수정 및 무수정을 통한 코드 재사용성 측정 모델링

(Modeling for Measurement of Code Reusability with without Modification)

김 형 섭 † 배 두 환 ††
(Hyungseob Kim) (Doohwan Bae)

요 약 소프트웨어 재사용은 소프트웨어 개발에서 생산성 향상과 품질 향상에 크게 기여한다. 이러한 소프트웨어 재사용을 위해서는 기존 소프트웨어 시스템에서 재사용성이 높은 부품들을 추출하는 것이 매우 중요하다. 소프트웨어 재사용은 수정 없는 재사용과 수정을 통한 재사용으로 구별 될 수 있다. 수정을 통한 재사용에서는 수정성이 가장 중요한 품질이 된다. 왜냐하면 수정에 드는 비용이 지나치게 많아서 부품을 새로 작성하는 것 보다 비용이 많이 든다면 굳이 수정을 통한 재사용을 할 필요가 없기 때문이다. 이러한 수정성을 높이기 위해서는 높은 이해성, 그리고 다른 부품들에 대한 낮은 파급효과가 필요하다. 반면에 수정 없는 재사용에서는 정보 은닉이 핵심이 된다. 왜냐하면 정보 은닉이 잘 되어 있을수록 소프트웨어 부품들의 구현상의 세부 사항들을 알지 않고 이용할 수 있으므로 수정 없는 재사용에 유리하기 때문이다.

본 연구에서는 수정 없는 재사용성과 수정을 통한 재사용성의 측정을 위한 새로운 모델을 제안한다. 또한 이 모델을 구성하는 인자들과 측정 기준도 제안한다.

Abstract Software reuse greatly increases productivity and enhances quality in software development. The extraction of reusable software components from existing software systems is critical to effective software reuse. Software reuse is classified into black-box reuse and white-box reuse. In white-box reuse, modifiability is the most important quality. If modification cost is greater than production cost then there is no need to reuse with modification. High understandability and low ripple-effect between software components are essential for increasing modifiability. In black-box reuse, however, information-hiding is the most important quality. The better the information-hiding, the better the black-box reuse, because we need not know the implementation details of every software components.

We propose a new model for measuring of black-box reusability and white-box reusability. Also the factors of the model, and the measurement criteria are proposed.

1. 개 요

소프트웨어 재사용(software reuse)이란 소프트웨어 시스템을 구축할때 모든것을 새로 다 만드는 것이 아니라, 기존의 소프트웨어를 이용하여 소프트웨어 시스템을 만드는 과정(process)을 말한다[1].

또, 재사용성(reusability)은 새로운 상황에서 이전의 개념이나 객체/부품들을 재사용하기 쉬운 정도를 말한다[2].

이러한 소프트웨어 재사용의 장점은, 생산성 향상, 소프트웨어 품질(신뢰성, 유지 보수성등) 향상, 소프트웨어 개발 기간 단축, 개발 비용 절감, 시스템과 시스템 구축 방법에 대한 지식 공유 등을 들 수 있다[3,4]. 소프트웨어 재사용에 대해서는 여러가지의 관점이 존재하는데 이를 정리하면 표 1과 같다.

재사용 가능 부품들을 만들기 위한 가장 좋고 빠른 방법은 기존 소프트웨어 시스템에서 그 부품들을 추출해내는 것이다[3]. 기존 소프트웨어 시스템은 많은 부품들로

† 비 회 원 : (주)나우콤 기술기획팀
†† 종신회원 : 한국과학기술원 전산학과 교수
논문접수 : 1996년 7월 10일
심사완료 : 1997년 3월 3일

이루어져 있는데, 이중에는 재사용성이 높은 부품도 있고, 그렇지 않은 것도 있다. 재사용성이 높은 부품들은 그만큼 더 쉽게 재사용이 가능하고, 재사용으로 인한 이점을 얻기 쉽다. 그런데, 재사용성이 높은 부품을 추출해 내기 위해서는 그에 적합한 기준과 그 기준에 따른 측정이 필수적이다. 이 측정에 의해 재사용성이 높은 부품들을 추출하고, 또는 기존의 부품들을 재사용성이 높아지도록 가공해서, 그 기능과 의미에 따라 분류하고 문서화(documentation)하면 우수한 소프트웨어 저장소(repository) 또는 라이브러리를 만들 수 있다.

표 1 에서 살펴본 바와 같이 여러가지 대상들이 재사용될 수 있는데, 본 연구에서는 그중에서 소스 코드를 연구대상으로 한다. [3]의 지적처럼, 코드의 재사용은 설계들의 재사용에 비추어 볼때, 프로그래밍 언어와 운영체계, 그리고 응용 분야에 종속되기 때문에 재사용성에서 다른 대상들 보다 더 많은 한계를 갖고 있다. 반면에 설계/아이디어등은 더 높은 재사용성을 갖는다[6]. 그러나 설계 정보등의 표현 방법(representation)은 코드에 비해 아직 잘 정립되어 있지 않으며 이는 재사용과 재사용성의 측정을 어렵게 하는 가장 큰 요인이다[6]. 따라서, 잠재적 재사용성은 코드 이상의 추상화 단계를 갖는 것들이 더 크지만, 현재의 실제적인 재사용은 오히려 코드 단계에서 더 많이 이루어 지고 있다. 이러한 이유 때문에 본 연구에서는 재사용성의 측정 대상을 소스 코드로 제한한다.

이러한 소스 코드의 재사용성의 측정은 다양한 방면에서 이용될 수 있다. 코드의 재사용성의 측정은 저장소 또는 라이브러리의 구축과 이용에 직접적으로 이용될수 있고, 또 소프트웨어 생명주기의 상위 단계로 피드백되어 재사용 중심의(reuse-based)소프트웨어 개발을 가능하게 한다. 그리고, 재사용성 측정에 의해 손쉽게 재사용성이 높은 부품들을 발견하고 이들을 사용해서 소프트웨어 시스템을 구성한다면, 그만큼 그 시스템의 이해성(understandability), 유지보수성(maintainability)이 좋아지고, 개발에 필요한 시간과 노력과 비용의 절감등 재사용으로 인한 제반이점을 쉽게 얻을 수 있다. 즉 재사용의 주요 목적인 요구 사항, 제반 명세서, 지식, 또는 정보에 대한 이해와 설명에 필요한 노력을 절감 시키는 것[7]을 달성하는데 도움이 된다. 그 이외에도 재공학(reengineering)과정에서 재사용성이 더 높은 부품의 사용을 유도할 수 있으므로 더 우수한 재공학을 가능케한다. 재사용성의 측정은, 또한, 재사용에 적합한 부품을 검색/추출하는 비용을 크게 절감시키는데도 유용하다. 재사용할 부품을 찾기 위해 방대한 소프트웨어 시스템 전체를 검색하기 보다는, 재사용성이 높은 부품들만 모아둔 상대적으로 소규모인 라이브러리에서 검색한다면 그만큼 비용과 시간이 절약된다[8].

표 1 재사용에 대한 여러 관점 [5]

본질	범위	과정	기술	방법	대상
아이디어/개념	수직적	계획적/체계적	구성적	수정 없음	소스 코드
부품	수평적	임기응변적	생성적	수정을 통함	설계
절차/기술					명세서
					객체
					문서
					구조

요약하면, 소프트웨어 재사용의 장점을 얻기 위해서는 재사용하기 좋은 부품들이 필요하고, 이러한 부품들을 발견/사용하기 위해서는 재사용성의 측정이 필수적이라 할 수 있다.

본 논문의 구성은 다음과 같다. 2 장에서는 재사용성 측정에 대한 관련 연구를 비판적으로 검토 해보고, 3 장에서는 재사용성을 어떤 방향으로 측정해야 하는지에 대해 정리한다. 4 장에서는 그 방향에 입각한 구체적인 측정 척도들을 열거하고 구체적 측정 방법을 정리한다. 5 장에서는 이 방법을 사용해서 실제의 소스 코드를 대상으로 한 사례 연구의 결과를 기술한다. 6 장에서는 결론과 앞으로 해야할 과제들을 제시한다.

2. 관련 연구

소프트웨어의 재사용성을 측정하는데 대한 연구는 [8], [9] 그리고[10]이 있다. 이중에서 [9]는 객체지향 소프트웨어와 객체 기본(object-based) 소프트웨어에 있어서 언어(특히 Ada)자체가 지원하는 재사용을 측정한다는 관점이다. 따라서 본 연구에서 초점을 맞추고 있는 소프트웨어 부품의 재사용성 측정과는 거리가 있다. 그러므로 본 연구에서는 [8]과 [10]을 관련 연구로 살펴보도록 한다.

[8]에서는 소프트웨어 부품의 재사용성을 4개의 척도(metric) 즉, Halstead의 software science에 의한 volume, McCabe의 cyclomatic complexity number, regularity, 재사용 빈도를 사용해서 측정한다.

이러한 방법의 측정은 다음과 같은 단점을 갖고 있다. 첫째, 측정방법을 복잡도(complexity)측정방법 — software science와 cyclomatic complexity는 주로 프

로그램의 복잡도를 측정하는데 사용되는 척도들이다 — 위주로 함으로써 재사용성의 측정을 복잡도 측정으로 한정짓는 결과를 가져왔다. 재사용성은 복잡도와 밀접한 관련이 있지만, 복잡도가 재사용성과 같은것은 아니다. 둘째, 복잡도 측정의 경우에서도 부품 하나 하나의 복잡도만 고려하고 각 부품들간의 관계에서의 복잡도에 대한 측정은 하지 않고 있으므로, 그 소프트웨어 시스템의 구조적 측면에서의 복잡도는 측정되지 않는 결과가 된다.

