객체지향 설계 행위를 보존하는 메트릭 기반 재구조화 기법

(A Metric based Restructuring Technique Preserving the Behavior of Object-Oriented Designs)

이 병 정 †

(Byung-Jeong Lee)

요 약 설계 재구조화는 설계 구성 요소들을 재조직함으로써 품질을 향상시켜 소프트웨어 유지보수 비용을 줄인다. 객체지향 메트릭은 설계 결함을 발견하고 설계 구성 요소들을 재조직하기 위한 변형을 찾는데 도움을 제공한다. 기본적으로 이러한 설계 변형은 초기 시스템의 행위를 보존해야 한다. 본 논문에서는 객체지향 설계 행위를 보존하는 메트릭 기반 재구조화 기법을 집합론에 기반하여 기술하고, 자바로 작성된 응용 프로그램에 적용하여 유효성을 확인한다. 그리고 재구조화의 효과성을 확인하기 위하여 시뮬레이티드 어닐링(simulated annealing) 알고리즘을 사용한 방법과 비교한다.

키워드 : 설계 재구조화, 메트릭, 행위 보존, 유전 알고리즘

Abstract Design restructuring improves software quality by reorganizing design elements and reduces maintenance cost. Object-oriented metrics can help to detect design flaws and find transformations to reorganize design elements. Basically, the transformations must preserve the behavior of an initial system. This paper describes a metric based restructuring technique preserving the behavior of object-oriented designs, founded on set theory, and gives its validity by applying the technique to applications written in Java. This paper also compares the technique with a technique using simulated annealing algorithm to show its effectiveness.

Key words : Design restructuring, Metric, Behavior preservation, Genetic algorithm

1. 서 론

소프트웨어 재구조화(software restructuring)[1]는 오랜 기간 계속 변경하여 훼손된 소프트웨어 구조를 복원함으로써 유지보수 비용을 줄이는 수단을 제공한다. 그 중 설계 재구조화는 설계 구성 요소들을 재조직함으로써 설계 품질을 향상시킨다. 좋은 설계는 설계의 각 구성 요소가 긴밀한 논리적 관계를 갖도록 응집되어야 하고 또한 구성 요소들 사이에 단단한 결합이 아닌 느슨한 결합을 가져야 한다[2]. 결합도가 낮을수록 변경의 파급 효과가 국소적이므로 설계를 수정하기 쉽다. 또한 규모가 크고 복잡한 시스템의 잦은 수정은 설계를 훼손할 뿐만 아니라 시스템 동작을 변경할 수 있다.

· 이 논문은 2002년도 서울시립대학교 학술연구조성비에 의하여 연구되었음.

† 종신회원 : 서울시립대학교 컴퓨터과학부 교수
bjlee@venus.uos.ac.kr

논문접수 : 2002년 8월 30일
심사완료 : 2003년 6월 25일

소프트웨어 메트릭을 사용한 정량적인 방법은 잠재해 있는 설계 결점을 발견하고 결점을 정정하기 위한 변형을 찾는데 개발자에게 도움을 제공한다[3]. 또한 이러한 설계 변형은 초기 설계 의도를 훼손하지 않도록 설계 행위를 보존해야 한다. 그리고 최근의 소프트웨어들은 규모가 방대하고 복잡하여 설계를 갱신하려면 너무 많은 시간과 노력이 요구되므로 자동화를 지원해야 한다. 그러나 이전 연구들은 메트릭을 사용하여 자동화를 지원하는 시스템 변경과 변경에서의 시스템 행위 보존을 거의 독립적으로 다루어 왔다. 본 논문의 객체지향 설계 재구조화는 소프트웨어 메트릭을 사용하여 정량적으로 설계 품질을 측정하고 그 측정을 기반으로 설계 변경의 자동화를 지원한다. 또한 객체지향 설계 행위를 보존함으로써 초기 설계 의도를 훼손하지 않는다. 그러므로 본 기법은 설계 도구를 사용하여 새로운 소프트웨어를 설계할 때 또는 기존 설계를 재공학할 때 설계 품질을 효과적으로 향상시켜 유지보수 비용을 줄이는 수단을 제공한다.

이전 연구에서는 자동 재구조화를 지원하기 위한 유전 알고리즘(genetic algorithm) 적용을 소개하였고[4], 객체지향 설계 메트릭과 재구조화 과정을 소개하였다[5]. 본 논문에서는 객체지향 설계 행위를 보존하면서 설계를 자동적으로 재구조화하는 기법을 집합론(set theory)에 기반하여 기술한다. 그리고 자바 응용 프로그램에 적용하여 실험적으로 유효성을 확인하고, 효과성을 보이기 위하여 시뮬레이티드 어닐링(simulated annealing) 알고리즘을 사용한 방법과 비교한다.

본 논문의 구성은 다음과 같다. 2장에서 기존 메트릭 기반 재구조화 연구와 행위 보존 연구를 기술한다. 3장에서 객체지향 시스템을 위한 설계 모델과 클래스와 객체지향 설계 응집도와 결합도 메트릭을 정의한다. 4장에서 재구조화 동작 집합을 정의하고 각 동작들이 유효한 재구조화 과정을 수행하는 것을 보인다. 그리고 유전 알고리즘을 이용한 자동 재구조화 과정을 간략히 기술한다. 5장에서 자바 응용 프로그램들에 적용한 실험 결과를 보이고, 6장에서 결론과 향후 연구 과제를 기술한다.

2. 메트릭 기반 재구조화와 행위 보존 변형

시스템의 코드 구조가 훼손되어 모듈화가 되어 있지 않은 경우에 프로그램 코드로부터 모듈과 그들의 관계를 추출하여 시스템 모듈 구조를 자동적으로 복원하기 위한 방법이 제안되었다[6]. 이 방법은 모듈화 과정을 최적화 문제로 간주하여 유전 알고리즘과 언덕 오르기 알고리즘(hill-climbing algorithm)을 사용한다. 모듈 사이의 관계를 모듈 의존성 그래프(module dependency graph)로 나타내고 서브시스템들로 분할한다. 그러나 이 방법은 모듈 사이에 교환되거나 공유되는 개체 개수를 의미하는 모듈 연결 강도를 고려하지 않았다. 그러므로 모듈 별로 연결 강도 크기 차이가 큰 시스템을 모듈화하는 경우에는 모듈화 결과가 바람직하지 않을 수 있다. 절차적 소프트웨어 재구조화를 위한 기준으로서 설계 응집도와 결합도가 제안되었다[7]. 재구조화는 시스템 요소들의 응집도를 올리고 결합도를 줄이도록 모듈의 분할과 합성을 통하여 수행된다. [7]의 재구조화는 어느 분할과 합성을 수행할 지를 개발자가 결정해야 하고, 재구조화된 시스템이 초기 시스템의 행위를 보존한다는 것을 보여주지 못한다. 그리고 앞에서 기술한 연구는 객체지향 개념을 지원하지 않는다[6,7]. 정량적으로 객체지향 시스템을 모듈화하기 위하여 통계적 방법인 클러스터 분석(cluster analysis)을 사용하였다[8]. 이 연구에서는 클러스터 분석을 위하여 클래스 사이 관계에 따라 각기 다른 가중치를 클래스 결합도로 할당한다. 예를 들면 클래스 사이 직접 상속은 3, 클래스 속성형은 2, 그리고 동작의 매개 변수 또는 변환형 등의 관계는 가중치 1을 가진다. 그리고 각 모듈이 하나의 클래스로 시작하는 계층적이고 집적적인 클러스터링(hierarchical agglomerative clustering) 방법을 사용하며 클러스터 개수가 원래 시스템 모듈 개수와 같아지면 정지한다. 그러므로 이 연구는 전체 모듈 개수는 최적화하지 않는다. 또한 실험적으로 검증하지 않고 클래스 사이 결합 종류에 따라 가중치를 할당하는 것은 모듈화된 시스템의 품질을 떨어뜨릴 수 있다. [9]에서 이전 연구[8]의 문제점을 인식하고 모듈 내부의 연결 정도를 나타내는 모듈화 장점 인자(MMF: Modularization Merit Factor)와 모듈 크기의 분산 정도를 나타내는 상대적 모듈 분산(RMD: Relative Module Dispersion) 메트릭을 정의하여 객체지향 시스템을 정량적으로 모듈화하였다. 모듈 개수가 각각 다른 여러 해들 중에서 모듈화 장점 인자를 최대화하는 해를 최종 해로 선택하므로 모듈 개수를 최적화한다. 그러나 이 연구는 클래스를 모듈 또는 서브시스템으로 모듈화하는 방법을 제안하였으나 클래스 멤버들의 모듈화는 지원하지 않는다[8,9]. 또한 재구조화된 시스템이 시스템 또는 설계 행위를 보존한다는 것을 보여주지 않는다.

