# 객체지향 분석 단계에서의 클래스 복잡도 측정

## (Measurement of Classes Complexity in the Object-Oriented Analysis Phase)

김 유 경 † 박 재 년 ††

(Yu-Kyung Kim) (Jai-Nyun Park)

**요 약** 구조적 개발 방법론에 적용하도록 만들어진 복잡도 척도들은 클래스의 상속성, 다형성, 메시지 전달 그리고 캡슐화와 같은 객체지향의 개념에 직접적으로 적용할 수 없다. 또한, 기존의 객체지향 소프트웨어에 대한 척도의 연구는 프로그램의 복잡도나, 설계 단계의 척도가 대부분이었다. 실제로 분석 단계 클래스의 복잡도를 낮춤으로써, 시스템의 개발 노력이나 비용 및 유지보수 단계에서의 노력이 크게 줄어들게 되므로, 분석 클래스에 대한 복잡도를 측량하기 위한 척도가 필요하다.

본 논문에서는 객체지향 개발방법론인 RUP(Rational Unified Process)의 분석 단계에서 추출되는 분석 클래스에 대하여 복잡도를 측정할 수 있는 새로운 척도를 제안한다. 협력 복잡도 CC(Collaboration Complexity)는 가능한 협력의 최대 수로서 클래스가 잠재적으로 얼마나 복잡할 수 있는지를 측정하기 위한 척도이며, 각 협력자들의 인터페이스를 이해하는 것과 관련된 총체적 어려움을 측정하는 인터페이스 복잡도 IC(Interface Complexity)를 정의하였다. 제안된 척도는 Weyuker의 9가지 공리적 성질에 대하여 이론적인 검증을 하였으며, 텍스트 마이닝 기법을 사용하여 사용자의 질문에 자동으로 응답하는 시스템의 분석 클래스에 대하여 제안된 척도를 적용하여 복잡도를 측정하였다. 제안된 CC와 IC의 값과 Chidamber와 Kemerer가 제안한 CBO와 WMC의 값을 비교해 본 결과, 제안된 복잡도 척도의 계산 결과 값이 큰 클래스의 경우에는 설계 이후 단계에서도 역시 복잡도가 커지게 되는 것을 알 수 있었다. 이로써 소프트웨어 개발 주기의 초기에 클래스에 대한 복잡도를 평가해 보고, 나머지 단계에 필요한 시간과 노력을 예측함으로써 보다 비용-효과적인 객체지향 소프트웨어를 개발할 수 있는 가능성이 높아질 것으로 기대된다.

***Abstract*** Complexity metrics have been developed for the structured paradigm of software development are not suitable for use with the object-oriented(OO) paradigm, because they do not support key object-oriented concepts such as inheritance, polymorphism, message passing and encapsulation. There are many researches on OO software metrics such as program complexity or design metrics. But metrics measuring the complexity of classes at the OO analysis phase are needed because they provide earlier feedback to the development project, and earlier feedback means more effective developing and less costly maintenance.

In this paper, we propose the new metrics to measure the complexity of analysis classes which draw out in the analysis phase based on RUP(Rational Unified Process). By the collaboration complexity, is denoted by CC, we mean the maximum number of the collaborations can be achieved with each of the collaborator and determine the potential complexity. And the interface complexity, is denoted by IC, shows the difficulty related to understand the interface of collaborators each other.

We verify theoretically the suggested metrics for Weyuker's nine properties. Moreover, we show the computation results for analysis classes of the system which automatically respond to questions of the user using the text mining technique. As a result of the comparison of CC and IC to CBO and WMC suggested by Chidamber and Kemerer, the class that have highly the proposed metric value

· 본 논문은 KISTEP 과제 지원에 의해 연구되었음.

† 정 회 원 : 숙명여자대학교 정보과학부
ykkim@cs.sookmyung ac kr
†† 종신회원 : 숙명여자대학교 정보과학부 교수
jnpark@sookmyung.ac.kr

논문접수 : 2000년 8월 18일
심사완료 : 2001년 8월 10일

maintain the high complexity at the design phase too. And the complexity can be represented by CC and IC more than CBO and WMC. We can expect that our metrics may provide us the earlier feedback and hence possible to predict the efforts, costs, and time required to remainder processes. As a result, we expect to develop the cost-effective OO software by reviewing the complexity of analysis classes in the first stage of SDLC(Software Development Life Cycle).

## 1. 서 론

최근 소프트웨어 개발은 하드웨어 기술의 발전으로 인하여 급격하게 복잡해지고 있으며, 대규모화, 다양화됨에 따라 정보 산업 분야 전반에 걸쳐 소프트웨어가 차지하는 비중이 점점 증가하고 있다. 이에 소프트웨어의 안정성과 신뢰성을 높이고 유지보수를 위한 노력을 최소화하기 위하여, 소프트웨어의 품질과 평가의 중요성이 널리 인식되고 있다. 그러나, 많은 소프트웨어 개발자들은 소프트웨어 유지보수 단계를 고려하지 않고 소프트웨어를 생산하고 있으며, 결과적으로 소프트웨어 유지보수단계의 비용이 증가하게 되었다. 이것을 방지하기 위해서는 요구분석 단계에서부터 구현 단계를 통한 이행이 자연스럽게 이루어져야 한다. 특히, 분석 단계에서의 결함은 최종 산출물인 소프트웨어의 결함으로 이어지기 때문에, 분석 단계에서 이루어지는 클래스의 설계에 대한 평가는 고품질의 소프트웨어를 생산하기 위하여 반드시 수행되어져야 한다.

많은 객체지향 개발 방법론들은 일반적으로 메트릭스의 적용을 크게 강요하지 않고 있다. 반면에 구조적 개발 방법론에서는 설계의 품질을 측량하기 위하여 복잡도 척도들이 사용되어 왔다. McCabe의 싸이클로메틱 복잡도(cyclomatic complexity)[1], Halstead의 프로그래밍 노력[2], 그리고 크기 메트릭[3] 등이 대표적으로 알려져 있다. 이러한 구조적 개발 방법론에 적용하도록 만들어진 복잡도 척도들은 클래스, 상속성(inheritance), 메시지 전달 그리고 캡슐화(encapsulation)와 같은 객체지향의 개념에 직접적으로 적용할 수 없다[4]. 그러므로, 객체지향 개발 방법에 적합한 척도가 요구되고 있다.

또한, 기존의 객체지향 소프트웨어에 대한 척도의 연구는 프로그램의 복잡도[5][6]나, 설계 단계의 척도[7][8][9]가 대부분이었다. 그러나, 실제로 분석 단계 클래스의 복잡도를 낮춤으로써, 시스템의 개발 노력이나 비용 및 유지보수 단계에서의 노력이 크게 줄어들게 되므로, 분석 클래스에 대한 복잡도를 측량하기 위한 척도가 필요하다.

본 논문에서는 객체지향 개발방법론인 RUP(Rational Unified Process)의 분석 단계에서 추출되는 분석 클래스에 대하여 복잡도를 측정할 수 있는 새로운 척도를 제안한다. 분석 단계의 클래스의 복잡도를 측량함으로써 구현될 시스템의 복잡도를 예측하고, 분석 클래스 및 클래스 사이의 관계를 개선함으로써 유지보수 비용의 절감 효과와 함께 개발비용의 절감 효과를 기대할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 객체지향 소프트웨어에 대한 척도와 기존 연구의 문제점에 대하여 살펴보고, 제안된 복잡도 척도의 적용 대상이 되는 RUP 분석 단계의 분석 클래스에 대하여 설명한다. 3장에서는 본 논문에서 새롭게 제안하고 있는 복잡도 척도에 대하여 설명하고, 4장에서는 제안된 척도에 대한 이론적인 평가와 사례연구를 통한 검증이 이루어진다. 마지막으로 5장에서 결론 및 향후 연구과제를 기술한다.

