능한 알고리즘을 찾는 것은 불가능하다. 본 논문에서 제시하는 클래스 삽입 방법은 기존의 프로그램을 기반으로 하여 클래스의 관계성을 유지하므로 다음과 같은 장점을 갖는다.

재사용 소프트웨어를 위한 구성요소를 위한 기본적인 틀을 제공 할 수 있다. 왜냐하면 이러한 방법은 갈리오 격자(Galois Lattice)로 클래스 인터페이스의 집합으로 만들 수 있다. 그러나 프로그램을 기반으로 할 경우는 오버라이딩 함수나 오버로딩 함수의 효율적인 관리를 하므로 클래스 계층 관계에 대한 확장이 용이하다. 사용자는 모듈내의 일치성을 강화할 수 있다. 왜냐

〈표 2〉 삽입에 대한 분류
〈Table 2〉 Classification of Insertion

데이타 멤버 관계	멤버 함수 관계	재구성 방법
동일관계	무관관계 (211)	기존의 데이타 멤버를 사용하므로 인접 클래스가 피프인 경우 공통성이 없는 나머지 데이타 멤버와 멤버 함수를 고려하여 하나로 합치거나 인접 클래스에 삽입한다. 그러나 함수는 이름이 같고 기능이 다른 함수는 가상 함수로 정한다.
	부분집합관계 (212)	기존 클래스에 포함되어 있으므로 삽입하지 않고 합병한다.
	포함관계(213)	인접 클래스가 피프인 경우 멤버 함수와 데이타 멤버의 개수를 고려하여 합치거나 인접 클래스에 삽입한다. 인접 클래스가 피프가 아닌경우 인접 클래스에 삽입한다.
	부분관계(214)	같은 데이터 멤버, 멤버 함수와 멤버 함수를 이름이 같고 기능이 다른 함수를 가상 함수로 정한다
	동일관계(215)	여러 존재하는 클래스이므로 삽입하지 않고 조정한다..
무관관계	무관관계(221)	무관한 클래스이므로 삽입하지 않는다. 단 삽입 클래스가 피프가 아닌 경우는 인접 클래스에 삽입한다.
	부분집합관계 (222)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
	포함관계(223)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
	부분관계(224)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
	동일관계(225)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
부분집합관계	무관관계(211)	기존의 데이타 멤버를 사용하므로 유사 클래스가 피프인 경우 공통성이 없는 데이타 멤버와 멤버 함수를 고려하여 하나로 합치거나 유사 클래스에 삽입한다. 그러나 함수는 이름이 같고 기능이 다른 함수는 가상 함수로 정한다..
	부분집합관계 (212)	기존 클래스에 포함되어 있으므로 삽입하지 않고 합병한다.
	포함관계(213)	유사 클래스가 피프인 경우 멤버 함수와 데이타 멤버의 개수를 고려하여 합치거나 유사 클래스에 삽입한다. 유사 클래스가 피프가 아닌 경우 유사 클래스에 삽입한다
	부분관계(214)	같은 데이터 멤버, 멤버 함수와 멤버 함수를 이름이 같고 기능이 다른 함수를 가상 함수로 정한다
	동일관계(215)	여러 존재하는 클래스이므로 삽입하지 않고 합병한다.
포함관계	무관관계(241)	동일 데이타 멤버와 기능이 다르고 함수명이 동일한 함수를 가상 함수로 정하고 남아있는 부분은 가상 클래스에 삽입한다.
	부분집합관계 (242)	삽입. 데이타 멤버만 남는 경우가 생긴다.
	포함관계(243)	공통 부분을 제외한 나머지를 인접 클래스의 하위 클래스에 삽입한다.
	부분관계(244)	공통 부분을 가상 삽입 클래스로 정하고 남겨진 부분은 가상 삽입 클래스에 삽입한다.
	동일관계(245)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
부분관계	무관관계(251)	동일 데이타 멤버와 기능이 다르고 함수명이 동일한 함수를 가상 함수로 가상 삽입 클래스에 올리고, 남아 있는 멤버 함수와 데이타는 가상 삽입 클래스에 삽입한다.
	부분집합관계 (252)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.
	포함관계(253)	같은 데이터 멤버, 멤버 함수와 [illegible] 나머지를 유사 클래스에 삽입한다.
	부분관계(254)	같은 멤버 데이터 멤버 함수를 [illegible] 남는 클래스는 가상 삽입 클래스에 삽입한다.
	동일관계(255)	삽입. 피프 클래스에 데이타 멤버만 남는 경우가 생긴다.

하면 독립적인 멤버 함수와 멤버 데이터의 범위를 제한 할 수 있다. 그러므로 클래스 구조를 그대로 유지한 채 데이터에서 다음과 같은 관계성을 이용하여 자동적으로 유지할 수 있도록 한다. 그러나 외부적으로는 완벽할 수 없으므로 변환 관계를 변환자에게 알리어 수정할 수 있도록 하여야한다. 또한 클래스 관계성을 구분함으로써 클래스 부품이 잘못 만들어진 클래스도 쉽게 구분할 수 있다. 클래스간의 유사도 측정은 유사도가 가장 높은 클래스들의 경로를 추출 한 다음 유사도가 있는 마지막 클래스에 [illegible] 멤버 및 멤버 함수를 세분화하여 유사성 [illegible] 클래스를 만든다. 데이터 멤버가 동일 할 때 [illegible] 함수에 [illegible]를 다음과 같이 구분할 수 있다.

첫째, 멤버 함수가 무관 관계일 경우에는 데이터 멤버는 기존의 것을 사용하며 새로운 멤버 함수를 가지는 클래스이므로 두 가지로 나누어 생각 할 수 있다.

① 멤버 함수명이 같고 기능이 다른 경우 새로운 삽입 클래스를 만들고 동일한 데이터 멤버와 가상 함수를 새로운 삽입 클래스를 만들어 일반화하여

올리고 남아 있는 데이터 멤버와 멤버 함수의 인접 클래스는 새로운 삽입 클래스의 자식 클래스로 하여 붙인다.

② 멤버 함수명이 모두 다른 경우에는 그 위치에 따라 하나로 합칠 수도 있고, 멤버 함수만 가지는 클래스를 인접 클래스를 부모 클래스로 하여 붙인다. 인접 클래스가 리프인 경우는 모두 적용이 가능하며 클래스 복잡도를 고려해 하나로 합칠 수도 있다. 리프가 아닌 경우는 인접 클래스를 부모 클래스로 하여 붙인다.

둘째, 부분집합관계일 경우에는 멤버 함수가 기존의 클래스에 포함되므로 삽입하지 않는다.

셋째, 포함 관계일 경우에는 삽입 클래스의 멤버 함수가 기존 클래스의 멤버 함수를 포함하므로 그 위치에 따라 하나로 합칠 수도 있고 멤버 함수만 가지는 클래스는 인접 클래스를 부모 클래스로 하여 붙인다. 인접 클래스가 리프인 경우는 모두 적용이 가능하며 클래스 복잡도를 고려해 하나로 합칠 수도 있다. 리프가 아닌 경우는 인접 클래스를 부모 클래스로 하여 붙인다.

넷째, 부분관계의 경우 기존의 클래스와 새로운 클래스가 서로 부분관계이므로 서로 같은 데이터 멤버와 멤버 함수, 멤버 함수중 이름이 같고 기능이 다른 함수를 가상 함수를 만들어 새로운 삽입 클래스를 만들어 일반화하여 올리고 남아 있는 데이터 멤버와 멤버 함수와 인접 클래스는 새로운 삽입 클래스의 자식 클래스로 하여 붙인다.

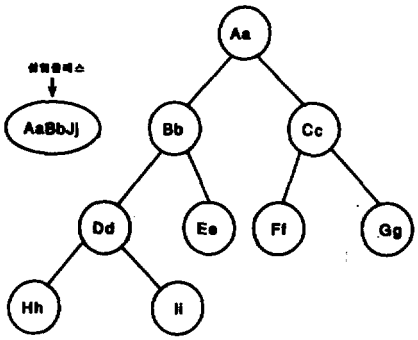
다섯째, 동일관계의 경우 이미 존재하는 클래스이므로 삽입하지 않고 합병한다.