행위 또는 구조를 보존하는 객체지향 재구조화는 스키마 진화(schema evolution)[10], 클래스 변형(class transformation)[11], 그리고 설계 변형(design transformation)[12] 등을 포함한다. 객체지향 스키마 진화는 스키마 불변(invariants) 성질 집합을 유도하고 각 스키마 변경에 대하여 불변 성질이 보존될 수 있는 규칙(rules) 집합을 정의하였다. 불변 성질과 규칙 집합을 적용하여 스키마 변경에 일관성을 유도하였다. 그러나 [10]의 프레임워크는 객체지향 소프트웨어 행위보다 자료 모델에 초점이 있다. 객체지향 데이타베이스 설계에서 데이타베이스를 다시 재구성하지 않기 위하여 객체-보존 클래스 변형이 제안되었다[11]. 실체(instantiable) 클래스와 추상(abstract) 클래스가 노드이고 상속(inheritance)과 부분(part-of) 관계가 에지인 클래스 사전 그래프(class dictionary graph) 모델을 정의하고 이 모델에 기반한 기본 변형 집합이 사용된다. 그러나 보편적인 객체지향 개념과는 달리 클래스 사전 그래프에서 실체 클래스는 세분화(specialization) 관계를 갖지 못하는 제약이 있다. 설계 패턴(design pattern)은 객체지향 설계를 변경하기 위하여 전문가의 지식을 활용한다. 따라서 설계 패턴을 활용하여 유지보수하면 품질이 우수한 설계를 얻을 수 있으므로 시스템 설계를 패턴화하기 위하여 초기 설계를 변형하는 리팩토링을 사용한다[12]. 리팩토링 자체가 행위를 보존하기 때문에 리팩토링으로 패턴화된 설계도 행위를 보존한다. 이 연구들은 객체지향 소프트웨어/설계 행위를 보존하지만 정량적인 기반이

없고 어디를 어떻게 변경할 지를 개발자가 모두 결정해야 한다. 상속 계층 구조를 재공학하는 방법은 계층 구조를 추론하는 방법[13]과 개념 분석(concept analysis)을 사용하는 방법[14]이 있다. 추론 방법에서 상속 계층은 잎(leaf) 노드인 객체와 그의 상위에 존재하는 클래스, 그리고 그들 사이의 링크로 구성되고 모든 노드는 시스템에 존재하는 메소드와 속성에 해당하는 특성(feature)을 갖는다. 추론 방법은 이러한 상속 계층을 자동적으로 추론하여 재구성한다. 그리고 재구성된 상속 계층은 객체들의 원래 구조를 보존하기 위하여 규약 집합을 만족한다. 추론 방법은 구현과 이해가 용이하다는 장점이 있으나 동적 언어를 대상으로 하고 있어 정적 타입 언어에는 제약이 있다. 또한 추론 방법과 개념 분석 방법은 정량적인 수단을 사용하지 않으므로 상속 계층의 품질을 정밀하게 측정하지 못하는 제약이 있다.

3. 객체지향 모델과 메트릭

3.1 객체지향 모델

객체지향 모델, 메트릭, 그리고 재구조화 동작을 일관성있고 명확하게 기술하기 위하여 객체지향 설계, 클래스, 메소드, 속성, 그리고 그들 사이의 관계를 [15]의 집합론에 기반한 형식주의(formalism)로부터 유도한다.

[정의 1] 객체지향 설계, 클래스, 메소드, 그리고 속성

- 객체지향 설계 D는 클래스 집합 C로 구성된다.
- $M(C)$는 객체지향 설계 D에 있는 모든 메소드 집합 그리고 $A(C)$는 객체지향 설계 D에 있는 모든 속성 집합이다.
- 각 $c \in C$에 대하여 $A(c)$는 클래스 c의 속성 집합 그리고 $M(c)$는 클래스 c의 메소드 집합이다.

클래스의 메소드 사이에는 호출 관계(call relationship), 그리고 메소드와 속성 사이에는 사용 관계(use relationship)가 있다. 호출 관계는 메소드 사이에 메시지를 주고받는 관계이고, 사용 관계는 메소드가 속성 자료를 읽거나 쓰는 관계이다.

[정의 2] 호출 메소드 집합(call(m)과 called(m))

$c \in C$ 이고 $m \in M(c)$ 일 때, 호출 메소드 집합을 다음과 같이 정의한다.

- $call(m) = \{m' | \exists d \in C : (m' \in M(d)) \wedge (m \ calls \ m')\}$
- $called(m) = \{m' | \exists d \in C : (m' \in M(d)) \wedge (m' \ calls \ m)\}$
- 각 $m \in M(C)$에 대하여 $para(m)$는 메소드 m의 입력 인자 집합을 나타낸다.

[정의 3] 사용된 속성 집합(use(m))과 사용하는 메소드 집합(used(a))

$c \in C$, $m \in M(c)$, 그리고 $a \in A(c)$일 때, 사용 관계에 기반한 속성 집합과 메소드 집합을 다음과 같이 정의한다.

- $use(m) = \{a' | \exists d \in C : (a' \in A(d)) \wedge (m \ uses \ a')\}$
- $used(a) = \{m' | \exists d \in C : (m' \in M(d)) \wedge (m' \ uses \ a)\}$

집합 C에 있는 클래스 사이의 관계는 상속 관계와 상호작용 관계로 구분한다. 상속 관계는 부모 클래스의 메소드와 속성을 상속받는 관계이고, 상호작용 관계는 상속 관계에 있지 않은 클래스 사이에 메시지를 보내거나 속성을 사용하는 관계이다.

[정의 4] 상속 관계 클래스 집합

$c \in C$ 일 때 상속 관계에 있는 클래스 집합을 다음과 같이 정의한다.

- $Parents(c) = \{d \in C | c \ inherits \ attributes \ and \ methods \ from \ d\}$
- $p(a, b)$가 $b \in Parents(a)$이면 참이고 그렇지 않으면 거짓일 때, $Ancestors(c) = \{d | \exists c_1, c_2, ..., c_n \in C (n \geq 2) : p(c_1, c_2) \wedge p(c_2, c_3) ... \wedge p(c_{n-1}, c_n) \wedge (c_1 = c) \wedge (c_n = d)\}$
- $Children(c) = \{d \in C | d \ inherits \ attributes \ and \ methods \ from \ c\}$
- $c(a, b)$가 $b \in Children(a)$이면 참이고 그렇지 않으면 거짓일 때, $Descendants(c) = \{d | \exists c_1, c_2, ..., c_n \in C (n \geq 2) : c(c_1, c_2) \wedge c(c_2, c_3) ..., \wedge c(c_{n-1}, c_n) \wedge (c_1 = c) \wedge (c_n = d)\}$

[정의 5] 상호작용 클래스 집합

$c \in C$ 일 때 상호 작용 클래스 집합을 다음과 같이 정의한다.

- $InteractFrom(c) = \{d \in C | (c \neq d) \wedge (d \notin Ancestors(c) \cup Descendants(c)) \wedge (m \in M(c) \wedge ((m' \in M(d) \wedge m' \in call(m)) \vee (a \in A(d) \wedge a \in use(m))))\}$
- $InteractTo(c) = \{d \in C | (c \neq d) \wedge (d \notin Ancestors(c) \cup Descendants(c)) \wedge ((m \in M(c) \wedge (m' \in M(d) \wedge m' \in called(m))) \vee (a \in A(c) \wedge (m' \in M(d) \wedge m' \in used(a))))\}$
- $Interact(c) = InteractFrom(c) \cup InteractTo(c)$

본 연구에서는 객체지향 설계를 CUG(Call-Use Graph)와 CAG(Class Association Graph)로 나타낸다. CUG에서는 객체지향 설계에 나타난 모든 클래스의 메소드, 속성, 그리고 그들 사이의 관계를 나타내고, CAG는 클래스들 사이의 관계를 보여준다. 그래서 CUG와 CAG는 서로 보완적인 그래프 모델이다.

[정의 6] 호출-사용 그래프(Call-Use Graph : CUG)

객체지향 설계 D의 CUG는 방향성있는 그래프 $CUG_D = (V_U, E_U)$로 표현되며, $V_U = \cup_{c \in C} M(c) \cup A(c)$는 객체지향 설계 D의 모든 속성과 메소드들의 집합이다. 그리고 $E_U = \{(x, y) \in V_U \times V_U | \ y \in call(x) \vee y \in use(x)\}$의 관계를 만족하는 가중치 W를 가진 에지(weighted edge)들의 집합이다. 메소드 사이 호출 관계를 나타내는 에지 집합을 E_c라 표시하면, 이 집합에 속하는 에지의 가중치는 정보 전달 양이다.

메소드와 속성 사이 사용 관계를 나타내는 에지 집합을 E_u라 표시하면, 이 집합에 속하는 에지의 가중치는 1이다. 그러므로 $E_U = E_c \cup E_u$ 이고, CUG 에지 가중치는 다음과 같다. 집합 E_c의 에지 가중치 계산에서 메소드 y의 입력인자 집합 $para(y)$은 메시지를 받는 객체 자체는 포함하지 않으므로 그 객체를 계산에 포함하기 위하여 1을 더한다.

$$W(e_c(x,y)) = |para(y)| + 1, \text{ if } e_c(x,y) \in E_c$$

$$W(e_u(x,y)) = 1, \text{ if } e_u(x,y) \in E_u$$

[정의 7] 클래스 연관 그래프(Class Association Graph : CAG)

객체지향 설계 D의 CAG는 방향성있는 그래프 $CAG_D = (V_A, E_A)$로 표현되며, $V_A = C$이다. 그리고 E_A는 가중치를 가진 에지(weighted edge)들의 집합이고, 클래스 사이의 상속(inheritance) 관계를 나타내는 에지 집합을 E_{ih}, 상호작용(interaction) 관계를 나타내는 에지 집합을 E_{ia}라 표시하면 $E_A = E_{ih} \cup E_{ia}$이다. $E_{ih} = \{(u, v) \in V_A \times V_A \mid v \in Parents(u)\}$이고, $E_{ia} = \{(u, v) \in V_A \times V_A \mid v \in InteractFrom(u)\}$이다. 에지 가중치는 CUG 가중치를 클래스 단계에서 합산한다.

$$W(e_{ih}(u,v)) = \sum_{x \in M(u) \cup A(u)} \sum_{y \in M(v) \cup A(v)} W(e_c(x,y)) + W(e_u(x,y)),$$
$$\text{if } W_{e_u}(u,v) \in E_{ih}$$

$$W(e_{ia}(u,v)) = \sum_{x \in M(u) \cup A(u)} \sum_{y \in M(v) \cup A(v)} W(e_c(x,y)) + W(e_u(x,y)),$$
$$\text{if } W_{e_u}(u,v) \in E_{ia}$$

CUG와 CAG의 각 노드는 유일한 이름을 가지며, CUG는 방향성있는 비순환(acyclic) 그래프이고, CAG에서 상호작용 관계는 순환(cyclic)할 수 있지만 상속 관계는 비순환한다. CUG와 CAG로부터 메소드 정보 교환 행렬(CMM)을 만든다.