## 2. 관련 연구

### 2.1 객체지향 척도

클래스의 복잡도와 관련된 객체지향 척도들 가운데 가장 많이 사용되고 있는 것은 Chidamber와 Kemerer가 제안한 객체지향 설계를 위한 척도이다. [7]에서 설계 규모와 복잡도에 영향을 주는 요소를 파악하기 위한 6가지 척도를 제시하였으며, 이 중 클래스의 복잡도에 관한 척도는 WMC(Weighted Methods per Class)와 RFC(Response For a Class)이다.

WMC는 클래스당 가중치를 갖는 메소드의 수로서, $c_i$가 n개 메소드 각각의 정적 복잡도일 때, 다음의 (식 1)과 같이 정의된다.

$$WMC = \sum_{i=1}^{n} c_i \quad (1)$$

만약, 모든 메소드 복잡도가 일정한 단위값을 갖게 된다면, WMC는 메소드의 개수인 n과 같다. 메소드는 클래스가 갖는 특성(properties)이고, 복잡도는 그 특성의 집합에 대한 크기(cardinality)에 의해 결정된다. 따라서, 메소드의 개수는 클래스 정의에 대한 측량일 뿐만 아니라, 클래스의 속성은 특성에 대응되기 때문에 속성들에 대한 측량이기도 하다는 것이다.

클래스 안에 선언된 메소드의 수나 메소드의 복잡도는 클래스를 개발하거나 유지보수 하는데 드는 노력을

나타내는 척도이다. 많은 수의 메소드를 갖는 클래스는 좀 더 특별해지기 때문에, 자녀 클래스에 미치는 잠정적인 영향이 커지고 재사용 범위가 좁아진다.

RFC는 클래스에 대한 반응도로서, 클래스 안에 선언된 메소드의 수와 이들 메소드에서 직접 호출된 또 다른 메소드들의 합이다. $RFC = |RS|$로 정의되며, 여기에서 RS는 클래스들의 응답집합으로 다음 (식 2)와 같이 주어진다.

$$RS = M_i \cup_{all\ i} \{R_v\} \quad (2)$$

(식 2)에서 $M_i$는 클래스 내에 있는 모든 메소드들의 집합이고, $R_i = \{R_v\}$는 $M_i$에 의해 호출되는 메소드들의 집합이다. RFC 값이 큰 클래스는 메시지에 반응해야 하는 의무가 그만큼 크다는 의미이며, 따라서 유지보수에 많은 비용이 들고, 테스트 단계에서 더 많은 노력이 필요하게 된다.

[10]에서는 객체지향 패러다임의 기본 컴포넌트를 정의하고, 각각의 기본 컴포넌트가 포함하고 있는 프로그래밍 습성(behavior)을 설명하였다. 또한, 이에 따른 5가지 척도를 제안하였다. 이 가운데 클래스의 복잡도에 관한 척도는 DAC(Data Abstraction Coupling), NOM (Number Of local Methods), 그리고 Size2이다.

DAC는 클래스 내에 정의된 ADT(abstract data type)의 수로서, 클래스 내에 선언된 서로 다른 타입의 단순하지 않은 속성의 수로서 표현된다. ADT의 선언은 잠재적으로 협력이 생성될 수 있다는 것을 의미하며, DAC의 값이 커질수록 클래스들간의 과도한 결합이 이루어지게 되고, 따라서 설계의 모듈화를 해치고 재사용을 어렵게 한다.

NOM은 인터페이스의 복잡성을 표현하는 것으로서, 클래스에 의해 지역적으로 제공되는 메소드들의 수를 나타낸다. NOM이 크다는 것은 그만큼 그 클래스가 제공하는 서비스와 책임이 많다는 의미가 된다. 클래스에 선언된 메소드가 많을수록 더 구체적인 클래스가 되고, 하위 클래스들에 영향을 많이 주어 재사용 하기 어려운 클래스가 된다.

Size2는 인터페이스에서 제공되는 서비스들의 수를 나타내는 것으로서, 속성의 수와 지역 메소드의 수를 합한 것으로 정의된다. 즉, NOA가 속성의 수이며, NOM이 지역 메소드들의 수를 나타낼 때 Size2는 아래의 (식 3)과 같이 정의된다.

$$Size2 = NOA + NOM \quad (3)$$

그러나 이 척도는 유용성이 적어서, 객체지향 시스템의 유지보수 노력을 예측하기 위한 간단한 회귀방정식에 포함되어 사용되고 있다.

Cohen은 [11]에서 객체지향 방법으로서 책임 기반(responsibility-driven)과 데이타 기반(data-driven)을 비교하여, [7]에서 제안한 척도들을 확장시키고 있다. 3가지 척도 가운데 클래스의 복잡도에 관한 척도는 WAC (Weighted Attributes per Class)와 NOT(Number Of Tramps)이다.

WAC는 WMC를 수정 보완하여 제안된 것으로서, 클래스 안에 정의된 속성을 가중치로 구한다. 클래스의 두 가지 성질로서 속성과 메소드는 클래스의 복잡도를 증가시키는 원인이 된다. WMC가 메소드에 대한 측량이라면, WAC는 속성에 대한 측량이 된다. 속성의 가중치는 그 크기로 정의된다.

NOT는 메소드에 포함된 매개변수의 타입과 수를 포함한다. 외부 매개변수는 메소드의 본체(body)에서 참조되지 않는 것으로서, 매개변수의 수가 증가할수록 복잡도가 증가하게 되고, 메소드에 의해 행해진 처리에 대해 잘못된 인식을 유도할 수도 있다. NOT는 클래스 내에 정의된 메소드의 표현에서 외부 매개변수의 전체 개수로서 정의된다.

### 2.2 기존 척도들의 문제점

기존의 척도들은 객체지향 분석 및 설계 단계에 적용하도록 제안되었다. 그러나, WMC나 LCOM을 비롯하여 메소드의 매개변수나 메소드의 복잡도를 속성으로 정의된 척도들은 분석 초기에 제공되는 클래스 정보보다 훨씬 더 상세한 정보를 필요로 하기 때문에 분석 단계에서는 적용할 수 없다[12]. 객체지향 분석은 실세계의 시스템을 소프트웨어로 모델링하여 설명하기 위한 단계로서, 문제 영역의 개념을 객체지향 모델 영역의 개념인 클래스로 추상화시키는 과정이다[11]. 이러한 분석 활동의 결과로서 작성되는 분석 모델은 전체 시스템을 구성하는 클래스와 클래스가 수행해야할 책임 및 그 책임을 수행하기 위하여 필요한 클래스간의 협력과 상속 관계를 포함한다. 따라서, 분석 모델에서 표현하고 있는 정보보다 훨씬 더 자세한 정보를 요구하는 척도들은 실제로 분석단계에서 사용할 수 없다. 예를 들어, RFC와 같은 경우는 클래스가 초기화되어 송신하는 각 메시지에 대한 정보를 요구하고 있다. 이것은 아마도 구현이 끝날 때까지는 계산할 수 없을 것이다. 그러므로, 실제로 객체지향 설계를 위한 척도라고 말할 수 없다. 이로 인하여 분석 단계에서 개발자들이 쉽게 적용할 수 있는 척도가 요구된다.

또한, 기존의 척도들이 단순한 객체지향의 개념과 척도를 정의하는데 그치고 있어서 척도를 적용하기 위한 구체적인 절차와 가설에 대한 충분한 설명이 없다는 문제점을 가지고 있다. 이들은 척도를 정의하는데 있어서

수학적 명세를 사용하지 않아서 이해하기 어려우며, 계산하는 자세한 절차를 설명하지 않으므로 적용하는데 혼란을 야기할 수 있다[13]. 따라서, 모델에 대한 가설과 제한사항을 충분히 설명하여 개발자들이 이해하기 쉽고 적용 방법을 혼돈하지 않도록 해야할 것이다.