[10]에서는 재사용 부품 평가 모델로 각 부품의 재사용성을 측정하는데, 이 모델의 인자로 모듈성(modularity), 일반성(generality), 신뢰성(reliability), 시스템 독립성(system independence), 자기 설명성(self documentation)을 들고 있다. 그리고 모듈성의 척도(metric)로 McCabe의 cylomatic complexity와 Yourdon의 제어폭(fan-out), 그리고 결합도(coupling)를 제시하고 있다. 이 논문은 시스템의 구조적 측면에서의 복잡도를 고려하고, 복잡도 이외에 재사용성과 관련되는 다른 품질들을 함께 고려 함으로써 [8]의 단점을 일부 극복하고 있다. 그러나 이 논문의 재사용 부품 평가 모델에도 몇가지 단점이 있다. 첫째, 결합도를 결정하는 방법에서 오해가 있다. 결합도는 두개의 모듈사이의 상호 의존도의 정도[11]로 정의된다. 따라서 결합도는 모듈들 사이의 관계에서 나타나는 개념으로 모듈 단독으로는 결정될 수 없다. 그러나 [10]에서는 하나의 함수만으로 결합도를 결정하고 있다. 예를 들어보면, 하나의 함수내에 전역변수가 존재하는 경우 공통 결합도(common coupling)로 결정하고 있다. 이는 결합도의 정의에 위배되는 것이다. 결합도는 모듈들 사이의 관계에서 결정되는 것이기 때문에 같은 모듈이라 하더라도 다른 모듈들과의 관계에서 서로 다른 결합도를 가질 수 있다. 둘째, 결합도는 모듈 단위의 품질을 측정하는데는 적합하지 않다. 결합도의 정의상 모듈 단위로 결정할 수 없기 때문에 이 논문의 재사용 부품 평가 모델과 같이 부품 단위의 품질(재사용성)을 측정하는데는 적합하지않다. 셋째, 이 논문에서는 재사용성의 구성인자의 하나로 일반성을 들고 있으며, 그 일반성의 척도로 2 이상의공유도(fan-in)을 제시하고 있다. 하지만 일반성은 응용성(applicability)과 서로 대립되는 관계에 있으므로, 일반성이 강조되는 경우 응용성이 떨어지는 문제가 있고, 이를 [6]은 재사용성의 3가지 딜렘마중의 하나로 제시하고 있다. 따라서 재사용성의 측정을 위해 일반성을 척도로 사용하는데는 일정 정도의 제한이 있어야 한다. 또한 2 이상의 공유도를 갖는 모듈이면 일반성이 있다고 결정하는 것은 객관적인 척도라고는 할 수 없다. 넷째, 신뢰성을 측정하기 위해서 statement coverage를 100% 수행하는 테스트 케이스를 생성하여 부품들을 테스트하는 방법을 제시하고 있는데, 각각의 모듈의 모든 문장들을 적어도 한번씩 수행하는 테스트 케이스를 만들어서 수행한다는 것은 현실적으로 매우 어렵다. 또한, statement coverage는 여러 테스트 방법중에서도 매우 신뢰성이 낮은 방법이기 때문에 한번 테스트에 통과 했다고 해서 그 부품이 신뢰성이 있다고 할 수는 없다.

3. 재사용성 측정의 방향

1 장의 표 1 에서 보았듯이, 재사용을 재사용 방법의 관점에서 보면 수정 없는 재사용과 수정을 통한 재사용으로 나누어 볼 수 있다. 그런데 수정없이 재사용하는 경우에는 이해성이 핵심적이지 않지만, 수정을 통한 재사용의 경우에는 이해성이 매우 중요해진다[12]. 그러나 수정 없이 재사용하는 경우에는 이해성 뿐만 아니라 다른 품질에서도 수정을 통해 재사용하는 경우와 서로 다른 특성을 갖는다. 수정 없이 재사용을 하는 경우에는 정보 은닉의 개념이 핵심이 된다[5]. 그 이유는 정보 은닉이 잘되어 있을 수록 그만큼 추상화(abstraction)가 잘되어 있게[13] 되는데, 이는 그 부품의 구현상의 세부 사항을 알지 않고도 그 부품의 기능등을 쉽게 재사용할 수 있게 하기 때문이다. 즉 소프트웨어 부품이 매우 복잡하고, 이해하기 어렵고 수정성(modifiablity)이 나쁘더라도, 정보 은닉이 잘되어 있다면, 그 부품이 특정한 요구 사항에 잘 부합하는 경우 수정 없이 재사용하기가 매우 쉽다. 이 정보 은닉의 개념은 Parnas에 의해 처음 소개가 되었는데, 가능한 한 많은 정보 즉 논리 제어와 자료 구조등의 세부 사항을 다른 외부의 부품들이 알지 못하게 하는 것을 의미한다. 정보 은닉을 높이기 위해서는 부품의 자기 완결성(self-containedness)과 높은 응집도, 되도록 적은 매개 변수(parameter)가 필요하다. 자기 완결성이 필요한 이유는, 그 부품에서 사용하는 자료 구조와 제어는 외부 부품과의 의존성이 적을 수록 정보 은닉에 유리하기 때문이다. 그리고, 응집도(cohesion)는 정보 은닉과 밀접한 관계가 있어서 정보 은닉이 잘 되어 있을 수록 응집도가 높아지게 된다. 왜냐하면, 응집도가 높다는 것은 그 부품의 각각의 부분들 모두가 매우 가까운 논리적 관계를 가지고 있다는 것을 의미하는데, 정보 은닉이 잘되어 있을 수록 외부에서 알 필요가 있는 정보가 적고 그만큼 부품내부의 각 부분들이 밀접하게 잘 연결되어 있게 되므로 응집도가 높아지게 되는 것이다. 물론 그 반대의 경우, 즉 응집도가 높다고 해서 항상 정보 은닉이 잘되어 있다고 하는 것은 항상 참은 아니다. 응집

도의 정의상 전역 변수를 사용함으로써 정보 은닉의 정도가 낮은 경우에도 응집도는 높을 수가 있기 때문이다. 매개 변수는 외부와의 인터페이스를 직접적으로 반영하며, 외부에서 그 모듈을 이용하기위해서 알아야 하는 정보를 나타낸다. 각각에 대한 자세한 내용은 4장에서 다루도록 한다.

반면에 수정을 통한 재사용의 경우에는 수정성이 가장 중요한 품질이 된다. 왜냐하면 수정을 통해서 재사용하는 경우, 수정에 드는 노력과 비용이 매우 커서 새로운 부품을 만드는 것보다 더 부담이 된다면 굳이 재사용할 필요가 없고, 또 손쉽게 수정이 가능하면 적은 비용과 시간으로 기존의 부품을 재사용할 수 있기 때문이다. 따라서 수정성을 측정하여 그 부품이 얼마나 용이하게 수정이 가능한가를 아는 것이 중요해진다. 이 수정성은 소프트웨어 시스템을 수정하는데 있어서의 상대적 용이성 또는 어려움을 말하며, 또 이러한 수정으로 인해 신뢰성이 저하될 수 있는 상대적 가능성을 뜻하는 것[14]으로 정의 된다. 따라서 수정성을 높이기 위해서는 높은 이해성, 그리고 수정으로 인해 발생하는 파급 효과(ripple effect)의 최소화가 필요하다. 소프트웨어의 수정을 위해서 가장 먼저 수행되어야 할 것이 소프트웨어에 대한 이해이고, 여기에서 상당히 많은 시간을 소모하므로 높은 이해성은 용이한 수정을 위해서 필수적이다. 또 수정의 효과가 가능한 한 국지적이어야만 수정 때문에 발생할 수 있는 영향의 파급을 줄임으로써 신뢰성이 낮아지는 정도를 줄일 수 있다.

이와같이, 수정 없는 재사용성(black box reusability)과 수정을 통한 재사용성(white box reusability)은 서로 다른 특성을 갖고 있으므로 소프트웨어 부품의 재사용성을 측정할 때는 이 두개를 구별해서 측정해야만 한다. 그런데 앞의 2장에서 살펴본 두 논문의 재사용성 측정은 이와같이 수정 없는 재사용성과 수정을 통한 재사용성을 구별하고 있지 않는데, 엄밀히 말하면 수정을 통한 재사용성을 측정한 것이라고 할 수 있다. 즉 소프트웨어의 복잡도가 낮고, 주석(comment)의 양이 많고, 신뢰성이 높은 부품이 재사용성이 높다고 하는 것은, 부품에 대한 이해와 수정을 통한 재사용을 암시하는 것이다. 그러나 이러한 재사용성 측정은 유닉스의 C 라이브러리 소스 코드와 같이 매우 높은 (수정 없는)재사용성을 갖는 부품들이 그 부품의 복잡도등 때문에 재사용성이 낮게 측정이 되는 문제를 야기하게 된다. 이와같이 재사용성을 수정 없는 재사용과 수정을 통한 재사용으로 구별하지 않고 포괄해서 측정하는 경우에는, 이 두가지 재사용성의 차이를 구별할 수 없게되어, 그 측정의 의미가 감소하게 될 뿐만 아니라 정확한 측정이 될 수 없게 되는 것이다. 그리고 소프트웨어 부품들은 구체적 필요성이 제기될 때에만, 그것이 수정 없이 재사용 될지, 수정을 통해 재사용될지, 아니면 완전히 새로운 부품을 만들어야 할 지를 알 수 있다. 즉 요구가 없는 경우에는 수정 없는 재사용이 될 지, 수정을 통한 재사용이 될지 알 수 없으며, 또 필요에 따라서는 하나의 부품이 수정 없이 재사용 될 수도 있고, 수정을 통해 재사용될 수도 있다. 그러므로 하나의 부품의 재사용성을 측정하는 경우 두가지 재사용성을 함께 그리고 별개로 측정해야 하는 것이다.