[정의 8] 메소드 정보 교환 행렬(Communication Matrix between Methods : CMM)

이 행렬은 설계 D의 메소드 개수 $|M(C)|$ 차원 정방행렬이고, CUG 모델 에지 값을 사용하여 정의하므로 대칭 행렬이다. CMM의 각 요소는 메소드 사이의 정보 교환 양, 즉 CUG 모델의 메소드 사이 호출 관계 에지 가중치 $W(e_c(x,y))$와 두 메소드가 공통으로 사용하는 속성 개수 $|share(x,y)|$를 더한 값을 나타낸다. 클래스 생성자(constructor), 소멸자(destructor), 그리고 접근 메소드(access method)는 정보 교환 행렬에 포함시키지 않는다. $id(x)$는 메소드 x의 숫자로 된 식별자를 나타낸다.

$$CMM(i,j) = W(e_c(x,y)) + |share(x,y)|,$$
$$\text{if } i = id(x) \text{ and } j = id(y)$$

$$share(x,y) = \{z \in A(C) \mid z \in use(x) \wedge z \in use(y)\}$$

CMM의 각 행은 한 메소드와 다른 모든 메소드들 사이의 정보 교환을 나타내므로 메소드의 행위 유형(behavioral type)을 나타낸다. 메소드 정보 교환 행렬에 생성자, 소멸자, 그리고 접근자를 포함시키지 않는 이유는 실험적으로 검증되지 않은 방법으로 응집도와 결합도에 영향을 미치기 때문이다[15].

만약 객체지향 설계가 중복 클래스들을 포함하고 있다면 그들은 동일한 행위를 보일 것이다. 예를 들어 클래스 c_i와 c_j가 중복 클래스라면, 그들은 다음과 같은 성질을 갖는다. 먼저 클래스 c_i와 c_j는 대응하는 메소드와 속성들을 가지고 있으며 또한 그들은 공통 클래스들과 상호작용 한다. 다음으로 클래스 c_i에 있는 메소드 쌍의 $W(e_c(x,y))$와 $|share(x,y)|$는 클래스 c_j에 있는 대응하는 메소드 쌍의 $W(e_c(x,y))$와 $|share(x,y)|$와 각각 같다. 마지막으로 클래스 c_i와 공통 클래스 사이의 메소드 쌍의 $W(e_c(x,y))$와 $|share(x,y)|$는 클래스 c_j와 공통 클래스 사이 메소드 쌍의 $W(e_c(x,y))$와 $|share(x,y)|$와 각각 같다. 이와 같이 본 기법은 CMM과 클래스 멤버들을 조사하여 중복 클래스 후보들을 자동적으로 식별할 수 있다. 그러나 이러한 성질을 가진 클래스들이 항상 중복 클래스는 아니기 때문에 최종적으로 개발자로부터 판단을 요구한다. 즉 자동적으로 식별된 위의 성질을 가진 후보 클래스들이 중복 클래스인지를 개발자가 최종적으로 결정하는 것이다. 개발자가 중복 클래스라고 결정하면 그 클래스들은 CUG와 CAG 모델로부터 삭제된다.

CMM 행 벡터는 메소드 행위 유형을 나타내므로 두 벡터의 코사인(cosine) 값은 메소드 행위 유형의 유사도를 나타낸다. 그래서 이 코사인(cosine) 값으로 메소드 유사도(SBM : Similarity Between Methods)를 정의하고, 이 메소드 유사도로부터 클래스 유사도(SBC : Similarity Between Classes)와 상속 계층에 있는 클래스 유사도(SBCH : Similarity Between Classes in a Hierarchy)를 정의한다. 클래스 유사도는 두 클래스에 속한 메소드 쌍의 유사도 합이다.

[정의 9] 메소드 유사도(SBM), 클래스 유사도(SBC), 그리고 상속 계층에 있는 클래스 유사도(SBCH)

$$SBM(m_s, m_t) = \frac{\sum_{k=1}^{|M(C)|} CMM(i,k) * CMM(j,k)}{\sqrt{\sum_{k=1}^{|M(C)|} CMM(i,k)^2} * \sqrt{\sum_{k=1}^{|M(C)|} CMM(j,k)^2}},$$
$$\text{if i = id}(m_s) \text{ and j = id}(m_t)$$

$$SBMH(m_s, m_t) = \begin{cases} SBM(m_s, m_t), & \text{if } CMM(id(m_s), id(m_t)) \neq 0 \\ 0, & otherwise \end{cases}$$

$$SBC(c_i, c_j) = \sum_{m_s \in M(c_i)} \sum_{m_t \in M(c_j)} SBM(m_s, m_t)$$

$$SBCH(c_i, c_j) = \sum_{m_s \in M(c_i)} \sum_{m_t \in M(c_j)} SBMH(m_s, m_t)$$

3.2 응집도와 결합도

지금까지 객체지향 설계를 위한 많은 응집도와 결합도 메트릭이 제안되었고 그들은 각각 나름의 기반을 가

지고 있다. 기존 응집도 메트릭은 공통 속성을 사용하는 메소드 쌍의 개수를 고려하고 메소드 호출은 고려하지 않거나[16], 공통 속성과 메소드 호출을 고려하나 메소드 사이에 교환되는 정보의 양은 고려하지 않는다[17]. 기존 결합도 메트릭은 결합된 클래스 개수를 고려하고 클래스 사이 교환되는 정보 양은 고려하지 않는다[16,18]. 정보-흐름 기반 응집도와 결합도는 메소드 호출과 메소드 사이 교환되는 정보 양을 모두 고려하나 메소드가 공통으로 접근하는 속성은 고려하지 않는다[19]. 본 논문에서는 메소드 행위 유사도에 기반한 객체지향 응집도와 결합도 메트릭을 제안한다. 메소드 행위 유사도의 기준은 앞 절에서 기술한 공통 속성, 메소드 호출, 그리고 메소드 사이 교환되는 정보의 양을 모두 포함한다.

객체지향 패러다임의 핵심 개념인 상속은 응집도와 결합도에 모두 영향을 미친다[2]. 클래스가 조상 클래스로부터 메소드와 속성을 상속받는 경우에 상속받은 멤버를 포함하여 후손 클래스의 응집도를 정의한다면 그 클래스의 응집도는 감소된다. 왜냐하면 조상 클래스에 정의한 멤버들과 후손 클래스에 정의한 멤버들 사이에는 밀접함이 적기 때문이다. 그래서 본 연구에서는 클래스에 정의된 멤버를 기준으로 응집도를 정의한다. 또한 상속은 결합도의 다른 형태를 만든다. 메소드와 속성을 상속받는 후손 클래스는 조상 클래스에서 정의한 멤버들을 접근하므로 자동적으로 조상 클래스와 결합되어 있다. 그래서 상속은 객체지향의 핵심 개념으로서 사용을 권장하는 바람직한 성질이지만 상속 관계를 가진 조상 클래스의 변경은 멤버를 상속받는 모든 클래스들에게 전파되기 때문에 주의하여 사용해야 한다.

클래스 응집도는 클래스내 구성 요소들의 긴밀 정도를 의미하므로, $SBC(c_i, c_i)$를 사용하여 클래스의 메소드 행위 유사도 평균으로 정의한다. 객체지향 설계 응집도는 클래스 응집도 $COH(c)$의 평균으로 정의한다.

[정의 10] 클래스 응집도($COH(c)$)와 객체지향 설계 응집도($COH(D)$)

$$COH(c_i) = \frac{SBC(c_i, c_i)}{|M(c_i)| * |M(c_i)|}$$

$$COH(D) = \frac{\sum_{c_i \in C} COH(c_i)}{|C|}$$

이 식에서 $|C|$은 객체지향 설계 D에 존재하는 클래스 개수이다.

본 연구에서 결합도는 상속 결합도와 상호작용 결합도로 구분한다. 클래스 상속 결합도는 같은 상속 계층에 있는 클래스들 사이의 연결 강도를 의미하므로 적당히 높은 것이 바람직하다. 그러나 상속 계층이 너무 깊으면 유지 보수성에 부정적인 영향을 미친다[20]. 반면에 상호작용 결합도는 상속 관계가 없는 클래스 사이의 연결 강도를 의미하므로 낮을수록 바람직하다. 본 연구에서는 클래스 사이의 연결 강도를 클래스 유사도를 사용하여 측정한다. 그래서 두 클래스에 속한 메소드들의 행위 유형이 유사할 수록 두 클래스 사이의 결합도가 크다고 정의한다.

[정의 11] 클래스 상속 결합도($COP_{IH}(c)$)와 클래스 상호작용 결합도($COP_{IA}(c)$)

$$COP_{IH}(c_i) = \sum_{c_j \in Ancestors(c_i)} SBCH(c_i, c_j)$$

$$COP_{IA}(c_i) = \sum_{c_j \in Interact(c_i)} SBC(c_i, c_j)$$

객체지향 설계 D의 상속 결합도와 상호작용 결합도는 $COP_{IH}(c)$의 합과 $COP_{IA}(c)$의 합으로 각각 정의한다.