본 논문에서 제안하고 있는 척도의 구별되는 특징은 분석 초기의 정보인 클래스의 책임과 협력 관계에 초점을 맞추었다는 점이다. 또한, 기존의 척도들이 가지고 있는 문제점을 해결하기 위하여 이해하기 쉬운 수학적 명세를 사용하였고, 척도를 적용하기 위한 자세한 계산 과정 및 가설을 설명하고 있다.

### 2.3 RUP의 분석 클래스

객체지향 분석은 사용자가 정의한 요구사항을 만족시키면서 작동되는 소프트웨어를 기술하는 모델을 개발하는 것이다. 분석 모델에는 도메인 정보와 기능, 행위 등이 나타난다. 현재 통합된 객체지향 분석 방법론인 RUP가 사용되고 있다.

RUP는 Booch의 OOD 방법론과 Rumbaugh의 OMT, Jacobson의 OOSE 등의 3가지 방법론을 중심으로 여러 객체지향 방법론들이 하나로 통합되어, OMG(Object Manager Group)에서 채택된 표준 객체지향 방법론이다[14]. RUP는 유스케이스 중심(Use-case Driven)이며 아키텍처 중심(Architecture Centric)의 개발 절차이고, 또한 반복적(Iterative)이면서 점진적인(Incremental) 소프트웨어 개발 절차이다[15].

특히, 분석은 유스케이스 모델을 입력으로 하여 요구사항을 구조화시키는 단계로서, 먼저 분석 패키지들 사이의 공통성 처리와 서비스 패키지를 식별하는 절차로 시작한다. 그리고 나서, 분석 패키지의 의존성을 정의하며, 개체 클래스를 식별하는 아키텍처 분석 절차로 진행된다. RUP 분석의 절차와 모델은 다음 그림 1과 같다.

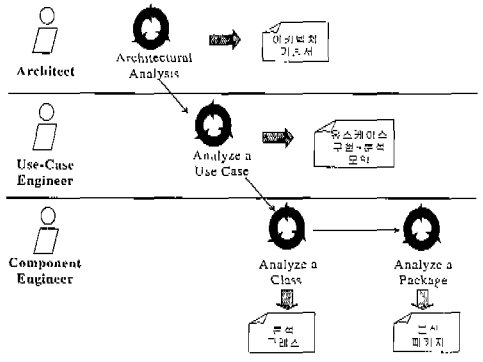


그림 1 RUP의 분석절차

유스케이스 분석은 분석 클래스를 식별하고, 분석 객체들의 교류를 기술하여 클래스도(Class diagram)와 교류도(Interaction diagram)에 나타낸다. 교류도는 객체간의 교류를 시간의 순서에 초점을 두어 나타낸 순차도(Sequence diagram)와 교류를 수행하는 객체들의 전체적인 조직과 상황(context)에 초점을 맞춘 협력도(Collaboration diagram)가 있다. 이들은 분석 클래스가 유스케이스를 어떻게 구현하여 수행하는가를 기술한다. 분석 클래스는 설계 단계의 하나 또는 여러 개의 클래스들 또는 서브시스템의 추상화를 나타내며, 클래스 분석 절차에서는 클래스의 책임과 속성을 식별하고, 연관관계 및 구성관계를 식별한다. 다음으로 일반화 관계를 식별해서 클래스도에 나타낸다.

본 논문에서 제안하고 있는 복잡도 척도는 이러한 분석 절차에서 생성되는 분석 클래스에 적용하여 설계와 구현단계에서의 복잡도와 개발 노력을 예측할 수 있도록 정의하였다. 각각의 클래스는 우리가 구현하고자 하는 업무 시스템의 최소 단위이며, 그 각자가 자신에 대한 책임을 지니고 자신의 인스턴스로서 객체를 만들어 내게 된다. 따라서, 클래스가 실세계의 대상을 부적절하게 모델링하고 있다면 시스템의 개발에 큰 장애 요소로 작용할 수 있다[16]. 그러므로, 개발 생명주기의 가장 초기 단계에서 개발자들이 적절하게 클래스로 모델링을 하고, 올바른 클래스 분할을 선택할 수 있도록 방향을 제시하는 것은 매우 중요한 일이다.

RUP의 분석 과정에서 생성되는 분석 클래스는 클래스에 대한 상세 설계가 이루어질 때 기초 명세서(specifications)로 사용된다. 분석 클래스들과 그들의 책임, 속성, 그리고 관계는 대응되는 설계 클래스들의 메소드, 속성, 그리고 관계를 생성하기 위한 논리적인 입력으로 제공된다[15]. 즉, 분석 단계의 결과는 설계 단계 및 소프트웨어 개발 절차의 나머지 과정에서 사용되어질 기술적인 인프라 구조를 제공하기 위해 확장되어지는 청사진이라고 할 수 있다. 그리고, 이러한 클래스 분석 절차의 목적은 요구사항을 좀 더 정확하게 이해하기 위한 것이며, 또한 유지보수가 쉽게 이루어지고 전체 시스템에 대한 구조를 제공하도록 요구사항을 형식적으로 나타내기 위한 것이다. 따라서, 분석 과정을 거쳐 나온 분석 클래스들은 매우 중요한 의미를 갖게 된다. 이러한 분석 절차에서 사용되는 모델은 클래스도와 교류도이다. 본 논문에서는 클래스도와 교류도의 일종인 협력도를 사용하여, 이 두 다이어그램에 나타난 분석 클래스와 클래스 사이의 관계를 기반으로 복잡도 척도를 정의하였다. RUP는 표준 모델링 언어인 UML을

사용하여 각 다이어그램을 작성한다. 그림 2는 클래스도와 협력도의 예를 보여주고 있다.

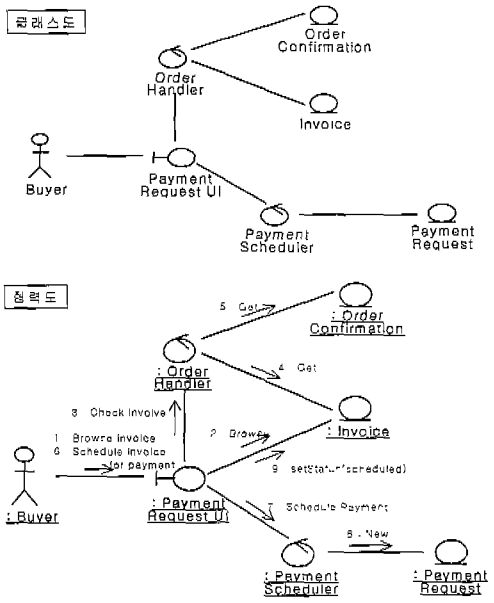


그림 2 지불 송장에 대한 클래스도와 협력도의 예

## 3. 클래스 복잡도 척도

### 3.1 척도를 정의하기 위한 가설

객체지향 개발단계에서 문제 영역을 표현하는 관련된 개체(entities)는 소프트웨어 시스템의 구현된 클래스로 대응된다. 각각의 클래스는 자신이 제공하는 일련의 서비스들에 의하여 설명되며, 이러한 일련의 서비스들을 책임(responsibilities)이라고 부른다. 클래스가 한 책임을 수행하기 위하여, 다른 클래스와 상호 작용하게 되며 이것을 협력(collaborations)이라고 한다. 또한, 협력하게 되는 그 클래스들을 협력자(collaborators)라고 정의한다. 클래스들 사이의 상호작용은 연관관계(association)나 구성관계(aggregation)를 통하여 이루어진다. 따라서, 이들 연관관계와 구성관계를 협력으로 정의한다.