물론 소프트웨어 공학상의 여러 품질들이, 수정 없는 재사용과 수정을 통한 재사용에서 서로 다르고 배타적으로 적용되는 것은 아니다. 이해성이 좋고, 그만큼 복잡하지 않고, 또 수정성이 있는 부품은 수정 없이 재사용하는 경우에도 높은 재사용성을 가질 수 있다. 다만 수정 없이 재사용 하는 경우에는 정보 은닉이 중심 품질이되고, 수정을 통해 재사용하는 경우에는 수정성이 중심 품질이 되어야 한다는 것을 의미한다.

재사용성 측정에서 한가지 더 명확하게 해야 할 것은 영역 또는 범위에 관한 것이다. 수직적 또는 특정 영역에서의 재사용에서는 소프트웨어의 여러 품질들 이외에 영역에 대한 지식(domain knowledge)이 매우 중요해진다. 즉 특정 영역에서의 재사용만을 고려할 때는, 정보 은닉이나 수정성 보다는 그 영역에의 적합성이 더 크게 작용하며, 이는 그만큼 특수성이 커진다고 할 수 있다. 본 연구에서는 특정 영역에서의 재사용을 연구 대상으로 하지는 않는다. 더 일반적인 관점에서의 수평적 재사용에 촛점을 두는데, 그 이유는 범용성 때문이다. 이제 여기에서 본 연구에서 측정하고자 하는 재사용성을 1장의 6가지 관점으로 자세하게 정리해 보면 다음과 같다. 먼저 본질적 관점에서는 부품을 대상으로 하는데 이는 현실적 실효성 때문이며, 범위에서는 위에서 언급한 바와 같이 수평적 입장을 취하고, 기술적 관점에서는 제품(product)을 그대로 재사용하는 생성적 재사용보다, 소프트웨어 개발에서 많이 사용하게 되는 구성적 재사용을, 대상적 관점에서는 역시 현실적 효용성때문에 소스 코드를 대상으로 하도록 한다. 특히 본 연구에서는 UNIX/C 환경에서의 소스 코드를 대상으로 한다. 따라서 C 언어의 특성상 함수와 프로시주어, 부품, 모듈등은 상호 호환성이 있는 용어로 사용하도록 한다. 재사용 방법이라는 관점에서는, 두가지 재사용성을 구별하여 모두 측정해야 한다는 것이다. 끝으로 과정적 관점에서 보면, 이러한 재사용성의 측정은 결국 계획적/체계적 재사용을

더 효율적으로 하기 위한 전제 조건이 된다.

4. 재사용성 측정 방법과 기준

측정은 명확하게 정의된 규칙에 따라서 현실 세계의 개체(entity)들의 속성(attribute)들에 숫자 또는 기호(symbol)을 할당하는 과정(process)으로 정의된다[15]. 여기에서의 개체는 하나의 객체(object)나 사건(event)이 될 수 있다. 또 속성은 그 개체의 특징 또는 특성을 말한다. 측정은 크게 두가지 유형으로 나눌 수 있는데, 하나는 직접 측정(direct measurement)이고, 또하나는 간접 측정(indirect measurement)이다. 재사용성은 소프트웨어의 다른 품질과 마찬가지로 추상적 개념이기 때문에 직접 관찰하거나 측정할 수는 없다. 그러나 재사용성과 관계가 밀접한 다른 관찰 가능한 현상들의 측정치를 통해 간접적으로 측정할 수는 있다. 이를 위해서는 측정기준을 먼저 만들 필요가 있는데, 수정 없는 재사용성과 수정을 통한 재사용성은 서로 구별해야 하기 때문에, 각각에 적합한 별도의 기준을 만들어야 한다.

4.1 수정을 통한 재사용성

3장에서 논의한 바와 같이 수정을 통한 재사용의 경우에는 수정성이 중심 품질이 된다. 그리고 수정성을 높이기 위해서는 높은 이해성과 낮은 파급 효과가 필요하다. 이를 BNF 표기법을 이용해서 정리하면 다음과 같다.

<수정성> ::= <이해성> + <파급 효과(ripple effect)> (1)

<이해성> ::= <내부 문서(internal documentation)>
+ <복잡도> (2)

<파급 효과> ::= <자기 완결성(self-containedness)>
+ 응집도 (3)

<내부 문서> ::= 주석(comment) + 자기 설명적 코드
(self-documenting code) (4)

<복잡도> ::= 부품내의 복잡도
+ 부품들 간의 복잡도 (5)

<자기 완결성> ::= 전역(global)/비지역(non-local)자료구조
+ 다른 부품의 호출 (6)

(1)의 식은 수정성이 이해성과 파급 효과를 함께 고려하는 것으로 정의해야 한다는 것을 나타낸다. 높은 이해성과 낮은 파급 효과를 가질때 수정성이 가장 높으며, 낮은 이해성과 높은 파급 효과를 가질때 수정성이 가장 낮다. 이해성은 (2)의 식과 같이 두가지로 나누어서 생각하는데, 하나는 내부 문서이고 또하나는 복잡도이다. 내부 문서가 잘되어 있고 복잡도가 낮을 수록 이해성이 좋은 것으로 판단한다. 파급 효과는 (3)의 식과 같이 자기 완결성과 응집도를 통해서 정의한다. 즉 자기 완결성이 높고 응집도가 높을 수록 수정때문에 발생하는 파급 효과는 적어진다. 자기 완결성이란 다른 외부 부품에의 의존도를 가능한 한 줄이고, 자기 부품내에서 자기의 작업(task)을 수행하는데 필요한 모든 자료와 제어를 최대한 많이 가지는 것을 말한다. 따라서, 완전히 자기 완결적이기 위해서는 다른 부품들의 호출도 제한되어야 한다[4]. 자기 완결성은 (6)의 식과 같이 정의되는데 전역변수 또는 비지역 변수를 적게 사용하고 다른 부품들을 적게 호출할수록 자기 완결성은 높아진다. 따라서 자기 완결성은 모듈 단위의 척도이면서, 다른 부품들과의 관계를 나타내는 척도로도 사용할 수 있다. 바로 이런 점에서 자기 완결성은 결합도와 다르다. 결합도는 모듈들 간의 관계를 나타내지만 모듈 단위로는 결정할 수 없는데, 자기 완결성은 모듈 단위 척도이면서도 다른 모듈들과의 관계를 나타낼 수 있다. 그래서 자기 완결성이 높으면 다른 부품과의 관계가 적고, 자기 완결성이 낮으면 다른 부품으로의 파급 효과가 커지게 된다. 또 응집도는 부품 내부에서의 파급 효과를 나타내는 척도가 된다. 즉 응집도가 높을 수록 부품 내부에서의 파급 효과가 적어진다. 그리고 이러한 면에서 응집도는 자기 완결성과 구별된다. 즉 자기 완결성은 부품내부 보다는 부품의부에 미치는 파급효과를 나타내고, 응집도는 부품내부적인 파급효과를 나타내는 것이다. 제어폭(fan-out)역시 다른 부품에의 영향을 나타내는 척도로 사용할 수 있는데, 여기서는 자기 완결성에서 한 부품이 다른 부품들에 미치는 영향을 감안했기 때문에, 척도로 사용하지 않는다. 프로그램의 이해성은 가독성(readability)과 깊은 관계가 있다. [16]에 의하면, 가독성은 두가지 방법으로 높일 수 있는데, 하나는 소스 코드 내에 들어가는 내부 문서이고, 또하나는 소스 코드 밖의 외부 문서(external document)이다. 본 연구에서는 소스 코드의 재사용성을 측정하는 것이 주 목적이므로 외부 문서에 대해서는 고려치 않는다. 내부 문서는 자기 설명적 코드와 주석으로 나누어진다. (4)의 식은 이를 나타낸 것이다. 코드내에 주석이 많고 코드의 의미가 명확할수록 내부 문서가 잘되어있고 이해성이 높아진다. (5)의 식은 한 부품의 복잡도를 결정하기 위해서는, 그 부품 내부의 복잡도와 그 부품이 다른 부품들과 맺는 관계에서의 복잡도를 함께 고려해야 한다는 것을 의미한다.

사례 연구에서의 계산을 위해 다음의 식으로 쓸 수 있다.

$$WR \cong M = U + R \quad (7)$$

U = ID + COM (8)

R = SC + COH (9)

여기서

WR = 수정을 통한 재사용성(white box reusability)

M = 수정성

U = 이해성

R = 파급 효과

ID = 내부 문서

COM = 복잡도

SC = 자기 완결성

COH = 응집도

4.2 수정 없는 재사용성

다음으로 수정없는 재사용성의 측정 기준을 보도록 한다. 역시 3장에서 논의한 바와 같이 수정 없는 재사용에서는 정보 은닉이 핵심이 된다. 이 정보 은닉을 BNF 표기법을 사용해서 식으로 정리하면 다음과 같다.

<정보 은닉> ::= <자기 완결성> + 응집도
+ 매개 변수 (10)

정보 은닉은 한 모듈내에 하나의 설계 결정(design decision)을 갖고 있고, 다른 모듈들이 이 설계 결정에 대해 알지 못하게 하는 것을 의미한다[17]. 설계 결정의 예를 들어 보면, 복잡한 자료 구조의 구현 또는 외부 장치와의 인터페이스에 대한 세부 사항의 구현등을 들 수 있다. [18]에 의하면, 모듈 단계의 정보 은닉을 측정하는데는 2가지 요소가 있어야 하는데 첫번째는, 그 모듈이 하나의 설계 결정을 가지고 있는지에 대한 것이고, 둘째는 그 인터페이스가 최소화 되어 있는지에 대한 것이다. 이 두가지 요소에서 다음의 공리를 이끌어 내고 있다.