데이터가 무관 관계일 때 멤버 함수는 부분집합관계, 포함관계, 부분관계, 동일관계로 나눌 수 있다. 멤버 함수 무관관계는 처음에 유사도 측정에 의해 제외되므로 참고하지 않는다. 그러나 삽입하여 올린 경우 즉 삽입 노드가 리프가 아닌 경우는 인접 클래스에 붙인다. 그리고 나머지의 경우 부분집합관계, 포함관계, 부분관계, 동일관계 조건이 성립하려면 동일 멤버 함수를 올리는 경우로 리프 클래스에 데이터만 남는 경우가 생길 수 있으므로 잘못된 클래스일 가능성이 있으므로 ERROR처리를 한다. 왜냐하면 참조하지 않는 데이터 멤버는 존재하지 않기 때문이다. 데이터 멤버가 부분집합일 경우에는 데이터가 이미 기존의 클래스에 데이터 멤버가 포함되어 있으므로 데이터 멤버가 동일관계일 때와 유사하다.

데이터 멤버가 포함 관계일 때 멤버 함수가 무관관계이면 같은 데이터 멤버와 기능이 다르고 함수명이 동일한 함수를 가상 함수로 새로운 삽입 클래스로 하여 일반화하여 올리고, 남아있는 멤버 함수와 데이터 멤버 그리고 인접 클래스는 새로운 삽입 클래스의 자식 클래스로 하여 붙인다. 가상 함수로 일반화하여 올라가는 멤버 함수가 없는 경우에는 인접 클래스를 부모 클래스화하여 붙인다. 인접 클래스가 리프인 경우는 클래스 복잡도를 고려해 하나의 클래스에 합칠 수도 있다. 데이터 멤버가 포함관계이면서 멤버 함수가 부분 집합관계일때와 멤버 함수가 동일관계일 경우에는 동일한 데이터 멤버와 멤버 함수 부분을 올리고 나면 리프에 데이터 멤버만 남는 클래스가 존재하므로 잘못된 클래스일 가능성이 있으므로 에러 처리한다. 왜냐 하면 처음의 삽입 클래스는 항상 리프 클래스이므로 참조하지 않는 데이터 멤버는 존재 할 수 없기 때문이다. 멤버 함수가 포함관계일 경우에는 공통인 멤버 함수와 데이터 멤버 부분을 제외한 나머지를 인접 클래스의 자식 클래스에 붙인다. 멤버 함수가 부분관계일 경우에는 공통인 데이터 멤버와 멤버 함수를 삽입 클래스로 일반화하여 올리고 남겨진 부분과 인접 클래스는 삽입 클래스를 부모 클래스로 하여 연결한다.

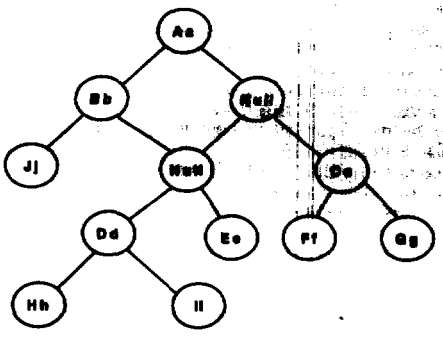
데이터 멤버가 부분 관계일 경우에는 멤버 함수가 무관 관계이면 동일한 데이터 멤버와 기능이 다르고 함수명이 동일한 함수를 가상 함수로 하여 새로운 삽입 클래스에 일반화하여 올리고, 남아있는 멤버 함수와 데이터 멤버 그리고 인접 클래스는 새로운 삽입 클래스를 부모 클래스화 하여 붙인다. 인접 클래스가 리프인 경우는 클래스 복잡도를 고려해 인접 클래스에 합칠 수도 있다. 가상함수로 일반화하여 올라가는 멤버 함수가 없는 경우에는 인접 클래스를 부모 클래스화 하여 붙인다. 멤버 함수가 부분 집합관계이거나 동일관계일 때 남겨지는 클래스가 데이터만 존재하므로 잘못된 클래스일 가능성이 있으므로 에러 처리한다. 멤버 함수가 포함 관계일 경우에는 공통부분을 제외한 나머지를 유사 클래스를 부모 클래스로 하여 연결한다. 또는 인접 클래스가 리프인 경우 클래스 복잡도를 고려해 유사 클래스와 합쳐 하나의 클래스를 만든다. 멤버 함수가 부분 관계일 경우에는 공통인 멤버 함수와 데이터 멤버 부분을 새로운 삽입 클래스에 일반화하여 올리고 남는 클래스와 인접 클래스 새로운 삽입 클래스의 자식 클래스화 하여 붙인다.

다음 예들은 클래스 삽입으로 인한 재구성 과정을 표현한 것이다. 아래 예는 인접 클래스를 찾는 형태 중 데이터 멤버와 멤버 함수가 포함관계인 경우를 나타내었다. (그림 1)은 새로운 클래스를 삽입하기 전의 클래스 계층구조이고 영문 대문자는 클래스내의 멤버 함수를 나타내고 소문자는 데이터 멤버를 나타낸다. 삽입하고자 하는 클래스 AaBbJj를 기존 클래스 계층구조에서 유사도가 있는 클래스의 마지막 유사 클래스는 Bb에 인접시킨다. 이때 유사성 임시 클래스는 AaBb가 된다.



(그림 1) 삽입 하기전 클래스 계층도
(Fig. 1) Class hierarchy diagram before insert

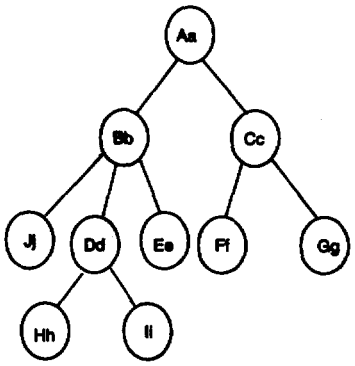
삽입 클래스 AaBbJj는 유사성 임시 클래스 AaBb와 데이터 멤버, 멤버 함수가 모두 포함 관계이다. 두 클래스의 공통인 데이터 멤버와 멤버 함수 부분을 올리면 아래 (그림 2)와 같이 새로운 삽입 클래스 AaBb가 되고 인접 클래스 Bb가 Null이 되면 다시 분해간 AaBb는 Aa클래스와 다시 관계를 비교하여 데이터 멤



(그림 2) 삽입하는 과정의 클래스 계층도
(Fig. 2) Class hierarchy diagram of insert process

버 및 멤버 함수가 모두 포함 관계가되므로 두 클래스의 공통부분을 올리면 Aa클래스가 Null클래스가 된다.

따라서 Null 클래스는 클래스 계층구조에서 있을 수 없으므로 Null 클래스를 삭제한 후 Null 클래스의 부모 클래스와 서브 클래스를 연결하면 아래와 같은 모습의 클래스 계층이 완성된다.

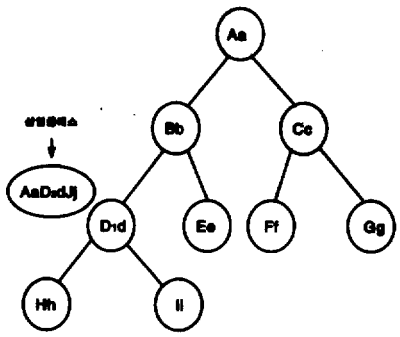


(그림 3) 삽입후 클래스 계층도
(Fig. 3) Class hierarchy diagram after insert

그러므로 (그림 3)은 결과적으로 분류 관계 243을 적용하면 멤버 함수, 데이터 멤버가 포함관계인 경우에는 삽입 클래스는 유사성 임시 클래스에 있는 멤버 함수와 데이터 멤버를 제외한 나머지와 인접 클래스를 새로운 삽입 클래스 아래 붙인다. 즉 공통인 멤버 함수와 데이터 멤버 부분을 일반화하여 올리는 형태이다.