또한, 시스템 내의 클래스들 사이에는 상속 계층구조가 형성되어 있고, 한 클래스는 부모 클래스로부터 책임을 상속받을 수 있으며 이것을 상속받은 책임이라고 정의한다. 상속받은 책임은 자녀 클래스의 인터페이스의 일부분이 된다. 책임의 나머지 부분은 상속받은 책임을 제외한 것으로서 자녀 클래스를 위하여 특별히 생성된다. 때때로 상속받은 책임들이 자녀 클래스에서 다시 구현될 수도 있으며, 이러한 경우에는 상속받지 않은 것으로 간주한다. 본 논문에서는 한 클래스의 책임이란 상속받은 책임을 제외하고, 자녀 클래스에서 생성된 책임을 표현하는 항목으로 사용할 것이다. 전체 책임(total responsibilities)이란 상속받은 책임과 클래스의 책임을 모두 합하여 표현하는 것으로 사용할 것이다.

다음은 분석 클래스의 책임 및 협력자의 수와 복잡도 사이의 관계에 대한 가설이다.

**가설 1.** 상속받은 책임을 제외한 클래스의 책임과 협력자의 수가 많을수록, 그 수가 적은 클래스보다 개발하기가 좀 더 복잡하다.

**가설 2.** 책임의 전체 개수가 많은 클래스는 그 수가 적은 클래스보다 사용하기(usage)가 좀 더 복잡하다.

**가설 3.** 클래스의 개발 복잡도는 상속받은 책임을 제외한 책임의 개수와 협력자의 수가 증가하는 것에 대하여 비 선형적(non-linearly)으로 증가한다.

**가설 4.** 사용상(usage)의 복잡도는 책임의 전체 개수가 증가하는 것에 대하여 비 선형적으로 증가한다.

**가설 5.** 한 클래스는 항상 자기 자신과 협력하고 있다.

위의 각 가설을 위한 논리는 다음과 같이 설명할 수 있다. 우선 클래스를 개발할 때의 어려움은 그 클래스의 책임의 수, 그리고 협력자의 수와 관련이 있다고 가정하였다. 이것은 클래스의 복잡도가 책임과 협력자의 수가 증가할수록 증가하게 된다는 것을 의미한다. 클래스의 개발자들은 열거된 책임을 구현해야 하고, 이것은 복잡도가 책임의 목록이 많아질수록 증가하게 된다는 것을 쉽게 보여준다. 또한 각 협력자와 상호작용하기 위하여, 클래스의 개발자들은 협력자의 인터페이스를 이해해야 하는 부가적인 노력을 들여야한다. 그러므로, 복잡도는 협력자의 수가 많아질수록 증가하게 된다.

그러나, 클래스의 모든 책임이 구현되어지는 것은 아니다. 일부 책임에 대한 구현은 사실상 구현이 이미 이루어진 부모 클래스로부터 상속되어진다. 그런 책임들은 상속되어 재 사용되어지고, 개발자는 이들 책임에 대한 메커니즘을 제공하지 않아도 된다. 구현이 제공되지 않는 경우는 부모 클래스가 추상 클래스(abstract class)일 때 발생할 수 있는데, 그들 책임은 상속받지 않은 것으로 간주된다. 그러므로 단지 상속받은 책임을 제외한 클래스의 책임에 대해서만 고려할 필요가 있다. 이러한 논의를 바탕으로 가설 1을 이끌어낼 수 있다.

가설 3은 종속 변수인 복잡도와 독립 변수인 책임 및 협력자 수 사이에 비 선형적인 관계가 존재함을 설명한다. 책임은 설계 단계에서 메소드의 그룹으로 사상되는

클래스 인터페이스 부분에 대한 추상적인 기술이다. 그러므로, 복잡도의 증가는 책임의 수에 반드시 비례한다고 할 수 없다. 또한, 협력자 집합이 크다는 것은 일반적으로 한 클래스가 많은 수의 연관관계나 구성관계 가지고 있다는 것을 설명해 준다. 즉, 이러한 협력 관계들은 비 선형적인 방법으로 복잡도에 기여할 것이라고 추정할 수 있다.

가설 2와 가설 4는 클래스들 사이의 협력에 대한 어려움은 각 협력자의 인터페이스 복잡도에 기인하게 된다는 것을 말한다. 앞에서 설명한 것과 같이, 한 클래스의 개발자는 협력자들 각각의 인터페이스를 이해하고 사용할 필요가 있다. 모든 클래스와 모든 협력자는 모든 사용자들이 이해해야만 하는 책임과 관련된 개념적인 상태(state)를 가지고 있다.

가설 2는 좀 더 적은 수의 책임을 가진 클래스의 인터페이스가 더 많은 수를 가진 클래스 보다 이해하기 더 쉽다는 것을 의미한다. 가설 4는 여러 개의 더 작은 클래스들의 인터페이스를 이해하는 것이 더 쉽다는 것을 말하고 있으며, 좀 더 작은 클래스들과 협력하도록 권장하고 있다. 왜냐하면, 전체 책임의 개수가 증가하는 것과 인터페이스 복잡도가 증가하는 것이 비 선형적인 관계를 유지하기 때문이다.

가설 5는 클래스가 자기 자신과 협력할 수 있다는 것이다. 이것은 클래스 내에 있는 어떤 책임도 자기 자신 또는 다른 어떤 책임을 재귀적으로 호출할 수 있다는 사실 때문에 매우 명백해진다. 또한, 클래스 E의 복잡도에는 클래스 E의 외적 복잡도가 포함되어 있다. 그러므로, 클래스 E의 개발자는 그 클래스의 인터페이스도 역시 이해 할 필요가 있다.

### 3.2 클래스 복잡도 척도의 정의

본 논문에서 제안하고 있는 분석 클래스에 대한 복잡도 척도는 협력 복잡도 *CC*(Collaboration Complexity)와 인터페이스 복잡도 *IC*(Interface Complexity)로 정의된다.

협력 복잡도는 가능한 협력의 최대 수로서 클래스가 잠재적으로 얼마나 복잡할 수 있는지를 알려주게 된다. 그러나 이것을 정확하게 알 수 없기 때문에, 구현되어지는 책임 각각에 대하여 협력자들이 사용된다고 가정한다. 따라서, 한 클래스 $E$에 대한 협력 복잡도 $CC$는 다음과 같다. 여기에서, $r$은 클래스 E의 책임의 개수이며 $c$는 협력자의 개수이다.

[식 1] $CC(E) = r(1+c)$

가설 3에서는 $r$과 $c$의 값에 대하여 복잡도가 비 선형적인 의존성을 가지고 있다고 보았다. 위의 [식 1]에서 협력 복잡도 $CC$가 비 선형적 성질을 만족하고 있다는 것을 알 수 있다. 여기에서 협력자의 수가 $(1+c)$가 된 것은 가설 5에 의하여 클래스는 자기자신과도 협력하기 때문이다. 따라서, $CC$는 모든 협력자$(1+c)$에 대하여 구현되어지는 책임 $r$개가 각각 사용되므로 한 클래스가 가질 수 있는 최대 협력의 개수를 측정하게 된다.

또한, 한 클래스에 대한 협력 복잡도는 완전한 한 시스템을 표현하는 클래스들의 집합에 대한 전체 복잡도를 계산하도록 확장할 수 있다. 클래스의 개수가 많아질수록 전체 시스템의 복잡도는 당연히 증가하게 된다. 따라서, 클래스의 구현에 따르는 비용과 노력을 비교하기 위한 의미 있는 값은 평균적인 협력 복잡도가 될 것이다. 그러므로, 만약 시스템 $S$가 $n$개 클래스들 $E_1, E_2, \cdots, E_n$로 이루어져 있다면, 즉 $S=\{E_1, E_2, \cdots, E_n\}$이라면, $S$의 평균 협력 복잡도는 [식 2]와 같다.

[식 2] $CC(S) = \frac{1}{n}\sum_{i=1}^{n} CC(E_i)$

협력으로 인한 클래스의 관계는 클래스 사이의 결합도를 나타내며, 높은 결합도는 클래스의 재사용을 어렵게 하고, 복잡도를 증가시키는 원인이 된다. 따라서, 각 클래스의 협력 복잡도는 낮아야 하며, 평균 복잡도보다 높은 클래스들은 다시 설계되어야 한다.

