소프트웨어 컴포넌트 재사용성 측정 메트릭
(Software Component Reusability Metrics)

박 인 근 † 김 수 동 ††
(In Geun Park) (Soo Dong Kim)

요 약 소프트웨어의 개발 생산성 향상과 고품질의 소프트웨어 생산을 위해서 현재 컴포넌트 기반 개발(CBD)이 보편화되고 있다. 이러한 CBD는 소프트웨어의 재사용성을 높여 주며 개발기간 단축과 개발 비용의 절감을 가져오고 있다. 이러한 장점 때문에 산업계에서는 많은 부분을 컴포넌트로 만들려고 노력하고 있다. 그러나, 만들어진 소프트웨어 컴포넌트에 대해서 얼마나 품질이 좋은지, 또한 얼마나 재사용성이 있는가에 대한 검증은 아직 미흡한 상태이다. 본 논문에서는 만들어진 소프트웨어 컴포넌트에 대한 품질 중 재사용성을 측정하는데 필요한 측정 방법을 제공한다. 재사용성의 측정 방법은 크게 간접적인 측정 기준과 직접적인 측정 기준으로 구분한다. 직접적인 측정 기준은 컴포넌트를 구성하는 클래스들과 컴포넌트의 인터페이스들을 조사해서 얻을 수 있는 방법을 말하는 것으로 컴포넌트의 크기, 복잡도, 결합도, 응집도 등을 측정한다. 간접적인 측정 기준은 이러한 직접적인 측정기준을 가지고 측정이 되는 기준으로서 이해도, 적용가능성, 수정가능성, 모듈화가능성이 있다. 이러한 간접적인 측정은 궁극적으로 재사용성의 측정에 사용이 된다. 이러한 직접적인 측정기준과 간접적인 측정을 이용해서 재사용성에 대한 측정을 해본 결과 소프트웨어 컴포넌트의 품질이 향상되면서 측정값이 재사용성에 도움을 주는 방향으로 변화하는 것을 확인 하였다.

키워드 : CBD, 소프트웨어 컴포넌트, 재사용성, 재사용성 측정 요소

Abstract Component Based Development(CBD) Methodology is widely used in software development lifecycle to improve software quality. The Component Based Development(CBD) results to improve software reusability and reduce development term and cost. For this reason, lots of Enterprises are trying to make their processes to components. But, there has been few quality assurance or reusability testing action to those components. Most software component users can not know how their components are reusable and what extent their components satisfy to their quality requirements. For this reason, this paper suggests that software components can be measured their reusability by metrics proposed by this paper. We propose that in measuring software component reusability, there are direct metrics and indirect metrics. The results made by direct metrics are suggested to measure indirect metrics, so results to obtain reusability metrics.

Key words : CBD, Software Component, Reusability, Reusability Metrics

1. 서 론

컴포넌트 기반 개발은 시스템의 전체를 각각의 독립적인 컴포넌트로 나누어서 개발한 다음 이것을 통합하여 어플리케이션을 구축하는 방법이다. 컴포넌트 기반 개발은 개발 기간과 비용의 단축을 가져오기 때문에 산업계에서는 이러한 개발방법을 도입하려고 노력하고 있다. 그러나, 개발된 소프트웨어 컴포넌트에 대한 품질 측정이나 재사용성 측정에 대한 연구는 많이 되어 있지 못한 실정이다. 따라서, 만들어진 컴포넌트에 대해서 재사용성이 얼마나 있는지 혹은 품질이 얼마나 좋은가에 대한 측정 기준은 거의 마련되어 있지 못한 실정이다. 본 논문에서는 그러한 문제점에 착안하여서 소프트웨어 컴포넌트의 재사용성 측정 방안에 관해서 측정 기준을 제시한다. 본 논문에서 제시하는 측정 기준은 크게 직접적인 측정 기준(Direct Metrics)과 간접적인 측정 기준(Indirect Metrics)으로 나뉘어 진다. 직접적인 측정 기준은 소프트웨어 컴포넌트의 내부 품질 메트릭인 크기, 복잡도, 결합도, 응집도 등을 조사하여서 측정하며, 간접

· 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음
† 비 회 원 : ㈜CJ Systems 소프트웨어 공학연구소 연구원
igpark@cj.net
†† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@ssu.ac.kr
논문접수 : 2003년 9월 18일
심사완료 : 2004년 3월 26일

적인 측정 기준은 이러한 직접적인 측정 기준을 이용하여서 소프트웨어 컴포넌트의 재사용성에 영향을 미치는 이해도, 적용가능성, 수정가능성, 모듈화가능성을 측정하는 것을 말한다. 이러한 직접적인 측정 기준과 간접적인 측정 기준을 이용하여서 상용되는 소프트웨어 컴포넌트의 재사용성에 대한 측정을 한 결과 컴포넌트의 버전이 높아지면서 품질이 향상되는 것과 같이 재사용성도 향상된 결과를 보여 주어 재사용성의 측정에 도움을 준다는 결론을 얻게 되었다.

2장에서는 현재까지 재사용성에 관한 연구에 대해서 언급하면서 이를 소프트웨어 컴포넌트에 적용할 경우 생기는 문제점에 대해서 논의 한다. 3장에서는 컴포넌트 재사용성 측정 기준에 대해서 언급을 하는데 이는 다시 간접적인 측정 기준과 직접적인 측정 기준으로 나뉘어 지고 이들에 대한 연관관계를 언급한 후 앞에서 제시한 직접적인 측정 기준을 평가한다. 4장에서는 3장에서 언급한 측정 기준을 이용해서 실제 소프트웨어 컴포넌트에 대한 측정을 한다. 여기에는 상용되는 컴포넌트가 쓰인다.

2. 관련 연구

기존에 연구 되어져 온 소프트웨어에 대한 재사용성의 측정 방법은 크게 두 가지로 나뉘어 질 수 있다[1]. 경험적인 측정 방법(empirical metrics)과 정성적인 측정 방법(qualitative metrics)이 있다. 경험적인 측정 방법은 소프트웨어에 대한 실험에 의해서 나온 객관적인 결과를 이용해서 측정하는 방법이고, 정성적인 측정 방법은 개인적인 기준에 의해서 측정하는 방법을 말한다.

2.1 경험적인 측정 방법

Prieto-Diaz와 Freeman[2]은 그들의 논문에서 소프트웨어 재사용성의 측정에 사용되는 다섯 가지 요소들을 나열하였다. 그들은 프로그램의 사이즈, 프로그램의 구조, 프로그램의 문서화, 프로그래밍 언어, 그리고 재사용 경험이 소프트웨어의 재사용성의 측정에 사용된다고 보았다.

Caldiera와 Basili[3]는 그들의 논문에서 소프트웨어의 재사용성은 소프트웨어의 정확성(Correctness), 가독성(Readability), 테스트가능성(Testability), 수정용이성(Ease of Modification), 그리고 성능(Performance)에 의하여 좌우가 된다고 보았다. 그러나, 이러한 요소들은 직접적인 측정이 어렵기 때문에 그들은 다음과 같은 방법으로 측정을 하였다. Halstead의 프로그램의 크기(Program Volume)를 이용하는 방법, McCabe의 Cyclomatic복잡도를 이용하는 방법, 정규도(Regularity), 그리고 재사용 빈도수에 의하여 소프트웨어의 재사용성을 측정 하였다.

2.2 정성적인 측정 방법

재사용성은 소프트웨어의 품질 측면에서 상당히 중요한 부분을 차지한다. 따라서, 소프트웨어의 품질을 향상시킬 수 있는 요소들이 역시 재사용성을 향상시킬 수 있는 요소가 될 수 있다[4].

표 1 재사용성이 높은 소프트웨어의 일반적인 특징

요소	설명
쉬운 이해성	소프트웨어는 이해하기가 쉬워야 하고, 주석이 잘되어 있어야 한다.
기능적인 완성도	요구사항에 맞는 기능이 잘 갖추어 있어야 하고, 미래의 기능도 포함하고 있어야 한다.
신뢰도	소프트웨어는 여러 가지 상황에서 주어진 기능을 에러 없이 정확하게 수행할 수 있어야 한다.
좋은 에러와 예외 상황처리	소프트웨어는 에러 상황이 발생 했을 경우 그 피해가 크지 않고, 정상적인 처리가 되어야 한다.
정보은닉	소프트웨어는 사용자로부터 정보를 숨길 수 있어야 한다.
이식성	소프트웨어는 특정한 하드웨어나 운영체제에 종속되어서는 안 된다.
높은 응집도와 낮은 결합도	소프트웨어는 외부의 영향을 최소화 하면서 기능을 수행 할 수 있어야 한다.

위의 표 1은 소프트웨어가 일반적으로 재사용성을 가지려면 갖추어야 할 요소들에 대해서 설명하고 있다[5].

ISO 9126[6]은 소프트웨어의 품질 측정에 관한 모델을 제시한다. ISO 9126은 소프트웨어의 품질을 6개의 특징(Characteristics)으로 나누어서 측정하며, 그 각각의 특징에는 하위 특징(Sub Characteristics)들이 존재한다. 이러한 하위 특징들은 내부의 측정기준(Internal Metrics)과 외부의 측정기준(External Metrics)으로 측정을 한다. ISO 9126은 소프트웨어의 일반적인 품질 측정에 관한 모델을 제시하는 반면, 본 논문에서 제시하고자 하는 소프트웨어 컴포넌트의 특징은 일반적인 소프트웨어의 특징과 차이가 나기 때문에 ISO 9126의 모델을 그대로 적용할 수가 없다.