> 하나 이상의 설계 결정을 은닉하고 있는 모듈은, 하나의 설계 결정만 은닉하고 있는 모듈보다 정보 은닉이 더 나쁘게 되어 있다.
>
> 인터페이스 내에 그 설계 결정에서 필요로 하지 않는 개체들을 많이 가지고 있을 수록, 그 모듈의 정보 은닉은 더욱 더 나쁘게 된다.

Rising은 이 공리에 의해 객체 기본 언어(object-based language)에 적용할 수 있는 모듈 단계의 정보 은닉 척도를 다음과 같이 제안하고 있다.

Information Hiding = 2 * [0|1]
+ Number of visible variables

첫째항의 경우, 하나의 설계 결정이 모듈내에 캡슐화(encapsulation)되어 있으면 0이고, 그렇지 않으면 1이 된다. 두번째 항은 전역 변수의 갯수를 의미한다. 이 식은 각각의 모듈의 정보 은닉의 정도를 순서적(ordinal)으로 나타낼 수 있게 한다. 본 논문에서는 객체 기본 언어보다 더 일반적으로 사용되고 있는 명령형 언어(imperative language)에서의 정보 은닉의 정도를 측정하려 한다. 그리고 Rising의 식의 첫째 항처럼 주관적 판단에 의존하는 것이 아니라, 더 명확한 속성들을 사용해서, 자동적으로 측정할 수 있게 하기 위해 위 식을 (10)의 식으로 변형했다. 즉 식(10)의 자기 완결성은 정보 은닉과 밀접한 관계를 가진다. 그리고 응집도 역시 정보 은닉의 개념과 관련이 있다[14]. 또한 매개 변수는 외부에서 그 부품을 사용하기 위해서 알아야 하는 정보이며, 인터페이스의 최소화를 측정하기 위한 척도로서의 의미를 갖고 있다. 그리고, Rising의 척도에서 불필요한 개체로서 계산되는 전역 변수등은 자기 완결성에서 함께 계산된다.

사례 연구에서의 계산을 위해 식으로 나타내면 다음과 같다.

BR ≈ IH = SC + COH + PA (11)

여기서

BR = 수정 없는 재사용성(black box reusability)

SC = 자기 완결성

COH = 응집도

PA = 매개 변수(parameter)

이 식에서 보면 (11)식의 정보 은닉과 (9)식의 파급효과에 모두 자기 완결성과 응집도가 공통적으로 들어가 있음을 알 수 있다. 이는 정보 은닉과 파급 효과가 서로 동일한 개념은 아니지만, 매우 밀접한 관련이 있기 때문이다. 즉 정보 은닉과 파급 효과는 서로 배치되는 개념이 아니기 때문에 정보 은닉이 잘 되어 있을 수록, 그 모듈의 수정으로 인해 발생하는 파급 효과는 그만큼 더욱더 적어지게 된다. 물론 자기 완결성과 응집도가 미치는 상대적 영향의 정도는 정보 은닉과 파급 효과에서 서로 다를 수 있다.

4.3 재사용성 측정을 위한 각 척도들의 계산 방법

지금까지 재사용성 측정을 위해 필요한 기준들과 그 구체적인 척도등을 알아 보았다. 이제 실제의 측정을 위해, 위의 척도들을 계산하기 위한 방법을 구체적으로 정리해 보도록 한다. 단 각각의 척도들은 계산방법과 결과값들의 의미가 모두 다르기 때문에 그 값들을 그대로 사용해서 비교한다는 것은 의미가 없다. 따라서 비교를 위

해 여러 척도들에 모두 100 점의 점수를 만점으로 주고, 각각의 특성에 따라 필요한 경우 몇개의 등급을 나누어 적절한 평가를 하도록 한다. 높은 점수를 받을 수록 좋은 품질을 가지고 있음을 나타낸다. 그리고 한가지 분명하게 할것은, 이러한 측정에 의해 구한 값들은 순서적(ordinal)의미를 갖는 것으로 상대적 비교를 위한 측정이라는 것이다.

4.3.1 복잡도

소프트웨어 부품의 복잡도는 부품 내부의 복잡도와 부품들간의 복잡도로 나누어 진다. 부품 내부의 복잡도는 코드 복잡도(code complexity) 척도로 측정하는데, 여기에는 LOC(lines of code), Halstead의 software science, McCabe의 cyclomatic complexity등이 주로 사용된다. 부품들간의 복잡도는 구조 복잡도(structure complexity) 척도로 측정하는데, 여기에는 Yourdon과 Constantine의 공유도(fan-in)/제어폭(fan-out) 척도와, Henry와 Kafura의 정보 흐름(information flow) 척도 등이 있다. 이들은 프로그램의 복잡도의 서로 다른 측면들을 나타낸다. 그리고, 두가지 복잡도를 모두 고려하는 [19]의 정보 흐름 척도의 혼합형(hybrid)도 있다. 이 혼합형의 식을 보면 다음과 같다.

$$HCp = Cip * (fan\text{-}in * fan\text{-}out) ** 2$$

HCp는 프로시주어 p의 복잡도를 말하며, Cip는 프로시주어 p의 내부 복잡도를 의미한다. 공유도(fan-in)는 주어진 한 부품을 호출하는 다른 부품들의 갯수이고, 제어폭(fan-out)은 주어진 한 부품에 의해 호출되는 부품들의 갯수이다. Cip는 코드 복잡도 척도들 중에서 아무거나 사용해도 된다. 이 혼합형은 복잡도의 두 측면을 모두 반영한다는 의미에서 더 정확한 복잡도 척도라고 할 수 있다. 그런데, 최근의 연구에서 복잡도를 나타내는데 있어서 공유도는 그리 중요하지 않다는 지적이 많이 있다[20]. 따라서 본 연구에서는 위의 혼합형 대신 McCabe의 cyclomatic complexity와 제어폭을 사용해서 두가지 측면의 복잡도의 척도로 사용하도록 한다. Cyclomatic complexity number를 계산하는 식은 다음과 같다.

$$V(G) = e - n + 2$$

V(G)가 의미하는 것은 독립적인 경로의 갯수이다. 그런데, 이 cyclomatic complexity는 몇가지 단점을 갖고 있다. [4]는 프로그램이 매우 복잡하고 이해하기 어려운 경우라도 데이타 중심의(data driven) 프로그램인 경우 V(G)의 값이 매우 낮게 나오기 때문에 이러한 유형의 프로그램의 복잡도를 잘 측정하지 못하며, 또 중첩된 반복(nested loop)과 중첩되지 않은 반복에 똑같은 가중치를 두고 있기 때문에 중첩된 반복이 더 복잡하다는 것을 반영하고 있지 못한다고 지적하고 있다. 또 [20]는, 이 척도는 순차적 문장의 복잡도는 무시하고 있으며, 서로 다른 종류의 제어 — 예를 들어서, 반복 구조와 IF_THEN_ELSE 문장, 또는 'case'와 중첩된 IF_THEN_ELSE 문장등 — 를 구별하지 않고 있다고 지적하고 있다. 하지만, 이러한 단점에도 불구하고 이 척도는 결정(decision)과 분기(branch)에 기초하고 있기 때문에, 설계와 프로그래밍의 논리 패턴과 일관성을 갖고 있으므로 실제적으로 많이 사용되고 있다[20]. McCabe는 하나의 프로그램 부품의 복잡도의 합리적 상한값으로 V(G)의 값 10 을 제시하고 있다[21]. 본 논문에서는 [22]의 extended cyclomatic complexity number($V(G)^+$로 표기)를 사용하도록 한다. 이는 cyclomatic complexity에 복합 조건문(예: IF <condition> AND <condition>)으로 인해 발생하는 추가적 복잡도를 반영할 수 있는 장점이 있다. 코드 복잡도의 또다른 척도들을 보면, LOC의 경우는 최근 연구에 의하면 복잡도와 LOC의 관계가 정비례 관계가 아니라 곡선 관계(curvilinear relation)임이 밝혀지고 있고, software science는 많은 비판을 받고 있으며 복잡도를 측정하는데 있어서의 유용성을 입증받지 못하고 있다[20]. 제어폭의 경우는 인간의 심리학적인 한계에 근거하여 7 ± 2 를 기준으로 제시[11] 하고 있다. 이러한 그동안의 연구 결과를 토대로 복잡도를 계산하기 위한 새로운 척도를 다음과 같이 제안한다.

$$ICp = ((V(G)^+ + fan\text{-}out) / 4) ** 2 \quad (12)$$

여기서

$V(G)^+$ = extended cyclomatic complexity number

여기서 ICp는 프로시주어 p의 종합 복잡도(Integrated Complexity)를 말하며, extended cyclomatic complexity number 즉 $V(G)^+$에 제어폭을 더해서 4로 나눈후 그 결과를 제곱하는 방법으로 계산한다. 두개의 요소를 더한 이유는 부품내의 복잡도와 부품간의 복잡도를 함께 고려하기 위한 이유 때문이고, 제곱을 하는 이유는 extended cyclomatic complexity number와 제어폭 모두 그 수치가 커질 수록 복잡도가 기하 급수적으로 증가한다는 것을 나타내기 위해서이고, 나누기 4를 한 이유는 기준치(즉 $V(G)^+$의 경우 10, 제어폭의 경우 7)내의 결과값의 차이를 적게 한다는 의미와, 각각의 기준치

의 단순 합인 17과, 그 기준치로 (12)의 식과 같이 계산한 결과 값 18.0625가 유사한 값을 갖는다는 데서 직관적으로 선택한 것이다. 이 종합 복잡도에서의 기준치는 $V(G)^+$의 기준치 10과 제어폭의 7을 함께 반영한 경우의 18.0625의 값으로 한다. 즉 종합 복잡도에서 18.0625 이하의 값을 갖는 부품은 적절한 복잡도를 갖는 것이고, 그 이상의 값인 경우는 모듈의 분할 혹은 설계의 변경을 통해 종합 복잡도를 낮추는 것이 좋다고 본다.