또 다른 복잡도 척도로서 각 협력자들의 인터페이스를 이해하는 것과 관련된 총체적 어려움을 측정하는 인터페이스 복잡도 *IC*(Interface Complexity)가 있다.

한 클래스의 인터페이스 크기 *IS*(Interface Size)는 클래스가 갖는 책임과 부모 클래스로부터 상속받은 책임을 모두 더한 경우 최대가 되며, 따라서 책임의 전체 개수로서 계산할 수 있다. 그러나 이것은 복잡도와 책임의 전체 개수 사이의 비 선형 관계를 가정한 가설 4에 위배된다. 따라서, 전체 책임의 개수 $R$이 상수 지수 $m$에 의해 증가되는 것으로 인터페이스 크기에 대한 복잡도를 측정하도록 정의하였다. 이 경우, $m$이 가질 수 있는 가능한 경우에 대하여, 유의 수준 5%이내에서 켄달(Kendall)의 순위 검정방법으로 시험하였다. 켄달의 순위 상관 계수 $\tau$를 사용하여, ATM 클래스[17]에 대하여 $R^2$과 $R^3$, $R^4$, $R^5$, $R^6$ 사이의 관계를 조사하였다. 아래 표 2에서 보듯이, 상관 관계에서 매우 미세한 차이를 보이고 있으며 따라서 좀더 구현이 간단해 지도록 $m=2$를 선택하였다.

따라서, 임의의 한 클래스의 인터페이스 크기 $IS=R^2$이 되며, 한 클래스와 협력 관계에 있는 클래스들의 인터페이스도 이해하는 것이 필요하다. 그러므로 인터페이스 복잡도는 한 클래스의 인터페이스 크기와 개별 협력

표 1 R에 대한 검정의 결과

| $R^m$ | $\tau$ | 유의 수준 |
|---|---|---|
| $R^3$ | 0.65 | 0.05 |
| $R^4$ | 0.66 | 0.05 |
| $R^5$ | 0.66 | 0.05 |
| $R^6$ | 0.66 | 0.05 |

자들의 인터페이스 크기를 더하여 구할 수 있다.

클래스 $E$가 전체 $R$개의 책임을 가지고 있고, $c$개의 다른 클래스들 $E_1, E_2, \cdots, E_c$ 과 협력하고, 인터페이스 크기를 $IS$라고 표현한다면, 인터페이스 복잡도 $IC$는 [식 3]과 같이 주어진다.

[식 3] $IC(E) = R^2 + \sum_{i=1}^{c} IS(E_i)$

여기에서, $IS(E_i)$는 클래스 $E_i$의 인터페이스 크기를 나타낸다.

또한, 한 클래스가 갖는 인터페이스 복잡도는 역시 완전한 한 시스템에 대한 전체 인터페이스 복잡도를 표현하도록 확장할 수 있다. 그러므로, 만약 시스템 $S$가 $n$개 클래스들로 이루어져 있다면, 즉 $S=\{E_1, E_2, \cdots, E_n\}$이라면, S의 평균 인터페이스 복잡도는 아래와 같다.

[식 4] $IC(S) = \frac{1}{n} \sum_{i=1}^{n} IC(E_i)$

인터페이스 복잡도는 한 클래스가 갖는 책임과 얼마나 많은 협력자들을 가지게 되는지를 나타낸다. 따라서, 평균 인터페이스 복잡도보다 높은 클래스들은 너무나 많은 책임을 갖거나 협력자들이 많은 경우가 되므로, 클래스의 분할이나 상속 관계를 통한 재 설계가 이루어지는 것이 바람직하다.

### 3.3 협동 그래프 COG(COllaboration Graph)의 정의

클래스들 사이의 협력 관계와 상속 관계를 표현하고, 복잡도 척도를 계산하기 위하여, 분석 클래스의 정보를 표현하고 있는 클래스도와 협력도로부터 협동 그래프 $COG$를 정의한다. 다음은 협동 그래프 $COG$에 대한 정의이다.

[정의 1] $COG$는 유방향 그래프 $G=(V, E)$이다. 여기에서, $V$는 정점(vertex)의 유한 집합 즉, $V=\{v_1, v_2, \cdots, v_n\}$ 이고, $E=\{e_1, e_2, \cdots, e_n\}$는 간선(edges)의 유한 집합이다. 또한, $V \neq \emptyset$이며, $E \subseteq V \times V$ 이다.

[정의 2] 임의의 한 분석 모델 M을 위한 협동 그래프는 $COG=(V, E)$이며, 여기에서 $V$는 M에 나타난 클래스를 표현하는 정점들의 집합이다. 각 정점은 가능한 책임의 수를 표현하는 가중치(weighted value)를 가지고 있으며, 가중치는 정수(integer)이다.

[정의 3] 임의의 한 분석 모델M을 위한 협동 그래프는 $COG=(V, E)$이며, 여기에서 $E$는 M에 나타난 클래스들의 관계를 표현하는 간선들의 집합으로, $E_c$가 협력 관계를 나타내고, $E_i$가 상속 관계를 나타낼 때, $E=E_c \cup E_i$ 이다.

$COG$는 클래스의 책임과 협력자에 대한 정보를 제공한다. 클래스도에 나타난 클래스가 갖는 책임의 개수는 각 노드에 정수 값으로 표현된다. 클래스 사이의 협력 관계는 협력도에 나타난 메시지 전달을 통하여 연관되어지는 클래스들 사이에 방향을 갖는 간선으로 표현된다. 다음의 그림 3은 클래스도와 협력도의 일부이며, 이들로부터 생성하게 되는 COG의 예가 그림 4에 나타나 있다.

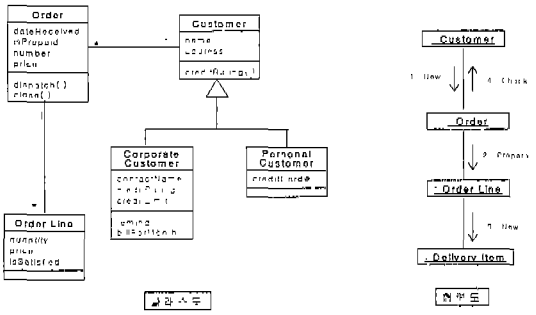


그림 3 클래스도와 협력도의 예

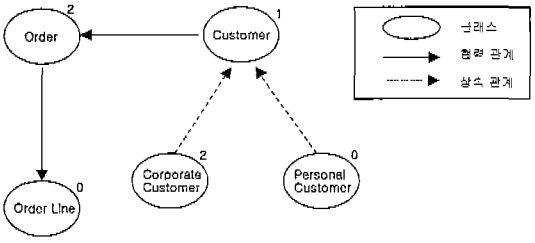


그림 4 그림 3에 대한 COG의 예

위의 그림 4로부터 우리는 제안된 척도를 사용하여 각 클래스의 협력의 복잡도와 인터페이스 복잡도를 측정할 수 있다. 클래스 *"CorporateCustomer"*와 *"Personal Customer"*는 부모 클래스로부터 1개의 책임을 상속받으므로, $R_{CorporateCustomer}=3$이고 $R_{PersonalCustomer}=1$이다. 나머지 다른 클래스들은 상속 관계를 형성하고 있지 않으므로, $r=R$이다. 따라서, 이들 클래스에 대한 협력 복잡도와 인터페이스 복잡도를 계산하면 다음과 같다.