[정의 12] 객체지향 설계 상속 결합도($COP_{IH}(D)$)와 상호작용 결합도($COP_{IA}(D)$)

$$COP_{IH}(D) = \sum_{c_i \in C} COP_{IH}(c_i)$$

$$COP_{IA}(D) = \sum_{c_i \in C} COP_{IA}(c_i)$$

Briand 등은 수학적 개념을 이용하여 정밀하고 특정 소프트웨어에 국한되지 않는 메트릭 프레임워크를 제안하였다[21]. 이 프레임워크는 응집도(cohesion) 그리고 결합도(coupling) 등의 메트릭에 대한 중요한 성질(property)을 정의하여 새로운 소프트웨어 메트릭을 정의할 때 만족해야 할 기준을 제공한다. 본 연구에서 제안한 응집도 메트릭 $COH(D)$와 결합도 메트릭 $COP_{IH}(D)$와 $COP_{IA}(D)$은 [21]에서 기술한 응집도와 결합도 성질을 만족한다[5]. 정의한 메트릭에 대한 보다 직관적인 이해를 위하여 간단한 예를 사용한 설명이 [5]에 기술되어 있다.

4. 재구조화 동작과 자동화 지원

재구조화를 위하여 먼저 객체지향 모델링 도구 또는 객체지향 언어로 작성된 코드에서 정보를 추출하여 CUG와 CAG를 만들고 CMM을 계산한다(그림 1).

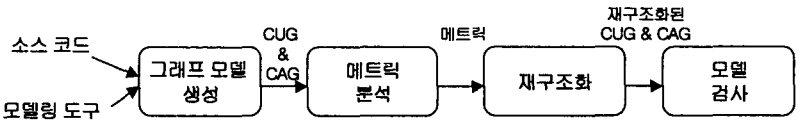

그림 1 객체지향 설계 재구조화 과정

*CMM*으로부터 객체지향 설계 메트릭을 유도하고, 이 메트릭을 기반으로 유전 알고리즘을 적용한다. 유전 알고리즘 수행시 유도한 설계 메트릭을 향상시키도록 기본 재구조화 동작을 사용하여 클래스를 새로 생성, 삭제 또는 변경함으로써 설계를 재구조화한다. 그리고 재구조화된 설계의 유효성을 확인하기 위하여 설계 모델을 검사한다. 본 장에서는 기본 재구조화 동작과 유전 알고리즘 적용에 대해서 기술한다.

4.1 재구조화 동작

설계 모델 *CUG*와 *CAG*에 대한 8 개의 기본 재구조화 동작(operations)을 정의한다. 정의한 동작은 클래스와 멤버 이름 변경, 클래스 에지 가중치 변경, 클래스 에지 추가/삭제, 메소드 또는 속성의 이동, 그리고 클래스 노드 추가/삭제 등이다.

동작 1 클래스 멤버 이름 변경 : 다른 클래스의 멤버와 이름이 중복되지 않도록 변경한다.

```
/* CAG_D = (V_A, E_A) where V_A = C and E_A = E_ih ∪ E_ia */
/* c is a node in V_A and m is a member of class c */
CAG::OP1(c, m) {
if ( (c ∈ V_A)∧(m ∈M(c)∪A(c)) )
  foreach c' ∈ V_A do
    foreach m' ∈ M(c')∪A(c') do
      if ( (m ≠ m')∧(m.name == m'.name) )
        m.name = new_name();
}
```

동작 2 클래스 노드 이름 변경 : *CAG*에서 다른 노드와 이름이 중복되지 않도록 변경한다.

```
CAG::OP2(c, name) { /* c is a node in V_A */
if ( c ∈ V_A ) {
  if ( name ≠ null )
    c.name = name;
  else
    foreach c' ∈ V_A do
      if ( (c ≠ c')∧(c.name == c'.name) )
        c.name = new_name();
}
}
```

동작 3 클래스 에지 가중치 변경 : 클래스에 속한 메소드 또는 속성들 사이의 호출 에지와 사용 에지를 더하여 *CAG*의 클래스 사이 상호작용 에지와 상속 에지 가중치를 갱신한다. 만약 두 클래스 모든 멤버 사이에 가중치 합이 0이면 두 클래스 사이의 에지가 삭제된다(동작 5).

```
CAG::OP3(c1, c2) {  /* c1 and c2 are nodes in V_A */
  sum = 0;
  foreach m1 ∈ M(c1) ∪ A(c1) do
    foreach m2 ∈ M(c2) ∪ A(c2) do
      sum = sum + W(e_c(m1, m2)) + W(e_u(m1, m2));
if ( sum > 0 )  // 멤버들 사이에 교류가 존재
    if ( c2 ∈ Parents(c1) ) W(e_ih(c1,c2)) = sum;
    else if ( c2 ∈ InteractFrom(c1) ) W(e_ia(c1,c2)) = sum;
else  // 멤버들 사이에 교류가 없음
    if ( c2 ∈ Parents(c1) ) OP5(c1, c2, R_IH);
    else if ( c2 ∈ InteractFrom(c1) ) OP5(c1, c2, R_IA);
}
```

동작 4 클래스 에지 추가 : 상속 에지가 추가되면 자식 클래스는 부모 클래스의 메소드와 속성을 상속받는다. 만약 상속 에지가 순환하면 그 추가 요구는 취소된다. 상속 관계에 있지 않은 두 클래스의 멤버 사이에 사용 에지 또는 호출 에지가 존재해야 그 클래스들 사이에 상호작용 에지가 추가될 수 있다. 추가된 에지 가중치는 동작 3에 의해 계산된다.

```
/* c1 and c2 are nodes in V_A and rel indicates a relationship
between c1 and c2 */
CAG::OP4(c1, c2, rel) {
  if ( rel == R_IH ) // 상속 관계
    if ( (no sequence of edges e_ih(x1,x2),e_ih(x2,x3), ...,e_ih
      (x_{n-1},x_n)∧(x1=c1)∧(x_n=c2))∧(e_ih(c1,c2)∉E_ih))
        E_ih = E_ih ∪ {e_ih(c1,c2)};
  else if ( rel == R_IA ) // 상호작용 관계
    if ( e_ia(c1,c2) ∉ E_ia )
      E_ia = E_ia ∪ {e_ia(c1,c2)};
  OP3(c1,c2);
}
```

동작 5 클래스 사이 에지 삭제 : 상속 에지가 삭제되면 자식 클래스는 더 이상 부모 클래스의 멤버들을 상속받지 않는다.

```
/* c1 and c2 are nodes in V_A and rel indicates a relationship
between c1 and c2 */
CAG::OP5(c1, c2, rel) {
  if ( rel == R_IH ) // 상속 관계
    E_ih = E_ih - {e_ih(c1,c2)};
  else if ( rel == R_IA ) // 상호작용 관계
    E_ia = E_ia - {e_ia(c1,c2)};
}
```

동작 6 메소드 또는 속성의 이동 : 에지를 가진 멤버의 이동으로 인해 멤버 이름이 중복되거나(동작 1), 클래스 사이 에지가 갱신되거나(동작 3) 또는 클래스가 삭제되기도 한다(동작 8). 그러나 멤버 사이 에지 가중치는 변경하지 않는다.

```
/* c1 and c2 are nodes in V_A and m is a member of class c1 */
CAG::OP6(c1, m, c2) {
 if ( (c1 ∈ V_A)∧(c2 ∈ V_A)∧(m ∈ M(c1)∪A(c1)) ) {
  foreach m' ∈ M(c2)∪A(c2) do
    if ( m.name == m'.name ) {
      OP1(c1,m); // 멤버 이름 변경
      break;
    }
  delete m from c1 and add m to c2;
  if ( M(c1)∪A(c1) == ∅ )
    OP8(c1); // 멤버가 없으면 클래스 삭제
  else // 가중치 갱신
      foreach m' ∈ call(m)∪use(m) do
```

```
    if ( (d ∈ V_A)∧(m' ∈ M(d)∪A(d)) )
       { OP3(c1,d); OP3(c2,d); }
    // m' ∈ called(m)∪used(m)에 대해서도 위와 유사하게
    OP3 호출
  }
}
```

동작 7 클래스 노드 추가 : 멤버가 없는 클래스 노드를 이름이 중복되지 않도록 추가한다.

```
CAG::OP7(name) {
  c = new class;
  OP2(c, name);
  V_A = V_A ∪ {c};
}
```

동작 8 클래스 노드 삭제 : 메소드와 속성이 하나도 없으면 클래스 노드를 삭제한다. 클래스 노드를 삭제하면 동작 5를 적용하여 연결된 에지도 함께 삭제한다.

```
CAG::OP8(c) { /* c is a node in V_A */
  if ( (c ∈ V_A)∧(M(c)∪A(c) == ∅) ) {
    foreach a ∈ Parents(c) do
      OP5(c,a,R_IH); // 부모와 연결된 에지 삭제
    foreach b ∈ Children(c) do
      OP5(b,c,R_IH); // 자식과 연결된 에지 삭제
    foreach a ∈ Parents(c) do
      foreach b ∈ Children(c) do
        OP4(b,a,R_IH); // 부모와 자식 노드 연결
    foreach a ∈ InteractFrom(c) do
      OP5(c,a,R_IA); // 현재 노드로부터 시작하는 에지 삭제
    foreach b ∈ InteractTo(c) do
      OP5(b,c,R_IA); // 현재 노드로 끝나는 에지 삭제
    V_A = V_A - {c}; // 클래스 노드 삭제
    delete c; // 클래스 노드 소멸
  }
}
```

4.2 기본 재구조화 동작 집합의 의미(semantics)

본 연구는 객체지향 설계 행위를 보존하기 위하여 재구조화 과정에서 설계 구성 요소인 메소드, 속성, 그리고 행위를 나타내는 그들 사이의 관계 에지가 새로 추가되거나 삭제되지 않고 그대로 보존되는 것이 목적이고 다음과 같이 정의한다.

