객체지향 모델로부터 정적 메트릭을 이용하여 컴포넌트 기반 시스템으로 변환하는 기법

(A Transforming Technique toward Component-based System from Object-oriented Model Using Static Metrics)

이 은 주 † 신 우 창 †† 이 병 정 ††† 우 치 수 ††††
(Eunjoo Lee) (Woochang Shin) (Byungjeong Lee) (Chisu Wu)

요 약 점차적으로 소프트웨어의 복잡도는 높아지고 그 생명주기는 짧아지므로, 기존의 검증된 소프트웨어 요소를 재사용하는 것이 필요하다. 그러나 기존의 객체 지향 기술은 광범위한 재사용을 유도하지 못하였다. 컴포넌트는 객체보다 규모가 크고 특정 도메인에 적합한 특성을 가지므로, 시스템 구조화, 설명 및 개발에 있어 더 나은 수단을 제공해 준다. 또한 그 규모나 재사용성에 있어 새로운 개발환경인 분산 시스템에 더 적합하다. 본 논문에서는 객체지향 시스템을 컴포넌트 기반 시스템으로 변환하는 프로세스를 제안하였다. 해당 프로세스는 두단계로 나뉜다. 우선, 클래스들 간의 합성 및 상속 관계를 이용하여 기본 컴포넌트를 생성한다. 그 후 기본 컴포넌트와 컴포넌트화 되지 않은 클래스들에 대해 본 논문에서 제안된 정적 메트릭과 가이드라인을 이용하여 정제하여 컴포넌트 기반 시스템으로 변환한다.

키워드 : 컴포넌트, 객체지향, 재사용, 메트릭

Abstract The increasing complexity and shorter life cycle of software have made it necessary to reuse software. Object-oriented development had not provided extensive reuse and computing infrastructures are evolving from mainframe to distributed environments. However, components provide more advanced means of structuring, describing and developing system, because they are more coarse-grained and have more domain-specific aspects than objects. And they are also suited for distributed environment due to their reusability and granularity. In this paper, we present a process including the static metrics and guidelines that can be applied to transform object-oriented systems into component-based systems. Our process consists of two parts: First, basic components are created based upon composition and inheritance relationships between classes. Second, intermediate system is refined into component-based system with the static metrics and guidelines we propose.

Key words : component, object-oriented, reuse, metric

1. 서 론

현재 소프트웨어 시스템의 복잡도가 증가되고, 그 생명주기는 점차 짧아지고 있는 추세이다. 따라서 검증된 기존의 소프트웨어 요소를 재사용 하여 개발 시에 시스템의 신뢰성을 향상시키고 효율을 증진시키는 것이 필요하다.

객체지향 기술은 소프트웨어의 재사용을 촉진시켰으나, 그 단위가 되는 클래스들이 지나치게 세부적이고 한정적이며, 클래스들은 컴파일 시나 링킹 시에 어플리케이션에 결합되어야 한다. 클래스를 이용하기 위해서는 그에 대해 상세히 알아야 하고 이는 소스코드가 늘 이용 가능해야 함을 뜻한다. 이러한 이유들로 인해 객체지향 개발은 광범위한 재사용을 유도하지 못하였다[1]. 오늘날 객체지향 기술에 더하여, 컴포넌트 기반 소프트웨어 기술이 자리를 잡아가고 있다. 컴포넌트 기술은 객체지향의 목표와 유사하게, 재사용을 증진시켜 저비용으로 신뢰성 있는 소프트웨어의 개발 및 사용을 지향한다[2]. 작은 단위의 재사용 개체로는 효율적으로 작업하기 어

· 본 연구는 한국과학재단 목적기초연구(R01-1999-00238)지원으로 수행되었음
† 비 회 원 : 서울대학교 전기컴퓨터공학부
banny@selab.snu.ac.kr
†† 비 회 원 : 서경대학교 인터넷정보학과
wcshin@imail.skuniv.ac.kr
††† 종신회원 : 서울시립대학교 컴퓨터과학부
bjlee@venus.uos.ac.kr
†††† 종신회원 : 서울대학교 전기컴퓨터공학부
wuchisu@selab.snu.ac.kr
논문접수 : 2003년 8월 6일
심사완료 : 2004년 3월 26일

려우나, 수많은 클래스와 객체들로 구성된 컴포넌트와 같은 보다 큰 단위의 도메인 기반 접근법을 이용하면, 이해나 유지보수가 어려운 대규모 시스템을 구조화, 이용, 개발하는 데 향상된 접근법을 제공해 준다[3,4].