Briand[12]는 소프트웨어의 측정 기준을 크게 크기(Size)에 관한 측정 기준, 길이(Length)에 관한 측정 기준, 복잡도(Complexity)에 관한 측정 기준, 응집도(Cohesion)에 관한 측정 기준, 결합도(Coupling)에 관한 측정 기준으로 분류를 하고, 각각의 측정 기준이 갖추어야 할 성질(Property)들을 정의 하였다. 그는 그의 논문에서 특히 소프트웨어 모듈(Module)들간의 응집도(Cohesion)과 결합도(Coupling)에 관한 측정 기준이 갖추어야 할 성질(Property)에 관해서 정의를 하였는데 응집도에 관한 측정 기준이 갖추어야 할 조건은 모두 4가지의 성질(Property)로 정의하였다.

3. 측정 메트릭

본 논문에서는 측정 모델을 간접적인 모델과 직접적인 측정 모델로 나누어서 제시한다. 간접적인 모델과 직접적인 모델의 연관관계는 간접적인 모델은 소프트웨어 컴포넌트의 재사용성을 측정하기 위한 개념적인 모델을 제시하는 반면에 직접적인 모델은 간접적인 모델 각각에 대해서 실제로 측정할 수 있는 방법을 제시한다. 소프트웨어 컴포넌트 재사용성은 4가지의 간접적인 측정 모델로 나뉘어 진다. 첫째는 이해성(Understandability)이다. 두 번째는 수정가능성(Customizability)이다. 셋째로 적용가능성(Adaptability)을 들 수 있다. 넷째로 모듈화 가능성(Modularity)을 들 수 있다. 이러한 간접적인 측정 모델에 대해서 6가지의 직접적인 측정 모델을 제시한다. 직접적인 측정 모델은 위에서 설명한 간접적인 측정 모델을 실제로 측정할 수 있는 모델과 연계하기 위해서 제시한다.

3.1 간접적인 측정 메트릭

3.1.1 이해성(Understandability)

소프트웨어 컴포넌트의 특징은 바이너리 코드로 배포가 된다. 두 번째 특징은 소프트웨어 컴포넌트는 블랙박스형태이고 반드시 인터페이스를 통하여서만 접근이 가능하다[7]. 따라서, 소프트웨어 컴포넌트를 사용하기 전에 사용하려는 소프트웨어 컴포넌트에 대한 이해가 전제 되어야 한다. 소프트웨어 컴포넌트의 인터페이스를 보고 얼마나 이해를 잘 할 수 있는 가는 그 소프트웨어 컴포넌트의 재사용성에 많은 영향을 끼친다.

3.1.2 수정가능성(Customizability)

컴포넌트 개발자는 소프트웨어 컴포넌트 개발 중에 공통성과 가변성에 대한 분석을 한다[9]. 이는 같은 업무 영역에서도 다른 부분들이 존재하기 때문이다. 또한, 소프트웨어 컴포넌트는 최초에 개발될 시점에서의 환경과 전혀 다른 환경에서도 사용이 될 수 있도록 만들어져야 한다. 이러한 가변성에 대한 지원은 소프트웨어 컴포넌트가 좀더 다양한 환경에서 사용 되도록 해준다. 수정가능성은 소프트웨어 컴포넌트가 이러한 가변성에 대한 지원을 얼마나 해주는가에 대한 측정이다. 이것은 또한 컴포넌트 사용자가 얼마나 쉽게 컴포넌트를 수정 할 수 있는가에 대한 측정이기도 한다.

3.1.3 적용가능성(Adaptability)

소프트웨어 컴포넌트는 그것이 만들어진 환경과 다른 환경에서도 사용되어지도록 만들어 져야 한다. 소프트웨어 컴포넌트의 특징 중에 "운영되어지기 전에 그 운영환경에 맞게 합쳐져서 실행되어야 한다"고 하였다[7]. 소프트웨어 컴포넌트가 여러 가지 다른 운영 환경에서 얼마나 알맞게 운영이 되는가는 소프트웨어 컴포넌트의 재사용성의 측면에서 매우 중요한 측정 기준이 된다.

3.1.4 모듈화 정도(Modularity)

소프트웨어 컴포넌트는 각각의 문제 단위로 나뉘어 질 수가 있으며, 응집도가 높고, 결합도가 낮다라고 하였다[7]. 이러한 특징은 소프트웨어 컴포넌트가 서로 모듈화 되어 있어야 함을 의미한다. 모듈화된 소프트웨어 컴포넌트는 적은 비용으로 사용을 할 수 있게 해주기 때문이다. 따라서, 소프트웨어 컴포넌트 간에 얼마나 모듈화가 되어있는가는 소프트웨어 컴포넌트의 재사용성에 중요한 측정 요소가 된다.

3.2 직접적인 측정 메트릭

직접적인 측정 메트릭은 소프트웨어 컴포넌트의 재사용성에 대한 측정을 컴포넌트 소스를 통해서 측정하는데 사용되는 메트릭을 말한다. 컴포넌트의 소스에서 측정할 수 있는 크기, 복잡도, 결합도, 응집도 등을 측정함으로써 컴포넌트의 재사용성을 예측하는 것이다. 소프트웨어 컴포넌트의 재사용성을 측정하기 앞서 측정에 사용되는 소프트웨어 컴포넌트의 구성원을 알아 볼 필요가 있다. 소프트웨어 컴포넌트는 자신이 제공해주는 서비스를 위하여 제공해주는 인터페이스(Provide Interface)가 있다[7]. 또한 자신이 다른 컴포넌트의 서비스를 사용하려고 하는 요구되어지는 인터페이스(Require Interface)가 있다[7].

정의 1. $\sum_{i=1}^{n} Sreq_i(C)$, $Sreq(C)$는 소프트웨어 컴포넌트 C의 요구되어지는 인터페이스(Require Interface)이며, $Sreq_i(C)$는 소프트웨어 컴포넌트 C의 요구되어지는 인터페이스(Require Interface)의 i번째 서비스

정의 2. $\sum_{j}^{m} Spro_j(C)$, $Spro(C)$는 소프트웨어 컴포넌트 C의 제공해주는 인터페이스(Provide Interface)이며, $Spro_j(C)$는 소프트웨어 컴포넌트 C의 제공해주는 인터페이스(Provide Interface)의 j번째 서비스

정의 3. $I(C) = Sreq(C) + Spro(C)$, $I(C)$는 소프트웨어 컴포넌트 C의 인터페이스.

정의 1은 소프트웨어 컴포넌트 C의 요구되어지는 인터페이스를 나타낸다. 요구되어지는 인터페이스는 내부에 존재하는 서비스들의 합으로 나타낸다. 정의 2는 소프트웨어 컴포넌트 C의 제공해주는 인터페이스를 의미한다. 이는 각각의 제공해주는 인터페이스의 서비스들의 합으로 나타낸다. 정의 3은 소프트웨어 컴포넌트 C의 인터페이스는 요구되어지는 인터페이스와 제공해주는 인터페이스의 합으로 나타내어 진다.

3.2.1 Component Complexity Number(CCN)

*CCN*는 컴포넌트의 복잡도를 컴포넌트 인터페이스에 있는 서비스들 각각의 흐름도를 측정하여 이들의 합으

로 정의한다. 소프트웨어 컴포넌트 인터페이스를 구성하는 요구되어지는 인터페이스와 제공해주는 인터페이스들의 합으로 구성한다.

정의 4.

$$CCN(C) = CCN(I(C)) = CCN(Sreq(C)) + CCN(Spro(C)) = CCN(\sum_{i=1}^{n} Sreq_i(C)) + CCN(\sum_{j=1}^{m} Spro_j(C))$$

. 여기서, C는 소프트웨어 컴포넌트를 나타내고, $Sreq(C)$는 C의 요구되어지는 인터페이스를 나타내며, $Spro(C)$는 C의 제공해주는 인터페이스를 나타낸다. 컴포넌트의 인터페이스의 수가 많아져서 복잡도가 높아지면 그 컴포넌트를 수정해서 사용하려는 사용자들이 이해하기도 어려울 뿐만 아니라 수정하기에도 시간이 많이 걸릴 것이다. 따라서, 컴포넌트의 복잡도가 높아지면 이해성이나 적용가능성이 떨어지게 되고 결국 컴포넌트의 재사용성에 악영향을 끼치게 된다. 컴포넌트의 개발자들은 내부의 복잡도를 높이지 않게 개발을 해야 재사용성이 좋은 컴포넌트를 만들 수 있을 것이다.