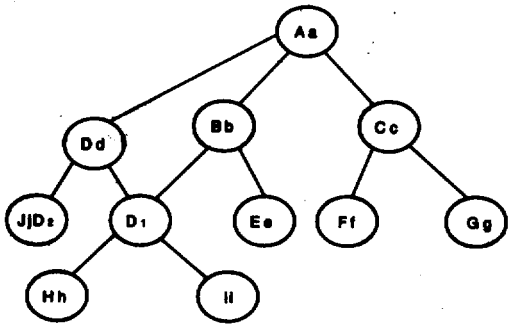
아래의 예는 데이터 멤버와 멤버 함수가 부분관계인 경우로 삽입 클래스 AaD_2dJj는 클래스 기존 클래스 계층 구조에서 유사도가 있는 클래스의 유사 클래스는 D_1d 인접시키고 이때 유사성 임시 클래스는 $AaBbD_1d$이다. 두 클래스는 데이터 멤버, 멤버 함수가 부분관계이며 멤버 함수 D_1, D_2는 이름은 같지만 기능이 다른 멤버 함수이다.

(그림 4)에서 분류 관계 254를 적용하면 두 클래스의 공통부분 멤버 함수와 데이터 멤버인 Aad와 이름은 같지만 기능이 다른 멤버 함수 D_1, D_2는 가상함수 D로 하여 올리면 AadD는 새로운 삽입 클래스가 된다. 처음 유사 클래스의 부모 클래스 Bb와 세분화한 유사성 임시 클래스 AaBb 와도 멤버가 부분 관계이므로 분류 관계 254 에서 공통부분을 Aa를 올린다 공통부분으로 올린 Aa는 Bb의 부모 클래스와 같으므로 클래스 이름



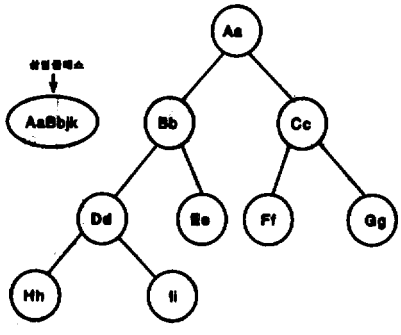
(그림 4) 삽입 하기전의 클래스 계층도
(Fig. 4) Class hierarchy diagram before insert

이 같은 경우는 하나의 클래스로 합병한다. (그림 5)에서 분류 관계 215 적용한다.



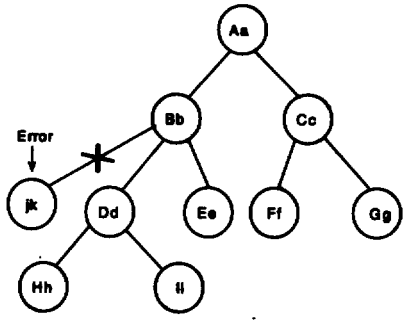
(그림 5) 삽입후 클래스 계층도
(Fig. 5) Class hierarchy diagram after insert

다음은 삽입 클래스가 ERROR 인 경우이다. (그림 6) 삽입 클래스가 ERROR인 경우는 공통부분인 데이터 멤버와 멤버 함수를 올린 후에 리프 데이터 멤버만 남는 경우이다. 삽입 클래스 AaBbjk는 인접 클래스Bb와 유사성 임시 클래스 (AaBb)는 멤버 함수는 동일관계이며 데이터 멤버는 포함관계이다. 멤버 함수가 동일관계이고 데이터 멤버가 포함관계일때는 인접 클래스에 포함되는 멤버만 제외하고 인접 클래스 아래 삽입 클래스를 붙일 수 있다.



(그림 6) 삽입전 클래스 계층도
(Fig. 6)Class hierarchy diagram before insert

그러나 아래 (그림 7)은 클래스를 삽입한 후의 계층도 인데 삽입후 데이터 멤버 jk 만 남는 클래스가 리프이므로 데이터 멤버만 남는 경우에는 잘못된 클래스일 가능성이 있다. 왜냐하면 참조가 없는 데이터 멤버만 존재하는 것은 의미가 없다. (분류 관계 245)