[정의 13] 객체지향 설계 행위 보존

설계 D와 D'가 주어졌을 때, 새로운 객체지향 설계 D'에 대하여 $CUG_{D'} = (V'_U, E'_U)$ 그리고 $CAG_{D'} = (V'_A, E'_A)$ 라고 가정한다. 만약 D와 D'가 아래 두 성질을 만족하면 D'는 객체지향 설계 D의 행위를 보존한다고 정의한다.

- CUG_D이 $CUG_{D'}$에 동형(isomorphic), 즉 $e(x,y)\in E_U$ 이면 $e'(f(x),f(y))\in E'_U$ 이고 $e'(f(x),f(y))\in E'_U$ 이면 $e(x,y)\in E_U$인 전단사 함수(bijection) $f : V_U \rightarrow V'_U$가 존재한다.
- CUG_D의 각 에지 $e(x,y)\in E_U$에 대하여 $W(e(x,y)) = W(e'(f(x), f(y)))$이 성립한다.

[정의 14] 객체지향 설계 재구조화

CUG와 CAG로 표시한 객체지향 설계 D의 변경을 R이라 할 때 $\Phi_R = \{(D, D') \mid D' \textit{ can be obtained from } D \textit{ by } R\}$이라 한다. 모든 $(D, D') \in \Phi_R$에 대하여 D'가 설계 D의 행위를 보존한다면 R을 객체지향 설계 재구조화라고 정의한다.

견고성(soundness)은 모든 재구조화 동작이 객체지향 설계 행위를 보존하는 것을 의미하고, 완전성(completeness)은 기본 재구조화 동작 집합이 모든 가능한 객체지향 설계 재구조화를 수행하는 것을 의미한다. 그리고 최소성(minimality)은 기본 재구조화 동작의 최소 부분 집합은 다른 동작들을 사용하여 유도될 수 없는 것을 의미한다.

[정리 1] (견고성) 모든 기본 재구조화 동작은 객체지향 설계 행위를 보존한다.

증명. 기본 재구조화 동작을 CUG와 CAG에 적용했을 때, 결과 그래프가 설계 행위를 보존함을 보인다.

- 동작 1 클래스 멤버 이름 변경

동작 1은 $CUG_D = (V_U, E_U)$ 노드 집합 원소 $x \in V_U$ 의 이름만 변경하므로 $CUG_D = (V_U, E_U)$와 변경된 $CUG_{D'} = (V'_U, E'_U)$ 사이에 $e(x,y)\in E_U$ 이면 $e'(f(x), f(y))\in E'_U$ 이고 $e'(f(x),f(y))\in E'_U$ 이면 $e(x,y)\in E_U$인 전단사 함수(bijection) $f : V_U \rightarrow V'_U$가 존재한다. 그리고 CUG_D의 각 에지 $e(x,y)\in E_U$에 대하여 $W(e(x,y)) = W(e'(f(x), f(y)))$이 성립한다. 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다.

- 동작 2 클래스 노드 이름 변경

동작 1과 유사

- 동작 3 클래스 사이 에지 가중치 변경

$CAG_D = (V_A, E_A)$의 클래스 사이 에지 $e(u,v) \in E_A$ 가중치 $W(e(u,v))$를 변경하므로 위와 마찬가지로 $CUG_D = (V_U, E_U)$ 노드 집합 V_U, 에지 집합 E_U, 그리고 에지 $e(x,y) \in E_U$ 가중치 $W(e(x,y))$를 변경하지 않는다. 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다.

- 동작 4 클래스 사이 에지 추가

$CAG_D = (V_A, E_A)$에서 클래스 사이 상호작용 에지 또는 상속 에지 추가는 $CUG_D = (V_U, E_U)$ 노드 집합 V_U, 에지 집합 E_U, 그리고 에지 $e(x,y) \in E_U$ 가중치 $W(e(x,y))$를 변경하지 않는다. 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다.

- 동작 5 클래스 에지 삭제

동작 4와 유사

- 동작 6 메소드 또는 속성의 이동

멤버를 현재 클래스에서 다른 클래스로 이동시키는 것은 $CUG_D = (V_U, E_U)$ 노드 집합 V_U, 에지 집합 E_U, 그리고 에지 $e(x,y) \in E_U$ 가중치 $W(e(x,y))$를 변경하

지 않는다. 그리고 이동시킨 후에 클래스 에지 가중치 갱신(동작 3)과 이름 변경(동작 1)도 설계 행위를 보존한다. 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다.

• 동작 7 클래스 노드 추가

$CAG_D = (V_A, E_A)$에서 클래스 노드 추가는 멤버가 없는 클래스 노드만 추가하는 것이므로 $CUG_D = (V_U, E_U)$ 노드 집합 V_U, 에지 집합 E_U, 그리고 에지 $e(x,y) \in E_U$ 가중치 $W(e(x,y))$를 변경하지 않고 이름이 중복된 경우에 해결하는 방법도 동작 2에 의하여 설계 행위를 보존한다. 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다.

• 동작 8 클래스 노드 삭제

멤버인 메소드와 속성이 없으면 클래스를 삭제하므로, $CUG_D = (V_U, E_U)$ 노드 집합 V_U, 에지 집합 E_U, 그리고 에지 $e(x,y) \in E_U$ 가중치 $W(e(x,y))$를 변경하지 않는다. 그리고 삭제될 클래스 c와 상속 관계를 가진 클래스 ($Parents(c) \cup Children(c)$) 그리고 상호작용 관계를 가진 클래스($Interact(c)$) 사이의 에지 삭제도 설계 행위를 보존한다(동작 5). 그러므로 결과의 객체지향 설계 D'는 D 행위를 보존한다. □

[정리 2] (완전성) CUG와 CAG로 표시한 임의의 설계 D와 D'가 객체지향 설계 행위를 보존할 때, D로부터 D'를 생성하는 일련의 기본 재구조화 동작이 존재한다.

증명. $CAG_D = (V_A, E_A)$ 그리고 $CAG_{D'} = (V'_A, E'_A)$일 때 다음과 같이 기본 재구조화 동작만을 적용하여 D를 D'로 재구조화한다.

1. D'에 존재하는 클래스를 D에 추가하기 위하여 동작 7을 사용한다. 이 동작은 D에 존재하지 않는 D'의 클래스가 모두 추가될 때까지 반복한다.

```
foreach c ∈ V'_A do
  if ( c ∉ V_A )
    OP7(c.name);
```

2. D'의 메소드와 속성에 대응하는 D의 메소드와 속성에 대하여 동작 6을 사용하여 D의 해당 클래스로 이동시킨다. 동작 6에서 이름을 변경해야 하는 경우에는 동작 1을 사용한다. 이 동작은 모든 메소드와 속성에 대하여 반복하여 수행한다.

```
foreach c ∈ V'_A do
  foreach m ∈ M(c)∪A(c) do
    m' = f⁻¹(m); // f⁻¹ : V'_U → V_U
    if ( (c' ∈ V_A) ∧ (m' ∈ M(c')∪A(c')) ) {
      c* = correspondTo(c); // c* in D corresponds to c in D'
      OP6(c', m, c*);
    }
```

3. D'에 존재하지 않는 D의 클래스들을 삭제하기 위하여 동작 8을 사용한다. 이 동작은 D'에 존재하지 않는 클래스가 D로부터 모두 삭제될 때까지 수행한다.

```
foreach c ∈ V_A do
  if ( c ∉ V'_A)
    OP8(c);
```

4. D의 클래스 사이 에지를 D'와 같게 만들기 위하여 동작 4를 사용하여 클래스 사이에 에지를 추가한다.

```
foreach e(f(x),f(y)) ∈ E'_A do
  if ( e(x,y) ∉ E_A )
    OP4(x,y,R(e(f(x),f(y)))); // R(e(f(x),f(y)))는 e(f(x), f(y))의 관계 유형(R_IH 또는 R_IA)을 반환
```

5. D의 모든 클래스 사이 에지에 동작 3을 적용하여 에지 가중치를 갱신한다.

```
foreach e(x,y) ∈ E_A do
  OP3(x,y);
```

위와 같은 순서로 재구조화 동작을 적용하면 D'는 객체지향 설계 D의 행위를 보존한다. 왜냐하면 D는 D'와 같은 멤버를 가진 클래스와 같은 가중치를 가진 에지를 모두 포함하고 있고 더 이상의 클래스와 에지는 없기 때문이다. □

[정리 3] (최소성) 기본 재구조화 동작의 최소 부분집합은 {동작 1, 동작 3, 동작 4, 동작 5, 동작 6, 동작 7, 동작 8}이다.

증명.

• 동작 1이외에 어떤 재구조화 동작 집합도 클래스 멤버 이름을 변경할 수 없다.

• 동작 2는 동작 1, 동작, 3, 동작 4, 동작 5, 동작 6, 동작 7, 그리고 동작 8의 조합으로 유도될 수 있다.