3.2.2 Inheritance In COmponents(HICO)

Kemerer가 제시하는 Number Of Children(NOC)는 클래스에서 상속에 의한 자식노드의 수를 의미한다[10]. 상속은 재사용의 한가지 방법으로 객체지향 설계에 많이 사용되었다. 컴포넌트를 구성하는 클래스가 상속을 많이 받게 되면 그 만큼 재사용성이 높다는 의미로 볼 수 있다. 그러나, Kemerer는 그의 논문에서 상속의 깊이가 너무 깊게 되면 이해하기가 오히려 어려워지고, 복잡도도 증가한다고 하였다[10]. 본 논문에서 제시하는 $HICO$는 소프트웨어 컴포넌트에서 내부의 클래스들이 얼마나 상속의 구조를 갖추고 있는가를 측정한다.

정의 5. $HICO(C) = \frac{\sum P_i(C)}{\sum P(C)}$, $P_i(C)$는 컴포넌트 C에서 상속의 구조를 갖춘 클래스의 개수.

위의 정의에서 $P_i(C)$는 컴포넌트C에서 상속의 구조를 갖춘 클래스를 의미하고, $P(C)$는 컴포넌트 C에 속하는 모든 클래스를 의미한다. $HICO(C)$는 하나의 컴포넌트 안에 존재하는 상속의 구조를 갖춘 클래스의 개수와 전체 클래스의 개수의 비율을 의미한다. 컴포넌트의 크기에 상관없이 상속의 구조를 갖춘 클래스가 전체 컴포넌트를 구성하는 클래스에서 얼마나 차지하고 있는가에 대한 비율을 나타낸다. 컴포넌트내에서 상속의 구조를 갖춘 클래스가 많으면 그 만큼 컴포넌트가 재사용을 많이 해서 만들어 졌다는 의미이다. 따라서, 컴포넌트의 재사용성은 높다고 볼 수 있다.

3.2.3 Coupling Between COmponents(CBCO)

Coupling Between COmponents($CBCO$)는 소프트웨어 컴포넌트 사이에 있는 결합도를 측정하기 위한 것이다. $CBCO$는 소프트웨어 컴포넌트 내부의 클래스들 중에서 다른 컴포넌트의 인터페이스를 참조하는 수의 비율로 정의 된다.

정의 6. $CBCO(C) = 1 - \frac{\sum I_r(C)}{\sum I(C)}$, $I_r(C)$는 소프트웨어 컴포넌트 C에서 외부 컴포넌트 인터페이스를 참조하는 인터페이스임.

위의 정의는 1에서 컴포넌트 C에서 외부 컴포넌트의 인터페이스를 참조하는 인터페이스의 수를 컴포넌트의 인터페이스의 수로 나눈 값을 뺀 것을 의미한다. 이는 컴포넌트 내부에 인터페이스가 얼마나 많이 외부의 컴포넌트를 참조하는 가에 대한 정도를 정규화(Normalization)하여 나타내었다. 따라서, 정규화가 되었기 때문에 컴포넌트의 인터페이스의 수가 많더라도 외부의 인터페이스를 참조하는 수에 대한 비율은 비교가 가능하다. 컴포넌트가 외부의 컴포넌트의 인터페이스를 참조한다는 것은 결국 다른 컴포넌트가 존재해야 기존의 컴포넌트가 자신의 기능을 수행 할 수 있다는 것이다. 이러한 외부 컴포넌트에 대한 인터페이스를 자주 참조하게 되면 컴포넌트에 대한 독립성이 결여가 될 수 있다[11]. 결국 컴포넌트의 결합도가 높아지게 되는데 이렇게 되면 컴포넌트의 모듈화 정도나 수정가능성에 영향을 미치게 된다.

3.2.4 Cohesion in Components(COHC)

Cohesion in Components($COHC$)는 소프트웨어 컴포넌트 내의 응집도를 측정한다. 즉, 한 소프트웨어 컴포넌트 인터페이스에 속하는 여러 가지 서비스들 각각의 응집도를 측정한다.

정의 7. 소프트웨어 컴포넌트 C의 인터페이스를 I라고 하면, $I(C) = \{S_1, S_2, S_3, \ldots, S_n\}$이며, 여기서 $S_1, S_2, \cdots S_n$은 인터페이스 I의 서비스들이다. Cls_i를 인터페이스 I의 서비스 S_i가 사용하는 컴포넌트 C내의 클래스들의 집합이라고 가정하면, 위의 인터페이스 I에는 n개의 집합이 있다. 즉, $C_{ls1}, C_{ls2}, C_{ls3}, \ldots, C_{lsn1}$. 이때, $P = \{(C_{lsi}, C_{lsj}) \mid C_{lsi} \cap C_{lsj} = \phi, i \neq j\}$라고 가정하고, $Q = \{(C_{lsi}, C_{lsj}) \mid C_{lsi} \cap C_{lsj} \neq \phi, i \neq j\}$라고 가정하면,

$COHC(C) = 1 - \frac{|P|}{|P \cup Q|}$, $|P|$는 컴포넌트 C의 인터페이스 I가 지니고 있는 서비스들 중 서로 사용하는 클래스가 관련이 없는 것들의 숫자이고, $|P \cup Q|$는 컴포넌트 C의 인터페이스 I가 지니고 있는 서비스들이 사용하는 클래스들의 숫자를 나타냄.

위의 정의에서 알 수 있듯이 $COHC$는 컴포넌트내에 존재하는 인터페이스들 사이의 응집도를 나타낸다. 컴포넌트 내의 인터페이스들 사이에서 같은 클래스를 사용한다는 것은 컴포넌트 내부의 인터페이스가 서로 응집도가 높다는 의미이다. 응집도가 높은 컴포넌트는 서로 분리되기가 어렵고 컴포넌트의 설계와 유지보수가 쉽다고 하

였다[11]. 또한 응집도가 높은 컴포넌트는 컴포넌트의 이해성과 수정가능성에 좋은 영향을 끼친다. 따라서, 응집도는 컴포넌트의 재사용성에 많은 영향을 끼친다.

3.2.5 Number of Patterns in Components(NOPC)

Number of Patterns in Components(NOPC)는 소프트웨어 컴포넌트 내에 존재하는 디자인 패턴의 수를 의미한다. 소프트웨어 컴포넌트 내부 구조가 패턴화 될수록 재사용성이 높아지기 때문이다.

정의 8. NOPC(C)= 소프트웨어 컴포넌트 *C*내에 존재하는 디자인 패턴의 수.

CBD는 결국 재사용을 위해서 만들어진 기술이다. 이러한 재사용을 위해서 사람들은 설계할 때에 컴포넌트 기반 개발 기술과 함께 디자인 패턴을 이용한다. 컴포넌트의 설계에 정형화된 패턴을 많이 사용했다는 것은 그만큼 신뢰할 수 있고, 이해하기에도 쉽다는 의미이다. 또한, 수정 시에 패턴에 대한 지식을 알고 있으면 수정을 쉽게 할 수 있어서 컴포넌트의 재사용성에 도움을 줄 수 있다.

3.2.6 Granularity in COmponents(GICO)

Granularity in COmponents(GICO)는 소프트웨어 컴포넌트의 크기를 측정한다.

정의 9. *GICO(C)* = 소프트웨어 컴포넌트 *C*내에 존재하는 Class의 수.

위의 정의는 소프트웨어 컴포넌트의 크기를 내부에 존재하는 클래스의 수로 정하였다. 설계자들은 설계 시 컴포넌트의 범위를 정하게 되는데, 너무 크게 범위를 잡게 되면 컴포넌트내에 존재하는 클래스의 수가 늘어나게 된다. 클래스의 수가 늘어나게 되면 유지보수하기도 어렵고, 컴포넌트를 이해하기에도 어렵게 되어서 궁극적으로 컴포넌트의 재사용성에 나쁜 영향을 미치게 된다. 따라서, GICO를 측정하는 것으로 컴포넌트의 재사용성을 예측할 수 있다.

3.3 직접적인 측정 메트릭과 간접적인 측정 메트릭과의 연관성

위에서 제시하는 6가지의 직접적인 측정 메트릭에서 간접적인 측정 메트릭과의 연관성을 고려해 보자. *CCN*는 컴포넌트의 복잡도를 나타내는데 복잡도가 높을수록 이해성, 그리고 적용가능성은 더 낮아진다. HICO는 컴포넌트내의 상속의 정도를 측정하는데, 상속이 많이 되면, 그 만큼 이해하기에도 좋고, 수정 할 때에도 적은 비용으로 수정을 할 수 있다. 그러나, 위에서도 언급되었듯이 너무 많은 상속은 오히려 복잡도를 높이고, 이해하기에도 어려운 컴포넌트로 만들 가능성이 있다[10]. 따라서, 적당한 상속은 컴포넌트의 이해성과 수정가능성에 도움을 준다. CBCO는 컴포넌트간의 결합도를 측정하는데, 결합도가 높아지면 서로간에 모듈화 정도가 작아져서 서로 다른 컴포넌트를 의지하게 된다. 또한, 결합도가 높아지게 되면 그 컴포넌트를 다른 어플리케이션에 적용할 때에도 결합관계에 있는 컴포넌트를 고려해야 하기 때문에 적용가능성도 떨어지게 된다. *NOPC*는 패턴(Pattern)의 수가 많이 존재하면 할수록 그 만큼 컴포넌트의 내부구조가 정형화되었다는 것을 의미한다. 따라서, 컴포넌트에 대한 이해성과 수정할 때에 수정가능성은 더 커진다. 아래의 표 2는 간접적인 측정 메트릭과 직접적인 측정 메트릭 간의 관계를 나타낸 것이다. 여기에서 (-)기호는 직접적인 메트릭 값이 증가할수록 간접적인 메트릭 값이 떨어지는 것을 의미한다. 반면, (+)값은 직접적인 메트릭 값이 증가할수록 간접적인 메트릭 값이 증가 함을 의미한다.