(그림 7) 삽입후 클래스 계층도
(Fig. 7) Class hierarchy diagram after insert

다음 알고리즘은 삽입에 따른 재구성에 나타낸 알고리즘이다.

```
〈삽입시 알고리즘〉
Procedure restruct_Class ( )
Create Temp_NewClass_Table(Class_name, member_Function, member_Variable)
 // 새로운 클래스정보 생성 //
Create LinkClass_Table(Temp_NewClass_Table, ClassTable,Inherant_Table)
// 유사도가 있는 클래스 정보및 인접 클래스 추출 //
Create Temp_MaxClass_Table( LinkClass_Table,ClassTable)
// 유사성 임시 클래스 생성//
if ( Maxdmf ∩ Newdmf = ∅ and  Maxdm ∩ Newdm = ∅)
 //데이터 멤버 무관관계, 멤버 함수 무관 관계 //
then LoopError  // 무관관계 //
Relation(Temp_MaxClass_Table, Temp_NewClass_Table)
   if (Maxdm=Newdm  OR Maxdm ⊃ Newdm)
  // 데이터 멤버 가 동일관계이거나, 데이터 멤버가 부분집합관계 일 때 //
  then  if  (Maxdmf ∩ Newdmf = ∅)
   then RelationCode=211     // 멤버 함수 무관관계 //
      else if (Maxdmf ⊃ Newdmf)
         then RelationCode=212  // 멤버 함수 부분집합관계 //
        else if (Maxdmf ⊂ Newdmf)
           then RelationCode=213      // 멤버 함수 포함관계 //
           else if (Maxdmf ∩ Newdmf ≠∅)
              then RelationCode=214       // 멤버 함수 부분관계//
              else if (Maxdmf = Newdmf)
                 then Relation Code=215    // 멤버 함수 동일관계 //
else if (Maxdm ∩ Newdm = ∅) //데이터 멤버가 무관관계일 때//
     then if (Maxdmf ⊃ Newdmf)
        then RelationCode=222 // 데이터 멤버 무관관계, 멤버 함수 부분집합관계 //
```

```
        else if (Mmemf ⊂ Newdmf)
            then RelationCode=223  // 데이터 멤버 부분관계, 멤버 함수 포함관계 //
            else if (Mmemf ∩ Newdmf) ≠∅
              then RelationCode=224 //데이터 멤버 부분관계
                                    멤버 함수 부분관계 //
              else if (Mmemf = Newdmf)
                 then RelationCode=225 //데이터 멤버 부분관계,
                                    멤버 함수 동일관계 //
        else if (Mmemm ⊂ Newdm) // 데이터 멤버가 포함관계 일때 //
            then if (Mmemf ∩ Newdmf = ∅)
                then RelationCode=241  // 데이터 멤버 포함관계,
                                    멤버 함수 부분관계 //
                else if (Mmemf ⊃ Newdmf)
                   then RelationCode=242 // 데이터 멤버 포함관계,
                                    멤버 함수 부분겹침관계 //
                   else if (Mmemf ⊂ Newdmf)
                      then RelationCode=243  // 데이터 멤버 포함관계,
                                    멤버 함수 포함관계 //
                      else if (Mmemf ∩ Newdmf) ≠∅
                         then RelationCode=244 // 데이터 멤버 포함관계,
                                    멤버 함수 부분관계//
                         else if (Mmemf = Newdmf)
                            then RelationCode=245 //데이터 멤버 포함관계,
                                    멤버 함수 동일관계 //

        else if (Mmemm ∩ Newdm) ≠∅ // 데이터 멤버가 부분관계일 때 //
            then if (Mmemf ∩ Newdmf = ∅)
                then RelationCode=251  // 데이터 멤버 부분관계,
                                    멤버 함수 부분관계 //
                else if (Mmemf ⊃ Newdmf)
                   then RelationCode=252 // 데이터 멤버 부분관계,
                                    멤버 함수 부분겹침관계 //
                   else if (Mmemf ⊂ Newdmf)
                      then RelationCode=253  // 데이터 멤버 부분관계,
                                    멤버 함수 포함관계 //
                      else if (Mmemf ∩ Newdmf ≠∅)
                         then RelationCode=254 // 데이터 멤버 부분관계,
                                    멤버 함수 부분관계//
                         else if (Mmemf = Newdmf)
                            then RelationCode=255
                           //데이터 멤버 부분관계, 멤버 함수 동일관계//
switch (RelationCode)
    case 211 : RestructureClass(Temp_MaxClass_Table, Temp_NewClass_Table) : breake
      // 가상 삽입 클래스에 클래스에 동일 데이터 멤버 멤버와 기능은 다르나 이름이 같은
          멤버 함수를 가상 함수로 등록하는 함수로 //
    case 212 : SubClass(Temp_MaxClass_Table, Temp_NewClass_Table) : breake
          // 이미 존재하는 기능이므로 삽입하지 않고 탈출한다 //
    case 213 : SubClass(Temp_MaxClass_Table, Temp_NewClass_Table) : breake
      // 공통 부분을 올라가서 Null 클래스가 생기면 이러한 경우 나머지 클래스만을 구현
          한 클래스 아래 올라가고 올라서 따라 만일 클래스에 등록하는 함수 //
    case 214 : RestructureClass(Temp_MaxClass_Table, Temp_NewClass_Table) : break
    case 215 :SubClass(Temp_MaxClass_Table, Temp_NewClass_Table) : break
          // 이미 존재하는 클래스이므로 삽입하지 않고 탈출한다. //
    case 221 :SubClass(Temp_MaxClass_Table, Temp_NewClass_Table) : break
         //만일 클래스와 탈출한다. //
    case 222 : Error : break // 정의된 클래스일 가능성이 있다 //
    case 223 : Error : break
    case 224 : Error : break
    case 225 : Error : break
    case 241 : RestructureClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 211과 동일 //
    case 242 : Error : break
    case 243 : SubClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 213과 동일 //
    case 244 : RestructureClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 214과동일 //
    case 245 : Error : break
    case 251 : RestructureClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 211과 동일 //
    case 252 : Error : break
    case 253 : RestructureClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 214와 동일 //
    case 254 : RestructureClass(Temp_MaxClass_Table,Temp_NewClass_Table) : break
         // 214와 동일 //
    case 255 : Error : break

RestructureClass(Temp_MaxClass_Table, Temp_NewClass_Table)  // 클래스 재구성 //
 Kdmf= NewClassdmfVal - (MaxClassdmfVal ∩ NewClassdmfVal)
 Kdm= NewClassdmVal - (MaxClassdmVal ∩ NewClassdmVal)
 // 클래스를 탐색첫번째의 비교함 //
 if ((MaxclassDmfName∩NewClassDmfName) ≠ ∅) // 기능은 다르나 함수명이 같은
                                    경우 //
 then Create Temp_VirtualFunction( ) // 가상함수 테이블을 작성 한다. //
 Creat Newclass_Table(Temp_MaxClass_Table, Temp_NewClass_Table,Temp_VirtualFunction )
   // 공통인 멤버 함수와 데이터 멤버로 일반화된 클래스 //
     Newclass_Table=(Temp_MaxClass_Table ∩ Temp_NewClass_Table) +Temp_ VirtualFunction
     //
 Creat RemainClass_Table(Temp_MaxClass_Table, Temp_NewClass_Table)
    // 공통인 멤버 함수와 데이터 멤버로 일반화된 나머지 클래스
       RemainClass_Table =Temp_NewClass_Table
         -(Temp_MaxClass_Table ∩ Temp_NewClass_Table)//
 Update Link_Class_Table(Temp_NewClass_Table, Temp_MaxClass_Table)
    //연결 클래스 테이블 수정//

  if (Kmdmf(=K and Kdm(=K) and  Link_classs(LinkClass_Table)=LeafNode)
   // 클래스 복잡도 기준값(K) 이하이고 연결 클래스가  리프 클래스일때//
     then  Link(RemainClass_Table,  LinkClass_Table)
        //클래스를 합한다//
       Exit

        else Link(RemainClass_Table,  LinkClass_Table, Newclass_Table)
   Delete from  Temp_NewClass_Table,Temp_VirtualFunction
   Drop Temp_MaxClass_Table // 유사성 임시 클래스 삭제//
   Temp_NewClass_Table = Newclass_Table //가상 삽입 클래스//
   Create Temp_MaxClass_Table( LinkClass_Table,ClassTable)//새로운 유사성 임시 클래스 생성//
   Relation(Temp_MaxClass_Table, Temp_NewClass_Table) // 새로운 관계성 함수로의 호출)

SubClass(Temp_MaxClass_Table,Temp_NewClass_Table) // 두클래스를 합쳐가나 탈한다//
Kdmf= NewClassdmfVal - (MaxClassdmfVal ∩ NewClassdmfVal)
Kdm= NewClassdmVal - (MaxClassdmVal ∩ NewClassdmVal)
if (Kmdmf(=K and Kdm(=K) and  Link_classs(LinkClass_Table)=LeafNode)
   // 비교값이 기준값(K) 이하이고 연결 클래스가  리프 클래스일때//
    then  Link(RemainClass_Table,  Link_Class_Table)
       //클래스를 합한다//
      Exit

    else
     Link_Class_Table  = Newclass_Table
     Link(RemainClass_Table,  Link_Class_Table)
```

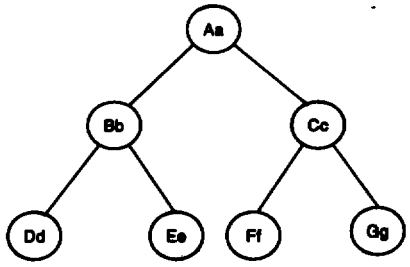
3.5 클래스 계층 구조에서 삭제 방법

본 논문에서 클래스 삽입은 클래스 유사도를 이용하여 클래스 계층 구조를 변경하였으며 클래스 삭제시 클래스의 미치는 영향을 정확히 파악하여야 한다. 클래스의 삭제로 인한 계층 구조의 변경도 상속관계를 변경하므로 클래스 내부 구현 코드가 변경되어야 한다. 그러므로 클래스가 삭제될 경우는 기존의 클래스 구조관계를 유지하기가 어렵다. 같은 모듈 내에서 클래스 참조관계 즉, 데이터 멤버의 참조관계와 멤버 함수의 참조관계를 고려하여야 한다. 따라서 이러한 참조관계에 관해 제시하는 클래스 삭제 방법은 다음과 같다. 클래스를 삭제하는 경우를 루트 클래스 삭제, 리프 클래스 삭제, 중간 클래스 삭제 등의 세 가지 형태로 나누었다. 각각에 대한 알고리즘을 아래와 같이 설명하면 다음과 같다.

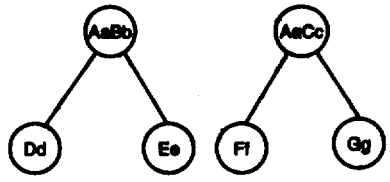
```
( #1루트 클래스 삭제시 알고리즘)

Search(DeletedClass)
if DeletedClass=RootNode  // RootNode 삭제시 //
 then if ModulePartition  // Module 분할될 때 //
       then for i=1 to n  // n은 Root Node의 SubNode의 개수 //
             SubClass(i)=SubClass(i)+DeletedClass;
            DelClass()  // 클래스 삭제//
       else DelModule() // 한 Module의 서브 트리 삭제 //
```

#1알고리즘과 같이 루트 클래스를 삭제할 때는 두 가지 형태가 있을 수 있다. 첫 번째 경우는 루트 클래스 아래 서브 트리를 서로 다른 서브 모듈로 분할할 때 루트 클래스의 데이터 멤버와 멤버 함수를 각 자식 클래스에 세분화하고 루트 클래스를 삭제하고 서브 모듈을 생성한다. 이러한 경우는 클래스의 계속적인 삽입으로 인한 경우로 클래스의 깊이 넓이를 고려해 모듈을 분해 생성하는 경우이다. 이러한 경우는 구성 객체(sub-object)로 구성하는 것이 바람직하다. 두 번째의 경우는 한 모듈 안에서 하나의 서브 트리를 모두 삭제하고자 하는 경우는 루트 클래스의 데이터 멤버와 멤버 함수를 각 자식 클래스에 세분화하고 루트 클래스를 삭제한다

(그림 9)는 삭제하기전 클래스 계층도 이고 (그림 10)은 루트 클래스 아래 서브 트리를 서브 모듈로 분할한 예이다. 루트클래스 Aa는 삭제되기 전에 서브 트리 Bb와 Cc아래 먼저 세분화되어야 한다. 루트 클래스의 멤버 함수와 데이터 멤버를 각각 자식 클래스에게 세분화한 후 루트 클래스를 삭제한 것이다.



