컴포넌트 복잡도, 특화성 및 재사용성 측정을 위한 메트릭

Software Component Metris for Complexity, Customizability, and Reusability

이 숙 희* 조 은 숙**
Sook-Hee Lee Eun-Sook Cho

요 약

최근 컴포넌트에 기초한 소프트웨의 발전은 효과적인 새로운 소프트웨어의 발전 시류로서 받아들여지고 있다. 1990년대 후반에 컴포넌트에 기초한 소프트웨어 공학(CBSD)의 도입이래 CBSD의 연구는 주로 컴포넌트의 모델링과 방법론, 구조및 컴포넌트 플랫폼에 초점이 맞춰져 왔으나, 시장에서 필요한 컴포넌트의 수가 증가함에 따라 다양한 특성의 컴포넌트를 명시하기 위한 측정기준을 제시하는 것이 더욱 중요해졌다.

이 논문에서는 소프트웨어 컴포넌트의 복잡성, 특화성, 재사용성 등을 측정하는 기준을 제시할 것이다. 복잡성 기준(metrics)은 다양한 컴포넌트들을 비교하고 측정하는데 사용될 수 있을 것이며 특화성 기준은 컴포넌트들이 조직의 특별한 요구조건을 얼마나 효율적이고 광범위하게 맞출 수 있느냐를 측정하는데 이용되어진다. 재사용성은 애플리케이션 개발에 재사용할 수 있는 정도를 측정하는데 사용될 수 있다.

Abstract

Recently, component-based software development is getting accepted in industry as a new effective software development paradigm. Since an introduction of component-based software engineering(CBSE) at later 90's, the CBSD research has focused largely on component modeling, methodology, architecture and platform. However, as the number of components available on the market increases, it becomes more important to make metrics to measure the various characteristics of components.

In this paper, we propose metrics for measuring the complexity, customizability, and resuability of software components. Complexity of metrics can be used to evaluate the complexity of components. Customizability is used to measure how efficiently and widely the components can be customized for specific requirements organization. Resuability can be used to measure the degree of features that is reused in building applications.

1. 서 론

1980년 대 이래로 객체(Object) 기술은 종종 소프트웨어 재사용 문제를 풀기 위한 핵심기술로 알려져 왔다. 하지만 객체기술은 기업적용 개발 프로젝트를 위해서는 너무 미미한 단위로 알려져 왔다. 컴포넌트 기술은 소프트웨어 개발에 있어 재사용성을 옹호해줄 새로운 기법으로 소개되어 왔다. COM+, EJB, CCM, 컴포넌트 모델링 기술, 개발 방법 및 개발 과정 등 다양한 컴포넌트 플랫폼들이 소개된다[1,3,5].

컴포넌트 지향의 소프트웨어 개발은 객체지향(OO) 방법에 기반으로 약간의 다른 접근을 요구한다. 객체지향 방법이 기능과 객체 모델을 정의하는 시스템을 개발하는 반면에 컴포넌트 지향 개발(CBD)은 공유성과 변동성 분석, 컴포넌트, 컴포넌트 접속장치, 그리고 컴포넌트들 간의 관계 등을 활용하는데 중점을 두고 있다[1]. 그러므로 다양한 객체기술 프로그램을 위해 발전된 측정기준(metrics)은 CBD 프로세스에 똑같이 적용할 수 없다. 그러므로 이 논문에서 우리는 효

* 정회원 : 서경대학교 인터넷정보학과 교수
oleesh@skuniv.ac.kr
** 정회원 : 동덕여대 데이터정보학과 강의전임교수
escho@dongduk.ac.kr

율적으로 CBD프로세스를 적용할 수 있는 컴포넌트 기준을 제시한다.

이 논문은 다음과 같이 구성되어 있다. 2장에서 관련 연구로 객체지향의 메트릭에 대해 언급한다. 이 메트릭들은 객체 구조에 중점을 두고 있는데, 메소드와 클래스와 같은 각각의 개별 복잡도와 객체들간의 상호 작용을 측정하는 외부 복잡도를 반영한다. 그리고, 기존의 객체지향 메트릭을 CBD에 적용할 때의 문제등을 제시한다.

3장에서는 컴포넌트 품질 즉, 복잡도(Complexity), 특화성(Customizability), 재사용성(Reusability)측정을 위한 3가지 메트릭(복잡성, 특화성, 재사용성)을 제시한다. 특히 각각의 메트릭에 대한 정의와 CBD에 적용가능성을 제안한다. 4장에서는 이렇게 제시된 측정기준에 대한 사례 연구를 제시하며 기존의 메트릭들과 비교한다. 마지막으로 5장에서는 결론 및 향후 연구 과제를 제시한다.

2. 관련 연구

2.1 객체지향 메트릭

지금까지 객체지향 시스템을 위한 여러 종류의 메트릭들이 제안되어 왔다. 객체지향 메트릭은 기본적인 구조들을 측정한다. 즉 부절적한 설계가 설계와 코드의 품질 속성에 미치는 영향을 측정한다. 기존의 객체지향 메트릭들은 주로 클래스, 결합도, 그리고 상속의 개념에 적용된다.

2.1.1 WMC(Weighted Methods per Class)

WMC는 클래스내에서 구현된 메소드의 수 이거나 메소드의 복잡도(메소드 복잡도는 사이클로메틱 복잡도로 측정된다)의 합을 의미한다. 이 메트릭은 구현하기가 어렵다. 왜냐하면 모든 메소드가 상속으로 인한 클래스계층 내에서 평가할 수 없기 때문이다. 많은 메소드와 연관된 메소드들의 복잡도는 클래스를 개발과 유지 보수에 얼마나 많은 시간과 노력이 필요한지를 알 수 있게 측정해준다. 클래스내의 메소드가 많아지면 많아질수록 자식 클래스에게 돌아가는 영향력은 점점 커진다. 자식 클래스는 부모 클래스 안에서 정의되는 모든 메소드를 상속받기 때문이다. 많은 메소드를 가진 클래스들은 재사용의 가능성을 제한하여 좀 더 애플리케이션 스팩으로 되려한다 [7,8.9.10].