표 2 직접적인 메트릭과 간접적인 메트릭 간의 관계

직접 / 간접	CCN	HICO	CBCO	COHC	NOPC	GICO
이해성	-	+			+	-
적용가능성	-		+	+		
수정가능성		+			+	
모듈화가능성			+	+		

위의 표 2에서 보듯이 간접적인 메트릭에 영향을 주는 직접적인 메트릭은 여러 가지이다. 따라서, 간접적인 메트릭에 대한 값을 구하기 위해서는 직접적인 메트릭을 이용하여서 측정한 값을 가감하여서 구해야 한다.

정의 10. $U(C) = \alpha CCN(C) + \beta HICO(C) + \chi NOPC(C) + \delta GICO(C)$, $|\alpha| + |\beta| + |\chi| + |\delta| = 1$여기서, U는 소프트웨어 컴포넌트 *C*의 이해성을 의미하며, 이는 CCN, HICO, NOPC, 그리고 GICO값의 합으로 나타난다. 이때, 각 값의 계수($\alpha, \beta, \chi, \delta$)는 평가하려는 컴포넌트의 종류에 따라서 다르게 줄 수 있다.

정의 11. $A(C) = \alpha CCN(C) + \beta CBCO(C) + \chi COHC(C)$, $|\alpha| + |\beta| + |\chi| = 1$. 여기서, *A*는 소프트웨어 컴포넌트 *C*의 적용가능성을 의미하며, 이는 CCN, CBCO, 그리고 COHC값의 합으로 나타난다. CCN, CBCO, COHC 모두 값이 증가하게 되면 적용가능성에 좋은 영향을 끼치기 때문에 계수를 모두 양수로 두었다. 이때, 각 값의 계수(α, β, χ)는 평가하려는 컴포넌트의 종류에 따라서 다르게 줄 수 있다.

정의 12. $C(C) = \alpha HICO(C) + \beta NOPC(C)$, $|\alpha| + |\beta| = 1$. 여기서, *C*는 소프트웨어 컴포넌트 *C*의 수정가능성을 의미하며, 이는 HICO, NOPC값의 합으로 나타난다. 상속의 구조를 갖춘 컴포넌트나 디자인 패턴이 많이 적용된 컴포넌트 일수록 컴포넌트의 수정가능성이 좋아짐을 의미한다.

정의 13. $M(C) = \alpha CBCO(C) + \beta COHC(C)$, $|\alpha| + |\beta|$. 여기서, M는 소프트웨어 컴포넌트 *C*의 모듈화 가능성을 의미하며, 이는 CBCO, COHC의 합으로 나타난다. 컴포넌트간의 응집도와 결합도가 컴포넌트의 모듈화 가능성에 중요한 역할을 함을 의미한다. 이때, 각 값의 계수(α,β)는 평가하려는 컴포넌트의 종류에 따라서 다르게 줄 수 있다.

3.4 메트릭 평가

위에서 6가지 직접적인 측정 메트릭과 4가지 간접적인 측정 메트릭에 대해서 알아 보았다. 본 절에서는 직접적인 측정 메트릭에 대해서 평가를 한다. 간접적인 측정 메트릭은 정량적인 수치를 측정 할 수 없는 메트릭이므로 본 논문에서는 직접적인 측정 메트릭에 대한 평가만을 한다. Briand가 제안한 메트릭의 종류에 따라서 본 논문에서 제안한 직접적인 메트릭를 분류하면 다음과 같다.

표 3 직접적인 측정 메트릭의 분류

직접 \ 종류	CCN	HICO	CBCO	COHC	NOPC	GICO
크기					O	O
길이						
복잡도	O					
결합도			O			
응집도				O		

표 3은 직접적인 측정 메트릭을 Briand[12]의 분류에 의해서 분류한 것이다. CCN은 컴포넌트의 복잡도를 측정하고, CBCO는 결합도, COHC는 응집도, 그리고, NOPC와 GICO는 크기 메트릭에 해당한다. 그러나, HICO는 Briand가 분류한 어떠한 부류에도 포함이 되지 않았다. 따라서, HICO는 Briand가 제안한 메트릭이 갖추어야 할 필요충분조건으로 평가하지 않고, Kemerer[10]가 제안한 객체 지향 메트릭이 갖추어야 할 조건을 가지고 평가해 본다. CCN은 복잡도를 나타내기 때문에 Weyuker[8]가 제안한 복잡도 메트릭이 갖추어야 할 조건을 가지고 적용해 본다. 나머지 메트릭들은 Briand[12]가 제안한 조건들을 적용해서 평가해 본다.

3.4.1 Component Complexity Number(CCN)

CCN은 컴포넌트 내의 복잡도에 대한 메트릭이다. 본 논문에서는 Weyuker[8]가 제안한 소프트웨어의 복잡도 메트릭 측정 기준에 대한 평가 방법 중 컴포넌트에 해당하는 부분만 선택을 해서 평가하도록 한다. Weyuker가 제안한 방법 중 성질 7은 객체지향 설계나 컴포넌트에는 맞지 않기 때문에 제외한다.

성질 1: Coarse. 성질 1이 의미하는 바는 서로 다른 복잡도 값을 가지는 컴포넌트가 존재한다는 것이다. 컴포넌트 P와 Q가 있다고 가정하자. *CCN(P)*의 값과 *CCN(Q)*의 값은 달라질 수 있다. P와 Q의 내부의 복잡도가 달라지면 *CCN*의 측정값은 달라질 수 있기 때문이다. 따라서, CCN은 성질 1을 만족한다.

성질 2: Nonnegative. 성질 2가 의미하는 바는 복잡도 값이 음수를 가져서는 안 된다는 내용이다. 컴포넌트의 CCN값은 복잡도를 의미한다고 하였다. 복잡도가 음수가 생기는 경우는 생겨나지 않는다. 아무리 작아도 최악의 경우에 0이 되는 경우는 있어도 음수가 되지는 않는다. 따라서, CCN은 성질 2를 만족한다.

성질 3: Nonuniqueness. 성질 3이 의미하는 바는 서로 다른 컴포넌트도 같은 복잡도 값을 가질 수 있다는 내용이다. 컴포넌트 P와 Q가 다른 기능을 하는 컴포넌트이지만 내부의 복잡도는 동일 할 수 있다. *CCN(P)*와 *CCN(Q)*가 서로 같은 측정값을 가질 수가 있다는 것이다. 따라서, CCN은 성질 3을 만족한다.

성질 4: Implementation. 성질 4가 의미하는 바는 서로 같은 기능을 수행 하지만 복잡도 값은 달라질 수 있음을 의미한다. 같은 기능을 수행하지만 내부의 구현방식이 복잡도 값의 차이를 결정함을 의미한다. P와 Q가 같은 기능을 하는 컴포넌트라고 가정 하자. P컴포넌트는 Java로 만들어 졌고, Q컴포넌트는 C++로 구현이 되었다면 P와 Q의 *CCN*은 서로 다를 수가 있다. 따라서, CCN은 성질 4를 만족한다.

성질 5: Monotonicity. 성질 5가 의미하는 바는 두 개의 컴포넌트를 합하게 되면 합하기 전의 복잡도보다 증가하게 됨을 의미한다. P와 Q컴포넌트가 있다고 가정하자. P와 Q컴포넌트가 서로 합쳐지게 되면 *CCN*도 증가하게 된다. 기존에 지니고 있던 복잡도가 합쳐지게 되기 때문에 복잡도는 증가하게된다. 따라서 기존의 P와 Q의 *CCN*보다 높아 진다. 따라서, CCN은 성질 5를 만족한다.

성질 6: Nonequivalence of Interaction. 성질 6이 의미하는 바는 컴포넌트 사이의 상호작용이 항상 같은 형태로 일어나지 않음을 의미한다. 컴포넌트 P,Q,R이 있고, 컴포넌트 P와 Q의 *CCN(P)*값이 *CCN(Q)*값과 동일하다고 가정하자. 컴포넌트 P,R를 합친 P+R의 *CCN(P+R)*의 값은 Q,R을 합친 *CCN(Q+R)*의 값과 항상 같지는 않다. 왜냐하면, 컴포넌트 P와 R를 합칠 때에 생겨나는 상호작용이 항상 컴포넌트 Q와 R를 합칠 때에 일어나는 상호작용과 같을 수는 없기 때문이다. 즉, *CCN(P+R)*의 값과 *CCN(Q+R)*의 값은 항상 같지 않다. 따라서, CCN은 성질 6를 만족한다.