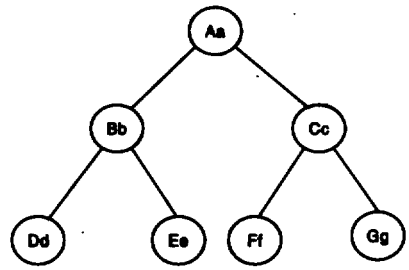
(그림 9) 삭제 하기전 클래스 계층도
(Fig. 9) Class hierarchy diagram before Delete



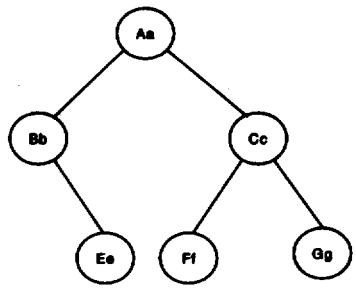
(그림 10) 삭제후 클래스 계층도
(Fig. 10) Class hierarchy diagram after Delete

```
( #2 리프 클래스 삭제시 알고리즘 )
   if DeletedClass=LeafNode
       then DelClass()   // 클래스 삭제//
```

#2 알고리즘은 삭제하려는 클래스가 리프 클래스인 경우 참조하는 자식 클래스가 없으므로 삭제하면 된다. (그림 11)은 리프 클래스 Dd를 삭제하기전의 클래스 계층을 나타내고 있고 (그림 12)는 Dd를 삭제한 후의 클래스 계층을 나타내고 있다.



(그림 11) 삭제 하기전 클래스 계층도
(Fig. 11) Class hierarchy diagram before Delete



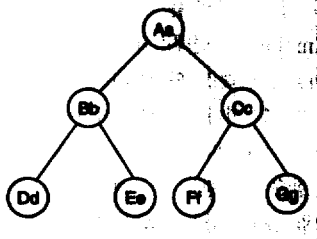
(그림 12) 삭제한 후 클래스 계층도
(Fig. 12) Class hierarchy diagram after Delete

```
( #3 중간 클래스 삭제 알고리즘 )
      if Alldmf and Alldm //중간 클래스 삭제시, 자식 클래스에서 삭제할 클래스를
                          참조하는 모든 멤버 함수와 데이터 멤버를 삭제하고자할때//
         then Search_and_Delete(AllSubClass,DeletedClass) // 자식 클래스에서
              삭제 클래스를 참조하는 멤버 함수와 데이터 멤버를 찾아 삭제한다 //
              for i=1 to n
              {
              Link(ParentClass,SubClass(i)):  //자식 클래스와 부모 클래스를 연결한다//
              DelClass() // 클래스 삭제 //
              }
```

```
else  Search_SharedClass(AllSubClass,DeletedClass) // 자식 클래스에서
      공통으로 참조하는 멤버 함수나 데이터 멤버를 찾는다 //
      Create(Search_SharedClass)  // 자식 클래스에서 공통으로 참조
      하는 멤버로 구성된 클래스를 작성한다 //
      DeletedClass=DeletedClass-Search_SharedClass
      for i=1 to n
      {
        Search(DeletedClass,TravleSubClass(i),Referencedmf,Referencedm)
//삭제하고자하는 서브 클래스에서 참조하는 멤버 함수와 데이터 멤버를 찾는다 //
        SubClass(i)=SubClass(i) + Referencedmf+Referencedm
//참조하는 멤버 함수와 데이터 멤버를 서브 클래스로 세분화하여 내려준다//
      }
        if Search_SharedClass ( ) Null
        then { Link (ParentClass,Search_ShardClass)
               ParentClass=Search_ShardClass }
          // 새로 작성된 공통참조 클래스가 Null이거나 더 공통 참조 클래스를
             부모 클래스와 연결하고 공통참조 클래스를 Parent클래스로 한다//
        for i=1 to n
        {
         Link(ParentClass,SubClass(i)); // 부모클래스에 자식 클래스를 연결 //
         DelClass() //클래스 삭제//
        }
```

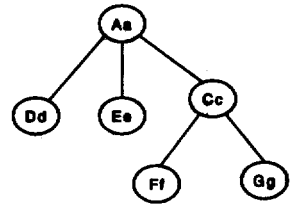
증간 클래스를 삭제할 때에는 삭제하는 클래스를 참조하는 자식 클래스를 고려해야한다. 첫 번째의 경우는 삭제 클래스와 연관되는 자식 클래스 데이터 멤버 및 멤버 함수를 삭제하려 할 때에는 자식 클래스를 방문하여 연관된 부분을 트리거 삭제하고 부모 클래스와 자식 클래스를 연결시킨 후 삭제하면 된다. 두 번째 경우는 중간 클래스의 한 서브 트리를 삭제 하고자 할 때 삭제 클래스와 연관되는 멤버를 삭제되지 않는 한 [illegible] 의 자식 클래스에 세분화하고 삭제 클래스의 부모 클래스와 자식 클래스를 연결시킨 후 삭제 클래스를 삭제시킨다. 세 번째 경우는 중간 클래스 이후의 모든 서브 트리를 삭제 하고자 할 때는 리프 클래스의 삭제와 동일하다.

(그림 13)은 중간 클래스 Bb를 삭제하기전의 클래스 계층이다.



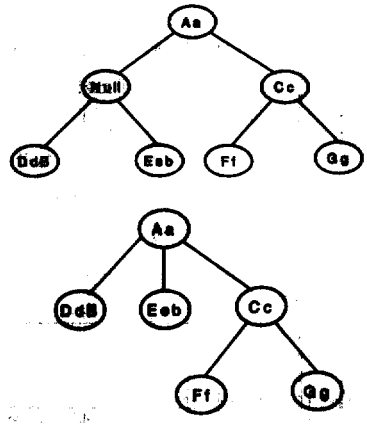
(그림 13) 삭제 하기전 클래스 계층도
(Fig. 13) Class hierarchy diagram before Delete

(그림 14)는 자식 계층에서 중간 클래스(Bb)를 참조하는 모든 멤버 함수와 데이터 멤버를 제거한 후 부모 클래스(Aa)와 연결시키고 중간 클래스를 삭제한 후의 모습이다. 중간 클래스(Bb)를 참조하는 모든 멤버를 자식 클래스로부터 제거하므로 결과적으로는 자식 계층을 바로 부모 클래스에 연결시킨 것과 같은 형태가 된다.



(그림 14) 삭제후 클래스 계층도
(Fig. 14) Class hierarchy diagram after Delete

(그림 15)는 삭제하려는 클래스중 자식 클래스에서 참조하는 멤버를 남겨두기위해 세분화하는 경우이다. 삭제 클래스 Bb는 자식 클래스 Dd와 Ee에서 각각 참조하는 B와 b가 세분화 된후 공통 참조 클래스가 Null 클래스이므로 부모 클래스와 연결한 후 중간 클래스를 삭제한다.