▷ 클래스 c의 이름 변경을 위하여 먼저 클래스 c의 모든 멤버들을 동작 6을 사용하여 다른 클래스로 이동시킨다.

▷ 동작 6에서 필요하면 동작 1을 사용하여 이름을 변경한다.

▷ 동작 8을 사용하여 클래스 c를 삭제하고 동작 5를 사용하여 연결된 에지를 삭제한다.

▷ 동작 7을 사용하여 생성하고자 이름을 가진 클래스 c'를 생성하고 클래스 c의 멤버들을 클래스 c'로 이동시킨다 (동작 6).

▷ 클래스 사이 에지를 추가하고(동작 4) 에지 가중치를 갱신한다 (동작 3).

• 동작 3이외에 어떤 재구조화 동작 집합도 클래스 사이 에지 가중치를 변경할 수 없다.

• 동작 4이외에 어떤 재구조화 동작 집합도 클래스 사이 에지를 추가할 수 없다.

• 동작 5이외에 어떤 재구조화 동작 집합도 클래스 사이 에지를 삭제할 수 없다.

• 동작 6이외에 어떤 재구조화 동작 집합도 멤버를 다른 클래스로 이동시킬 수 없다.

• 동작 7이외에 어떤 재구조화 동작 집합도 클래스 노

드를 추가할 수 없다.

- 동작 8이외에 어떤 재구조화 동작 집합도 클래스 노드를 삭제할 수 없다. □

4.3 자동 재구조화 지원

재구조화를 자동화하기 위하여 유전 알고리즘(이하 GA로 표기)을 적용한다. GA 적용에 대한 보다 자세한 내용은 [4]를 참조할 수 있고 요약하면 그림 2와 같다. GA를 실행시키기 위하여 모든 메소드 사이의 유사도를 계산하고 초기 설계의 응집도와 결합도를 계산한다. 초기 설계에서 응집도가 가장 낮은 클래스를 선택하여 해(solution)들의 모임인 모집단(population)을 생성하고 적합도(fitness) 함수를 사용하여 해들을 평가한다. 그리고 적합도에 비례하는 확률을 사용하여 해들을 선택(selection)하여 교배(crossover)시키고, 또 변이(mutation)시킨 후에 모집단의 품질이 낮은 해를 대치(replacement)한다. 이런 과정이 세대가 흘러가면서 반복되어 적합도가 높은 해들이 모집단의 많은 부분을 차지하면 이 중에서 가장 좋은 적합도를 가진 해를 최종의 해로 선택한다. 그러나 GA는 실행 시간이 길어질 수 있으며, 또한 지역 최적해 근처에서 지역 최적해를 찾는데 효율적이지 못한 특징이 있다. 그래서 일부 GA에서는 이 부분을 보완하기 위하여 미세 조정(fine tuning)을 위한 지역 최적화 방법(local-optimization)을 사용한다. 이렇게 클래스 응집도가 좋지 않은 클래스 구조를 차례로 변경함으로써 좋은 메트릭 값을 가진 클래스들로 점진적으로 재조직하는 것이다. 해를 평가하기 위한 적합도는 이전 연구의 정의를 갱신하여 다음과 같이 정의한다.

```
compute similarity for every method pair;
compute cohesion and coupling of the initial design;
do {
    select the class with the lowest cohesion;
    create an initial population of fixed size p;
    compute the fitness of the population;
    do {
        select parent1 and parent2 from the population;
        offspring = crossover(parent1, parent2);
        mutation(offspring);
        local-optimization(offspring);
        compute the fitness of the offspring
        replace(population, offspring);
    } while ( population is not converged and generations
      < g_max);
} while ( there exists a class to be restructured );
return the best solution;
```

그림 2 GA를 적용한 재구조화 방법

[정의 15] 적합도 함수(*Fitness(D)*)

$$Fitness(D) = w_1 * t_{coh} + w_2 * t_{cop}$$

where

$$t_{coh} = \frac{COH(D_{INI})}{COH(D)}$$

$$t_{cop} = \frac{COP_{IA}(D) * VOL(D)^{\alpha}}{COP_{IH}(D)} * \frac{COP_{IH}(D_{INI})}{COP_{IA}(D_{INI}) * VOL(D_{INI})^{\alpha}}$$

D_{INI}는 초기 설계, w_i는 메트릭에 대한 상대적인 가중치, α는 조정자로서 적합도 계산에서 $VOL(D)$의 상대적인 중요도 조정자, 그리고 $VOL(D)$는 $\sum_{Hi \in H(D)} D(H_i) * W(H_i) * R(H_i)$로서 각 구성 요소의 의미는 다음과 같다.

- $H(D)$는 설계 D에 존재하는 상속 계층들의 집합이다.
- $D(H_i)$는 상속 계층 H_i의 최대 깊이이다.
- $W(H_i)$는 상속 계층 H_i의 최대 폭이다.
- $R(H_i)$는 상속 계층 H_i에 있는 직접 상속 관계들의 개수이다.

재구조화 과정 동안에 객체지향 설계 응집도와 결합도에 따라 메소드와 속성들을 최적화된 클래스들로 나누며, 이 과정에서 $COP_{IA}(D)$는 감소하고, $COH(D)$와 $COP_{IH}(D)$는 증가하도록 적합도 함수 값을 최소화한다. 적합도 함수에서 $VOL(D)$는 상속 계층 구조 크기들의 합을 나타내며 상속 결합도 $COP_{IH}(D)$와 비례 관계이다. 객체지향 패러다임에서 상속은 중요한 특징이나 상속 계층이 너무 깊으면 유지 보수성에 부정적인 영향을 미치므로[20], 재구조화 과정에서 상속 계층의 깊이와 폭이 일방적으로 증가하지 않도록, 즉 상속 결합도를 보수적으로 증가시키기 위하여 $VOL(D)$를 추가하였다. 본 재구조화 방법의 시간 복잡도(time complexity)는 객체지향 설계 D의 전체 메소드 개수를 m, 전체 클래스 개수를 n, 클래스 당 최대 세대 반복 횟수를 g_{max}라고 가정할 때 $O(g_{max} * n * m^2)$로 단순화된다[5].

5. 실험 및 토론

본 논문의 클래스 상속 결합도(정의 11)는 단일 상속과 다중 상속을 포함하고 정의한 메트릭은 메소드 호출 관계와 속성 사용 관계에 기반한 메소드 행위 유사도(정의 9)로부터 유도한다. 반면 인터페이스는 메소드 몸체(body)가 없는 메소드 선언들의 모임으로서 메소드 호출 관계 또는 속성 사용 관계를 갖지 않으므로 메소드 행위 유사도에 기반한 결합을 갖지 않는다. 또한 인터페이스 메소드들은 클래스에서 오버라이딩되므로 다형성에 기반한 결합도를 유도할 수 있으나[22], 다형성에 기반한 메트릭은 클래스 또는 상속 계층 구조의 오버라이딩 메소드 개수를 기준으로 하고 메소드 행위는 고려하지 않는다. 따라서 본 논문에서는 메소드 행위 유사도에 기반한 클래스 관계를 대상으로 한다. 그림 1의 모델 검사에서는 재구조화된 설계의 클래스 상속 관계

가 순환하는 지를 검사하고, 부모 클래스로부터 자식 클래스로의 호출 또는 사용관계가 있는 지를 검사한다. 또한 자바 언어로 작성된 시스템의 설계는 클래스 다중 상속을 갖지 않아야 한다.

본 연구의 유효성을 실험적으로 평가하기 위하여 SUN Java Development Kit (JDK) 버전 1.3과 함께 배포된 이미지 맵 도구(표 1의 응용 1)와 자바 음악 재생기(표 1의 응용 2), 그리고 모델링 도구(표 1의 응용 3)[23]의 설계를 재구조화하였다. 표 1은 실험을 위한 응용 프로그램들에 대한 자료를 보여준다. 첫 번째로 응용 1은 클래스 평균 메소드 개수와 상속 개수가 많고 두 번째로 응용 2는 익명 클래스(anonymous class)를 포함하고 있으며 클래스들은 라이브러리 클래스와 상속 관계를 갖지만 응용 2의 프로그램 내부에서는 상속 관계가 없다. 세 번째로 응용 3은 클래스와 메소드 개수가 많고 클래스 사이에 상속 관계도 많이 존재한다. 응용 프로그램들은 라이브러리 클래스로부터 상속받아 메소드와 속성을 사용하나 본 연구의 재구조화 범위는 라이브러리 클래스와의 상속 관계는 제외하며 응용 프로그램 클래스에서 정의한 메소드와 속성만을 대상으로 한다. 즉 응용 프로그램내의 클래스에서 정의하지 않은 메소드와 속성의 접근은 고려하지 않는다. 그리고 실험은 Pentium Ⅲ 650 MHz CPU에 리눅스를 설치한 시스템에서 수행하였고 프로그램은 자바로 구현하였다.