시스템의 수행 환경은 여러 다른 플랫폼에서 다른 언어로 개발된 어플리케이션들이 수행되는 분산환경으로 바뀌고 있다. 이러한 분산 환경상의 많은 어플리케이션들이 새로운 컴포넌트의 생성과 기존 컴포넌트의 이용에 초점을 맞추는 컴포넌트 기반 시스템들이다[5]. 이렇게 새로운 수행 환경이 도입, 확산되면서, 과거에는 '레거시 시스템(legacy system)'이 C나 FORTRAN, COBOL과 같은 기존의 절차적 언어로 구현된 시스템을 일컬었으나, 오늘날에는 객체지향 시스템을 포함하는 의미로 바뀌게 되었다[6]. 따라서 객체지향 시스템을 분산 환경에 적합하게 재공학 하고 광범위하게 재사용 하기 위해서, 객체지향 시스템으로부터 재사용 가능한 컴포넌트들을 추출하여 컴포넌트 기반 시스템으로 변환시켜 이용하는 것이 효율적일 것이다.

레거시 시스템을 객체기반의 분산 시스템으로 재공학하는 연구가 수행되었다[7-9]. 연구 대상이 되는 레거시 시스템은 절차적 언어로 개발된 시스템이다. 또 분석단계 객체 모델에서 컴포넌트를 식별하는 연구가 제안되었으나[4], 객체지향 시스템을 컴포넌트들로 구성된 시스템으로 변환하는 완전한 방법론을 제공해 주고 있지 않다.

본 논문에서는 객체지향 시스템을 컴포넌트 기반 시스템으로 변환하는 프로세스를 제안하고 있다. 우선 클래스간의 관계에 기반 하여 기본 컴포넌트를 생성하고, 분산환경에서 응집력 있는 기능적 단위로 사용되는 컴포넌트를 추출하는데 이용되는 메트릭과 컴포넌트의 품질을 측정하는 메트릭을 정의하였다. 최종적으로 이러한 메트릭을 이전 단계의 결과물인 기본컴포넌트와 컴포넌트화 되지 않은 클래스들에 적용하여 최종 컴포넌트를 생성한다.

본 논문의 구성은 다음과 같다. 2장에서는 기본 연구와 문제점을 살펴보고 3장에서는 시스템 명세 모델을 정의한다. 4장에서는 객체지향 시스템의 기본 단위인 클래스들로부터 컴포넌트를 생성하는 프로세스를 제안한다. 5장에서는, 4장에서 정의한 프로세스를 객체지향 시스템에 적용하여 컴포넌트 기반 시스템으로 변환하는 예를 보이며 6장에서는 결론 및 향후 과제에 관해 기술한다.

2. 관련연구

레거시 시스템을 분산 객체 환경으로 점진적으로 옮기는 방법론이 제안되었다[7]. 이러한 점진적인 이주를 위해서는 레거시 시스템을 일시적으로 사용하면서, 점차적으로 새로운 환경으로 옮겨야 하며, 이를 위해 기존의 레거시 시스템을 컴포넌트화하는 과정은 다음과 같다. 우선 ISA(Identification of Subsystems based on Associations)방법을 이용하여 레거시 시스템을 서브시스템으로 계층화하고, 각각의 서브시스템을 객체 기반으로 변형한 후, 래퍼(wrapper)를 이용하여 컴포넌트화 한다. 컴포넌트들이 "인터페이스로 래핑(wrapping)된 객체들의 집합"으로 정의된 것은 본 연구와 유사하지만, 그 대상은 절차적 시스템이며 결과 시스템은 객체기반 분산 시스템이다. 또한 품질과 같은 컴포넌트의 성질에 대해서는 다루고 있지 않다.