2.1.2 RFC(Response for a Class)

RFC는 클래스의 객체에 응답하는 메시지나 클래스의 클래스내의 몇 몇의 메소드로부터 호출될 수 있는 메소드 집합의 수이다. 또한 RFC는 클래스 계층 안에서 평가되는 모든 방법을 포함한다. 이 메트릭은 많은 메소드들과 다른 클래스와의 통신의 양으로 클래스내의 복잡도의 조합을 찾는다. 메시지를 통해 클래스에서 호출되는 메소드가 많아지면 많아질수록 클래스의 통신과 디버깅은 복잡하게 된다. 왜냐하면 테스팅과 디버깅은 테스터의 역할에서 높은 이해력을 요구하기 때문이다. 가능한 응답에 대한 최악의 경우의 값은 적절한 시험 시간의 분배를 조정하게 될 것이다 [7,8.9.10].

2.1.3 LCOM (Lack of Cohesion)

응집도의 결여는 인스턴스 변수나 속성으로 클래스내의 메소드들의 상이점을 측정한다. 응집도가 높은 모듈은 단독으로 사용되어야 한다. 높은 응집도는 클래스의 세분화가 잘됨을 의미한다. 낮은 응집도나 응집도의 결여는 복잡도를 증가시켜서 개발 과정 동안에 에러의 양을 증가시킨다. 응집도가 높을수록 단순하고 재사용성이 높음을 의미한다. 낮은 응집도를 가진 클래스들은 응집도가 높아지는 두 개 혹은 그이상의 서버 클래스들로 나뉘어 질 수 있다[7,8,9,10,11].

2.1.4 CBO(Coupling Between Object Classes)

CBO는 어떤 한 클래스가 결합되는 다른 클래스들의 수이다. CBO는 클래스가 의존하는 비상

속 관련 클래스 계층의 수로 계산함으로써 측정된다. 과도한 결합도는 모듈 설계를 어렵게 하고 재사용을 막는다. 클래스가 독립적일수록 다른 애플리케이션에서의 재사용을 쉽게 해 준다. 결합의 수가 많아질수록 다른 부분의 디자인의 변화에 민감하게 되고, 따라서 유지 보수가 어렵게 된다. 강한 결합도는 시스템을 복잡하게 한다. 왜냐하면 클래스가 다른 클래스들과 상호 관련되어있을 경우 자체적으로 변경, 교정, 또는 이해를 더 어렵게 하기 때문이다. 시스템 설계시 클래스들간의 가장 약한 결합도를 가지게 하는 것은 시스템의 복잡도를 줄여준다. 이런 점은 모듈성을 좋게 하고 캡슐화를 증대시킨다.

2.1.5 DIT(Depth of Inheritance Tree)

상속 구조를 가진 클래스의 깊이는 클래스 노드로부터 트리의 루트 노드까지의 최대의 수이며, 조상 클래스의 개수에 의해 측정된다. 계층내의 클래스가 많아질수록 클래스가 상속하는 메소드가 많아져서 행위를 측정하는 것을 더 어렵게 한다. 메소드와 클래스들이 더 많이 관련되어 있기 때문에 트리의 깊이가 깊을수록 디자인의 복잡도를 더 크게 한다. 그러나 상속되는 메소드들의 재사용성은 더 커진다. DIT를 뒷받침하는 메트릭은 상속되는 메소드들의 수이다.

2.1.6 NOC(Number of Children)

자식의 개수는 계층 내에서 어떤 클래스의 하위에 있는 서버 클래스들의 개수이다. 이것은 클래스가 설계와 시스템 상에서 가질 수 있는 잠재적인 영향력을 보여주는 지표이다. 자식의 개수가 많아질수록 부모의 부적절한 추상화가 더 커지며 서브 클래싱의 잘못 사용되는 예가 될 수 도 있다. 그러나 자식의 개수가 많아질수록 재사용성의 형태인 상속으로 인해 재사용성은 더 커진다. 만약 클래스가 많은 자식을 가지고 있다면 그 클래스는 그 클래스의 메소드에 테스팅을 더 요구하게 되고, 따라서 테스팅 시간이 늘어나게 된다 [6,7,8,9,10].

2.2 기존 객체지향 메트릭의 문제점

이 절에서는 기존의 객체지향 메트릭을 컴포넌트 개발과 CBSD에 적용하는데 있어서의 문제점에 대해서 논하고자 한다. 이런 점에서 논의된 객체지향 메트릭 자체를 그대로 컴포넌트 품질 측정을 하는데 적용하는 것은 부적절하다. 이유는 다음과 같다.

첫째, 측정 단위가 다르기 때문이다. 객체지향 메트릭은 객체나 클래스에만 초점을 맞춘다. 컴포넌트는 한 개 이상의 클래스와 한 개 이상의 인터페이스로 구성된다. 기존의 객체지향 메트릭은 컴포넌트의 복잡도, 응집도 혹은 결합도를 측정하는데 있어서 컴포넌트 그 자체를 고려하지 않거나 컴포넌트의 인터페이스를 고려하지 않는다. 그러므로 컴포넌트 그 자체의 복잡도를 측정하는 새로운 메트릭이 요구된다.

둘째, 측정 요소가 충분하지 않다. 객체지향 애플리케이션들이 클래스들로만 개발되기 때문에 거의 모든 객체지향 메트릭은 클래스들, 메소드들 그리고 클래스 계층의 깊이를 고려하여 복잡도나 재사용성을 측정한다. 그런데 이러한 요소들만 고려하는 것은 컴포넌트의 복잡도나 재사용성을 측정하기에 충분하지 않다. 왜냐하면 컴포넌트는 훨씬 많은 정보, 예를 들면 인터페이스, 인터페이스 메소드 등을 가지고 있기 때문이다. 기존의 객체지향 메트릭이 클래스나 객체들의 특화성을 고려하지 않는 반면 컴포넌트의 특화성은 CBD에 있어서 매우 중요하다. 왜냐하면 컴포넌트의 특화성이 CBD안의 컴포넌트의 재사용성에 영향을 주기 때문이다.