그런데 여기에서는 복잡도 자체를 다루는 것이 아니고, 재사용성을 측정하기 위한 하나의 요소로서 복잡도를 다루는 것이며, 이 경우 종합복잡도와는 달리 높은 점수일 수록 좋은 품질을 나타내고 있으므로 (8)의 식의 COM은 다음의 (13)의 식과 같은 방법으로 계산한다.

$$COM = \begin{cases} 100 - ICp \\ 0, \ ICp\text{의 값이 } 100\text{보다 큰 경우} \end{cases} \quad (13)$$

여기서 ICp가 100 이상의 값을 갖는 경우에는 COM의 값이 음수가 되는데, 이때는 0으로 계산한다. 이 COM의 값이 100에 가까울 수록 복잡도가 낮고, 따라서 그만큼 더 이해성이 좋음을 나타낸다.

4.3.2 응집도

모듈 응집도를 객관적이고, 자동적으로 측정할 수 있는 방법을 개발하기 위한 여러가지 시도가 있었다. 그중 가장 대표적인 것을 보면 다음 두가지를 들 수 있다[23].

첫째, 연관성(association)을 기초로한 접근 방법으로, 모듈의 코드내의 데이타 종속성에 관한 법칙(rule)들로서의 처리 요소(processing element)들 사이의 연관성의 개념을 형식화(formalize)하는 것이다. 이 방법은 Lakhotia에 의해 개발되었으며, 코드 단계 정보에 대한 분석을 필요로 한다. 둘째, 단편(slice)를 기초로 한 접근 방법으로, [24]에서 제안한 방법이다. 여기서는 모든 출력 변수에 대한 단편에 관계되는 데이타 토큰들 사이의 연결(connection)을 기준으로 함수적(functional) 응집도를 측정한다. 이 방법도 코드 단계의 정보를 분석해야 한다는 것은 같다. 즉 두가지 방법 모두 설계상의 응집도 측정이 아니라 소스 코드상의 응집도를 측정한다.

본 연구에서는 단편을 기초로 한 접근 방법으로 응집도를 측정하도록 한다. [23]에 따르면, 연관성을 기초로 한 접근 방법은 모듈내의 연결(connection)들의 갯수의 변화를 잘 반영하지 못하지만, 단편을 기초로 한 접근 방법은 이를 잘 반영한다. 또 단편을 기초로한 접근 방법은 핵심적 부품의 상대적 갯수, 고립된 부품의 상대적 개수등을 잘 반영하지만, 연관성을 기초로 한 접근방법은 그렇지 못하다.

그런데, 단편을 기초로 한 접근 방법은 응집도의 여러 단계(함수적, 연속적,통신적등등)들을 측정하는 것이 아니라, 함수적 응집도만을 측정한다. 하지만 어떤 의미에서는, 여러 단계의 응집도들은 함수적 응집도의 또다른 단계이고, 따라서 가장 낮은 응집도인 동시적(coincidental) 응집도를 갖는 모듈은 이 방법의 3가지 측정값이 거의 0에 가깝게 나올 것이라고 기대 할 수 있다[24].

[24]에서 제안하는 응집도 측정 방법은 다음과 같다. 여기에서는 한 모듈의 데이타 단편(data slice)에 기초를 둔 "glue" 데이타 토큰 또는 "adhesive" 데이타 토큰의 상대적 갯수로 함수적 응집도를 계산한다. 한 변수에 대한 데이타 단편은, 그 변수와 종속(dependence)관계를 갖는 일련의 데이타 토큰들이다. 이 데이타 단편들은 그 프로시주어의 각각의 출력(output)에 대해서 구해진다. 이 출력물(output object)은 한 프로시주어의 함수성(functionality)을 나타내는 것으로, 출력의 경우를 보면 출력 인자(output parameter), 전역 변수의 수정, file에의 출력등이 있을 수 있다. glue 토큰은 하나 이상의 데이타 단편에서 공통적으로 나타나는 데이타 토큰이다. 한 모듈내의 모든 데이타 단편에서 공통적인 glue 토큰이 super-glue 토큰이다. glue 토큰들은 여러 단편들을 결합시키는 효과를 갖기 때문에, glue 토큰과 super-glue 토큰들의 분포는 개별적인 단편들이 얼마나 강하게 서로 결합되어 있는지를 나타낸다. 개별적인 glue 토큰들은 그들이 결합시키는 단편들의 갯수에 따라 응집도에 영향을 미치는 정도가 달라질 수 있다. 한 데이타 토큰의 adhesiveness 는 그 데이타 토큰을 공유하는 데이타 단편들의 갯수로, 한 프로시주어 내에서 그 'glue'의 상대적 강도를 나타낸다. 함수적 응집도를 측정하기 위한 세가지 도구가 있는데, 그것은 WFC(Weak Functional Cohesion), SFC(Strong Functional Cohesion), A(Adehesiveness)이다. WFC는 한 프로시주어내의 모든 토큰 개수에 비한 glue 토큰의 비율이다. SFC는 한 프로시주어내의 모든 토큰 갯수에 비한 super-glue 토큰의 비율이다. A는 최대로 가능한 adhesiveness에 비한 adhesiveness양의 비율이다. 함수적 응집도를 구하는 예를 보면 예1과 같다. SFC는 함수적 응집도의 원래의 정의(모듈내의 모든 요소들은 한가지 문제와 연관된 작업 수행에 기여해야 한다[11].)에 가장 가까운 측정 도구이고, WFC는 함수적 응집도 이외의 단계에서의 응집도를 나타내는데 적합한데 특히 최하의 단계인 동시적 응집도를 나타내는데 적합하다. A

는 이 세가지 측정 도구중에서 가장 민감하게 나머지 5개 단계의 응집도를 나타내고 있다.

예 1 소스코드에서 함수적 응집도 계산하는 방법

sum	max	avg	statement
			void Sum_Max_Avg
1	1	1	(int arr[],
1	1	1	int n,
1		1	int *sum,
	1		int *max,
1		1	float *avg)
			{
1	1	1	int i;
2		2	*sum = 0;
	3		*max = arr[0];
5	5	5	for(i = 0;i < n;++i){
4		4	*sum = *sum + arr[i];
	3		if(arr[i] > *max)
	3		*max = arr[i];
			}
3		3	*avg = *sum / n;
			}

tokens = 29
glue tokens = 19
superglue tokens = 8

WFC(Sum_Max_Avg) = 19 / 29 = 0.66
SFC(Sum_Max_Avg) = 8 / 29 = 0.28
A(Sum_Max_Avg) = (11 * 2 + 8 * 3) / (29 * 3) = 0.53

예 1 에서와 같이 WFC(p)와 SFC(p)와 A(p)는 하나의 프로시주어 내의 단편과 데이타 토큰들의 갯수, 그리고 glue 토큰과 super-glue 토큰의 갯수만을 이용해서 계산해낼 수 있다. 프로시주어 p에 대한 이 세가지 측정 도구의 결과 값을 크기 순으로 나열해 보면 다음과 같다.

$$SFC(p) <= A(p) <= WFC(p)$$

이 3가지 측정도구의 성격을 이용해서 응집도의 척도를 계산하는 방법을 다음과 같이 정리한다.

$$COH = \begin{cases} SFC(p) * 100, & SFC(p)의\ 값이\ 1.000000인\ 경우 \\ WFC(p) * 100, & WFC(p)의\ 값이\ 0.000000인\ 경우 \\ A(p) * 100, & 그\ 이외의\ 경우 \end{cases} \quad (14)$$

여기서 SFC(p)의 값이 1.000000인 경우는 모듈 p의 응집도가 함수적 응집도인 경우이고, WFC(p)가 0.000000인 경우는 응집도중 가장 낮은 단계인 동시적 응집도(coincidental cohesion)일 때이다. 그외인 경우에는 가장 민감하게 응집도를 나타내는 A(p)로 응집도를 나타낸다. (14)식에서는 응집도의 점수화를 위해 100을 곱한 값을 COH의 값으로 사용한다.