비즈니스 도메인을 표현하는 객체 모델로부터 컴포넌트를 식별하는 방법이 소개되었다[4]. 이 방법에서는 미리 정의한 규칙과 휴리스틱을 이용한 클러스터링 알고리즘을 적용하고 있다. 클래스간의 연관 관계로부터 파악된 정적 관계와 유스케이스 다이어그램과 시퀀스 다이어그램에서 이끌어 낸 동적 관계를 이용하여 계층적이고 집적적으로 클래스들을 클러스터링 한다. 품질 개선을 위해 컴포넌트 내부 클래스의 부모 클래스가 컴포넌트 외부에 있다면 그 부모 클래스를 컴포넌트 내부로 복제하도록 한다. 또한 클러스터링 알고리즘으로부터 나온 결과를 정제하기 위해 설계자나 시스템에 의해 각각 수동적, 자동적으로 휴리스틱들을 적용하도록 한다. [4]에서는 컴포넌트 인터페이스에 관한 언급이 없으며, 실제 프로세스를 수행할 때, 유스케이스 다이어그램이나 시퀀스 다이어그램 같은 분석 모델이 존재해야 한다. 또한 한 클래스의 여러 자식 클래스들이 모두 각기 다른 컴포넌트에 산재해 있을 경우에는 비효율적이다.

컴포넌트의 결합도(coupling)와 응집도(cohesion), 의존, 인터페이스, 규모, 아키텍처를 포함하는 컴포넌트 식별 방법이 제안되었다[10]. 이 방법에서는, 컴포넌트의 응집도와 결합도를 컴포넌트 식별의 주요 요소로 보고 있다. 응집도를 측정하기 위해 유스케이스 다이어그램에서 주요 기능을 식별하고, 시퀀스 다이어그램과 콜레보레이션 다이어그램을 이용하여 각 기능에 해당되는 클래스를 중요도에 따라 다섯 가지로 분류를 한다. 또한 결합도를 상호관련(interaction) 커플링과 정적 커플링으로 분류하여, 시퀀스 다이어그램에서 호출된 메서드의 수를 이용해서 상호관련 결합도를 측정하고, 클래스 다이어그램에서 각 클래스들의 관계를 통해 정적 결합도를 측정한다. 이 값들을 기반으로 하여 도출된 클래스 관계 그래프에 대해 클러스터링 알고리즘을 수행하여 컴포넌트를 생성한다. 이 방법은 분석, 설계정보인 유스케이스, 시퀀스, 콜레보레이션 다이어그램 등을 필요로 하므로 컴포넌트에 기반한 개발(CBD)시에 유용한 방법

이다.

재사용 가능한 컴포넌트의 생성과 식별에 도움을 주기 위해 소프트웨어의 설계와 구현 단계에서 반복적으로 사용될 수 있는 재사용성 메트릭들이 제안되었다[11]. 설계자가 주관적으로 객체지향 코드를 분석하여 클래스들을 재사용성 수준에 따라 일반(general) 클래스 레벨 0에서 특수(special) 클래스까지 수준을 나누고, 향후 함께 이용될 만한 클래스들을 연관짓는다(Related). 그리고, 일반·특수 클래스와 그들간의 'Related' 관계를 바탕으로 여덟 가지 결합도 유형을 제안하여, 향후 재사용을 증진시키는 데 이용한다. 예를 들어, 결합도 개수(Coupling Count) 유형1은 모든 일반 클래스들에 대해 그들과 연관된 일반 클래스들간의 결합도 개수로, 이러한 결합도는 바람직하다. 그리고 유형5는 모든 특수 클래스들에 대해 그들과 연관된 일반 클래스와의 결합도 개수이며, 재사용성을 향상시키기 위해서는 유형1로 변경되어야 한다. 이 방법을 이용하기 위해서 소프트웨어 설계자들은 어플리케이션 도메인과 시스템 종류에 대해 확고한 지식을 가지고 있어야 한다.

클러스터 분석을 이용하여 객체지향 시스템을 모듈화하는 정량적인 방법이 제안되었다[12]. 이 연구에서는 분류 구조에 따라 클래스 사이 관계를 구분하고 각 범주에 가중치를 할당하여 상대적인 클래스 사이 결합도로 정의한다. 그리고 이 상대적인 결합도를 사용하여 클래스 사이 상이도를 정의하고 클러스터 분석을 수행한다. 그러나 모듈이 컴포넌트가 될 수 있으나 모듈 결합도는 클래스들간의 정적인 관계만을 고려하여 클래스들간에 전달되는 정보의 양은 생각하지 않는다. 또한 클러스터링 전에 모듈의 개수가 미리 정해진다.