$CC(Order)=r(1+c)=2(1+1)=4$, $IC(Order)$
$=2^2+IS(OrderLine)=4+0=4$
$CC(Customer)=1(1+1)=2$, $IC(Customer)$
$=1^2+IS(Order)=1+2^2=5$
$CC(OrderLine)=0(1+0)=0$, $IC(OrderLine)$
$=0+0=0$
$CC(CorporateCustomer)=2(1+0)$
$=2$, $IC(CorporateCustomer)=(2+1)^2+0=9$
$CC(PersonalCustomer)=0(1+0)$
$=0$, $IC(PersonalCustomer)=(0+1)^2+0=1$

그림 4의 클래스 "*Order*"와 클래스 "*Corporate Customer*"를 비교해 보면, 같은 책임의 개수를 갖는다해도, 클래스 "*Order*"는 협력자 클래스 "*OrderLine*"과의 협력 관계를 가지고 있기 때문에, 클래스 "*Corporate Customer*"보다 협력 복잡도 *CC* 값이 더 크다. 그러나, 인터페이스 복잡도는 클래스 "*CorporateCustomer*"가 상속관계로 인하여 전체 책임의 개수가 증가하기 때문에 *IC* 값이 더 커진다.

## 4. 복잡도 척도에 대한 평가

이 장에서는 제안된 척도에 대한 이론적인 검증이 이루어진다. 복잡도 척도가 만족해야만 하는 공리적 성질을 정의하고, 이들을 만족하는지를 검증하게 된다. 그리고 나서 사례연구의 결과를 분석하고 설명한다.

### 4.1 이론적 평가

Weyuker[18]는 절차적(procedural) 프로그램을 위한 복잡도 척도에 요구되는 성질들을 정의하였다. 이 성질들은 Chcrniavsky[19]등에 의하여 비판을 받고 있지만, 엄격한 이론적 검증방법을 제공하고 있다[8]. Weyuker에 의해 보여진 것처럼, 이 성질들을 이용하여 본 논문에서 제안하고 있는 복잡도 척도를 검증할 수 있을 것이다.

다음은 Weyuker의 성질을 설명하기 위하여 사용되는 표현법이다.

- E ; F는 서로 다른 임의의 클래스 E와 F의 결합(combination)을 나타낸다.
- 서로 다른 두 클래스 E와 F가 같은 책임의 집합을 가지고 있다면, E와 F는 동등하다고 말하고, E ≡ F와 같이 표시한다.
- 한 클래스 E의 전체 책임의 개수는 $R(E)$이며, 책임의 개수는 $r(E)$이다.
- $|E|$는 클래스 E의 복잡도로서 $|E| \geq 0$이다.

**성질 1.** $(\exists E)(\exists F)\ (|E| \neq |F|)$.

비공식적으로 이 성질은 모든 클래스들이 같은 척도 값을 가지 수 없음을 나타내고 있다. 다시 말하자면, 서로 다른 복잡도를 갖는 클래스들이 적어도 두 개는 존재한다는 것을 의미한다.

제안된 척도의 경우, 클래스에 정의된 책임의 개수와 상속 및 협력 관계에 있는 클래스의 개수에 기반을 두고 정의되었다. 일반적으로, 서로 다른 두 개의 클래스가 같은 협력 클래스들을 갖으며, 같은 상속 관계를 유지하는 경우는 매우 드문 일이다. 즉, 대부분의 서로 다른 클래스가 갖는 *CC*와 *IC* 값은 달라진다. 그러므로, $E \neq F$에 대하여 $CC(E) \neq CC(F)$이고, $IC(E) \neq IC(F)$라는 사실을 알 수 있다.

**성질 2.** c가 음이 아닌 수라면, 주어진 시스템 내에서 복잡도가 c인 클래스는 유한하다.

이 성질은 같은 복잡도를 갖는 클래스들이 유한개가 존재한다는 것을 나타내고 있다. 즉, 복잡도가 c인 클래스가 주어지면, 같은 복잡도를 갖는 클래스들의 수는 유한해야만 한다는 것이다. [CK 94]는 일반적으로 모든 시스템이 유한개의 클래스로 이루어지므로 클래스 수준에서 측정된 어떤 척도에 의해서도 충족될 것이라고 설명하고 있다.

제안된 두 척도에 할당되는 값은 $c+1$가지만이 존재하고, 각 척도의 값이 상수로 주어진다면 클래스에 대한 $c$, $r$, 그리고 $R$값과 그 협력자에 대한 $R_i$값의 조합만큼의 유한한 수가 존재하게 된다. 따라서, 성질 2를 만족하게 된다.

**성질 3.** $(\exists E)(\exists F)\ (E \neq F$ and $|E| = |F|)$.

이것은 같은 복잡도를 갖는 서로 다른 두 클래스가 존재한다는 것을 의미한다. 즉, 두 클래스가 같은 설계 특성을 가질 수 있다는 것이다. 일반적으로 척도 정의는 수식으로 표현되기 때문에, 같은 척도 값을 갖는 클래스를 찾을 수 있다.

주어진 한 클래스 E가 가지는 책임들 가운데 한 책임의 내용(text)을 바꾼다면, 같은 척도 값을 갖는 새로운 클래스 F를 얻을 수 있다. 그러므로 제안된 척도는 이 성질을 만족한다.

**성질 4.** $(\exists E)(\exists F)\ (E \equiv F$ and $|E| \neq |F|)$.

이 성질은 서로 같은 기능을 수행하지만, 복잡도가 같지 않은 두 클래스가 존재한다는 것을 나타낸다.

정확하게 서로 같은 인터페이스를 제공하는 동등한 두 클래스들이 서로 다른 내부 메커니즘을 가질 수 있으므로 제안된 척도에 대하여 이 성질은 성립한다. 다시 말하면, 같은 인터페이스를 제공한다고 해도 서로 다른

상속 계층이나 서로 다른 협력자의 집합을 가질 수 있다는 것이다.

**성질 5.** $(\forall E)(\forall F)(|E| \leq |E;F| \text{ and } |F| \leq |E;F|)$.

이것은 두 클래스의 결합에 대한 복잡도가 두 요소 클래스의 복잡도 보다 결코 작아질 수 없다는 것을 의미한다. 일반적으로 두 클래스 E와 F가 결합된다면, 책임의 개수와 협력자의 전체 개수가 증가하게 된다. 따라서 상속되거나 또는 새로 정의되는 책임의 개수도 증가하게 된다.

만약 두 클래스가 협력 관계를 가지고 있다면, 두 클래스의 결합은 책임의 개수가 증가함에 따라 복잡도가 증가할 것이다. 또한, 협력 때문에 발생할 수 있는 새로운 기능들이 추가될 수 있고, 결합된 클래스의 협력 클래스의 개수도 증가하게 되므로 명백하게 복잡도는 증가하게 된다. 만약 두 클래스 E와 F가 상속 관계를 가지며, E가 클래스 F의 부모 클래스라고 한다면, 결합된 클래스의 복잡도는 $CC$와 $IC$가 각각 $IC(E)$ 또는 $IC(F)$, $2R_ER_F$ 만큼 더 증가하게 된다. 그러나, 만약 클래스 F가 부모 클래스의 모든 책임을 재정의(overriding)하고 있다면, $IC$는 증가하지 않을 것이다. 따라서, 제안된 척도는 성질 5를 만족하고 있다.

**성질 6.**

(a) $(\exists E)(\exists F)(\exists G)(|E| = |F| \text{ and } |E;G| \neq |F;G|)$.

(b) $(\exists E)(\exists F)(\exists G)(|E| = |F| \text{ and } |G;E| \neq |G;F|)$.

기본적으로 성질 6은 같은 복잡도를 갖는 두 클래스 E, F가 다른 클래스 G와 각각 결합되었을 경우, 결합의 순서와 상관없이 결합된 클래스는 서로 다른 복잡도를 갖게 된다는 것을 의미하고 있다.

이것은 비록 E와 F가 같은 복잡도를 갖는다고 해도, 협력자의 수나 책임의 수 또는 상속 계층 구조와 같은 그들의 개별적 속성은 다르기 때문에 다른 클래스 G와 결합이 이루어진 후의 $|E;G|$와 $|F;G|$는 매우 달라지게 된다. 따라서, 이 성질도 제안된 척도에 대하여 성립한다.