성질 7: Renaming. 성질 7이 의미하는 바는 이름만 바꾼 컴포넌트라고 하면 그 복잡도 값은 같다는 내용이다. 컴포넌트 P의 이름만 Q로 바꾸어 놓은 두 개의 컴

포넌트가 있다고 가정하면, 그 두 개의 컴포넌트의 CCN값은 동일할 것이다. 내부구조 및 구현 방식이 모두 같기 때문에 CCN값도 같다. 따라서, CCN은 성질 7을 만족한다.

성질 8: Interaction Increases Complexity. 성질 8이 의미하는 바는 컴포넌트를 합치게 되면 합치기 전보다 복잡도가 증가하게 된다는 것이다. 만일, P와 Q컴포넌트를 합치게 된다면, P의 인터페이스와 Q의 인터페이스를 연결해주는 부분이 생겨나야 할 것이다. Katharine Whitehead는 컴포넌트를 합치는 과정에서 발생하는 상호작용 때문에 어뎁터(Adapter) 컴포넌트를 두어서 컴포넌트간의 독립성을 유지시키면서 합쳐야 한다고 하였다[7].

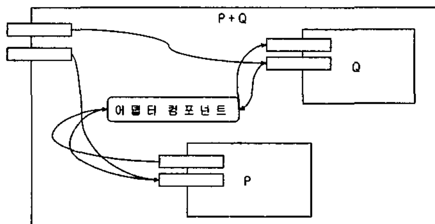


그림 1. P와 Q컴포넌트를 합할 경우 생기는 상호작용

그림 1은 P와 Q를 합할 경우 생기는 상호작용에 대해서 설명하고 있다. 위의 그림에서 알 수 있듯이 P와 Q를 합하게 되면 그 만큼 복잡도 값이 늘어남을 알 수 있다. 이러한 경우에 합친 후의 복잡도 값은 합치기 전의 복잡도 값보다 증가 한다. 따라서, CCN은 성질 8을 만족한다.

3.4.2 Inheritance In COmponents(HICO)

HICO에 대한 평가는 Kemerer가 제안한 메트릭이 갖추어야 할 요건에 비교해서 평가해 본다. Kemerer는 그의 논문에서 객체 지향 설계를 측정하기 위한 메트릭에는 기본적으로 갖추어야 할 요건을 6가지의 성질로 요약을 하였다[10]. 그가 제안한 성질 6가지 중 6번째의 성질은 복잡도 메트릭에 대한 것이어서 여기서는 제외한다.

성질 1: Noncoarseness. 성질1이 의미하는 바는 내부구조와 기능이 동일한 컴포넌트들의 측정값은 서로 다른 값을 가질 수 있어야 한다. 컴포넌트 P의 HICO와 컴포넌트 Q의 HICO값은 달라 질 수 있다. 내부 구조의 차이로 인해서 다를 수도 있고, 구현 방법의 차이로 다를 수도 있다. 따라서, HICO는 성질 1을 만족한다.

성질 2: Nonuniqueness. 성질 2가 의미하는 바는 서로 다른 컴포넌트이지만 측정값은 서로 같아 질 수 있음을 의미한다. 컴포넌트 P와 Q가 서로 다른 컴포넌트라고 할 때에 HICO의 값은 서로 같아 질 수 있다. 왜냐하면, 서로 다른 컴포넌트이지만 상속의 구조를 갖춘 클래스의 수는 같을 수가 있기 때문이다. 따라서, HICO는 성질 2를 만족한다.

성질 3: Design Details are important. 성질 3이 의미하는 바는 서로 같은 기능을 하는 컴포넌트이지만 그 측정값은 달라질 수 있음을 의미한다. 컴포넌트 P와 Q가 서로 같은 기능을 하는 컴포넌트라고 가정해 보자. P는 C언어로 만들어졌고, Q는 Java언어로 만들어 졌다고 하면, HICO(P)의 값은 0이다. 왜냐하면, C언어는 상속을 지원하지 않기 때문이다. 그러나, HICO(Q)의 값은 0일 수도 아닐 수도 있다. 따라서, 하는 기능은 동일하지만 내부의 구현 방식이 HICO값을 결정한다. 따라서, HICO는 성질 3을 만족한다.

성질 4: Monotonicity. 성질 4가 의미하는 바는 P와 Q컴포넌트를 통합한 컴포넌트 P+Q의 측정값은 P와 Q의 측정값보다 최소한 같거나 커진다는 의미이다. Kemerer는 P와 Q의 측정값은 두 개를 합하게 되면 늘어난다고 하였다. P와 Q의 컴포넌트의 HICO가 각각 3/4, 1/10이라고 가정하자. 두 개의 컴포넌트가 서로 관련되는 부분이 없다고 가정하면, P+Q의 HICO는4/14가 될 것이다. 이것은 P의 HICO값보다 작다. 따라서, 항상 성질 4가 만족하지는 않는다. 따라서, HICO는 성질 4를 만족하지 않는다.

성질 5: Nonequivalence of Interaction. 성질 5가 의미하는 바는 P와 Q, R컴포넌트가 있고, P와 Q의 측정값이 동일하다면, 항상 P+R의 값과 Q+R의 측정값이 동일하지 않음을 의미한다. 즉, P와 Q가 같다고 P+R과 Q+R의 측정값도 같지는 않음을 의미한다. P와 Q의 HICO값이 동일하다고 가정하자. 그림 4와 같이 P와 Q를 합할 때에 생기는 상호작용으로 어뎁터 컴포넌트가 생겨날 수 있다[7]. 마찬가지의 이유로 P와 R을 통합하거나 Q와 R을 통합할 때에도 내부에 생기는 상호작용 때문에 어뎁터 컴포넌트가 생성이 될 수 있다. 그러나, P와 R을 통합할 때에 생성되는 어뎁터 컴포넌트와 Q와 R을 통합할 때에 생성되는 어뎁터 컴포넌트는 항상 같지는 않다. 이러한 경우 HICO(P+R)의 값과 HICO(Q+R)의 값은 달라질 수 있다. 따라서, HICO는 성질 5를 만족한다.

3.4.3 Coupling Between COmponents(CBCO)

CBCO는 컴포넌트들 사이의 결합도를 측정한다. 본 논문에서는 Briand[12]가 제안한 결합도 측정기준이 갖추어야 할 조건에 대해서 CBCO에 적용해서 평가해 본다.

성질 1: Nonnegativity. 성질 1이 의미하는 바는 컴포넌트 사이의 결합도는 음수가 나올 수 없다는 의미이다. CBCO는 컴포넌트 사이의 참조되는 횟수를 측정한

다. 만약, 참조하는 외부의 인터페이스가 없다면 0이 될 것이다. 즉, 아무리 작아도 음수는 되지 않는다는 것이다. 따라서, CBCO는 성질 1을 만족한다.

성질 2: Null Value. 성질 2가 의미하는 바는 외부에 참조하는 인터페이스가 없다면 결합도는 null값을 의미한다. CBCO는 컴포넌트내에서 외부의 컴포넌트 인터페이스를 참조하는 수의 비율을 의미한다. 위에 정의된 CBCO의 정의에 의해서 만일 외부의 인터페이스를 참조하는 내부의 인터페이스가 전혀 없다면 CBCO의 값은 1일 것이다. 따라서, null값을 의미하는 값이 CBCO에 존재하기 때문에 CBCO는 성질 2를 만족한다.

성질 3: Monotonicity. 성질 3이 의미하는 바는 컴포넌트간의 관계를 높이는 것은 결합도를 줄이지 않음을 의미한다. 즉, 외부의 컴포넌트에 대한 참조가 늘어나게 되면 컴포넌트의 결합도는 증가하게 된다는 의미이다.

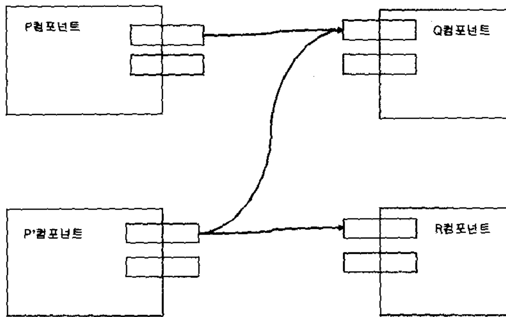


그림 2 P컴포넌트에 인터페이스가 증가하는 경우(P')

위의 그림과 같이 컴포넌트 P와 Q가 있고 P컴포넌트의 인터페이스의 수는 3개라고 가정하자. 그림에서 보는 바와 같이 3개의 인터페이스 중에서 1개가 외부의 Q컴포넌트를 참조한다고 가정하자. 이 경우 CBCO(P)의 값은 1-1/3=2/3이다. P컴포넌트에 외부의 Q컴포넌트를 참조하는 인터페이스 이외에 R컴포넌트를 참조하는 인터페이스가 한 개 추가 되었다고 가정해보자. 그림의 P'컴포넌트를 보면 기존의 P컴포넌트에 외부의 R컴포넌트를 참조하는 인터페이스가 추가되었다. 이러한 경우 CBCO(P')의 값은 1-2/4=2/4이다. 따라서, 기존의 CBCO(P)의 값(2/3)보다 값이 감소하였다. 즉, CBCO의 값이 감소하였다는 의미는 컴포넌트의 결합도가 증가 하였음을 의미한다. CBCO의 값은 비율로 나타내었기 때문에 컴포넌트 간에 외부 인터페이스를 참조하는 수가 늘어나게 되면 그 비율이 줄어들기 때문이다. 즉, CBCO의 값이 줄어드는 것은 그 만큼 결합도가 늘었다는 것을 의미한다. 따라서, CBCO는 성질 3을 만족한다.