3. 시스템 명세 모델

본 논문에서 가정하고 있는 시스템 모델은 다음과 같다.

• 상속관계

한 클래스 X가 다른 클래스 Y로부터 멤버 함수나 멤버 변수를 직접적으로 받아서 재사용 할 때, 'X와 Y는 상속관계를 가진다' 또는 'X는 Y의 자식이다' 또는 'Y는 X의 부모이다' 라고 한다. 그리고 한 클래스 X가 어떤 클래스 $Y_1...Y_n$ ($n \geq 1$인 정수)의 멤버함수 혹은 멤버 변수를 직, 간접적으로 받아서 재사용 할 때, $Y_1...Y_n$는 X의 조상 클래스라고 하고, 한 클래스 X의 멤버함수 혹은 멤버 변수를 어떤 클래스 $Z_1...Z_m$ ($m \geq 1$인 정수) 이직, 간접적으로 받아서 재사용 할 때, $Z_1...Z_m$은 X의 후손 클래스라고 한다.

• 합성 관계

한 클래스 X가 다른 클래스 Y의 한 인스턴스 y를 자신의 멤버 변수로 가지되, y가 X에 강한 소속력을 가지고 생명주기를 함께 할 때, 'X와 Y는 합성관계이다' 또는 'X는 Y를 합성 멤버 클래스로 갖는다' 라고 한다.

• 사용 관계

한 클래스 Y의 멤버함수의 인자로 X의 한 인스턴스 x를 가지거나, Y가 X의 한 인스턴스 x를 합성관계를 제외한 멤버 변수로 갖거나, Y의 멤버함수 내에 X의 한 인스턴스 x가 존재할 때 'Y는 X를 사용한다' 라고 한다.

• 의존관계

컴포넌트 Y가 컴포넌트 X가 제공해주는 서비스를 이용할 때 'Y는 X에 의존한다'고 한다.

• UID는 유일한 식별자들의 전체집합이다.

• DTYPE는 모든 자료형의 전체집합이다.

- DTYPE = PTYPE ∪ ETYPE ∪ UC
  - PTYPE은 *integer, real, bool, string*등과 같은 기본 자료형의 집합이다
  - ETYPE은 모든 *enumeration* 형의 전체 집합이다.

• SIG는 메서드의 인자와 결과 자료형의 전체집합이다.

- SIG = $\{<name, s_1 \times ... \times s_n, s> \mid name \in UID, s_1 \times ... \times s_n, s \in DTYPE \cup \{\varnothing\}\}$
  - $\varnothing$는 존재하지 않는 값(empty value)이다.
- SORTS(x) = { s_1, ..., s_n, s }
  - $x = <name, s_1 \times ... \times s_n, s>$

• UC는 클래스들의 전체집합이다.

UC내의 각 클래스 c는 다음과 같은 튜플 <*name*, ATTRSET, MSET, GEN, CP, USE>로 정의된다.

- *accessor*는 멤버 변수(속성) 혹은 멤버 함수에 대한 접근 제어자이다.

  accessor ∈ { *public, private, protected* }
- *name* ∈ UID
- ATTRSET은 c의 속성인 <*type, accessor*>의 집합이다.
  - attr ∈ DTYPE
  - ATTRSET(c)는 클래스 c의 ATTRSET 원소를 추출해 주는 함수이다.
- MSET은 c의 멤버 함수를 나타내는 $<name_M, sig, body, accessor>$ 의 집합이다.
  - $name_M$ ∈ UID
  - *sig* ∈ SIG
  - *body*는 메서드의 실행가능한 코드이다.
  - SIG(m)은 메서드 m에 대해 그 *sig*를 추출해 주는 함수이다.
  - SIGLIST(m) = SORTS(SIG(m))
- GEN은 부모 클래스들의 집합이다.
  - GEN = { $p \mid p \in$UC, c 는 p를 상속받음 }

- GEN(c)는 클래스 c에 대해 그의 GEN 원소를 추출해 주는 함수이다.
- CP는 합성 멤버 클래스들의 집합이다.
  - CP = { d | d ∈UC, composition(c, d) }
  - *composition*(x, y)는, 만일 클래스 x가 클래스 y를 합성 멤버 클래스로 가지고 있을 때 참이고 그렇지 않으면 거짓이다.
- USE는 c가 사용하는 클래스들의 집합이다.
  - USE = {<u, m> | u ∈ UC, m = ∅ ∧ m ∈ MSET(u) }
  - MSET(x)는 클래스 x에 대해 그의 MSET 원소를 추출해 주는 함수이다.