**성질 7.** E의 책임이나 메소드의 순서 배치에 의하여 F가 구성되고, $|E| \neq |F|$인 클래스 E와 F가 존재한다.

[7]과 [18]에서는 객체지향 설계에서 클래스들은 문제 영역을 추상화한 것이며, 따라서 클래스 정의 내에서 명령문의 순서는 실행하거나 사용하는데 아무런 영향을 주지 않는다고 설명하고 있다. 그러므로 이것은 클래스 수준에서 정의된 어떤 척도에 대해서도 성립하게 된다. 본 논문에서 제안한 척도가 클래스에 대한 책임의 수와 협력 클래스의 개수만을 고려하고, 책임의 순서에 대한 배열은 고려하지 않기 때문에, 이 성질은 성립한다.

**성질 8.** E가 F를 다시 명명한 것이라면, $|E| = |F|$이다.

이 성질은 클래스의 이름을 변화시킨다고 해도 클래스의 복잡도에는 영향을 주지 않아야 한다는 것을 의미한다.

제안된 척도의 측량은 클래스의 이름에 기반을 두지 않았기 때문에 이 성질을 만족하고 있다는 것은 명백하다.

**성질 9.** $(\exists E)(\exists F)[(|E| + |F|) \leq (E;F)]$.

이것은 결합된 클래스가 구성 요소보다 좀 더 복잡해진다는 것을 명시하고 있다. 즉, 두 클래스가 결합될 때, 클래스들 간의 상호작용은 복잡도를 증가시킬 수 있다는 것이다. 그러나, [7]에서는 클래스의 결합이 이루어질 경우, 결합된 클래스의 복잡도가 낮아질 수 있으며, 성질 9는 객체지향 척도에 대한 필수적인 특징이 될 수 없다고 말하고 있다. 따라서, 제안된 척도에 대해서도 성질 9는 고려하지 않도록 한다.

아래의 표 2는 제안된 척도를 Weyuker의 9가지 성질에 관하여 평가한 결과를 보여준다. 평가의 결과로서 제안된 척도는 성질 9를 제외한 모든 성질들을 만족하고 있다는 것을 알 수 있다.

표 2 Weyuker의 성질을 이용한 평가 결과

| 척도 \ 성질 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| CC | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | – |
| IC | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | – |

따라서, 본 논문에서 제안한 척도는 위에 있는 성질들을 모두 만족하는 것으로 나타났기 때문에, 객체지향 척도로서의 유용성에 대한 이론적인 검증이 이루어졌다고 말할 수 있다.

### 4.2 실 사례를 통한 복잡도 측정

국내 S 기업의 b시스템은 텍스트 마이닝 기법을 사용하여 사용자의 질문에 자동으로 응답하는 시스템이다. 이 시스템은 RUP를 적용하여 개발되었으며, Java 언어를 사용하여 구현되었다. 본 연구에서 제안한 복잡도 척도를 분석하기 위하여 b시스템의 분석 모델과 설계 모델을 이용하여 실험하였다. 분석 모델의 클래스의 수는 98개로서, 각 클래스에 대하여 $CC$와 $IC$를 계산하였다. 계산 결과의 일부를 사례로 보이기 위하여, b 시스템에서 기업의 데이타 소스로부터 필요한 자료를 추출하여

정제, 변형, 전송, 적재하는 일련의 과정을 지원하는 한 서브 시스템을 선택하였다. 그림 5는 이 서브 시스템의 분석모델 가운데 DataMapping이라는 유스케이스로부터 생성된 클래스도의 일부를 COG로 표현한 것이다.

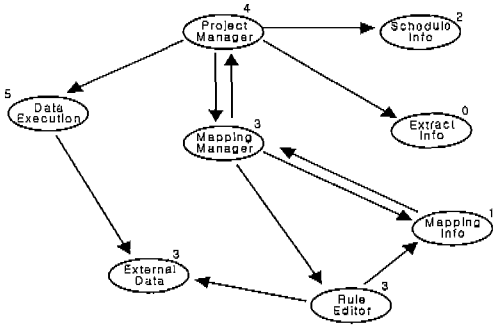


그림 5 DataMapping에 대한 클래스도의 일부

그림 5에 나타난 각 클래스에 대하여 *CC*와 *IC*를 계산한 결과가 그림 6에 있다.

| 클래스 명 | 부모 클래스 명 | r | c | R | CC | IC |
|---|---|---|---|---|---|---|
| DataExecution | | 5 | 1 | 5 | 10 | 34 |
| ProjectManager | | 4 | 4 | 4 | 20 | 54 |
| ScheduleInfo | | 2 | 0 | 2 | 2 | 4 |
| MappingInfo | | 1 | 1 | 1 | 2 | 10 |
| MappingManager | | 3 | 3 | 3 | 12 | 36 |
| ExtractInfo | | 0 | 0 | 0 | 0 | 0 |
| RuleEditor | | 3 | 2 | 3 | 9 | 19 |
| ExternalData | | 3 | 0 | 3 | 3 | 9 |

그림 6 그림 5에 대한 계산 결과

### 4.3 측정 결과에 대한 분석

98개 클래스에 대하여 *CC*와 *IC* 값을 계산하였고, 설계 모델로부터 98개 클래스에 대해 생성된 의사코드(pseudo code)를 이용하여 [7]의 WMC와 CBO 척도를 적용한 값과 비교하였다. 그림 7은 IC와 WMC의 계산 결과 값을 비교한 그래프이다. *IC*가 전체 책임의 개수 $R$에 대한 $R^2$으로 계산되어지기 때문에 값의 범위는 매우 크지만, WMC와 유사한 결과를 보이고 있다. 몇 개

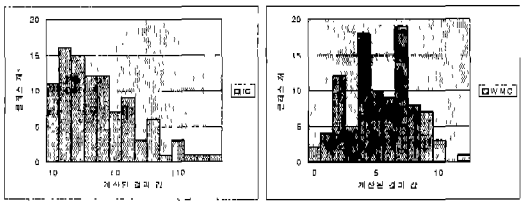


그림 7 IC와 WMC 값의 분포

외 클래스에서 다른 결과를 나타내고 있는데, 이 경우는 *IC*의 값은 작으나 설계 단계에서 책임을 수행하기 위한 복잡한 연산이 정의됨으로써 WMC의 값이 커지게 되는 것이다.

*IC*는 클래스 내에 있는 책임의 개수가 늘어남에 따라 큰 폭으로 증가하는 경향이 있다. 이것은 책임의 개수가 증가함에 따라 얼마나 더 복잡해지는지를 쉽게 판단할 수 있는 지표가 될 수 있다. WMC의 경우는 단순히 클래스가 가지고 있는 책임의 개수가 되므로, 그 값의 범위가 7±2의 범위를 크게 벗어나지 않고 있다. 그러나, *IC*는 협력 관계에 있는 협력자들의 인터페이스 크기를 고려하기 때문에, 책임의 개수가 적은 클래스라고 해도 협력 관계로 인하여 인터페이스 복잡도가 증가하는 경향을 보이게 된다.

아래의 그림 8은 *CC*와 CBO 값의 분포를 표현한 그래프이다.