성질 4: Merging of Modules. 성질 4가 의미하는 바는 두 컴포넌트를 합해서 측정한 결합도는 합하기 전에 각각의 컴포넌트에서 측정한 결합도의 합보다 더 커질 수 없음을 의미한다. 이것은 두 개의 컴포넌트 사이에 공통적인 부분이 있게 되면, 합하면서 한 개의 기능으로 통합되기 때문에 그 측정값은 아무리 커 보아야 합하기 전의 두 개의 결합도보다 더 커질 수 없음을 의미한다.

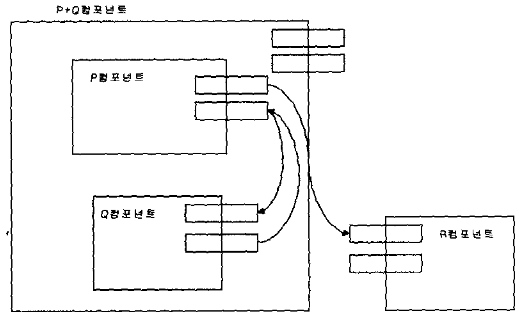


그림 3 서로 관련 있는 컴포넌트끼리 합할 경우 결합도의 변화

위의 그림과 같이 컴포넌트 P와 Q,R이 있고, 각각의 컴포넌트 내에는 3개의 인터페이스가 존재한다고 가정해보자. P컴포넌트와 Q컴포넌트는 서로 간에 참조를 하고 있으며, P컴포넌트는 R컴포넌트를 참조하고 있다. CBCO(P)의 값은 1-2/3=1/3이다. CBCO(Q)의 값은 1-1/3=2/3이다. 만일 P와 Q를 합한 컴포넌트를 P+Q컴포넌트라고 가정하면, CBCO(P+Q)는 1-1/6=5/6이다. 즉, CBCO(P)와 CBCO(Q)의 값의 합은 CBCO(P+Q)의 값보다 더 커지게 된다. 그러나, 만일 P컴포넌트의 인터페이스3개가 전부 Q컴포넌트를 참조하고 있고, Q컴포넌트의 인터페이스 3개가 모두 P컴포넌트를 참조하고 있다고 한다면 CBCO(P)의 값은 1-3/3=0이 될 것이고, CBCO(Q)의 값은 1-3/3=0이 될 것이다. 따라서, P와 Q의 CBCO값의 합은 0이 될 것이다. P와 Q컴포넌트를 합한 P+Q컴포넌트의 CBCO(P+Q)의 값은 1-0/6=1이 될 것이다. P와 Q를 합한 컴포넌트에서는 외부에 참조하는 인터페이스가 존재하지 않기 때문이다. 이런 특수한 경우에는 합하기 전의 컴포넌트의 CBCO값보다 합한 후의 컴포넌트의 CBCO값이 오히려 증가하게 된다. 따라서, CBCO의 정의는 성질 4를 만족하지 못한다. CBCO의 정의가 성질 4를 만족하지 못하는 원인은 CBCO는 비율을 나타내는데 비해서 Briand가 제안한 성질 4는 결합도의 단순한 개수를 의미하기 때문이다.

성질 5: Disjoint Module Additivity. 성질 5가 의미하는 바는 서로 관련이 없는 두 개의 컴포넌트가 통합될 경우 통합된 컴포넌트의 결합도는 통합되기 전의 컴포넌트 각각의 결합도의 합과 같음을 의미한다.

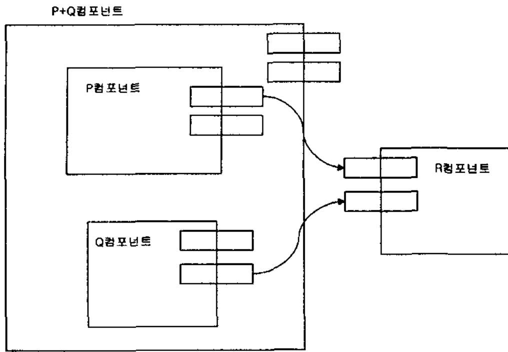


그림 4 서로관련 없는 컴포넌트끼리 합할 경우의 결합도의 변화

위의 그림에서 처럼 컴포넌트 P,Q,R이 존재하고, 각각의 인터페이스의 수는 개라고 가정하자. 위의 그림에서 보는 바와 같이 P컴포넌트와 Q컴포넌트는 서로 참조하는 인터페이스가 하나도 존재하지 않는다. 이러한 경우 CBCO(P)의 값은 1-1/3=2/3이다. CBCO(Q)의 값은 1-1/3=2/3이다. 만일, P와 Q를 합하게 되면 P+Q의 CBCO(P+Q)의 값은 1-2/6=4/6가 되어서 CBCO(P)와 CBCO(Q)의 합인 4/3와 같지 않게 된다. 따라서, CBCO정의는 성질 5를 만족하지 못한다. CBCO가 성질 5를 만족하지 못하는 이유는 Briand가 정의한 성질 5는 단순히 결합도의 개수측면에서 고려한 성질이다. CBCO는 단순한 결합도의 개수가 아니라 결합도의 비율을 나타내기 때문에 Briand가 제안한 성질 5에는 부적합하다.

3.4.4 Cohesion in Components(COHC)

COHC는 Briand[12]가 제안한 응집도 측정 기준이 갖추어야 할 기본적인 요건에 비추어 평가해 본다.

성질 1: Nonnegativity and Normalization. 성질 1이 의미하는 바는 응집도 측정값은 어떤 특정한 범위 내에 존재해야 함을 의미한다. COHC는 정의에서 알 수 있듯이 비율로 나타난다. 따라서, 최대값은 1이고, 최소값은 0이다. COHC의 값은 응집도가 크면 클수록 커지고, 응집도가 작아지면 COHC의 값도 작아진다. 따라서, COHC는 성질 1을 만족한다.

성질 2: Null Value. 성질 2가 의미하는 바는 컴포넌트 내부에서 서로 관련있는 인터페이스가 전혀 없다면 응집도의 값은 null값을 의미한다. 만일, 컴포넌트 내에서 인터페이스가 공유하는 클래스가 전혀 없다면 COHC의 값은 0이 될 것이다. 왜냐하면, |P|의 값이 |P∪Q|의 값과 동일해지기 때문이다. 따라서, COHC는 null값을 의미하는 0을 가질 수 있기 때문에 성질 2를 만족한다.

성질 3: Monotonicity. 성질 3이 의미하는 바는 컴포넌트 내부의 상호 작용이 늘어나게 되면 응집도 값은 최소한 같거나 늘어나게 된다는 의미이다. 컴포넌트 P 내부에 인터페이스가 더 늘어난다고 가정해보자. 만일, 추가된 인터페이스가 기존의 인터페이스들이 사용하는 클래스를 사용하게 되면 COHC의 정의에 의해서 |P∪Q|의 값은 늘어나게 되고, 따라서 COHC의 값은 늘어나게 된다. 만일, 추가된 인터페이스가 기존의 인터페이스들이 사용하는 클래스를 전혀 사용하지 않고 다른 컴포넌트의 외부 인터페이스를 사용한다면, 그 인터페이스는 추가하려는 컴포넌트에 맞지 않은 인터페이스가 된다. 따라서, COHC는 컴포넌트 내부의 상호작용이 늘어나게 되면 값이 증가한다. COHC는 성질 3을 만족한다.

성질 4: Cohesive Modules. 성질 4가 의미하는 바는 서로 관련이 없는 두 개의 컴포넌트가 통합될 경우 통합된 컴포넌트의 응집도는 통합되기 전 두 개의 컴포넌트 응집도의 최대값보다 더 커질 수 없다는 의미이다. 컴포넌트 P와 Q가 서로 관련되는 부분이 전혀 없다고 가정한다면, 두개의 컴포넌트를 통합하게 되면 COHC의 값은 커봐야 두개의 컴포넌트의 최대값이 될 것이다. 예를 들면, P와 Q 컴포넌트 각각의 COHC값이 COHC(P)=1-2/4=2/4이고, COHC(Q)= 1-1/5=4/5라고 가정하자. P와 Q를 통합하게 되면 공통되는 부분이 없고, 상호작용도 없기 때문에 컴포넌트 P+Q의 COHC(P+Q)=1-3/9=6/9가 되어서 P와 Q의 최대값인4/5보다 작게 된다. 따라서, COHC는 성질 4를 만족한다.

3.4.5 Number of Patterns in Components(NOPC)

NOPC는 Briand[12]가 제안한 크기 메트릭이 갖추어야 할 요건에 대해서 NOPC에 적용해 본다.

성질 1: Nonnegativity. 성질 1이 의미하는 바는 컴포넌트의 크기 측정값은 항상 0이상이어야 한다는 의미이다. 컴포넌트에 존재하는 패턴의 수는 음수가 될 수는 없다. 따라서, NOPC는 성질 1을 만족한다.