3. 컴포넌트 메트릭 정의

이 장에서 우리는 복잡성, 특화성, 재사용성 등

을 측정할 수 있는 측정기준을 제시한다. 설계된 컴포넌트들의 메트릭을 측정하기 위한 기준을 정의하는 것뿐 아니라 이미 구현된 컴포넌트들의 메트릭을 측정하는 기준을 제시한다. 그러므로 제안된 측정기준은 설계기준과 구현기준으로 분류된다.

3.1 복잡도 측정

복잡도를 측정하기 위해서는 전통적인 프로그램에서 순환(Cyclomatic)복잡도가 사용된다. 순환복잡도는 전통적인 방법에서 알고리즘의 복잡성을 평가하는데 사용된다. 이것은 전통적인 방법을 테스트하기 위해 필요한 수많은 테스트 케이스들 중의 하나이다. 순환복잡도(Cyclomatic Complexity)를 산출하는 공식은 에지(edge)의 수에서 노드(nodes)의 수를 차감한 후 2를 더한다. 낮은 순환 복잡도를 지닌 방법이 일반적으로 나은 방법이다. 순환복잡도는 컴포넌트의 고유성 때문에 컴포넌트의 복잡도를 측정할 수 없고 개별 방법들의 순환복잡도는 컴포넌트의 복잡도를 평가하기 위해 다른 측정방법과 연결하여 사용할 수 있다. 그러므로 우리는 순환복잡도를 결합함으로써 컴포넌트의 복잡도를 측정하는 새로운 복잡도 측정 기준을 제시하며 그것은 컴포넌트 복잡도 메트릭(CCM)이다. 우리는 CCM을 복잡도 측정기준의 4가지 종류로 구분하며 그것들은 다음과 같다. 컴포넌트 일반복잡도(CPC), 컴포넌트 정적복잡도(CSC), 컴포넌트 동적복잡도(CDC)와 컴포넌트 순환복잡도(CCC)이다. 이러한 컴포넌트 복잡도 측정기준들 중에 컴포넌트 순환복잡도가 컴포넌트 구현 단계에서 사용되어지는 반면에 다른 복잡도 기준들은 컴포넌트 설계 단계에 적용될 수 있다.

3.1.1 컴포넌트의 일반 복잡도(CPC)

각각의 컴포넌트의 복잡도를 측정하기 위해 사용되는 첫번째 접근 방법은 컴포넌트의 일반 복잡도이다.

컴포넌트의 일반 복잡도는 클래스, 추상화클래스와 인터페이스의 합을 계산함으로써 컴포넌트의 복잡도 그 자체를 측정하며, 클래스들과 메소드들의 복잡도를 측정하는 기준이다. 컴포넌트의 일반 복잡도는 다음과 같은 식으로 표현된다[7]:

[정의 1] $CPC(C) = CmpC + \sum_{i=1}^{n} CC_i + \sum_{j=1}^{n} MC_j$

여기서,

CmpC : 클래스들, 추상화클래스들 그리고 인터페이스들을 합산하여 계산된다.

$\sum_{i=1}^{n} CC_i$: 각 클래스의 복잡도, 그리고

$\sum_{j=1}^{n} MC_j$: 각 메소드의 복잡도.

CmpC 는 클래스들, 추상화 클래스들, 인터페이스들 그리고 메소드들의 수를 합산하여 계산된다. CmpC의 정의는 다음과 같다.

[정의 2]

$$CmpC = \sum_{i=1}^{n}(Count(C_i) * W(C_i)) + \sum_{j=1}^{n} Count(I_j) + \sum_{k=1}^{n}(Count(M_k) * W(M_k))$$

여기서,

Count(C) : 컴포넌트에 포함된 클래스의 수

W(C) : 각 클래스의 가중치

I : 컴포넌트에서 사용되거나 제공되는 인터페이스

Count(M) : 컴포넌트에 포함된 클래스들의 메소드들의 수, 그리고

W(M) : 각 메소드의 가중치

컴포넌트 내에 결합되어 있는 클래스들은 내부 클래스와 외부클래스로 구분된다. 외부 클래스는 다른 재사용된 라이브러리(Library) 또는 패키지로부터 불러 사용한 클래스이며, 내부 클래스는 컴포넌트 영역 내에서 컴포넌트 분석과 설계를 통해 이 연구에서는 외부클래스가 구현된 클래스이기 때문에 내부 클래스에 가중치를 두며, 외부 클

래스의 메소드들은 단지 불러 쓰기 때문에 내부 클래스의 메소드에 가중치를 부여한다. 컴포넌트에 포함된 각 클래스들의 복잡도(CC)는 개별 속성(SA)과 복합 속성(CA) 등과 같은 각 클래스의 개별 속성들의 합으로 계산된다. CC는 다음과 같이 정의한다.

[정의 3]

$$CC = \sum_{i=1}^{n} (Count(SA_i) + \sum_{j=1}^{n} (Count(CA_j) \times W(CA_j))$$

여기서,

Count(SA) : 개별 속성의 수

Count(CA) : 복합 속성의 수

W(CA) : 각 각의 복합 속성의 가중치.

클래스의 각각의 메소드에 대한 복잡도는 각 메소드의 매개변수를 더함으로써 계산된다. 단순 매개변수는 SP로 계산되며 객체들과 같은 복합 매개변수는 CP로 계산한다. 다른 매개변수들을 포함하는 복합 매개변수(Complex Arguments)에는 가중치가 주어진다.