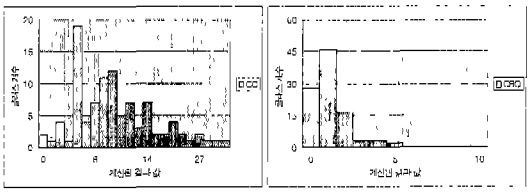


그림 8 CC와 CBO의 값의 분포

*CC*는 외부 클래스와의 결합정도를 나타내는 것이기 때문에 *CC*의 값이 낮은 경우, CBO의 값도 작다는 것을 알 수 있었다. *CC*와 CBO의 상관관계가 높지만, 몇몇 클래스에서 다른 결과가 나왔는데, 그러한 경우는 CBO의 값은 작지만 클래스의 책임이 많은 경우이다. 그러나, *CC*의 값이 고르게 분포되어 있는 반면, CBO의 값은 2보다 큰 값은 거의 없이 분포되어 있음을 알 수 있다. 이렇게 값이 차이가 나는 것은 CBO가 단순히 참조하고 있는 외부 클래스의 개수만으로 복잡도를 평가하였기 때문이며, *CC*는 이들 외부 클래스에 대하여 반응하는 클래스의 책임들을 이해하게 되는 복잡성을 고려하였기 때문이다.

## 5. 결론 및 향후 연구과제

본 논문에서는 RUP의 분석단계에서 생성되는 분석 클래스에 대한 복잡도를 측정하기 위한 척도를 정의하였다. 분석 단계는 문제 영역에 존재하는 중요한 대상이나 개념들을 클래스로 모델링하고, 시스템은 이들 분석

클래스로부터 생성된 객체들에 의하여 구축된다. 그러므로, 안정적이고 유지보수와 확장이 용이한 시스템을 구축하기 위해서는 클래스 단위의 설계가 잘 이루어져야 한다. 뿐만 아니라, 분석 단계에서 시스템의 복잡도를 줄이게 되면, 개발 및 유지보수에 소요되는 시간과 노력을 크게 감소시킬 수 있게 된다. 따라서, 분석 클래스에 대한 복잡도 측정은 필수적이다. 그러나, 기존에 연구되었던 척도들이 분석 단계에 나타나는 것보다 훨씬 더 상세한 정보를 요구하였기 때문에, 실제로 객체지향 분석 척도라고 말할 수 없었던 것이 사실이다. 또한, 설계 척도의 일부는 구현이후의 소스 코드에 대하여 측량할 수 있는 것이었다. 이러한 문제를 개선하기 위하여, 제안된 척도는 분석 초기의 클래스에 대한 정보에 기반을 두고 정의되었다. 즉, 분석 과정에서 추출되는 분석 클래스에 대한 정보로서 클래스가 갖는 책임과 협력관계 및 상속 관계를 기반으로 하였다.

제안된 척도는 9가지 공리적 성질에 대하여 이론적인 검증을 하였다. 그리고, 텍스트 마이닝 기법을 사용하여 사용자의 질문에 자동으로 응답하는 시스템에 대하여 사례 연구를 하였다. 이 시스템의 분석 클래스에 대하여 제안된 척도 *CC*와 *IC*의 값을 계산하고, 설계 모델을 사용하여 [CK 94]에서 제안한 CBO와 WMC 값을 계산하였다. *CC*와 CBO, *IC*와 WMC의 값을 비교해 본 결과 제안된 복잡도 척도의 계산 결과 값이 큰 클래스의 경우에는 설계 이후 단계에서도 역시 복잡도가 커지게 되는 것을 알 수 있었다. 따라서, 개발 초기인 분석 단계에서 클래스의 복잡도를 측정하여 피드백(feedback)을 제공할 수 있게 된다. 개발 초기에 제공되는 피드백은 나머지 단계에 대한 계획이나 수정(correction) 작업을 줄여줄 수 있다. 그 결과로 유지보수 비용의 절감 효과와 함께 개발 비용 및 노력의 절감 효과를 기대할 수 있다.

향후 연구과제로는 복잡도를 측정한 결과를 이용하여, 기능성, 신뢰성, 효율성 그리고 이식성과 같은 소프트웨어 품질 특성들을 어떻게 평가할 수 있는지에 대한 연구가 필요하다. 또한, 복잡도를 측정하기 위하여 필요한 자료를 자동으로 수집하고 복잡도를 계산하는 자동화 도구가 현재 개발중이다.

## 참 고 문 헌


[1] T. J. McCabe, "A Complexity Measure." IEEE Transactions on Software Engineering, December 1976, Vol. SE-2, No.(4), pp.308-320.

[2] Halstead, M., *Elements of Software Science*, New York : Elsevier North-Holland, 1977.

[3] Basil, W. G., Zelkowitz, M. V., "Program Complexity Using Hierarchical Abstract Computers," Computer Language, Vol. 13, No. 3, pp.109-123, 1988.

[4] Wilde N., Huitt R., "Maintenance Support for Object-Oriented Programs," IEEE Transactions on Software Engineering, Vol.18, pp.1038-1044, 1992.

[5] E. M. Kim, O. B Chang, S. Kusumoto, and T. Kikuno, "Analysis of Metrics for Object-Oriented Program Complexity," Proceedings of COMPSAC '94, pp.201-207, 1994.

[6] Xiaowei Liu, "Object-Oriented Software Metrics," Ph. M Thesis, Department of Computer Science, University of Manitoba, Canada, 1999.

[7] S. R. Chidamber and C. F. Kemerer, "A Metric Suite for Object Oriented Design," IEEE Transactions on Software Engineering, Vol. 17, No. 6, pp.636-638, June 1994.

[8] B. Henderson S., *Object-Oriented Metrics : Measures of Complexity*, Prentice-Hall, 1996.

[9] F. B Abreu and W. Melo, "Evaluation the Impact of Object-Oriented Design on Software Quality," Proceedings of the 3rd International Software Metrics Symposium, Berlin, Germany, pp.90-99, March 1996.

[10] Li, Wei and S Henry, "Maintenance Metrics for the Object-Oriented Paradigm," Proceedings of First International Software Metrics Symposium, *IEEE Computer Society Press*, pp.52-60, 1993.

[11] R. Sharble, S. Cohen, "The Object-Oriented Brewery : A Comparison of Two Object-Oriented Development Methods," SIGSOFT Software Engineering Notes, Vol. 18, No. 4, pp. 60-73, 1993.

[12] 김유경, 박재년, "클래스 기반 분석 모델에 대한 복잡도 메트릭", 제27회 한국 정보과학회 춘계학술발표대회 논문집, 제27권, 제 1호, pp.516-518, 2000.

[13] 김은미, 전형수, 장옥배, Shinji KUSUMOTOD, Tohru KIKUBO, and Yoshihiro TAKADA, "C++ 프로그램의 복잡도 척도 및 평가 도구", 정보과학회 논문지(C), 제 3권, 제6호, pp.656-665, 1997.

[14] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley Longman, 1998.

[15] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Software Development Process*, Addison-Wesley Longman, 1998.

[16] 채흥석, 권용래, 배두환, "객체지향 시스템의 클래스에 대한 응집도", 정보과학회 논문지, 제26권, 제9호, pp.1095-1104, 1999.

[17] R. W. Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall. 1990.

[18] E. J. Weyuker, "Evaluating software complexity measures," IEEE Transactions on Software Engineering, Vol. 14 No. 9, pp.1357-1365, 1988.
[19] J. C. Cherniavsky and C. H. Smith, "On Weyuker's axioms for software complexity measures," IEEE Transactions on Software Engineering, Vol. 17 No. 6, pp.636-638, June 1991.

김 유 경

1991년 숙명여자대학교 수학과 학사. 1994년 숙명여자대학교 전산학과 석사. 2001년 숙명여자대학교 컴퓨터과학과 박사. 현재 숙명여자대학교 정보과학부 강의전담교수. 관심분야는 객체지향 방법론, 소프트웨어 품질 평가, 소프트웨어 공학

박 재 년

1966년 고려대학교 학사. 1969년 고려대학교 석사. 1981년 고려대학교 박사. 1979년 ~ 1983년 전남대학교 전산통계학과 교수. 1983년 ~ 현재 숙명여자대학교 정보과학부 교수. 관심분야는 시스템 개발 방법론, 소프트웨어 품질 평가, 컴포넌트 기반 소프트웨어 공학