• UCOMP는 컴포넌트들의 전체집합이다.

UCOMP의 각 컴포넌트 q는 다음과 같은 튜플 <*name*, INTERFACES, CSET, COMPDEP> 으로 정의된다.

- *name* ∈ *UID*
- INTERFACES는 q가 제공해 주는 모든 인터페이스들을 나타낸다. INTERFACES의 각 인터페이스 i는 다음과 같이 정의된다.
  - i = <$name_I$, *sig*>, where $name_I$ ∈ UID, *sig* ∈ SIG
- CSET은 q를 구성하는 클래스들의 집합이다.
  - CSET ⊂ UC
  - CSET(x)는 컴포넌트 x에 대해 그의 CSET 원소를 추출해 주는 함수이다.
- COMPDEP은 q가 의존하는 컴포넌트들의 집합이다.
  - COMPDEP ⊂ UCOMP

• 객체지향 시스템 S는 상호작용하는 클래스들의 집합이다.

- S ⊂ UC

• 변환된 시스템 S′는 컴포넌트들의 집합이다.

- S′ ⊂ UCOMP

이러한 가정을 바탕으로, 시스템을 튜플과 집합을 이용하여 명세하였다. 이러한 정형적인 모델은 시스템에 대한 이해를 명료하게 하여주고 연산을 정확히 수행할 수 있도록 해준다.

4. 변환 프로세스

본 장에서는 객체지향 시스템을 컴포넌트 기반 시스템으로 변환하는 프로세스에 관해 기술한다. 프로세스는 두 단계를 거치는데, 첫 번째 단계로 클래스들간의 합성 및 상속 관계를 기반으로 하여 기본 컴포넌트를 생성하며, 두 번째 단계로 중간단계의 컴포넌트 및 클래스들을, 정의한 메트릭을 이용하여 고품질의 컴포넌트들로 구성된 최종 컴포넌트 기반 시스템으로 클러스터링 한다. 이러한 과정을 그림 1에 나타내었다.

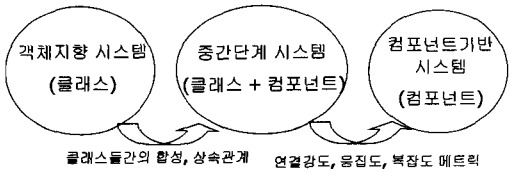


그림 1 변환 프로세스

4.1 기본 컴포넌트 식별

기본 컴포넌트는, 클래스들 간의 상속 관계와 합성 관계를 이용해서 생성된 중간 단계의 컴포넌트이다.

• **단계 1.** 합성 기본 컴포넌트

한 컴포넌트 내의 응집도는 최대, 컴포넌트간의 결합도는 최소가 되도록 하는 것이 이상적인 시나리오이다 [4]. 보통 합성관계를 가진 클래스들의 경우는 하나의 기능적인 단위로 작용을 하는 경우가 많다. 합성관계에 있는 두 클래스들은 생명 주기를 함께 하므로, 두 클래스가 각기 다른 컴포넌트에 속한다면 두 컴포넌트 간에도 일종의 합성관계가 생기게 될 것이다. 예를 들어, 그림 2.a에서 클래스 a와 b가 한 컴포넌트 *comp_a*에 속하고, c가 다른 컴포넌트 *comp_c*에 속한다고 하면, *comp_a*가 생성 될 때, *comp_c*도 생성이 되어 *comp_a*가 소멸될 때까지 계속 남아 있어야 한다. 결국 이들 클래스들이 다른 컴포넌트들에 분산되어 이용된다면 컴포넌트내의 응집도는 낮아지고 컴포넌트간의 결합도는 높아질 것이다. 따라서 본 논문에서는 아래와 같이 이들을 하나의 기본 컴포넌트로 만든다.