4.3.3 내부 문서

내부 문서는 (4)의 식에서와 같이 자기 설명적 코드와 주석으로 이루어 지는데,자기 설명적 코드가 갖는 특징으로는 의미있는 변수 이름의 사용, 코딩 표준(coding standard) 준수등이 있다. 내부 문서의 품질을 정확히 측정하기 위해서는 의미(semantic) 정보와 구문(syntax) 정보를 모두 잘 파악해야 하는데 의미 정보를 정확하게 파악할 수 있는 방법은 아직 개발되어 있지 않다. 따라서 여기서는 구문적 분석만을 사용하도록 한다. 우선 변수 이름이 의미가 있게 만들어 진것인지에 대한 척도를 보자. 이에 대해서는 기존에 만들어져 있는 척도가 없으므로 적절한 척도를 만들 필요가 있다. [25]는 변수 이름의 길이가 너무 짧으면 가독성이 떨어진다고 지적하고 있고, 그 예로 Fortran77이 변수 이름에 최대 6 자의 문자까지 밖에 사용을 못했기 때문에 의미가 함축된 이름을 사용하는 것이 거의 불가능 했음을 지적하고 있다. 이 예를 참고로 해서 본 연구에서는 6 자 이상의 길이를 갖는 변수는 의미있는 이름의 변수로 본다. 이의 척도로는 한 부품내에서 6 자 이상의 변수가 전체 변수의 갯수에 비해 어느 정도의 비중이 되는 지로 한다. 코딩 표준의 준수 여부에 대해서는 구문적으로 측정하기가 어렵고, 또 다양한 변이가 있기 때문에 여기서는 고려하지 않기로 한다. 다음으로 주석이 프로그램의 가독성에 미치는 영향을 측정하기 위해서, 주석 대 코드의 비율(comment-to-code ratio)을 계산한다. 이는 주석이 있는 라인 수를 주석이 아닌 라인(non-blank line) 수로 나누어서 구하는데, C 프로그램에서는 보통 1.0 이상이면 좋고, 적어도 0.8은 넘어야 한다고 한다[16]. 역시 보편적 인정을 받는 기준 수치는 아직 알려져 있지 않다. 그런데 이 주석을 쓰는 방법에도 다양한 변이가 있다. 즉 코드 옆에 주석을 달 수도 있고, 함수의 머리부(header)에 달 수도 있고, 또 화일 단위로 전체적인 주석을 화일의 선두에 모두 기술해두고, 각각의 함수내에서는 간단하게만 주석을 다는 등 다양한 방법이 있다. 이 때문에,한 화일 내부에 여러개의 함수가 있는 경우 각각의 함수 내부의 주석만 고려하는 방법은 정확성이 떨어질 가능성이 많다. 따라서 본 연구에서는 주석대 코

드 비율을 화일 단위로 하고, 이 비율을 그 화일내에 있는 함수에 모두 동일하게 적용하도록 한다. 이상의 기준을 근거로 내부 문서의 척도는 다음과 같이 계산한다.

$$ID = MV + CCR \quad (15)$$

MV = 함수내에서 6자 이상의 이름의 변수의 비율 * 100 / 2 (16)

CCR = 주석 대 코드 비율 * 100 / 2 (17)
50, 주석 대 코드 비율의 값이 1.0 이상인 경우

MV와 CCR 모두 100을 곱한 후에 2로 나눈 것은 내부 문서의 총점이 100점이기 때문에 각각의 만점을 50점으로 하기 위한 것이다. 그리고 CCR의 경우는 1.0 이상의 값이 가능한데, 이경우는 코드 보다 주석이 더 많은 경우로 코드 이해에 충분하다고 판단 되어 CCR에서의 만점인 50점으로 계산한다.

4.3.4 자기 완결성

자기 완결성의 척도는 전역/비지역 변수와 다른 함수의 호출의 두가지로 이루어진다. 전체 변수 중에서 전역변수와 비지역 변수가 있는지, 그리고 읽기 전용(read only)인지, 또는 쓰기(write)가 있는지의 여부와, 다른 함수를 호출하는 경우에는 그 호출하는 모듈의 갯수(즉 제어폭)가 몇개인지를 척도로 계산한다. 자기 완결성을 계산하는 방법을 다음과 같이 제안한다.

SC = (18)

- 100점, 모든 변수가 지역 변수(local variable)이고, 제어폭이 0 인 경우
- 91점, 모든 변수가 지역 변수이고, 제어폭이 7 이하인 경우
- 82점, 모든 변수가 지역 변수이고, 제어폭이 8 이상인 경우
- 73점, 비지역 변수/전역 변수가 모두 읽기 전용이고, 제어폭이 0 인 경우
- 64점, 비지역 변수/전역 변수가 모두 읽기 전용이고, 제어폭이 7 이하인 경우
- 55점, 비지역 변수/전역 변수가 모두 읽기 전용이고, 제어폭이 8 이상인 경우
- 45점, 비지역 변수만 쓰기(write)이고, 제어폭이 0 인 경우
- 36점, 비지역 변수만 쓰기이고, 제어폭이 7 이하인 경우
- 27점, 비지역 변수만 쓰기이고, 제어폭이 8 이상인 경우
- 18점, 전역 변수에도 쓰기이고, 제어폭이 0 인 경우
- 9점, 전역 변수에도 쓰기이고, 제어폭이 7 이하인 경우
- 0점, 전역 변수에도 쓰기이고, 제어폭이 8 이상인 경우

여기에서 전역변수와 비지역변수, 지역변수를 구분한 이유는 전역 변수는 비지역 변수보다 더 많은 모듈에 영향을 미치고, 지역변수는 다른 모듈로 영향을 미치지 않는 차이점을 가지고 있기 때문이다. 또 읽기전용과 쓰기를 구분한 이유도 쓰기의 파급효과가 훨씬 크기 때문이다. 그리고 제어폭의 경우에는 [11]의 기준숫자 7을 기준으로해서 7이하와 8이상을 구분하고, 제어폭이 0인 경우는 제어폭이 있는 경우와 구별하기 위해 구분하고 있다.

4.3.5 매개 변수

매개 변수가 많을 수록 그 부품에 대해 알아야 할 정보가 많아진다. 따라서 매개 변수가 적을 수록 정보 은닉에 유리하다. 그 계산 방법은 다음과 같다.

$$PA = 100 - (\text{매개변수의 갯수} * 10) \quad (19)$$
0, 매개 변수가 11개 이상인 경우

매개 변수의 경우는 [11]의 기준숫자 7을 기준으로 하기가 어렵다. 왜냐하면 실제 프로그램의 매개 변수는 대부분 7개 이하이기 때문이다. 따라서 매개 변수의 갯수를 그대로 사용해서 계량화한다. 매개 변수의 갯수가 11개 이상이 되어 PA의 값이 음수가 되는 경우는 역시 0으로 계산한다. 매개 변수에서는 매개 변수의 자료구조의 유형도 고려해야 하는데, 왜냐하면 하나의 필드(a single field), 배열, 구조체(struct), 유니온(union)등은 각각 다른 특성을 갖고 있기 때문이다. 특히 구조체와 유니온의 경우는 서로 이질적인 자료들이 섞여 있을 수 있고, 구조체와 유니온내에 다시 구조체 또는 유니온이 포함될 수 있기 때문에 더욱 복잡해진다. 따라서 이들을 각각의 구조체 또는 유니온 내에 포함되어 있는 원소 단위로 세는 경우에는 지나치게 복잡해질 수 있다. 본 연구에서는 이러한 복잡함을 피하기 위해 매개 변수의 자료 구조의 유형과는 관계없이 모든 단위 매개 변수를 1개로 세어 계산한다.

5. 사례 연구

4장의 재사용성 측정 방법과 기준을 적용해서, 실제 C 소스 코드의 각각의 함수들의 재사용성을 측정해보도록 한다.

5.1 사례 연구 환경

사례 연구의 대상이 된 소프트웨어 시스템은 모두 128,307라인(헤더 화일 포함)의 C 언어 프로그램으로 2,005개의 함수로 구성되어 있다. 이 시스템은 표 2 와 같이 3개의 그룹으로 나눌 수 있다. 이 시스템은 국내의 한 PC통신 서비스 회사의 서비스 시스템의 일부이다.

표 2 재사용성 측정 대상 소프트웨어 시스템의 특징

명 칭	함수 갯수	특 징
SVC	1,177개	DB검색 시스템, 전자 우편/채팅등 통신용 시스템, 홈 쇼핑과 같은 주문 거래 시스템등의 단위 서브 시스템으로 구성
COMM	448개	SVC의 여러 단위 서브 시스템에서 공통으로 사용하는 함수들의 집합
LIB	380개	SVC와 COMM에서 필요에 의해 호출하는 사용자 정의 라이브러리 함수들의 집합

표 3 재사용성 측정을 위해 사용한 도구

이 름	특 징	사용 목적
csize[1]	C 프로그램의 크기를 측정하는 도구	식(17)의 CCR을 계산하기 위해 사용
Mas[2]	유지보수와 관련된 여러 척도들을 구하기 위한 도구	식(12)의 V(G)+, 식(16)의 MV, 식(12,18)의 제어폭, 식(19)의 PA를 계산하기 위해 사용
FUNCO[3]	C 프로그램에서 함수적 응집도를 측정하는 도구	식(14)의 COH를 계산하기 위해 사용

표 2 의SVC의 함수들과 COMM의 함수들은 함께 합쳐져서 여러 단위 서브 시스템을 구성하는데, 각각의 단위 서비스들에 모두 공통적으로 들어가는 함수들만 모아 놓은 것이 COMM 그룹이고, 그외에 단위 서비스에서 고유하게 필요한 함수들을 모아 놓은 것이 SVC 그룹이다. SVC의
모든 단위 시스템에서, COMM내의 함수들의 대부분을 반드시 호출한다는 점에서, 필요에 의해 일부 함수들이 호출되는 LIB와 COMM은 다르다. 하지만 COMM은 모든 SVC의 서브 시스템에서 호출되는 함수들이라는 면에서 LIB와 공통점을 갖는다. 즉 COMM은 LIB와 유사한 성격을 갖는 일반 함수들이라고 할 수 있다.

이 3가지 그룹의 소스 코드의 재사용성을 측정하기 위해 사용한 도구는 표 3과 같다. 수정을 통한 재사용성과 수정 없는 재사용성을 측정하기 위한 4장의 여러 식들은 이 도구들을 사용해서 그 값들을 계산해낼 수 있다. 단 식(18)의 함수내의 변수들의 사용 유형(유형 1: 모든 변수가 지역 변수인 경우, 유형 2 : 비지역 변수/전역 변수가 읽기 전용인 경우, 유형 3 : 비지역 변수에 쓰기인 경우, 유형 4 : 전역 변수에 쓰기인 경우)은 이들 도구들을 사용해서 알 수가 없기 때문에 수작업을 통해 판별했다.(물론 이 역시 별도의 도구를 구현하여 자동화할 수 있다.)