MC는 다음과 같은 공식으로 정의된다:

[정의 4]

$$MC = \sum_{i=1}^{n} Count(SP_i) + \sum_{j=1}^{n} (Count(CP_j) \times W(CP_j))$$

여기서,

Count(SP) : 단순 매개변수의 수

Count(CP) : 복합 매개변수의 수, 그리고

W(CP) : 각 각의 매개변수의 가중치

3.1.2 컴포넌트 정적 복잡도(Component Static Complexity)

각 컴포넌트의 복잡도를 측정하기 위한 두 번째 접근방법은 컴포넌트의 정적 복잡도이다. 컴포넌트의 정적 복잡도는 컴포넌트에서 선언된 클래스들, 인터페이스들, 메소드들 및 매개변수들의 수에 초점을 맞춘 반면에 컴포넌트의 정적 복잡도는 컴포넌트의 내부 구조가 얼마나 복잡하느냐에 중점을 두며, 컴포넌트의 정적 복잡도는 정적인 관점에서 컴포넌트내의 내부구조의 복잡도를 측정하는 기준이다. 그러므로 각 컴포넌트의 정적인 복잡도는 컴포넌트에 포함된 클래스들 간의 상호관계를 측정함으로써 계산된다. 다음과 같은 공식으로 컴포넌트의 정적 복잡도를 정의한다.

[정의 5] $$CSC = \sum_{i=1}^{m} (Count(R_i) \times W(R_i))$$

여기서,

Count(R) : 클래스들 사이의 각 각의 관계의 수, 그리고

W(R) : 각 관계의 가중치

UML[4]에는 클래스들간의 4가지 상호관계가 있다. 클래스들 사이의 접근성에 따라 상호관계의 가중치 규모가 정해진다. 우리는 다음과 같은 우선 순위로써 가중치를 부여한다.

의존도(Dependency) < 연관화(Association) < 일반화(Generalization) < 집단화(Aggregation) < 합성(Composition)

상호관계를 설명함에 있어, 클래스들 간에 N진 관계가 있다면 N진 관계는 반드시 2진 관계로 변환되어야 한다.

3.1.3 컴포넌트 동적 복잡도(Component Dynamic Complexity)

각 컴포넌트의 복잡도를 측정하기 위한 세 번째 접근방법은 컴포넌트 동적 복잡도이다. 컴포넌트 동적 복잡도가 컴포넌트의 내부 구조가 얼마나 복잡하느냐에 중점을 두는 반면 컴포넌트 동적 복잡도 는 컴포넌트 내에서 얼마나 많은 메시지가 전해지느냐에 중점을 둔다. 컴포넌트 동적 복잡도는 동적인 관점에서 컴포넌트 내부에 내부 메시지 전송의 복잡성을 측정하는 기준이다. 그러

므로 각 컴포넌트의 동적 복잡도는 컴포넌트에 있는 클래스들 간에 전달된 메시지 수를 셈하므로써 계산된다. 컴포넌트의 동적 복잡도를 다음과 같은 공식으로 정의한다:

[정의 6] $CDC = \sum_{i=1}^{m} DC(IM_i)$

여기서,

$\sum_{i=1}^{m} DC(IM_i)$: 각각의 인터페이스 메소드의 복잡도

[정의 7]

$$DC(IM) = \sum_{i=1}^{n} (Count(Msg_i) \times Freq(Msg_i) - MC(Msg))$$

여기서,

Msg : 클래스들 사이에 전달되는 메시지

Freq(Msg) : 클래스들 사이에 전달되는 메시지들의 빈도수, 그리고

MC(Msg) : 정의 4에서 정의된 MC와 같은 각 메시지의 복잡도

3.1.4 컴포넌트 순환 복잡도(Component Cyclomatic Complexity)

각 컴포넌트의 복잡도를 측정하기 위한 네 번째 접근방법은 컴포넌트 순환 복잡도이다. 이전의 세 가지 측정기준들이 컴포넌트 설계시에 사용되어지는 반면에 컴포넌트 순환 복잡도는 컴포넌트의 구현이후에 사용되어진다. 그러므로 다른 세 가지 측정기준들은 클래스 다이어그램, 상호 교류 다이어그램 및 컴포넌트 다이어그램을 사용함으로써 계산되어지는 반면에 컴포넌트 순환 복잡도는 개발된 소스 코드를 사용함으로써 계산되어진다. 컴포넌트 일반 복잡도 와 컴포넌트 순환 복잡도 사이의 차이점은 컴포넌트의 인터페이스 내에 선언된 인터페이스 메소드에 대한 복잡도가, 전통적 프로그램에서 사용된 순환적인 복잡도 측정기준에 기초하고 있다는 점이다. 컴포넌트 순환 복잡도는 다음과 같은 공식으로 정의되어진다.

[정의 8] CCC = $CmpC + \sum_{i=1}^{n} CC_i + \sum_{i=1}^{n} MC_i + \sum_{i=1}^{n} CCM_i$

여기서,

CmpC : 정의 2에서의 클래스, 인터페이스 그리고 인터페이스 메소드의 합

$\sum_{i=1}^{n} CC_i$: 컴포넌트에 포함된 각 클래스의 복잡도의 합, 그리고

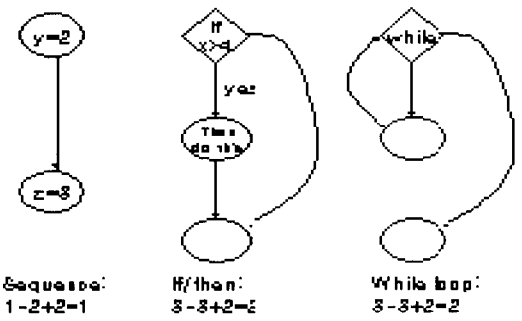
$\sum_{i=1}^{n} MC_i$: 각 각의 인터페이스 메소드의 복잡도의 합

각 클래스와 인터페이스 메소드의 복잡도는 정의 3과 정의 4와 동일하다. 그러나 한 개의 클래스에서 구현된 각 메소드의 순환 복잡도는 컴포넌트 순환 복잡도가 구현된 컴포넌트의 소스코드를 사용하기 때문에 이를 통해서 산정 된다. 순환 복잡도 방법은 다음과 같은 공식으로 정의한다.