자신의 합성 멤버 클래스를 가지고 있는 각 클래스들에 대해

i) 클래스 a가 어떤 합성 기본 컴포넌트에도 속하지 않으면, 컴포넌트 *comp_a*를 생성하고 a와 a의 모든 합성 멤버 클래스들을 자식 클래스의 유, 무에 따라 *comp_a*에 복제 혹은 이동한다(그림 2(a)).

ii) 클래스 b가 이미 컴포넌트 *comp_a*에 포함되어 있다면, b의 모든 합성 멤버 클래스들을 자식 클래스의 유, 무에 따라 *comp_a*에 복제 혹은 이동한다(그림 2(b)).

본 단계에서 "클래스가 복제된다"는 것은 S에 그 원본 클래스가 존재하여 기본 컴포넌트 생성 시 이용될 가능성이 있다는 것을 의미한다. 그리고 "클래스가 이동된다"는 것은 클래스가 S에서 제거되어 기본 컴포넌트 식별 단계에서 사용되지 않음을 뜻한다. 합성 기본 컴포넌트를 생성하는 대상 클래스들의 자식 클래스가 있는 경우에는, 차후 상속 기본 컴포넌트 생성 시 이용되므로 복제하며, 자식 클래스가 없는 경우는 이동한다.

만일 시스템 자체의 설계 문제로 클래스간의 합성 계

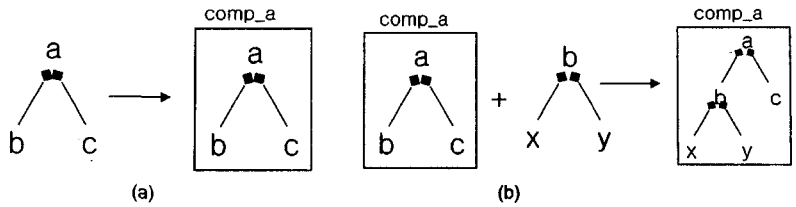


그림 2 합성 기본 컴포넌트

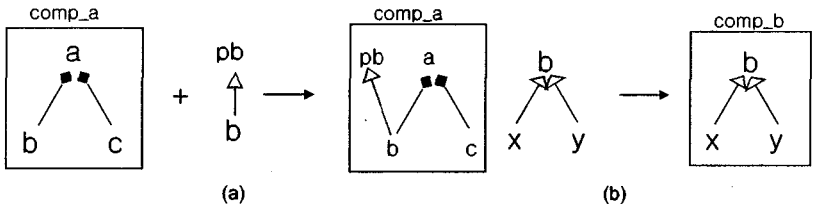


그림 3 상속 기본 컴포넌트

층이 과도하여, 생성된 합성 컴포넌트 내의 계층 구조가 복잡하다면 설계자가 적절히 그 계층을 분할할 수 있도록 한다.

그림 2에서는 UML 표기법과 같이 채워진 다이아몬드를 가진 실선으로 합성관계를 표기한다. 예를 들어 그림 2(a)와 그림 2(b)에서 클래스 a는 클래스 b를 합성 멤버 클래스로 갖는다.

• **단계 2.** 상속 기본 컴포넌트

클래스간의 상속관계가 있다면 그들간의 강한 응집력 때문에 이들을 한 컴포넌트 안에 두는 게 타당하다. 만일 상속관계에 있는 클래스들이 여러 컴포넌트로 분산된다면 컴포넌트간의 의존도도 높아질 것이다[4]. 그러나 만일 시스템이 여러 상속관계의 클래스들로 구성되었을 때, 해당 말단 클래스들마다 그들의 모든 직/간접 부모 클래스들을 포함시켜 컴포넌트를 만드는 것은 실용적인 접근법이라 볼 수 없다. 또한 공통의 부모를 갖는 모든 말단 클래스들을 하나의 컴포넌트로 묶는다면 컴포넌트의 복잡도가 크게 증가할 것이다. 또한 많은 객체지향 프로그램에서 가상함수를 통한 인터페이스 재사용이 행해지고 있으므로[13], 단말 클래스를 기준으로 컴포넌트를 만든다면, 추상 부모 클래스의 메서드 호출 시에 적절한 컴포넌트를 결정하는 부분에 관한 고려가 필요할 것이다.