(그림 15) 삭제후 클래스 계층도
(Fig. 15) Class hierarchy diagram after Delete

4. [illegible] 평가

4.1 [illegible] 예

본 논문에서는 재사용을 위하여 설계 부품 정보 및 원시 프로그램을 모듈 단위로 저장하고 탐색할 수 있도록 하였다. 그 모듈은 하나의 사건을 처리하는 프로그램 단위에서 상위의 추상화 클래스로 일반화시켜 최상위 수준의 클래스를 생성하는 단위이다. 그리고 모듈간

에 참조할 수 있는 인터페이스 클래스도 추출하여 저장한다. 객체 지향의 특징은 각 모듈의 참조 관계, 상속 관계, 복합 관계, 인터페이스 관계를 생성하고, 세부적인 다형성 관계, 프랜드 관계 등의 추가적인 정보를 생성하게 된다. 따라서 클래스 유사성 분석시 클래스 상속 관계를 그래프로 해석하고 클래스 삽입 및 삭제로 인한 클래스 계층 구조 재구성 정보를 알 수 있다. 그리고 원시 프로그램은 화일 단위로 저장하여 아이콘화 하였다. 본 논문에서는 학사 관리 업무 중 수강신청 관리를 적용하여 클래스 계층 구조 재구성을 적용하였다. 구현언어는 비쥬얼 베이직과 SQL을 사용하였다. 수강신청 모듈에서 추출된 클래스는 Example, Course, Person, Schedule, Student, Employee, ScheduleCourse, Clerk, Professor로 분류되었다. 그리고 자료 구조를 구현하기 위한 Collection을 인터페이스 클래스로 추출하였다. 인터페이스 테이블은 서로 다른 모듈에서 참조하는 클래스로 클래스의 복잡도에 의거 할당 된 클래스이다. 그리고 정보 은닉화에 어긋나는 프랜드 함수는 재구성에는 적용하지 않는다. (그림 16)은 한 모듈 프로그램에서 각 프로그램을 다음과 같은 형태로 분석한다. 클래스에 정의된 이름과 상위 클래스 관계를 정의하고 그 클래스에 사용하는 데이터 멤버와 멤버 함수의 형태로 분석한다.

(그림16)에서 분석을 기반으로 하여 분석한 수강 신청 모듈의 클래스로 클래스간의 상속 관계는 (그림17)과 같은 형태로 데이터 베이스에 저장한다.

클래스 명 : Collection		
member variable	Example **list; int size; int filled; static int GrowIncrement	
member function	name	used value and function
	void add_alike(String &);	filled , list[0]->alike(name), add(element);
	Collection()	list, GrowIncrement, size, filled;
	int get_size()	filled;
	void add(Example *);	search(element),filled,size, list,GrowIncrement
	Example* search(String &);	filled, list[i]->identify(key)
	Example* search(Example *);	filled, list[i]
	void remove(Example *);	list, filled;
	void print();	filled; list[i]->print();

	void maintain();	print(); search(selection)
		sel_example->print();
		remove(sel_example);
		sel_example->maintain();
		add_alike(selection);

	Example* select();	print();search(String selection);

Class Name : Course: public Example		
member variable	String number; String name; int capacity; static Collection *All; protect: Collection* Course::All = new Collection;	
member function	name	used value and function
	Course(String nu, String na);	number(nu),name(na),capacity(25) All->add(this);
	void print()	number , name
	bool identify(String &);	
	Example* alike(String &);	All->search(name), new Course(name,"")
	void maintain();	capacity; name; print();
	bool add(Example *)	none
	Example* select()	none

(그림 16) 모듈의 프로그램 관계 테이블
(Fig. 16) Program analysis Relationship table of module

superClass	subClass	relation	approach
Example	Course	IS_A	pub
Example	Person	IS_A	pub
Person	Employee	IS_A	pub
Person	Student	IS_A	pub
Employee	Clerk	IS_A	pub
Employee	Professor	IS_A	pub

(그림 17) 클래스 상속 관계 테이블
(Fig. 17) Class inheritance Relationship table

(그림 16)에서 분석을 기반으로 하여 (그림 18)에서는 멤버 함수의 참조관계 정보를 분석하여 저장된 형태이다. 예를 들면 Collection 클래스의 add(Example*) 멤버 함수는 자기클래스의 search(element)와Example 클래스의 생성자 함수를 사용한 것을 알 수가 있다. 그리고 Employ 의 클래스와 Person 클래스는 상속 관계로 Person 클래스 의 Print함수를 재정의 하여 사용한 것을 알 수 있다. 그뿐만 아니라 중복 정의된 멤버 함수는 인자형에 의해 결정된다. 그러므로 참조 테이블에 멤버 함수 인자형을 구분하므로 알 수가 있다. 그러므로 (그림 18)과 같이 멤버 함수 참조 테이블을 작성하였을 때 멤버 함수 이름으로만 구분 할 수 없는 오버로딩의 함수를 구분이 된다. 그리고 상속 테이블과 같은 멤버 함수를 조인하면 오버라이딩 된 함수도 구분이 된다.

(그림19)는 각 클래스 멤버 함수가 클래스에 정의된 데이터나 다른 클래스에 정의 된 데이터를 참조를 나타낸 것이다.

Class1	Method1	class2	Method2	Used Type
Collection	add(Example*)	Collection	search(element)	
Collection	add(Example*)	Example	Example()	instance
Collection	add_alike(String&)	Example	alike(name)	instance
Collection	add_alike(String&)	Collecton	add(element)	
Collection	get_size()			
Collection	maintain()	Collection	search(selection)	
Collection	maintain()	Example	print()	instance
Collection	maintain()	Collection	remove(sel_example)	
Collection	maintain()	Example	maintain()	instance
Collection	maintain()	Collecton	add_alike(selection)	
Collection	print()	Example	print()	
Collection	remove(Example*)			
Collection	search(String &)	Example	identify(key)	instance
Collection	search(Example*)			
Collection	select()	Collection	print()	
Collection	select()	Collection	search(selection)	
Example	add(Example*)			
Example	alike(String &)			
Example	identify(String &)			
Example	maintain()			
Example	print()			
Example	select()			
Course	add(Example*)			
Course	alike(String&)	Collection	search(name)	instance
Course	alike(String&)	Cource	Cource(name,"")	
Course	identify(String&)			
Course	maintain()	Course	print()	
Course	print()			

(그림 18) 멤버 함수에 [illegible] 참조 [illegible] 테이블
(Fig. 18) Reference [illegible] table of member function and member function

Class1	Method1	Class2	MemberData
Collection	add(Example*)	Collection	int filed
Collection	add(Example*)	Collection	int size
Collection	add(Example*)	Collection	static int GrowIncrement
Collection	add(Example*)	Example	public
Collection	search(String&)	Example	public
Collection	search(String&)	Collection	int filed

(그림 19) 멤버 함수와 멤버 데이터 참조 테이블
(Fig. 19) Reference Relationship table of member func -tion and member data

(그림 20)는 클래스 정의 테이블이다. 클래스 정의 테이블은 각 클래스에 정의된 데이터와 멤버함수를 클래스명, 멤버명, 멤버의 종류, 접근 형태, 변수의 정의, 결과 자료형, 함수 인자형을 구분 정의하여 데이터 정보를 저장하였다.