성질 2: Null Value. 성질 2가 의미하는 바는 만일 컴포넌트 내부에 클래스가 존재하지 않는다면 크기 측정값은 null값을 가진다는 의미이다. 만일, 컴포넌트 내부에 패턴이 존재하지 않는다면 NOPC의 값은 0이 될 것이다. 따라서, NOPC는 성질 2를 만족한다.

성질 3: Module Additivity. 성질 3이 의미하는 바는 컴포넌트를 구성하는 서브 컴포넌트가 서로 공통되는 부분이 전혀 없을 경우 컴포넌트의 크기 측정값은 서브 컴포넌트들의 크기 측정값의 합계와 같다는 의미이다. 컴포넌트 P가 있고, 내부에 서로 공통되는 부분이 없는 서브 컴포넌트 P′와 P″이 있다고 가정하자. 컴포넌트 P의 NOPC값은 P′와 P″의 NOPC값의 합계이다. 공통되는 부분이 없기 때문이 P의 패턴의 수는 내부에 존재하는 P′와 P″내에 존재하는 패턴의 수와 같기 때문이다. 따라서, NOPC는 성질 3을 만족한다.

3.4.6 Granularity In COmponents(GICO)

GICO는 Briand가 제안한 크기 메트릭이 갖추어야 할 요건을 GICO에 적용하여 평가해 본다.

성질 1: Nonnegativity. GICO는 컴포넌트내에 존재하는 클래스의 개수를 측정하기 때문에 최소한 0이상이 된다. 컴포넌트내에 클래스가 하나도 존재하지 않는다면 GICO의 값은 0이 되지만 음수 값은 나올 수 없다. 따라서, GICO는 성질 1을 만족한다.

성질 2: Null Value. 컴포넌트가 클래스를 하나도 가지고 있지 않다고 가정해보자. GICO의 값은 0을 의미한다. 이러한 경우는 거의가 일어나지 않지만 일어난다고 해도 GICO는 0이 되어서 성질 2를 만족한다.

성질 3: Module Additivity. 컴포넌트 P가 있고 P컴포넌트의 내부에는 서브 컴포넌트 P′와 P″이 있고 P′와 P″는 서로 공통되는 부분이 없다고 가정하자. GICO(P)의 값은 서브 컴포넌트 P′의 GICO값과 P″의 GICO값을 합한 값이다. 왜냐하면, 공통되는 부분이 없기 때문에 내부의 클래스들은 두 개의 컴포넌트에 나누어서 존재하기 때문이다. 따라서 GICO는 성질 3을 만족한다.

4. 사례 연구

본 논문의 사례 연구에서는 소프트웨어 컴포넌트를 위에서 제시한 측정 메트릭을 이용하여서 측정하여 보았다. 사례 연구에 사용될 소프트웨어 컴포넌트는 OpenEJB라는 컴포넌트이다. 이 컴포넌트는 시간이 지남에 따라서 버전이 증가하면서 그 기능이 늘어났다. 본 논문에서는 이러한 컴포넌트들이 버전이 증가하면서 본 논문에서 제시한 측정기준과 Caldiera와 Basili[3]가 제안한 메트릭으로 측정한 측정값이 어떻게 변화하는가를 알아보았다. Caldiera와 Basili[3]는 그의 논문에서 재사용성의 측정에는 Halstead의 프로그램의 사이즈를 이용하여 측정하고, McCabe의 복잡도를 이용하여 측정하며, 정규도(Regularity), 그리고 재사용 빈도수를 이용하여 측정 할 수 있다고 하였다. 본 논문에서도 Caldiera와 Basili[3]가 제시한 측정 방법인 프로그램의 사이즈, McCabe의 복잡도, 그리고 정규도를 이용하여 재사용성에 대한 측정을 하였다. 그러나, 재사용 빈도수는 실제 다른 컴포넌트에 얼마나 많이 재사용이 되었는지 정확한 확인이 어려워 본 논문에서는 제외하기로 하였다.

4.1 직접/간접적인 측정기준에 의한 측정

먼저 본 논문에서 제시한 측정 기준인 간접적인 측정값을 알아보기 위하여 위에서 정의한 각각의 간접적인 측정 기준에 실제 값을 대입하였다. 간접적인 측정 기준인 이해성은 다음과 같이 대입하였다.

수식1: $U(C)=--0.3CCN(C)+0.2HICO(C)+0.3NOPC(C)-0.2GICO(C)$. 수식1은 본 사례 연구에서 측정하려고 하는 이해성에 대한 수식이다. 수식1에서 보는 바와 같이 *CCN, GICO*의 계수는 음수를 두었고, *HICO, NOPC*는 양수를 두었는데 CCN의 값과 GICO의 값은 증가할수록 이해성에 나쁜 영향을 끼치기 때문이다. 따라서, $U(C)$의 값이 증가할수록 이해성은 높아진다.

수식2: $A(C)=-0.4CCN(C)+0.3CBCO(C)+0.3COHC(C)$. 수식2는 적용가능성을 나타낸다. CCN는 그 수가 증가할수록 적용가능성에 나쁜 영향을 끼치는 반면 *CBCO, COHC*는 그 수가 증가할수록 적용가능성에 좋은 영향을 끼친다. $A(C)$의 값이 증가할수록 적용가능성은 높아진다.

수식3: $C(C) = 0.5HICO(C) + 0.5NOPC(C)$. 수식3은 수정가능성을 나타낸다. HICO, *NOPC*는 수정가능성에 좋은 영향을 끼친다. 상속의 구조와 패턴의 수가 증가 한다는 것은 컴포넌트의 내부 구조가 좀더 체계화 되어 있다는 증거이기 때문에 수정을 더 용이하게 할 수 있기 때문이다. 수식 3의 경우는 측정자가 HICO의 값과 NOPC의 값을 수정가능성에 동일하게 중요시 하였기 때문에 0.5로 같은 계수값을 주었다. $C(C)$의 값이 증가 할수록 수정가능성은 높아진다.

수식4: $M(C) = 0.5CBCO(C)+0.5COHC(C)$. 수식4는 모듈화 가능성을 나타낸다. *CBCO, COHC*는 그 값이 증가 할수록 모듈화 가능성이 늘어난다. 이유는 결합도가 떨어지고, 응집도가 늘어난다는 것은 컴포넌트의 구조가 독립적인 구조를 갖추었다는 증거이기 때문이다. $M(C)$의 값이 증가할수록 모듈화 가능성은 높아진다.

본 사례 연구에서 측정하려고 하는 소프트웨어 컴포넌트는 OpenEJB는 시간이 지남에 따라 계속해서 버전이 증가하였다. OpenEJB컴포넌트는 2002년 5월부터 2003년 6월까지 모두 7번의 수정이 있었다. 이러한 수정과정에서 컴포넌트의 버전이 증가하면서 내부의 기능도 추가가 되었으며, 성능면에서 이전 버전의 제품 보다는 향상된 것을 추정할 수 있다.

표 4는 OpenEJB의 버전에 따른 직접적인 측정값을 나타낸다. 위의 표에서 보는 바와 같이 버전이 증가하면서 각각의 측정값이 증가하는 것을 보인다. 이는 버전이 증가하면서 컴포넌트 내부 구조가 재사용성에 맞게 최적화 되어감을 의미한다.

그림 5는 OpenEJB컴포넌트의 버전이 변하면서 HICO, CBCO, COHC의 값이 어떻게 변화하는 가를 나타낸 그래프이다.

그림 6은 그림 5와 같이 OpenEJB컴포넌트가 버전이 변화하면서 CCN, GICO의 값이 어떻게 변화하는가를 나타내었다. 위의 그림 5,6은 표 4의 측정값의 변화를 그래프로 표현한 것이다. 위의 그림 5와 6에서 가로축은 OpenEJB의 버전의 증가를 나타내고, 세로축은 직접적

표 4 OpenEJB컴포넌트의 버전에 따른 직접적인 측정값

Product / Metric	OpenEJB 0.8	OpenEJB 0.8.1	OpenEJB 0.8.2	OpenEJB 0.8.3	OpenEJB 0.9.0	OpenEJB 0.9.1	OpenEJB 0.9.2
CCN	82	83	75	77	79	79	79
HICO	0.247	0.337	0.395	0.405	0.569	0.591	0.603
CBCO	0.312	0.378	0.401	0.442	0.578	0.712	0.815
COHC	0.501	0.538	0.513	0.662	0.691	0.701	0.773
NOPC	2	2	3	3	5	5	5
GICO	160	167	153	151	151	152	151

표 5 OpenEJB컴포넌트의 버전에 따른 간접적인 측정값

Product / Metric	OpenEJB 0.8	OpenEJB 0.8.1	OpenEJB 0.8.2	OpenEJB 0.8.3	OpenEJB 0.9.0	OpenEJB 0.9.1	OpenEJB 0.9.2
이해성	-54.5	-57.6	-52.1	-49.3	-43.5	-41.8	-38.1
적용가능성	-31.7	-32.7	-29.7	-27.2	-25.8	-23.4	-19.1
수정가능성	1.1	1.2	1.6	1.9	2.0	2.3	2.5
모듈화가능성	0.4	0.4	0.5	0.7	0.8	0.9	1.0

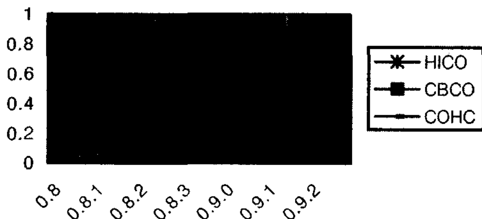


그림 5 OpenEJB컴포넌트의 버전에 따른 직접적인 측정값의 변화 1

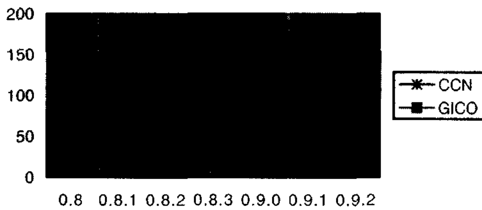


그림 6 OpenEJB컴포넌트의 버전에 따른 직접적인 측정값의 변화 2

인 측정값을 나타낸다. 위의 그림에서 보듯이 HICO, CBCO, COHC는 버전이 증가하면서 그 값이 점차 증가하는 것을 나타내고, CCN, GICO는 버전의 변화에 따라서 그 값이 점차 감소함을 나타낸다.