[정의 9] = $\sum_{i=1}^{n} CCM_i = edges - nodes + 2$

정의 9는 [5]을 참조하면 된다. 순환 복잡도를 계산하는 공식은 에지의 수에서 노드의 수를 차감한 후 2를 더한다.

단 한가지 경로가 있는 곳, 선택과 옵션이 없는 에지에서의 연속성을 위해 단지 한 개의 테스트 케이스가 필요하다. If문은 2개의 선택을 가진다. 만일 조건이 맞다면 한가지 경로(One Path)가 테스트되며, 아니면 다른 경로가 테스트된다. 그림 1은 4가지 기본 프로그램 구조를 위한 순환 복잡도 계산의 예를 보여준다.



(그림 1) 순환 복잡도의 예

3.2 특화성 측정

컴포넌트 특징중의 하나가 컴포넌트 특화성(Customizability)이다. 만약 컴포넌트가 특화를 위한 인터페이스를 공급하지 못한다면, 컴포넌트의 재사용성은 저하된다. 그 이유는 어플리케이션 개발자들이 그들의 목적에 맞게 컴포넌트를 재사용하도록 맞추기를 원하기 때문이다. 그러므로 컴포넌트의 특화성은 컴포넌트의 개발 과정에서 고려되어야 한다. 이 논문에서 우리는 컴포넌트 설계 단계 또는 개발 후에 사용되는 특화성 측정기준을 제시한다. 특화성을 측정하기 위하여 우리는 다음과 같은 공식으로 컴포넌트의 가변성(Variability)을 사용한다.

[정의 10] $$CV = \frac{\sum_{i=1}^{m} Count(CVM_i)}{\sum_{j=1}^{n} Count(CIM_j)}$$

여기서,

CV : 특화성을 측정하기 위한 컴포넌트 가변성

Count(CVM) : 특화를 위한 메소드의 수

Count(CIM) : 각 각의 인터페이스 내에 선언된 메소드의 수

정의 8에 따라, CVM 은 다음 식으로 재 정의된다.

[정의 11]

$$CVM = \sum_{i=1}^{m} (Count(CVMa_i)) + \sum_{j=1}^{k} Count(CVMm_j) + \sum_{k=1}^{o} Count(CVMw_k))$$

여기서,

Count(CVMa) : 속성 특화를 위한 메소드의 수

Count(CVMm) : 행위 특화를 위한 메소드의 수

Count(CVMw) : 워크플로우 특화를 위한 메소드의 수

W(CVMm) : 행위 특화 메소드에 대한 가중치, 그리고

W(CVMw) : 워크 플로우 특화 메소드에 대한 가중치

정의 11에서 주어진 바와 같이, 우리는 행위 특화 메소드와 워크플로우 특화 메소드에 가중치를 부여한다. 그 이유는, 그러한 메소드들이 속성 특화 메소드에 비해 더 복잡하기 때문이다. 더구나, 워크플로우 특화 메소드들은 여러 가지 비즈니스 방법들을 포함하기 때문에 행위 특화 메소드보다 더욱 복잡하다. 그래서 우선 순위는 다음과 같이 표현된다.

속성 특화 메소드들 < 행위 특화 < 워크플로우 특화 메소드들

3.3 재사용성 측정

이 논문에서 컴포넌트의 재사용성을 측정하기 위해 두 가지 접근방법을 제안한다. 첫째는 컴포넌트가 얼마나 재사용성이 높은가를 측정하는 방법이고, 다른 하나는 컴포넌트가 특별한 어플리케이션내에서 얼마나 많이 재사용 되는지를 측정하는 방법이다.

첫번째 접근 방법은 컴포넌트 자체의 재사용성(CR)이다. 컴포넌트 자체의 재사용성 측정방법은 컴포넌트 개발 과정 중 설계 단계에서 사용될 수 있을 것이다. 컴포넌트 자체의 재사용성은 전체 인터페이스 메소드들의 합에서 공통기능을 공급하는 인터페이스 메소드들의 합을 나눔으로써 계산된다. 컴포넌트 자체의 재사용성은 다음과 같이 정의된다.

[정의 12] $$CR = \frac{\sum_{i=1}^{n} (Count(CCM_i))}{\sum_{j=1}^{m} Count(CIM_j)}$$

여기서,

Count(CCM) : 도메인내에서 여러 개의 어플리케이션 사이에 공통기능을 제공하기 위한 각 각의 인터페이스 메소드의 수, 그리고

Count(CIM) : 컴포넌트로 제공되는 인터페이스내에 선언된 메소드들의 수

두번째 접근 방법은 컴포넌트 소프트웨어개발(CBSD)에서 매 어플리케이션마다 특별한 컴포넌트의 재사용 수준을 측정하는 방법이다. 컴포넌트 재사용 수준(CRL)은 CRL_{LOCs}과 CRL_{Func}로 나눌 수 있다. CRL_{LOCs}가 코드 라인들(LOC)을 사용함으로써 측정되는 반면에, CRL_{Func}은 컴포넌트가 어플리케이션 내에서 기능성을 지원 받을 수 있도록 그 기능성을 나눔으로써 측정된다

CRL_{LOCs}는 다음 식에서 처럼 특정 어플리케이션에 대해서 퍼센트로 표현된다.

[정의 13] $CRL_{LOCs}(C) = \frac{Reuse(C)}{Size(C)} \times 100\%$

여기서,

Reuse(C) : 어플리케이션 내에 재사용된 컴포넌트의 코드라인 수,

Size(C) : 어플리케이션 내에 만들어진 전체 코드라인 수.