본 논문에서는 상속관계를 구체, 추상의 두 유형으로 분류하였다. 클래스들 간의 상속 관계 구조에서 구체 클래스들간의 상속 관계로 이루어진 그룹에 대해 새 상속 기본 컴포넌트가 생성되고(그림 3(b)) 생성된 컴포넌트 외부에 추상 클래스가 부모 클래스로서 존재하면 그 추상 클래스를 해당 컴포넌트 내부로 복제한다(그림 3(a)). 컴포넌트간의 상속관계는 바람직하지 않으므로, 한 컴포넌트 내의 클래스의 조상 클래스가 컴포넌트 외부에 있다면 조상 클래스들은 컴포넌트 내부로 복제된다(그림 3(a)). 상속 관계를 가진 클래스들의 그룹 내의 추상 클래스들은 향후 다른 클래스의 부모 클래스가 될 가능성이 있으므로 컴포넌트 내로 복제되고 구체 클래스들은 컴포넌트 내부로 이동된다.

본 프로세스는 아래와 같이 수행된다.

단계 1에서 생성된 모든 합성 기본 컴포넌트와 다른 클래스들에 대해

i) 컴포넌트 *comp_a* 내의 클래스 *b*가 컴포넌트 외부에 존재하는 다른 클래스와 상속 관계가 있으면 그 클래스의 모든 조상 클래스가 해당 컴포넌트 내부로 복제된다(그림 3(a)).

ii) 상속 관계를 가지는 구체 클래스 *x*, *y*, *b*에 대해 새 컴포넌트 *comp_b*가 생성된다(그림 3(b)).

그림 3에서 끝이 빈 삼각형인 화살표는 UML 표기법과 같이 상속 관계를 나타낸다. 즉 그림 3(a)에서 *pb*는 *b*의 부모 클래스이다.

• **단계 3.** 불필요한 클래스 삭제

만일 한 클래스가 하나 이상의 컴포넌트로 복제되었다면, 클래스의 중복과 불필요한 다음 단계의 정제 프로세스를 줄이기 위해 삭제된다.

본 단계에서 기술된 프로세스는 각 컴포넌트 내에 포함될 클래스들을 제한하여 컴포넌트들의 복잡도를 제어하고 상속관계를 가진 클래스들을 두 유형으로 분류함으로써 컴포넌트 기반 시스템으로 변환하는 프로세스의 효율을 고려하였다.

4.2 컴포넌트 정제

본 절에서는 연결강도(Connectivity Strength, 이하 CS로 표기), 응집도, 복잡도를 포함한 메트릭을 정의하고 이 메트릭을 이용하여, 전 단계의 결과인 기본 컴포넌트 및 컴포넌트화 되지 않은 클래스들을 클러스터링 함으로써 좀더 강력한 기능적 단위인 컴포넌트를 생성한다. 최종적으로 생성된 컴포넌트들에 대해 각각의 인터페이스가 생성된다.

4.2.1 메트릭

연결강도 CS(p,q)는 요소 p와 q간의 연결 강도를 의미한다. 여기서 대상이 되는 요소 p와 q는 메서드나 클래스 혹은 컴포넌트가 될 수 있다. 응집도 COH(p)는 요소 p에서 내부 요소들간의 관계 긴밀도를 의미하며, p는 컴포넌트나 시스템 전체가 될 수 있다. 이는 각각 클래스들간의 정적인 호출 관계 및 정적 관계를 기반으로 측정된다. 복잡도 COX(p)는 요소 p의 복잡도를 의미하며 p는 컴포넌트나 클래스가 될 수 있다. 클래스 복잡도는 클래스의 멤버 변수의 자료형과 멤버 함수의 인자 및 결과 자료형을 고려하여 측정한다. 컴포넌트 복잡도는 클래스간의 관계 복잡도와, 클래스 구현코드 자체의 복잡도를 기반으로 하여 산출한 것으로, 컴포넌트의 이해도나 적응도에 영향을 미친다. 한 컴포넌트는, 주어진 임계 복잡도를 넘지 않는 범위 내에서 응집력 있게 연결된 다른 클래스나 다른 컴포넌트들을 포함할 수 있다.