GA를 적용한 실험에서 모집단의 크기 p는 50으로 설정하였고 해의 길이는 선택된 클래스의 메소드 개수($M(c_i)$)이다. 적합도 정의에서 $VOL(D)$의 조정자 α는 3으로 설정하여 상속 계층 크기가 보수적으로 증가하도록 유도하였다. w_1은 응집도에 대한 가중치이고 w_2는 결합도에 대한 가중치로서 문제의 특성상 모든 문제에 공통적으로 최적인 가중치(w_1과 w_2)를 찾는 것은 어려우나, w_1과 w_2 어느 한 값이 상대적으로 매우 크면, 예를 들어 w_1이 0.9 이상이고 w_2이 0.1 이하인 경우에 응집도의 영향이 강해져 응집도가 크게 향상되나 클래스 개수가 매우 증가하여 객체지향 설계 결합도가 높아진다. 그러나 w_1이 w_2보다 너무 크지 않으면, 예를 들어 w_1 = 0.4~0.6 이고 w_2 = 0.4~0.6 이면, 응집도와 결합도가 모두 개선된 설계를 얻을 가능성이 높다는 것을 확인하였다. 그래서 응집도 영향이 조금 크도록 가중치를 각각 w_1 = 0.6과 w_2 = 0.4로 설정하였다. 그림 2 유전 알고리즘의 첫 번째 연산자인 선택(select) 연산으로서 룰렛-휠(roulette-wheel) 연산을 사용하였는데, 그 과정은 모든 해들의 적합도 합을 계산한 후에, 적합도를 첫 번째 해부터 더하면서 처음으로 0과 그 합 사이의 난수보다 크거나 같은 적합도를 가진 해를 선택하는 것이다. 교배(crossover)의 기능은 부모를 사용하여 자식을 생성함으로서 과거의 행적을 기억하면서 개선된 자손을 만드는 것이다. 본 실험에서는 자식 해의 각 유전자를 부모 해의 두 유전자에서 임의로 선택하여 만드는 균등 교배 연산자를 사용하였다. 변이(mutation) 연산은 모집단에 다양성을 다시 부여하여 전역 최적해를 향한 새로운 공간을 탐색하게 한다. 이 연산자에서는 해의 각 유전자에 대하여 0과 1 사이의 난수를 발생시켜 모집단의 수렴 비율에 비례하는 변이율과 비교한다. 만약 난수가 변이율보다 크면 유전자를 새로운 임의의 값으로 변경한다. 대치(replace) 연산에서는 부모 해와 자식 해의 적합도를 비교하여 부모 해보다 자식 해의 적합도가 작으면 부모 해를 대치하고, 그렇지 않으면 모집단 중에서 가장 품질이 나쁜 해를 대치하는 방법을 사용하였다. 새로 생성한 자식 해로 부모 해를 대치하게 되면 모집단의 다양성을 유지할 수 있는 확률이 높다. 그림 2에서 한 클래스에 대한 GA 수행은(내부 **do-while** 루프) 동일한 품질을 갖는 해가 모집단의 80 퍼센트를 차지하거나 또는 내부 반복횟수가 g_{max}에 도달했을 때 정지한다. 본 실험에서 g_{max}는 선택된 클래스의 메소드 개수에 모집단 크기를 곱한 값($M(c_i)*p$)으로 설정하였다. 그리고 GA의 공간 탐색 효율성과 클래스 개수 복잡도를 조절하기 위하여 변경된 설계의 최대 클래스 개수를 설정하는데, 본 실험에서는 초기 설계의 클래스 개수보다 일정 비율(β) 증가한 수로 제한하였다. 응용 1은 클래스가 많지 않으므로 β = 0.15, 응용 2와 응용 3은 β = 0.1로 설정하였다. 실험에서 응용 1과 응용 2는 전체 클래스를 대상으로 재구조화를 수행하고, 응용 3은 클래스 개수가 많으므로 클래스 응집도가 낮은 순서로 전체 클래스의 50 퍼센트를 대상으로 수행하였다.

본 기법의 효과성을 확인하기 위하여 시뮬레이티드 어닐링 [24](이하 SA로 표기)을 사용한 방법과 비교한다. SA(그림 3)는 탐색 공간의 한 점으로부터 다른 점으로 임의적으로 이동한다. 이렇게 이동한 점이 품질이 더 좋은 점이면 받아들이는데, 이 이동을 긍정적 이동

표 1 응용 프로그램 자료

	LOC	클래스 개수	메소드 개수	속성 개수	상속 개수
응용 1	1.3K	15	113	71	11
응용 2	3.7K	46	170	196	0
응용 3	7K	64	420	241	23

```
get an initial design D;
get an initial temperature T > 0;
while (not yet frozen) {
        repeat L times {
                pick a random neighbor D' of D;
                d = cost(D') - cost(D);
                if d ≤ 0 (positive move)
                    D = D';
                if d > 0 (negative move)
                    D = D' with probability 1/e^(d/T);
        }
        T = r * T;  // reduce temperature
}
return D;
```

그림 3 SA를 적용한 재구조화 방법

(positive move)이라 한다. 그러나 만약 품질이 나쁜 점이라면 임의의 확률로 받아들인다. 이 이동을 부정적 이동(negative move)이라 하는데, 부정적 이동의 확률은 시간이 지남에 따라, 즉 초기 온도(T)로부터 시작하여 온도가 일정 비율(r)로 감소함에 따라 줄어든다. 이 부정적 이동은 SA가 지역 최적해로부터 벗어날 수 있게 한다. 그리고 매 온도마다 일정 횟수(L)만큼 해 공간을 반복 탐색한다. 그림 3에서 초기 설계 D의 임의의 이웃(random neighbor)을 얻기 위하여 현재 선택된 클래스의 한 메소드를 임의의 다른 클래스로 이동시킨다. SA 실험에서 GA의 적합도 함수를 SA의 비용 함수(cost function)로 사용하였고 온도 길이 L은 현재 선택된 클래스 메소드 개수($M(c_i)$)의 3.5~6배 그리고 온도 감소 비율 r은 0.8로 설정하였다. 긍정적 이동과 부정적 이동을 합해 받아들이는 비율이 0.1보다 작을 때 SA는 그림 3의 안쪽 repeat 반복문을 벗어난다. 각 클래스를 선택할 때마다 현재 설계의 비용 함수 값을 초기 온도 T로 설정하고 얼지 않을 때까지, 즉 0.01도가 될 때까지 수행한다. GA와 SA를 적용한 실험에서 수행 시간 차이가 너무 크지 않으면서 품질이 우수한 설계를 생성하도록 실험 인자 값을 설정하였다. 재구조화된 설계의 결과 값은 응용 프로그램에 각 방법을 30번 수행하여 얻은 평균이다.

표 2는 응용 1의 초기 설계와 GA와 SA를 적용하여 재구조화된 설계를 본 논문에서 정의한 메트릭과 기존 메트릭을 사용하여 측정한 결과를 보여준다. 기존 메트릭은 각 클래스의 메트릭 값의 합을 클래스 개수로 나누어 평균을 표시하였다. 응집도 메트릭 [16](이하 LCOM1으로 표기), [17](이하 LCOM2로 표기)의 정의에서 상속된 멤버를 포함하는지 명확히 기술되어 있지 않으므로 포함하지 않는 것으로 가정하였다. 표 2에서 보면 GA를 적용한 재구조화된 설계의 $COH(D)$이 0.52로 향상되었으나 SA를 적용한 결과는 오히려 감소하였다. $COP_{IH}(D)$과 $COP_{IA}(D)$는 각각 모두 증가하고 감소하여 향상되었다. 작을수록 바람직한 LCOM1과 LCOM2은 GA를 적용한 재구조화된 설계는 감소하여 개선되었으나 SA를 적용한 재구조화 결과에서 LCOM2는 증가하였다. 상속 관계와 상호작용 관계를 모두 포함한 결합도 메트릭 [16](이하 CBO로 표기)은 상속 결합이 증가하였기 때문에 GA를 적용한 결과에서 약간 증가하였으나 SA 결과에서 크게 증가하여 클래스 개수로 본 결합도가 매우 높아졌다. 그리고 상속 계층 깊이 DIT[18]는 상속 관계가 새로 추가되어 두 결과에서 모두 증가하였으며 SA 결과에서는 매우 증가하였다. 이것은 SA를 적용한 재구조화 설계의 $COP_{IH}(D)$가 크게 증가한 것과 일치하는 결과를 보여준다.

표 2 응용 1의 재구조화 결과

메트릭 \ 설계	초기 설계	GA 적용	SA 적용
COH(D)	0.51	0.52	0.44
COP_{IH}(D)	46.28	69.95	93.4
COP_{IA}(D)	11.83	3.42	4.51
LCOM1	41.53	34	33.1
LCOM2	3.53	3.46	3.84
CBO	3.13	3.71	5.18
DIT	0.8	1.15	1.33
클래스 개수	15	16.8	15.9
시간(초)		352	392

표 3 응용 2의 재구조화 결과

메트릭 \ 설계	초기 설계	GA 적용	SA 적용
COH(D)	0.61	0.67	0.55
COP_{IH}(D)	0	1.02	7.29
COP_{IA}(D)	96.64	89.78	69.31
LCOM1	7.35	7.32	9.52
LCOM2	2.67	2.54	3.1
CBO	1.41	1.49	1.92
DIT	0	0.02	0.12
클래스 개수	46	49.4	41
시간(초)		678	692

표 3에서 보면 응용 2의 GA를 적용한 재구조화된 설계 $COH(D)$이 0.67로 향상되었으나 SA를 적용한 결과는 오히려 감소하였다. $COP_{IH}(D)$과 $COP_{IA}(D)$는 모두 증가하고 감소하여 향상되었다. LCOM1과 LCOM2은 GA를 적용한 재구조화된 설계는 감소하여 개선되었으나 SA를 적용한 재구조화 결과는 오히려 증가하였다. GA를 적용한 결과는 클래스 개수가 증가하였지만 CBO는 약간 증가하였고, SA를 적용한 결과는 클래스 개수가 감소하고 CBO도 크게 증가하여 클래스 개수 결합도는 매우 높아졌다. 그리고 DIT는 상속 관계가 새로 추

표 4 응용 3의 재구조화 결과

메트릭 \ 설계	초기 설계	GA 적용	SA 적용
COH(D)	0.37	0.43	0.38
$COP_{IH}(D)$	1.95	5.31	7
$COP_{IA}(D)$	206.99	194.62	182.3
LCOM1	21.67	19.92	20.6
LCOM2	4.64	4.36	4.52
CBO	1.7	1.75	1.81
DIT	0.45	0.11	0.15
클래스 개수	64	68.9	66.1
시간(초)		2161	2415

가되어 GA 결과는 약간 증가하였으나 SA 결과는 크게 증가하였다.