이 사례 연구는 sun workstation에서 Solaris 2.4 환경하에서 수행했다.

5.2 측정 결과

먼저 위의 도구들을 사용해서 구한 척도들의 결과 값의 평균을 보면 표 4 와 같다. 표 5 는 수작업으로 이 소스 코드의 함수들의 변수 사용 유형을 분석한 것이다. 이 두개의 표를 통해서 이 소프트웨어 시스템의 전체적인 특징을 살펴 볼 수 있다. Extended cyclomatic complexity number인 V(G)+를 보면 COMM과 LIB가 상당히 유사한 값을 보여 주고 있고, SVC는 이 두개 그룹보다 상당히 큰 값을 나타내고 있다. 제어폭의 경우는 LIB가 SVC의 절반 이하의 값을 갖는데 이는 LIB의 특성이 LIB내의 함수들을 다른 함수들이 호출해서 사용되는 것이기 때문에 LIB의 함수들은 다른 함수들의 호출의 대상이 되는 경우가 많고, 그만큼 자신이 다른 함수를 호출하는 경우는 적게 된다. COMM역시 SVC에서 대부분 호출하는 함수들로 구성되어 있기 때문에 LIB만큼은 아니지만 제어폭이 작다. 이 두가지 척도로 미루어 볼때 LIB의 종합 복잡도가 가장 낮다고 판단할 수 있다. 응집도의 경우 LIB가 가장 우수한데 이는 LIB의 함수들은 주로 작은 단위의 작업들 위주로 잘 분리되어 있기 때문으로 보인다. 함수내에서 6자 이상의 변수를 사용하는 비율은 COMM 그룹이 가장 높고, LIB가 가장 낮다. SVC는 거의 평균에 가까운 값을 갖고 있다. 주석 대 코드 비율을 보면 3개 그룹 모두 지나치게 낮은 값을 갖고 있는데, 이는 이 시스템이 거의 주석을 갖고 있지 않다는 것을 의미한다. 전체 평균이 0.056458이라면 이는 소스 코드내에 공백이 아닌 코드 100라인에 주석이 있는

1) csize는 Christopher Lott가 1994년에 개발한 도구로, C 프로그램의 전체 라인 수, 공백 라인 수, 코드 라인 수, 주석 라인 수 등을 계산한다. 본 연구에서는 편의상 csize의 결과 값 출력시 주석 대 코드 비율을 함께 출력하도록 소스의 일부를 수정하여 사용했다.

2) Mas는 1992년에 Idaho대학에서 Hewlett-Packard Corporate Engineering의 지원을 받아 개발한 도구로 extended cyclomatic complexity number, Halstead의 Software Science, 제어폭등 여러가지 척도들을 계산한다. 본 연구에서는 의미있는 변수의 기준을 6자 이상의 이름을 갖는 것으로 하기 위해 소스의 일부를 수정하여 사용했다. csize와 Mas는 모두 Christopher Lott가 http://www.qucis. queensu.ca/Software-Engineering/Cmetrics.html에 수집해둔 도구들 중의 일부이다.

3) FUNCO는 Colorado 주립 대학의 Byung-Kyoo Kang이 [24]를 근거로 1995년에 구현한 도구로 C 프로그램의 함수의 함수적 응집도를 측정한다.

표 4 측정 도구를 사용해 구한 각 척도들의 평균 값

	V(G)*	제어폭	응집도[4]	의미있는 변수비율(%)	주석 대 코드 비율	매개 변수갯수
SVC 평균	10.598980	13.711130	0.582819	53.780732	0.045964	2.799490
COMM 평균	7.377232	9.910714	0.683082	63.816986	0.062388	1.732143
LIB 평균	7.618421	6.118421	0.814917	43.389972	0.081974	2.205263
전체 평균	9.314214	11.422942	0.649210	54.053925	0.056458	2.448379

표 5 변수 사용 유형

	유 형 1 함수갯수/비율(%)	유 형 2 함수갯수/비율(%)	유 형 3 함수갯수/비율(%)	유 형 4 함수갯수/비율(%)
SVC	91 / 7.73	507 / 43.08	319 / 27.10	260 / 22.09
COMM	41 / 9.15	269 / 60.04	74 / 16.52	64 / 14.29
LIB	78 / 20.53	179 / 47.11	102 / 26.84	21 / 5.53
전체	210 / 10.47	955 / 47.63	495 / 24.69	345 / 17.21

라인은 5 내지 6 라인 정도에 불과함을 나타낸다. 이는 그만큼 소스 코드를 이해하는데 어려움이 많다는 뜻이다. 매개 변수 갯수를 보면 COMM이 가장 적고 SVC가 가장 많으며, LIB는 중간 정도의 값을 보여주고 있다. 전체 평균으로 매개변수의 갯수는 3개 이하로 많은 편은 아니다. 함수 내에서 변수를 사용하는 유형을 보면 부작용(side-effect)을 일으키는 유형 3, 유형 4의 경우 SVC 그룹이 49.19 %로 가장 많고, 부작용이 없는 유형 1과 유형 2의 경우는 COMM과 LIB가 각각 69.39 %와 67.62 %로 SVC보다 많다. 따라서 COMM과 LIB가 더 부작용이 적음을 알 수 있다.

이들 각각의 척도들로 4장에서 제시한 방법을 기준으로 재사용성을 측정하기 위한 점수들을 산출할 수 있다. 즉 V(G)* 와 제어폭의 값으로 식(12)와 식(13)의 식을 사용하면 복잡도 COM의 값을 구할 수 있고, 응집도 COH, 내부 문서 ID등도 역시 4장의 식들을 이용해서 그 값을 모두 구할 수 있다. 이러한 계산 결과를 토대로 각 함수들의 수정을 통한 재사용성과 수정 없는 재사용성의 측정 결과 값을 식(7), (8), (9)와 식(11)을 통해 구해 보면 표 6와 같다.

표 6 재사용성 측정 결과의 평균 값

	수정을 통한 재사용성[5]	수정 없는 재사용성[6]
SVC 평균	195.7417	172.3040
COMM 평균	232.8326	201.9845
LIB 평균	247.1845	216.7917
전체 평균	213.7791	187.3674

기존 소프트웨어 시스템에서 재사용성이 높은 부품을 추출하기 위해서는 재사용성의 측정 뿐만 아니라, 그 측정을 기반으로 한 부품의 선택 기준이 있어야 한다. 본 연구에서 제시한 모델의 각 척도들의 특성을 고려하여

4) 응집도를 측정하는 FUNCO는 출력이 없는 함수의 경우에는 측정하지 못하는 단점이 있다. 이는 [24]이 함수의 출력을 기본으로 응집도를 측정하기 때문이다. 따라서 이런 경우의 함수들은 수작업으로 응집도를 계산했다. 즉 함수적 응집도인 경우는 1.000000, 동시적 응집도인 경우는 0.000000등의 값을 부여했다. 출력이 없는 함수의 갯수는 모두 158개로 전체 함수의 7.88 %를 차지한다. 출력이 없는 함수의 예를 들면 일정 시간 동안 시간 지연만 시키는 함수, 또는 스스로는 아무 출력도 하지 않으면서 다른 함수들을 호출만 하는 함수들을 들 수 있다.

5,6) 수정을 통한 재사용성 WR은 400점 만점, 수정 없는 재사용성 BR은 300점 만점이다. 즉 식(7), (8), (9)에 의해 수정을 통한 재사용성은 WR ≅ ID + COM + SC + COH이고, ID, COM, SC, COH가 모두 100점씩이므로 400점이 만점이 되고, 또 식(11)에 의해 수정없는 재사용성은 BR ≅ SC + COH +PA 가 되고 역시 SC, COH, PA가 각각 100점씩이므로 300점이 만점이 된다.

선택 기준을 제시하면 표 7, 8 과 같다.

표 7 각 척도에서의 선택 기준(최소 기준)

복잡도 COM	내부 문서 ID	자기 완결성 SC	응집도 COH	매개 변수 PA
81.9375	67	64	67	70

표 8 재사용성이 높은 부품 선택 기준(최소 기준)

수정을 통한 재사용성이 높은 부품 선택 기준	수정 없는 재사용성이 높은 부품 선택 기준
279.9375	201

복잡도 COM의 경우에 V(G)+의 10, 제어폭의 7을 기준으로 해서 식(12)와 식(13)에 의해 계산하면 81.9375의 값을 구할 수 있다. 즉 COM의 값이 이 값 이상이면 좋다는 것이다. 나머지 4개의 척도는 100점 만점을 상/중/하 3개의 그룹으로 나눌 때 상에 해당하는 점수대를 기준으로 한 것이다. 자기 완결성 SC의 경우는 점수들이 단속적으로 끊어지기 때문에 67점에 가장 가까운 64점을 기준으로 하고, 매개 변수의 경우도 마찬가지 이유로 70점을 기준으로 했다. 이 기준에 근거하여 재사용성이 높은 부품을 선택하기 위한 기준을 만들 수 있다. 먼저 수정을 통한 재사용성의 경우는 식(7), (8), (9)에 의해 279.9375가 되고, 수정 없는 재사용성의 경우는 식(11)에 의해 201이 된다. 이 선택기준을 각각의 척도 별로 AND 조건을 적용해서 각 척도를 모두 만족해야 하는 것으로 하지 않고, OR에 의해 합산을 한 이유는 재사용성이 종합적인 소프트웨어 품질이기 때문이다. 즉 예를 들어 보면, 복잡도의 점수가 약간 나쁘더라도 내부 문서가 잘 되어 있다면 종합적으로는 복잡도에서의 단점을 보완할 수 있다고 볼 수 있다.