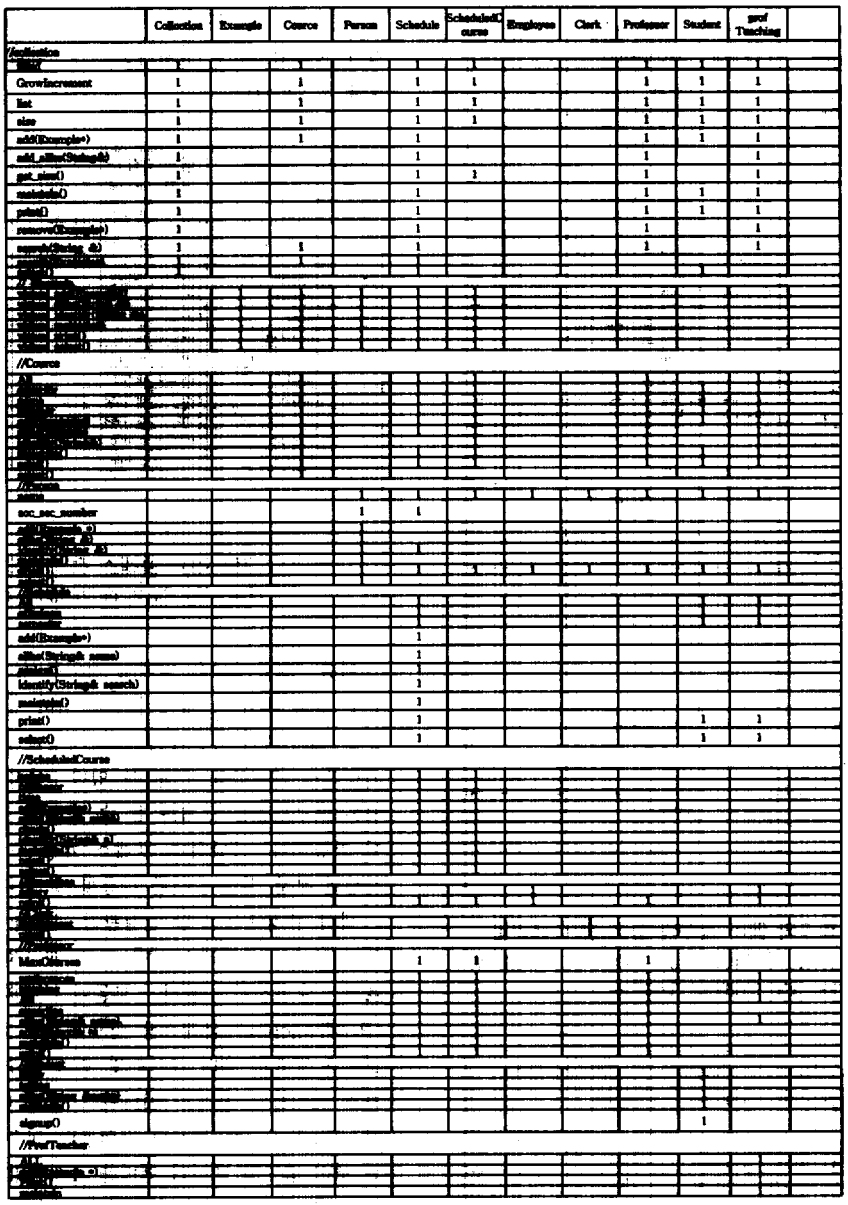
클래스명	멤버명	멤버 종류	접근 상태	변수의 정의	결과 자료형	함수 인자형
Student	taking	data	private	Collection *		
	Body	data	private	static Collection *		
	signup	function	public		void	
	Student	function	public			String
	alike	function	public		Example *	String &
	maintain	function	public		void	
Clerk	department	data	private	String		
	print	function	public		void	
Person	name	data	private	String		
	soc_sec_number	data	private	String		
	Person	function	public			
	Person	function	public			String

(그림 20) 클래스 정의 테이블
(Fig. 20) Class definition table

다음 (그림 21)은 앞에서 정의한 상속 테이블, 멤버 함수 참조 테이블, 멤버 함수내 멤버 데이터 참조 테이블을 조인하여 다음과 같은 유사성을 측정 할 수 있는 메트릭스 테이블을 만들 수 있다. 이러한 메트릭스 구성은 프로토콜을 구체화 할 수 있으며 점진적인 재구성이 가능하다. 왜냐하면 클래스 계층 유지 보수시 이미 작성된 부분 작업 순서를 사용할 수 있다.

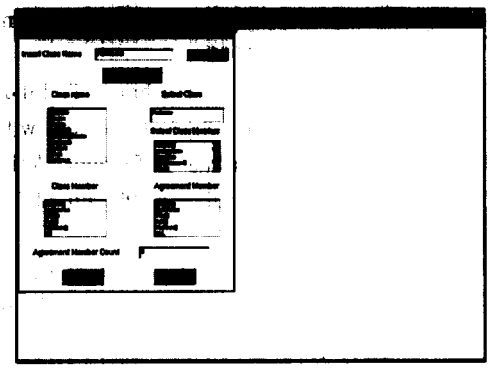
학사 관리에서 수강 신청관리 모델을 나타낸 것이다. 본 논문에서는 클래스 계층 구조 재구성을 위한 적용에 있어서 수강신청 관리 모듈에 강의 전담 교수(ProfTeacher) 클래스를 삽입하고자 한다. 강의 전담 교수가 필요한 멤버 데이터및 멤버 함수를 멤버 데이터 참조 테이블과 클래스 메트릭스 상속 테이블을 고려하여 (그림 21)에 표시된 강의전담 교수 클래스에 필요한 [illegible] 데이터 멤버를 표시하였다.

(그림 22)은 제시용 설계자는 우선 연관된 모듈을 [illegible] 모듈명중 수강 신청 관리 관리를 클릭 하고 Insert Class Name에 삽입할 클래스 이름과, Class Member에 삽입할 클래스 멤버 데이터와 멤버 함수를 삽입후 SelectClass에 선택 클래스 이름을 삽입하면 Agreement Member Count에 유사성 멤버의 개수를 체크하여 유사 경로를 분석한다. 여기에서 삽입할 클래스는 Profteacher이고 유사 클래스는 {Example. Person, Employ, Professor} 그리고 위 경로의 클래

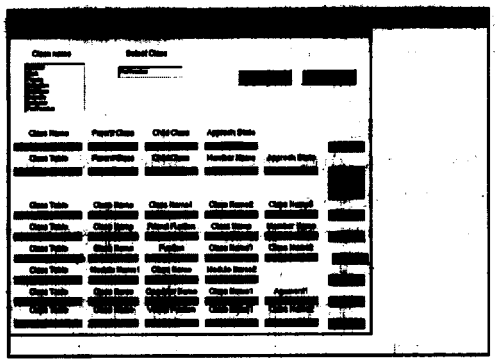


(그림 21) 클래스 메트릭스 테이블
(Fig. 21) Class matrix table

스의 데이타 멤버 및 멤버 함수를 세분화하여 Professor에 유사성 임시 클래스를 만든다. 유사 관계성은 멤버 데이타와 멤버 함수는 부분 집합에 연관되므로 RelationCode 254에 해당되므로 동일한 멤버 데이타와 동일한 멤버 함수 이름이 같으나 기능이 다른 함수를 가상 함수로 하여 상위의 새로운 삽입 클래스를 만든다. 그리고 다음 인접 클래스인 Employ 클래스에 데이타 멤버 및 멤버 함수를 세분화하여 Employ에 유사성 임시 클래스를 만든다. 새로 삽입된 Teacher와의 관계가 데이터 멤버와 멤버 함수가 포함관계이므



(그림 22) 유사성 검사를 위한 폼 윈도우
(Fig. 22) Form window for similarity measure



(그림 23) 클래스 삽입, 삭제를 위한 폼 윈도우
(Fig. 23) Form window for class insert and delete

로 공통인 멤버 데이터와 멤버 함수를 제외한 나머지는 인접 클래스인 Employ의 하위 클래스가 된다. 이것은 RelationCode 243이 해당된다. (그림 23)는 클래스 삽입, 삭제를 위한 폼 윈도우로 연관된 테이블을 화면에 나타나면 삽입 클래스명을 확인한 후 변화된 테이블을 확인후 프로그램 코드를 수정하면 된다. 클래스를 삭제하는 경우 모듈명을 삽입하면 모듈명에 연관된 클래스 이름이 출력되면 삭제될 클래스를 선택 한후 그 연관된 클래스 관계 테이블이 삭제 윈도우에 표현된다. 그 클래스와 연관된 관계형 테이블이 화면에 디스플레이 된다. 중간노드 삭제는 상속 관계 테이블과 참조관계 테이블을 조인한 후 참조관계가 있는 멤버 함수나 데이타 멤버를 세분화하여 수정한다. 그리고 클래스의 멤버 함수 참조테이블, 클래스 멤버 함수내 데이터 멤버 참조 테이블, 클래스 정의 테이블, 클래스 상속 테이블을 검사하여 연관된 클래스 수정후 트리거 삭제하면 된다.

4.2 평가

본 논문에서는 프로그램을 분석한 정보에 의해 유사성 기반으로 하는 하여 자동으로 새로운 클래스 등록 및 클래스 계층 구조를 표현 할 수 있다. 그리고 코드 재사용을 기반이므로 구현 인수를 고려하므로 재구성 단계 이르러서 계층의 정당성,모델에 대한 이해력을 높일 수 있다. 논문에서 제시한 클래스 메트릭스를 생성할 수 있고 이것은 결국 갈리오 격자로 클래스 인터페이스 집합으로 표시 할 수 있다. 일반적으로 갈리오 격자는 프로토콜 일치를 구체화 할 수 있고, 점진 적인 갱신의 장점을 가지고 있다. 본 논문에서 제시하는 클러스터링 방법은 정형화된 모듈 단위의 계층 구조로 표현되며 유사성을 기반으로 클래스 계층 구조를 재구성 할 수 있고 확장성이 좋다.