CRL_{Func}은 다음처럼 표현된다:

[정의 14] $CRL_{Func}(C)$≈컴포넌트 내에 제공되는 기능성의 합/어플리케이션 내에서 요구되는 기능성의 합

정의 14에 따르면, 컴포넌트에서 지원되는 기능이 많으면 많을수록 어플리케이션 내에서 컴포넌트의 재사용성은 증가한다. 만약 이러한 측정방법을 같은 도메인 내에서 다른 어플리케이션이 사용된 컴포넌트에 적용한다면, 하나의 도메인 내에서 컴포넌트의 재사용성을 얻을 수 있다.

4. 사례 연구

복잡도, 재사용성, 특화성을 측정하기 위해 은행업무에 처리되는 몇 개의 분야에 대해 제안된 메트릭을 적용한다. 같은 목적을 위한 여러 가지 구성 요소들이 같은 도메인내에서 개발되면 그 구성요소들의 복잡도, 재사용, 특화성을 측정할 수 있기 때문이다. 이 장에서는, 메트릭을 컴포넌트 설계와 구현에 적용하여 측정결과를 논증하고 또한 현존하는 메트릭과 제안된 메트릭의 다른 점에 대해서도 말한다. 앞에서 설명했던 메트릭을 이용한 측정결과를 통해 우리는 컴포넌트 개발 혹은 컴포넌트를 기반으로한 소프트웨어 개발에 들어가는 비용과 노력을 추정할 수 있다. 더 나아가 컴포넌트 저장소에 개발된 컴포넌트를 기록하므로써 컴포넌트의 질을 측정할 수 있다.

4.1 복잡도 측정 결과

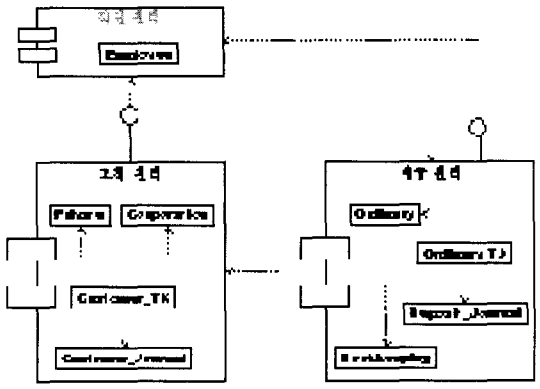
컴포넌트의 설계 단계에서 각 컴포넌트의 복잡도를 측정하기 위해, 우리는 먼저 컴포넌트 다이어그램을 개발해야 한다. 우리는 제안한 메트릭을 은행업무 도메인에 적용한다. 컴포넌트 다이어그램의 예를 그림 2에서 보여준다. 그림 2는 '고객 관리', '직원 관리' 및 '예금 관리'와 같은 은행 컴포넌트 다이어그램의 일부를 보여준다. 그림 2에 보여준 바와 같이, '고객 관리', '직원 관리'와 '예금 관리' 등 3가지가 있다. 또한 각 컴포넌트에는 한 가지 또는 그 이상의 클래스가 있다. 우리는 제시된 컴포넌트 일반 복잡도와 컴포넌트 정적 복잡도를 이용함으로써 각 컴포넌트의 복잡도를 측정한다. 예를 들어 '고객 관리'와 '예금 관리'에 대한 컴포넌트 일반 복잡도와 컴포넌트 정적 복잡도는 다음과 같이 측정된다.

CPC(고객 관리) = 47 + 66 + 13 = 126,

여기서, CmpC= 10 + 1 + 36 = 47,

$\sum CC_i$ = 30 + 9 * 4 = 66, 그리고

$\sum MC_i$ = 5 + 2 * 4 = 13.



(그림 2) 컴포넌트 다이어그램

(표 1) 관계에 대한 가중치 표

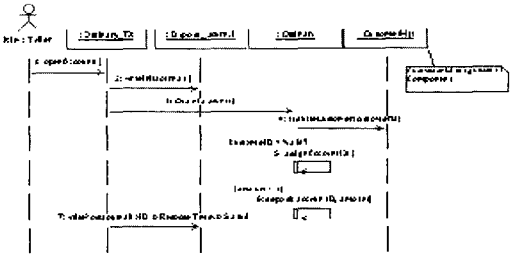
관 계	가중치
의존도	2
연관화	4
일반화eralization	6
집단화Aggregation	8
합성	10

CSC(고객 관리) = (0*2) + (1*4) + (4*6) + (0*8) + (0*10) = 28

표 1에서 보다시피, 우리는 가중치 표에 기초를 두고 각각의 상호 관계를 위한 가중치를 준다.

각 컴포넌트의 컴포넌트 동적 복잡도를 측정하기 위해, 사용 경우마다 순차도를 사용한다. 순차도의 예는 그림 3에서 보여준다. 그림 3은 '예금 관리', USE CASE내에 존재하는 클래스들 사이의 상호작용을 보여준다. Open Account()는 예금관리의 인터페이스 내에 선언된 인터페이스 메소드이다.

그림 3에서 보여진 것과 같이, 예금 컴포넌트와 같은 컴포넌트 안에 포함된 클래스들 사이에 여러 가지 메시지 흐름(flow)이 있다. 단 하나의클래스 다이어그램 또는 컴포넌트 다이어그램으로는 예금 관리의 복잡도를 측정하기 어렵다. 그러므로 상호교류 다이어그램으로 각 컴포넌트의 동적 복잡도를 측정할 수 있다. 이런 예의 다이어그램에 적용한다면 측정 결과는 다음과 같다.