표 5는 위의 표 4에서 측정한 OpenEJB컴포넌트의 직접적인 측정값을 이용하여서 수식 1,2,3,4에 대입하여 얻은 값이다. 위의 표 4와 5에서 보는 바와 같이 값이 버전이 증가 함에 따라 점차 증가함을 보인다. 위의 표 5에서 보는 바와 같이 이해성과 적용가능성의 측정값은 음수가 나왔는데 이러한 원인은 이해성은 CCN과 GICO에 비중을 두어서 계수에 음수를 부여 했기 때문이다. 따라서, CCN의 값은 많은 값을 차지하는 반면 HICO나 NOPC는 상대적으로 적은 값을 가졌기 때문에 결과로 음수가 나오게 된 것이다. 그러나, 음수가 나오더라도 점점 그 음수가 양수 쪽으로 이동하고 있음은 곧 그 컴포넌트의 이해성이 향상되고 있음을 의미한다고 볼 수 있다. 적용가능성도 마찬가지로 CCN의 값이 다른 CBCO와 COHC의 값보다 훨씬 많이 차지 했기 때문에 음수가 나오게 되었다. 그러나 적용가능성 역시 점점 음수가 양수 쪽으로 이동하고 있음은 적용가능성이 좋아지고 있음을 말해준다.

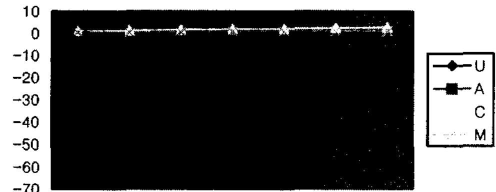


그림 7 OpenEJB컴포넌트의 버전에 따른 간접적인 측정값의 변화

위의 그림 7은 표 5에서 얻은 간접적인 측정값을 그래프로 표현한 것이다. 그림의 오른쪽에 있는 U는 이해성(Understandability)를, A는 적용가능성(Adaptability), C는 수정가능성(Customizability), 그리고 M은 모듈화가능성(Modularity)을 의미한다. 위의 그림 7에서 보는 바와 같이 버전이 증가하면서 간접적인 측정 값은 점차 증가하는 것으로 나타났다. 이는 버전이 증가 하면서 컴포넌트의 품질이 향상되고, 재사용성도 늘어난다는 추측과 일치한다.

표 6 OpenEJB컴포넌트의 버전에 따른 Caldiera와 Basili [3]의 측정기준에 의한 측정값

Metric \ Product	OpenEJB 0.8	OpenEJB 0.8.1	OpenEJB 0.8.2	OpenEJB 0.8.3	OpenEJB 0.9.0	OpenEJB 0.9.1	OpenEJB 0.9.2
프로그램 사이즈	8976	9053	9148	8871	8504	7993	7901
복잡도	21.1	23.6	21.8	19.3	18.8	18.1	17.3
정규도	0.77	0.72	0.66	0.83	0.89	0.90	0.89

4.2 Caldiera와 Basili[3]의 측정기준에 의한 측정

두 번째 측정 방법으로 Caldiera와 Basili[3]가 제안한 프로그램의 사이즈, McCabe의 복잡도, 그리고, 정규도를 이용하여 위에서 제시한 OpenEJB컴포넌트에 대해서 측정을 한다. 측정방법은 위에서 설명한 직접/간접적인 측정 기준을 측정할 때와 동일하게 컴포넌트의 버전이 증가함에 따라서 측정값이 어떻게 변화하는지를 측정한다.

위의 표에서 보는 바와 같이 프로그램의 사이즈는 버전이 증가하면서 작아졌으며, 복잡도도 버전이 증가하면서 작아졌다. 반면, 정규도는 버전이 증가하면서 1에 가까워졌다. Caldiera와 Basili의 논문에 따르면 정규도는 1에 가까울수록 재사용성이 높은 소프트웨어라고 하였다[3]. 따라서, 버전이 증가하면서 컴포넌트가 재사용성이 높은 컴포넌트로 발전하였음을 알 수 있다. 따라서, 본 논문에서 제시하는 컴포넌트에 관한 재사용성 측정 기준은 타당성이 있다고 할 수 있다.

5. 결 론

컴포넌트 기반 개발(CBD)은 개발 기간 단축과 개발 비용의 감소를 가져 온다는 이점 때문에 산업계에서 많이 이용되고 있다. 그러나, 개발된 컴포넌트에 대해서 재사용성 측면에서의 품질 측정에 대한 연구는 아직까지 미흡한 실정이다. 본 논문에서는 소프트웨어 컴포넌트의 재사용성 측면에서 측정 기법을 제시하였다. 이를 위하여 소프트웨어 컴포넌트의 재사용성 측정 기준을 직접적인 측정 기준과 간접적인 측정 기준으로 나누어 정의하였다. 직접적인 측정 기준에는 6가지가 있었으며, 간접적인 측정 기준에는 4가지가 있었다. 직접적인 측정 기준은 컴포넌트의 소스코드와 설계를 기반으로 측정하는 방법을 말하고, 간접적인 측정기준은 이러한 직접적인 측정 기준을 이용해서 얻을 수 있는 좀더 추상화 된 측정 기준을 말한다. 본 논문에서는 직접적인 측정 기준을 이용하여서 간접적인 측정 기준을 측정 하고, 궁극적으로 소프트웨어 컴포넌트의 재사용성을 측정하는 방법을 제시하였다.

참 고 문 헌

[1] Jeffrey S. Poulin, "Measuring Software Reusability," Proceeding of the Third International Conference on Software Reuse, Rio de Janeiro, Brazil, 1-4 November 1994, pp.1-5.

[2] Prieto-Diaz, Ruben and Peter Freeman, "Classifying Software for Reusability," IEEE Software, Vol. 4, No. 1, January 1987, pp.6-16.

[3] Caldiera, Gianluigi and Victor R. Basili, "Indentifying and Qualifying Reusable Software Components," IEEE Software, Vol. 24, No. 2, Febuary 1991, pp.61-70.

[4] Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw-Hill, 2002.

[5] STARS, "Repository Guidelines for the Software Technology for Adaptable, Reliable Systems (STARS) Program," CDRL Sequence Number 0460, 15 March 1989.

[6] ISO/IEC, FCD 9126-1.2 Information Technology-Software product quality-Part 1:Quality model, 1998.

[7] Katherine Whitehead, Component-based Development: Principles and Planning for Business Systems, Addison Wesley, 2002.

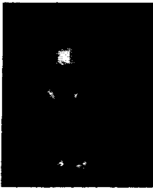
[8] Elaine J. Weyuker, "Evaluating Software Complexity Measures," IEEE Transactions on Software Engineering. vol. 14. NO.9, September 1988, pp.1357-1365.

[9] Atkinson C., Bayer J., Bunse C., Kamsties E., Laitenberger O., Laqua R., Muthig D., Paech B., Wüst J, Zettel J., Component-based Product Line Engineering with UML, Addison Wesley, 2001.

[10] Shyam R. Chidamber, Chris F. Kemerer, "A Metrics Suit for Object Oriented Design," IEEE Transactions on Software Engineering. vol. 20. NO.6, June 1994, pp.476-493.

[11] John Cheesman, John Daniels, "UML Components: A Simple Process for Specifying Component-Based software," Addison-Wesley, 2001.

[12] lionel C. Briand, Sandro Morasca, Victor R. Basili, "Property-Based Software Engineering Measurement," IEEE Transactions on Software Engineering, vol 22. NO. 1, January 1996, pp.68-86.



박 인 근

1992년 3월~1999년 2월 연세대학교 전산학과(학사). 2001년 8월~현재 숭실대학교 대학원 컴퓨터 공학과(석사). 2003년 3월~현재 ㈜CJ Systems 소프트웨어 공학연구소 연구원. 관심분야는 OOA/D, CBD, UML, EJB, MDA, Software Reusability

김 수 동

정보과학회 논문지 : 소프트웨어 및 응용 제 31 권 제 1 호 참조