〈표 3〉 기존 클러스터링과의 비교
〈Table 3〉 comparison of existing clustering

	계층 표현	확장성	객체 지향 지원여부	자동화 여부	적용 시스템
비구문	표현 못함	좋다	없다	가능	GURU
퍼셋	표현 못함	좋다	없다	가능	DIAZ
시멘틱 넷	표현한다.	나쁘다	없다	어려움	AIRS
실전 분류	표현한다.	나쁘다	없다	가능	
상속에 따른 방법	표현한다.	좋다	있다	가능	CARS
본 논문	모듈단위의 계층구조를 표현	좋다	있다	가능	

6. 결 론

객체 지향 개발 방법론은 소프트웨어 생산성과 품질에서 많은 장점을 가진다. 그러나 초기의 개발비용이 많이 소요되며, 클래스 라이브러리 부품 설계 방법이 미흡하다. 또한 이들의 정보를 검색하는데 있어 방대한양을 사람의 기억력에 의존 할 수 없으므로 하나의 자동화된 방법이 필요하다. 또, 개발자가 모든 응용 분야를 경험에 의해본다 는 것은 어렵기 때문에 클래스 계층 구조 재구성시 많은 어려움이 발생한다. 따라서 본 논문에서는 프로그램을 기반으로 클래스의 각 정보를 모듈별로 클래스 관계성을 분해하여 데이터 베이스에 저장한다. 기본적인 클래스 삽입과 삭제로 인한 클래스 재구성시 계층의 작성과 유지는 전형적인 방법을 사용하고 있으나 프로그램을 기반으로 재구

성하므로 다른 개념의 클러스터링과 비교했을 때 다음과 같은 장점을 가지고 있다.

첫째, 프로그램을 기반으로 분석된 계층 구조이므로 기존의 일반화, 세분화 개념을 사용 할 수 있다.

둘째, 기존의 시멘틱 개념인 오버라이딩 및 오버로딩 쉽게 구분되므로 재구성시 도움을 줄 수 있다.

셋째, 이미 만들어진 클래스 계층 구조의 잘못된 문제점을 발견하기가 용이하다.

본 논문에서 제안된 방법은 모듈에서의 클래스 관계성을 분해하고 추출하여 데이터베이스에 저장하였다. 검색 기법은 키워드 매칭법을 사용하였다. 그리고 모듈 안의 클래스 삽입으로 인한 재구성은 데이타 멤버와 멤버함수의 유사성을 측정하여 그 삽입 위치를 결정하고, 그 유사성에 따른 클래스 계층 분류에 의하여 데이타 멤버와 멤버 함수를 일반화 세분화함과 동시에 클래스 재구성을 효율적으로 할 수 있도록 하였다. 또한 클래스 삭제는 클래스 참조 관계를 고려하였으며 삭제의 위치 및 삭제의 의도에 따라 할 수 있는 알고리즘을 제시했다. 그러므로 본 논문에서 제시하는 방법은 기존의 방법 보다 확장성 및 계층 구조 표현이 용이하다. 그러나 클래스 계층이 복잡한 구조일 때 재구성시 모든 경우의 수를 개발자가 분석하는데 많은 시간이 필요로 하는 단점을 가지고 있다. 앞으로의 연구과제는 분산 객체 지향 환경 하에서의 클래스간의 일관성 문제와 재구성에 관한 정형적인 명세서를 모형화 할 수 있는 표준 설계안이 필요하다.

참 고 문 헌

[1] Amnos Tversky, "Feature of Similarity", Psychological Review, Vol. 84, No.4 pp.327-352 Jul, 1997.

[2] D.C.Halbert and P.D.O'Bien, "Using Types and Inheritance in Object-oriented programming", IEEE Software, pp 71-79, 1987.

[3] Edurado Casais, " An Incremental Class Reorganization Approach",ECOOP, 92, pp 114-132, 1992.

[4] Edurado Casais, "Managing Class Evolution in Object-Oriented Systems", Technical rep -ort ,Uninversity of Geneva, 1990.

[5] G.Booch, "Object-Oriented Design in applic -ations", The Benjamin Publishing Company ins. , pp. 132-152, 1991.

[6] H. Dicky, C. Dony , M. Huchard, T. Libou -rel, "On Automatic Class Insertion with Overloading" Special issue of Sigplan Notice Proceeding of ACM OOPSLA '96, pp. 251-2 67, 1996.

[7] Ivan Moore, "Automatic Inheritance hierarchy Restructuring and Method Refactoring" Spe -cial issue of Sigplan Notice Proceeding of ACM OOPSLA '96, pp. 235-250, 1996.

[8] K. J. Lieberherr and A.J. Riel ,"Contribution to teaching Object-Oriented Design and Programming", Special issue of Sigplan No -tice Proceeding of ACM OOPSLA '89, pp.1 1-22, 1989.

[9] Lehman M.M , "Programs, life-cycles and the laws of software evolution", Proc IEEE 15(3), pp. 225-252, 1980.

[10] T.C. Jones, "Reusability in Programming: A Survey of State Of the Art ", IEEE Trans. on Software Engineering, Vol. Se-10, No.5, pp. 40-60, Sep, 1984.

[11] R.Prieto-Diaz, "Implementing Faceted Classification For Software Reuse" comm ACM. v ol.34, N0. 5 pp. 89-97, May,1991.

[12] William B. Frakes and Ricardo Baeza-Yates, "Information Retrieval", Prentice-Hall, pp. 419-422, 1992.

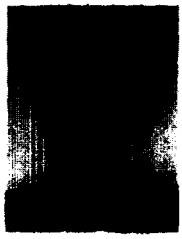
[13] Yie-Farm Chen, Michael Y. Nishimoto, C.V. Ramamoorthy, "The C information abstract system", IEEE trans. on Software Eng, Mar. 1990.

[14] Y. Wand, R. Weber,"An Ontological Model of an Information System " IEEE Trans. on se., Vol. 16, No.11, pp. 1282-1292, 1990.

[15] 김갑수, 신영길 , "소프트웨어 재사용을 위한 C++ 클래스 계층 구조 변경 방법", 한국 정보과학회 논문지 제22권 제1호, pp. 88-93, 1995.

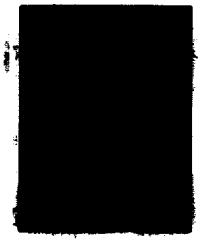
[16] 김상욱, 신영길 ,이정태, "공유 객체를 지원하는 객체 중심 시각 프로그램 환경 개발", 한국 정보과학회 논문지(C) 제2권 2호, pp. 130-141, 1996.

[17] 진영택, "객체 지향 시스템에서의 클래스 라이브러리 계층의 재구성에 관한 연구", 중앙대학교 박사 학위 논문 pp. 2-12, 1991.
[18] 천상호, 양해술, "객체 지향 언어 C++프로그램을 위한 정보 분석기", 한국 정보과학회 가을 학술발표 논문집 Vol. 21, No. 2, pp. 685-688, 1994.
[19] 홍은지, "객체지향 데이터베이스에서 스키마 일관성을 유지하는 스키마 수정",서울대학교 박사학위 논문 pp.141-165, 1996



정 계 동

1985년 광운대학교 전자계산학과 졸업(이학사)
1992년 광운대학교 산업대학원 전자계산학과(이학석사)
1992~현재 광운대학교 전자계산학과 박사과정
관심분야 : 객체 지향 프로그래밍언어 및 데이터베이스, 병렬처리 프로그래밍언어, 객체지향 분산 컴퓨팅



최 영 근

1980년 서울대학교 수학교육과 (학사)
1982년 서울대학교 계산통계학과 (이학석사)
1989년 서울대학교 계산통계학과 (이학박사)
1992~현재 광운대학교 전자계산학과 교수
1992~현재 광운대학교 전자계산소 소장
관심분야 : 프로그래밍 언어, 병렬 프로그래밍언어,객체지향 프로그래밍언어 및 설계, 객체지향 분산 컴퓨팅