(그림 3) 예금 관리의 'Open Account' 순차도

(표 2) 인터페이스 메소드들의 측정결과

인터페이스 메소드들	예금 관리
OpenAccount	13
CloseAccount	20
InquireAccount	9
InquireTransactionHistory	13
InquireCustomerAccount	9
Deposit	22
WithDraw	24
Transfer	17
InquireBookKeeping	11
AssignAccountId	2
CallInterest	5

DC(OpenAccount())=1+1+1+2+1+3+4=13

그림 3에서 보여준 것 같이 Open Account 인터페이스 메소드에는 6가지 메시지 흐름이 있다. 그러므로 정의 7의 공식을 적용함으로써 DC(Open Account)의 값을 구할 수 있다. 예금 관리 컴포넌트의 각 인터페이스 메소드들에 대한 결과는 DC를 사용함으로써 구해지며 결과는 표 2에 있으며 우리가 구한 컴포넌트 동적 복잡도(예금 관리)의 값은 다음과 같다.

컴포넌트 동적 복잡도(예금 관리) = 13 + 20 + 9 + 13 +9 +22 +24 +17 +11+2 +5=145

또한 우리는 코드 라인을 사용함으로써 각 컴포넌트에 대한 컴포넌트 순환 복잡도 가치를 계산한다.

우리는 EJB 빈의 형태에서 개발되었으며 순환

복잡도와 컴포넌트 일반 복잡도를 결합함으로써 각 컴포넌트의 컴포넌트 순환 복잡도(CCC)를 측정한다. 그 결과들은 다음과 같다.

CCC(고객 관리) = 47 + 66 + 13 + 98 = 224
CCC(예금 관리) = 114 + 69 + 68 + 70 = 321

여기서 우리는 각 메소드의 순환 복잡도를 적용함으로써 CCM을 계산한다. 컴포넌트에 포함된 각 각의 클래스에서 각 메소드에 대한 CCM이 계산된 후에, 각 CCM 값의 합은 고객 관리 컴포넌트의 CCM이 된다. 그 결과치는 98이다. CmpC의 값, 클래스 복잡도의 합, 메소드 복잡도의 합이 CPC의 값과 같다.

우리는 CCC의 측정 결과가 CPC의 측정 결과 보다 더 크다는 것을 알았다. 이것은 CPC의 복잡도가 커질수록 CCC의 복잡도도 커진다는 것을 의미한다.

4.2 특화성 측정 결과

각 컴포넌트의 특화성을 측정하기 위하여, 우리는 각 컴포넌트 명세를 위한 컴포넌트 명세를 사용한다. 컴포넌트 분석 단계에서 컴포넌트들의 확인된 공통성과 가변성이 한 도메인 내에서 개발될 것이다. 확인된 가변성 메소드들은 설계 단계에서 컴포넌트 명세내에 특화 메소드로 기술된다. 고객 관리와 예금 관리에 대한 특화 메소드들은 표 3에 기술되어 있다.

우리는 CV 메트릭을 사용하여 고객 관리 컴포넌트와 예금 관리 컴포넌트의 특화 메소드를 측정하며 그 산출된 결과는 다음과 같다.

(표 3) 특성화 메소드들

IcustomerManagement	IdepositManagement
SetCorporationID (corporationIDFlag): Boolean	SetAccountFlag (accountFlag):Boolean
SetCustomerIDFlag (customerIDFlag): Boolean	SetInterestFlag (interestFlag1, interestFlag2):Boolean

CV(고객 관리) = 2/110.18
CV(예금 관리) = 2/11 0.18

예를 들면, 예금 관리에 있어 2가지 특성화 메소드가 있다. CV(예금 관리)는 전체 인터페이스 메소드에서 특화 메소드를 나눔으로써 계산된다.

4.3 재사용성 측정 결과

우리는 설계된 컴포넌트에 CRL_{Func} and CRL_{LOCs}을 사용함으로써 재사용성을 측정한다. CRL_{LOCs}은 컴포넌트의 많은 부분들 중 얼마만큼이 어플리케이션 내에서 재 사용할 수 있는지를 퍼센트로 측정하기 때문에 CRL_{LOCs}은 구현된 어플리케이션에 적용된다. 예를 들어 예금 관리 컴포넌트와 고객 관리 컴포넌트의 CRL_{Func}은 다음과 같이 구할 수 있다.

CRL_{Func} (고객 관리) = 9/9=1
CRL_{Func} (예금 관리) = 9/110.819

우리는 고객 관리와 직원 관리, 그리고 예금 관리를 사용한 컴포넌트에 기반한 은행업무 시스템을 개발했다. 그리고 우리는 코드의 라인을 통해 은행업무 시스템의 개발에 있어 각 컴포넌트의 재사용 수준을 측정한다.

각 컴포넌트의 측정 결과는 다음과 같다.

CRL_{LOCs} (고객 관리) = 34/576*100% ≈ 5.9%
CRL_{LOCs} (예금 관리) = 28/576*100% ≈ 4.9%

4.4 평가

이 장에서 컴포넌트의 질과 그들의 장점과 단점 등을 측정하기 위해 제안된 다른 측정기준들을 보고자 한다. 표 4는 컴포넌트의 질을 측정하기 위한 접근 방법과 요소들을 나열했다. 표 4에서 보는 바와 같이, 설계 단계에 적용된 측정 메트릭

의 주요 요소의 수는 구현 단계에서 적용된 측정 메트릭의 수보다 적다. 그러므로 CCC와 CRL_{LOCs}을 사용하여 얻은 복잡도와 재사용성에 대한 측정 결과는 컴포넌트 일반 복잡도(CPC), 컴포넌트 정적 복잡도(CSC), 컴포넌트 동적 복잡도(CDC) 및 컴포넌트 재사용성(CR)을 사용하여 얻은 결과보다 정확하다.

하지만 컴포넌트 일반 복잡도, 컴포넌트 정적 복잡도, 컴포넌트 동적 복잡도 및 컴포넌트 재사용성은 컴포넌트 기반 소프트웨어 개발 단계에서 일찍 측정될 수 있기 때문에 컴포넌트의 개발과 컴포넌트에 기반한 소프트웨어 개발에 요구된 컴포넌트의 크기와 비용 및 노력의 정도를 예측할 수 있다.