표 9 재사용성이 높은 부품 선택 기준 적용 결과

	함수 갯수	수정을 통한 재사용성이 높은 함수 갯수 / 퍼센트	수정 없는 재사용성이 높은 함수 갯수 / 퍼센트
SVC	1177	204 / 17.33 %	432 / 36.70 %
COMM	448	179 / 39.95 %	247 / 55.13 %
LIB	380	138 / 36.31 %	250 / 65.78 %
전체	2005	521 / 25.98 %	929 / 46.33 %

이 기준을 적용해서 재사용성이 높은 부품을 추출해 보면 표 9와 같다. 전체적으로 25.98 % 의 함수들이 수정을 통한 재사용성이 높은 부품으로 선택이 되고, 또 46.33 % 의 함수들이 수정 없는 재사용성이 높은 부품들로 선택이 되었다. 부품에 대한 이해와 그것을 바탕으로 한 수정 보다는, 정보은닉과 추상화에 의해 부품의 세부적인 구현 정보를 모르고도 재사용 할 수 있는, 수정 없이 재사용 할 수 있는 함수의 개수가 더 많은 것으로 나타났다. 그리고 단위 작업 별로 비교적 구분이 잘 되어 있는 라이브러리 성격의 함수들(LIB와 COMM의 함수들)이 수정을 통한 재사용성과 수정 없는 재사용성 모두에서 우수한 것으로 드러났다.

6. 결 론

본 논문에서는 기존 소프트웨어 시스템을 구성하고 있는 각 부품들의 재사용성을 측정하기 위한 모델을 제안했다. 부품을 재사용하는 방법에는 수정을 통한 재사용과 수정하지 않고 그대로 재사용하는 방법이 있는데, 이 두가지 경우에 있어서 서로 다른 특징을 갖는다. 따라서 제안한 모델에서는 이 두가지를 서로 구분해서 측정할 수 있도록 했다. 또 이 모델을 실제 소스 코드에 적용하여 사례 연구를 수행하였다. 이 사례 연구를 통해서 각 부품의 재사용성을 정량화 하여 측정할 수 있음을 보였다. 또한 이 모델은 재사용성의 측정을 자동화 할 수 있게 하며, 따라서 매우 효과적으로 기존 소프트웨어 부품에서 재사용성이 높은 부품들을 선택할 수 있게 한다. 그리고 각 부품들을 수정을 통한 재사용성과 수정 없는 재사용성이라는 관점에서 그 상대적 비교를 가능하게 한다.

제안한 모델은 소프트웨어 품질의 일정 측면을 나타내는 여러가지 척도들로 구성되어 있는데, 그 각각의 척도들은 서로 다른 가중치를 가질 수 있다. 예를 들어서 수정을 통한 재사용성을 구하기 위한 척도들 중에서 복잡도와 내부 문서는 그 중요성에 있어서 서로 비중이 다를 수 있고, 이는 서로 다른 가중치를 갖게 함으로써 표현할 수 있다. 특히 수정을 통한 재사용성 측정에서의 파급 효과와, 수정 없는 재사용성에서의 정보 은닉의 측정을 위한 척도들은 중복이 되는 부분들이 있는데, 각각의 경우에 서로 다른 가중치를 가질 수 있을 것이다. 풍부한 사례 연구와 그를 통한 자료 축적을 토대로 이 부분에 대한 추가적인 연구가 필요하다. 본 논문에서는 일반적인 가중치를 결정할 수 있을 만큼 자료가 축적되어 있는 것이 아니기 때문에 각 척도들의 가중치를 1로 가정했다.

또한 이 모델을 개선해 나가기 위한 향후 연구가 필요하다. 즉 이 모델에서 제시한 방법과 기준에 의해 재

사용성이 높은 것으로 판단된 부품들을 이용한 소프트웨어 개발과 그렇지 않은 부품들을 이용한 소프트웨어 개발의 비용과 기간과 생산성, 그리고 산출된 소프트웨어의 여러 품질, 유지 보수의 용이성등을 종합적으로 검토해서 이 모델의 장단점을 분석하고 문제점이 드러나는 경우 개선해 나가는 작업이 필요하다. 아울러 재사용에 필요한 비용을 예측하는 모델을 개발하는 방향의 연구도 가능하다. 재사용에 드는 비용은 크게 재사용 가능한 부품을 찾기 위한 검색 비용과 필요한 경우 수정 하는데 따르는 수정 비용으로 이루어진다. 본 논문의 재사용성 측정 모델은 소프트웨어 부품을 이용해서 실제로 재사용하는데 소요되는 비용을 더 높은 정확도를 갖고 예측하기 위해서 이용할 수 있는데, 역시 향후의 연구과제가 될 수 있다.

재사용을 통한 품질 향상과 생산성 증가등의 효과는 코드 단계의 재사용 보다 상위 단계 즉 설계, 분석 단계에서의 재사용이 더욱 크다. 이들 분석,설계 단계의 산출물들의 재사용성을 측정하고, 그 측정을 이용해서 재사용성이 높은 분석과 설계를 수행하는 부분에 대해서도 추가적인 연구가 필요하다. 이러한 연구를 위해서는 분석, 설계 단계에서의 산출물의 정형화 및 표준화 작업에 관한 연구가 선행되어져야 한다.

참 고 문 헌

[1] C.W.Krueger, "Software Reuse," *ACM Computing Surveys*, Vol.24, No.2, Jun.1992, pp.131-184.

[2] R.Prieto-Diaz and P.Freeman, "Classfying Software for Reusability," *IEEE Software*, Jan. 1987, pp.6-16

[3] C.McClure, *The Three Rs of Software Automation*, Prentice Hall, Englewood Cliff.NJ. 1992

[4] I.Sommerville, *Software Engineering(4th ed.)*, Addison-Wesley Publishing Company, Reading, Mass., 1992

[5] R.Prieto-Diaz, "Status Report:Software Reusability," *IEEE Software*, May 1993, pp.61-66

[6] T.J.Biggerstaff and C.Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, Mar. 1987, pp.41-49

[7] C.Paredes and J.L.Fiadeiro, "Reuse of Requirements and Specifications — a Formal Frame-work — ," *Proc. ACM Symp. Software Reusabiliry.(SSR'95)*, Apr. 1995, pp.263-266

[8] G.Caldiera and V.R.Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, Feb 1991, pp.61-70

[9] J.M.Bieman and S.Karunanith, "Measurement of Language-Supported Reuse in Object-Oriented and Object-Based Software," *The Journal of Systems and Software*, 39(1995)3, PP.271-294

[10] 김갑수,신영길,우치수, "소프트웨어 부품 자동 추출, 분류 및 검색," *한국정보과학회 봄 학술발표논문집*, Vol. 21, No. 1, 1994, pp. 625-628

[11] M.Page-Jones, *The Practical Guide to Structured System Design*, YOURDON Press, New York, N.Y. 1980

[12] STARS, "Reusability Guidelines," *Document Number STARS-QC-00340/001/01*, May 1989

[13] D.L.Parnas, P.C.Clements, and D.M.Weiss, "Enhancing Reusability with Information Hiding," *Proceedings of ITT Workshop on Reusability in programming*, Sep. 1983

[14] A.Marco, *Software Engineering concepts and management*, Prentice Hall, Englewood Cliff. NJ. 1990

[15] N.Fenton, "Software measurement : A Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, Vol.20, No.3, Mar. 1994, pp.199-206

[16] W.B.Frakes, C.J.Fox, and B.A.Nejmeh, *Software Engineering in the UNIX/C Environment*, Prentice Hall, Englewood Cliff, NJ. 1991

[17] D.L.Parnas, "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, Vol.15, No.12, Dec. 1972, pp.1053-1058

[18] L.S.Rising, "An Information Hiding Metric," *The Journal of Systems and Software*, Vol.26, 1994, pp.211-220

[19] S.M.Henry and C.A.Selig, "Predicting Source-Code Complexity at the Design Stage," *IEEE Software*, Mar. 1990, pp. 36-44.

[20] S.H.Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley Publishing, 1995

[21] T.J.McCabe, "A Complexity Measure," *IEEE Transactions on Softeare Engineering*, Vol. SE-2, No.4, 1976, pp.308-320

[22] G.Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *ACMSIGPLAN notices*, Oct. 1977, pp.61-64

[23] B.K.Kang and J.M.Bieman, "Design Level Cohesion Measures: Derivation, Comparison, and Applications," *Technical Report CS-96-104*, Colorado State University, Jan. 1996

[24] J.M.Bieman and L.M.Ott, "Measuring Functional Cohesion," *IEEE Transactions on Software Engineering*, Vol.20, No.8, Aug. 1994, pp.644-657

[25] R.W.Sebesta, *Concepts of Programming Languages (2nd ed.)*, Benjamin/Cummings Publishing Company, Inc. 1993

김 형 섭

1984년 서울대학교 축산학과 학사. 1987년 ~ 1989년 삼일경영경제연구원. 1989년 ~1991년 한국경제신문사(KETEL 개발). 1992년 ~ 1994년 한국 PC통신. 1994년 ~ 1996년 한국과학기술원 정보 및 통신공학과 석사. 1994년 ~ 현재 (주) 나우콤 기술기획팀 팀장.

배 두 환

1980년 서울대학교 조선공학과 학사. 1987년 위스콘신-밀워키대학 전산학 석사. 1992년 플로리다 대학 전산학 박사. 1992년 ~ 1994년 플로리다 대학 전산학과 조교수. 1995년 ~ 1996년 한국과학기술원 정보 및 통신공학과 조교수. 1996년 ~ 현재 한국과학기술원 전산학과 조교수.