표 4에서 보면 GA를 적용하여 재구조화된 응용 3 설계의 $COH(D)$이 0.43로 매우 향상되었으나 SA를 적용한 결과는 약간 향상되었다. $COP_{IH}(D)$과 $COP_{IA}(D)$는 모두 증가하고 감소하여 향상되었다. LCOM1과 LCOM2은 GA를 적용하여 재구조화된 설계가 SA를 적용한 재구조화 결과보다 더 개선되었다. 재구조화된 설계의 클래스 개수가 증가하고 상속 결합도가 증가하여 두 결과 모두 CBO는 증가하였지만 SA 결과가 크게 증가하여 클래스 개수로 본 결합도는 더 높아졌다. 그리고 DIT는 GA와 SA 결과 모두 매우 감소하였는데, 그 이유는 응용 3의 초기 설계에서 부모 클래스와 자식 클래스 멤버들 사이에 상대적으로 교류가 없는 상속을 많이 사용하고 또한 자식 클래스의 생성자가 부모 클래스와 교류하더라도 본 기법의 메트릭 계산에서 생성자는 제외되기 때문이다. 그리고 응용 1, 응용 2, 그리고 응용 3의 재구조화된 설계(응용 2의 SA 적용 제외) 모두 클래스 개수가 증가하였는데, 이것은 일부 클래스들이 관련이 적은 메소드나 속성을 멤버로 포함하고 있다는 것을 보여준다. 객체지향 시스템 모듈화 연구에서도 모듈화한 후에 모듈 개수가 증가한다는 것을 통계적 기법을 사용하여 보였다[9]. 이것은 일반적으로 클래스 또는 모듈이 긴밀한 요소들로만 구성되지 않고 관련이 적은 요소들도 일부 포함하고 있다는 것을 나타낸다.

실험 결과로부터 GA를 적용한 재구조화는 응집도와 결합도가 고르게 향상되었지만 SA를 적용한 재구조화는 응집도가 거의 같거나 더 낮아지고 $COP_{IH}(D)$과 $COP_{IA}(D)$가 크게 개선되었지만 CBO와 DIT도 크게 증가하였다는 것을 알 수 있다. 따라서 GA를 적용한 방법이 SA를 적용한 방법보다 객체지향 설계를 효과적이고 안정적으로 재구조화하여 좋은 품질의 설계를 생성한다고 기대된다. 실험에서 클래스 전체 또는 반수를 대상으로 재구조화를 수행하여 시간이 길게 소요되었으나 실제에서는 품질이 떨어진 몇 개 클래스만을 선택적으로 변경하면 품질도 개선되고 수행 시간도 많이 단축된다.

6. 결 론

본 논문에서는 객체지향 설계 행위를 보존하는 메트릭 기반 설계 재구조화 기법을 집합론에 기반하여 기술하고 자바 응용 프로그램에 적용하여 유효성을 실험적으로 평가하였다. 먼저 설계 품질을 측정하기 위하여 메소드 행위 유형의 유사도를 기반으로 응집도와 결합도 메트릭을 정의하였고 자동적으로 재구조화하기 위하여 유전 알고리즘을 적용하였다. 본 기법의 재구조화 동작들은 설계 행위를 보존하므로 초기 시스템 동작을 훼손하지 않는다. 그리고 자동화를 지원하므로 모델링 도구와 통합되면 규모가 크고 복잡한 시스템의 설계 재구조화에 효율적으로 적용이 가능하다.

본 기법은 클래스들을 세밀하게 재구성하므로 재구조화된 설계의 전체 클래스 개수가 증가하는 경향이 있다. 여기서 한가지 고려할 점은 각 클래스가 재구성되어 메트릭 값은 우수하지만 클래스 개수 증가가 설계 복잡도에 영향을 미칠 수 있다는 것이다. 따라서 확장성, 재사용성, 또는 응용의 특정성 등의 요인들을 고려할 때 재구조화된 설계를 설계자가 일부 변경할 수 있다. 세밀하게 재구성된 클래스들을 합병하거나 또는 재구조화하지 않도록 고정하는 것이 메트릭 값을 좋지 않게 만들지라도 설계자의 응용 지식과 경험을 기준으로 볼 때 바람직할 수 있기 때문이다. 따라서 재구조화 과정 중에 설계자의 지식과 경험을 적용하면 실질적으로 더 유용한 설계를 얻을 수 있다.

현재 모델 검사는 재구조화된 설계가 객체지향 개념에 적합한 지를 확인하는 기능만 담당하고 재구조화된 시스템의 동적인 특성에 대한 검사는 하지 않는다. 왜냐하면 현재 *CUG* 모델은 호출과 사용 관계만을 나타내고 메소드들 사이의 순서와 의존 관계는 지원하지 않기 때문이다. 따라서 시스템의 동적인 특성에 대한 결함을 검사하기 위하여 순서와 의존 개념을 지원하도록 현재 모델을 확장하여 객체지향 설계 행위의 의미를 세밀하게 규정할 것이다. 또한 일반 객체지향 모델링 도구를 사용하여 작성된 설계도 재구조화할 수 있도록 UML과 호환되게 모델을 확장할 것이다.

참 고 문 헌

[1] R. Arnold, "Software Restructuring," *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 607-617, 1989.

[2] I. Sommerville, *Software Engineering*, 5th ed., Addison Wesley, Harlow, England, 1995.

[3] H. Sahraoui, R. Godin, and T. Miceli, "Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation?," *In Proceedings of International Conf. on Software Maintenance*, pp. 154-162, 2000.

[4] B. J. Lee and C. S. Wu, "Restructuring Object-Oriented Design using Genetic Algorithms," *In Proceedings of International Conf. on Software Engineering and Applications*, pp. 167-172, 2000.

[5] B. J. Lee and C. S. Wu, "Restructuring of Object-Oriented Designs using Metrics," *Journal of KISS : Software and Applications*, Vol. 28, No. 6, 2001.

[6] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," *In Proceedings of the 6th International Workshop on Program Comprehension*, pp. 45-52, 1998.

[7] B. K. Kang and J. M. Bieman, "A Quantitative Framework for Software Restructuring," *Journal of Software Maintenance*, Vol. 11, No. 4, pp. 245-284, 1999.

[8] F. Abreu, G. Pereira, and P. Sousa, "A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems," *In Proceedings of 4th European Conference on Software Maintenance and Reengineering*, 2000.

[9] F. Abreu and M. Goulo, "Coupling and Cohesion as Modularization Drivers: Are we being over-persuaded?" *In Proceeding of Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[10] J. Banerjee, W. Kim, H. J. Kim, and H. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *ACM SIGMOD*, 1987.

[11] P. Bergstein, "Object-Preserving Class Transformations," *In Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, 1991.

[12] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," *In Proceedings of International Conf. on Automated Software Engineering*, pp. 174-181, 1999.

[13] I. Moore and T. Clement, "A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies," *In Proceedings of Technology of Object-Oriented Languages and Systems(TOOLS) Europe*, pp. 173-184, 1996.

[14] G. Snelting and F. Tip, "Reengineering Class Hierarchies Using Concept Analysis," *ACM. Sigsoft Software Engineering Notes*, Vol. 23, No. 6, pp. 99-110, 1998.

[15] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, Vol. 3, No.1, pp. 65-117, 1998.

[16] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. on Software Engineering*, Vol. 20, No. 6, pp. 476-493, 1994.

[17] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *In Proceedings of International Symposium on Applied Corporate Computing*, 1995.

[18] W. Li and S. Henry, "Object-Oriented Metrics That Predict Maintainability," *Journal of Systems and Software*, Vol. 23, pp. 111-122, 1993.

[19] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *In Proceedings of International Conference on Software Quality*, 1995.

[20] R. Harrison, S. Counsell, and R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *The Journal of Systems and Software*, Vol. 52, No. 2-3, pp. 173-179, 2000.

[21] L. C. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. on Software Engineering*, Vol. 22, No. 1, pp. 68-86, 1996.

[22] C. Pons, L, Olsina, M. Prieto, "A Formal Mechanism for Assessing Polymorphism *in Object-Oriented Systems," In Proceedings of Asia-Pacific Conference on Quality Software*, pp. 53-62, 2000.

[23] 강유훈, 박찬진, 우치수, 김민정, "UML 추상 구문을 이용한 정적 설계 제한조건 검사 도구", 한국 소프트웨어공학 학술대회 논문집, Vol. 3, No.1, 2001.

[24] D. S. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation, Part I, Graph Partitioning," *Operations Research*, Vol. 37, pp. 865-892, 1989.

이 병 정

1990년 서울대학교 계산통계학과(학사) 1998년 서울대학교 전산과학과(석사) 2002년 서울대학교 전기컴퓨터공학부(박사). 1990년 1월~1998년 1월 현대전자 소프트웨어연구소 주임연구원. 2002년 3월~현재 서울시립대학교 컴퓨터과학부 전임강사. 관심분야는 소프트웨어 재공학, 소프트웨어 품질평가, 소프트웨어 개발방법론 등