표 4에서처럼 제안된 컴포넌트 지향의 메트릭들은 요구되는 시험 노력과 개발, 이해도, 유지보수성 및 재사용성을 평가하는데 도움을 준다. 이러한 정보는 표 5에 요약되어 있다. 제안된 컴포넌트 지향의 메트릭들은 컴포넌트의 개발자, 컴포넌트 조립자, 애플리케이션 개발자 및 프로젝트 관리자에게 중요한 정보를 제공한다.

(표 4) 다른 메트릭들과의 비교 결과

	CPC	CSC	CDC	CCC	CV	CR	CRL_{LOCs}
Class	O			O			O
Interface	O			O			O
Class Method	O		O	O			O
Interface Method	O		O	O	O	O	O
Attributes	O			O			O
Parameters	O		O	O			O
Relationship		O					
Messages			O				O
Cyclomatic complexity				O			
Customization Methods				O	O		O
Common Method				O	O	O	O
Lines of Code				O			O

(표 5) 컴포넌트 기반 메트릭 영향도

메트릭	Objective	C.T.E	Und	Main	C.D.E	CBSD.E	Cust
CPC	↓	↓	↑	↑	↓	↓	
CSC	↓	↓	↑	↑	↓		
CDC	↓	↓	↑	↑	↓		
CCC	↓	↓	↑	↑	↓		
CV	↑	↑	↓	↑	↑	↓	↑
CRL_{Func}	↑		↑	↑		↓	↑
CRL_{LOCs}	↑		↑	↑		↓	↑

* C.T.E : 컴포넌트 테스팅 영향도, Und : 이해성, Main : 유지보수성, C.D.E : 컴포넌트 개발 영향도, CBSD.E : CBSD 영향도, Cust : 특화성

5. 결 론

우리는 이 논문에서 은행 업무 도메인에서 컴포넌트 개발 과정 중에 만들어진 컴포넌트의 복잡도, 특화성 및 재사용성을 측정했다. 이러한 목적을 위한 CPC, CSC, CDC, CCC, CV, CR과 CRL 등과 같은 여러 가지 측정 메트릭들이 있다. 특히 우리는 컴포넌트에 기초한 은행업무 시스템에서 개발된 컴포넌트의 재사용 수준을 측정하기 위해 CRL을 적용했다.

우리는 컴포넌트의 복잡도가 컴포넌트의 크기를 추정하는데 도움이 될 수 있다는 것을 발견하였으며, 컴포넌트들의 재사용성과 특화성이 컴포넌트에 기반한 소프트웨어 개발 과정에서 컴포넌트의 재사용성에 영향을 미친다는 것 또한 발견했다.

마지막으로 우리는 컴포넌트들의 코드 라인들이 CBSD에서 재사용성의 측정을 위해 적절하다는 것을 알아냈지만, 각 컴포넌트에 대한 기술적인 복잡성은 고려하지 않았다. 컴포넌트의 복잡도와 재사용성은 기능 포인트(Function Points)들을 사용함으로써 산정 될 것이다. 전통적인 기능 포인터는 컴포넌트에 기반한 소프트웨어 개발에는 적절치 않을 것이다. 향후 우리는 컴포넌트 지향 기능 포인터와 복잡도 메트릭(Complexity Metrics)을 연구해야 할 것이다.

참 고 문 헌

[1] Szyperski C., Component Software: Beyound Object-Oriented Programming, Addison Wesley Longman, Reading, Mass., 1998.

[2] Linda H. Rosenberg, "Applying and Interpreting Object-Oriented Metrics", at URL:http://satc.gsfc.masa.gov/support/STC_APR98/apply_oo/apply_oo.html.

[3] Sun Microsystems Inc., Enterprise JavaBeans Specifications, at URL: http://www.javasoft.com.

[4] Rational Software Corp., Unified Modeling Language(UML) Summary, 1997.

[5] Object Management Group, CORBA Components, at UR: http://www.omg.org, March 1999.

[6] Norman E. Fenton and Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.

[7] Chidamber, Shyam and Kemerer, Chris, "Ametrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, June, 1994, pp. 476~492.

[8] Lorenz, Mara and Kidd, Jeff, Object Oriented Tool User's Instructions, 1994.

[9] McCbe & Associates, McCabe Object Oriented Tool User's Instructions, 1994.

[10] Rosenberg, Jan, "Metrics for Object Oriented Environments", EFAITP/AIE Third Annual Software Metrics conference, December, 1997.

[11] Hudli, R., Hoskins, C., Hudli, A., "Software Metrics for Object Oriented Designs", IEEE, 1994.

◐ 저 자 소 개 ◑

이 숙 희

1979년 숙명여자대학교 독문학과 졸업(학사)
1982년 동국대학교 경영대학원 정보처리학과 졸업(석사)
1991년 성균관대학교 대학원 통계학과 졸업(박사)
1987년~1993년 동신대학교 전자계산학과 교수
1993년~현재 : 서경대학교 인터넷정보학과 교수
관심분야 : 소프트웨어 공학, 소프트웨어 테스트, 컴포넌트기반 소프트웨어 개발기법
E-mail : oleesh@skuniv.ac.kr

조 은 숙

1993년 동의대학교 전산통계학과 졸업(이학사)
1996년 숭실대학교 대학원 컴퓨터학과 졸업(공학석사)
2000년 숭실대학교 대학원 컴퓨터학과 졸업(공학박사)
2000년 9월~현재 : 동덕여자대학교 정보학부 데이터정보학과 강의전임교수
관심분야 : 컴포넌트기반 소프트웨어 개발 기법, 객체지향 모델링, 객체지향 메트릭, 컴포넌트 정형 명세
E-mail : escho@dongduk.ac.kr