정제단계 이후, 컴포넌트들 간의 결합도는 낮아지고 한 컴포넌트 내의 응집도는 높아져서, 결과적으로 이전보다 강력한 기능적 단위 컴포넌트들로 구성된 시스템이 될 것이다. 컴포넌트의 크기는 임계 복잡도로써 조절 가능하며, 컴포넌트 설계자가 결정할 수 있다.

a. CS

컴포넌트 기반 시스템 S'가 컴포넌트 $C_1 ... C_k$로 이루어 졌다고 가정하자($S'=\{C_1 ... C_k\}$ $k \geq 1$). CS는 컴포넌트와 클래스, 메서드들을 포함한 개체들이 얼마나 긴밀하게 연결되어 있는지 나타낸다. 메서드, 클래스, 컴포넌트 간의 CS는 다음 식으로 정의된다. C_i, c_i, m_x는 각각 컴포넌트, 클래스, 메서드를 가리킨다.

$$CS(C_i, C_j) = \sum_{c_u \in CSET(C_i)} \sum_{c_v \in CSET(C_j)} CS(c_u, c_v)$$

$$CS(c_u, c_v) = \sum_{m_x \in MSET(c_u)} \sum_{m_y \in MSET(c_v)} CS(m_x, m_y) + \sum_{m_x \in MSET(c_v)} \sum_{m_y \in MSET(c_u)} CS(m_x, m_y)$$

$$CS(m_x, m_y) = \begin{cases} |primitive\_arg(m_y)| * w_{pri} + (|abstract\_arg(m_y)| + 1) * w_{abs}, & \text{if } m_x \text{ calls } m_y \\ 0, & otherwise \end{cases}$$

여기서

w_{pri}, w_{abs}: 기본형과 사용자 정의형 인자에 관한 각각의 가중치. ($w_{pri} + w_{abs} = 1$, 보통 $w_{abs} > w_{pri}$)

$|primitive\_arg(m_y)|$, $|abstract\_arg(m_y)|$: 각각 메서드 m_y에 대한 기본형과 사용자 정의형의 갯수

b. 응집도

컴포넌트 내의 클래스들은 서비스를 제공하기 위해 서로 긴밀하게 연관되어 있어야 한다. 그들의 관계 긴밀도를 나타내는 응집도를 정량적으로 측정하기 위한 식을 다음과 같이 정의하였다.

$$COH(C_i) = \begin{cases} 1, & \text{if } |C_i| = 1 \\ \dfrac{\sum_{c_u \in CSET(C_i)} \sum_{c_v \in CSET(C_i)} w_R(c_u, c_v)}{|C_i| * (|C_i| - 1)}, & \\ & 1 \leq i \leq k, \ otherwise \end{cases}$$

여기서

$|C_i|$는 컴포넌트 C_i 내부의 클래스 개수.

$$w_R(c_u, c_v) = \begin{cases} w_u, \ w_g, \ w_c \ (w_u < w_g < w_c), (c_u \neq c_v) \\ \quad , \text{if } c_u \text{ has a relationship with } c_v \\ 0, \ otherwise \end{cases}$$

w_u, w_g, w_c는 각각 사용관계, 상속관계, 합성관계에서의 가중치.

윗 식에서는 컴포넌트의 응집도를 측정하기 위해 컴포넌트를 구성하는 클래스들 간의 모든 쌍들이 가지는 관계값의 총합을 이용한다. 클래스들의 관계값은 합성, 상속, 사용관계에 대해 각각 가중치를 준 것인데, 일반적으로 합성, 상속, 사용 관계 순으로 관계의 응집도가 높다.

c. 복잡도

CPC, CSC, CDC, CCC를 포함한 컴포넌트 복잡도 메트릭이 제안되었다[14]. CPC는 클래스, 인터페이스, 클래스와 메서드의 복잡도의 총합으로 각각의 컴포넌트의 복잡도를 측정한다. CSC는 컴포넌트의 내부 구조의 복잡도를 정적인 관점에서 측정한 것이며 CDC는 컴포넌트 내에 발생하는 메시지 전달 횟수를 기반으로 측정하며 컴포넌트의 동적인 면에 초점을 둔다. CCC는 CPC와 유사하지만 컴포넌트의 인터페이스 복잡도가 순환 복잡도(cyclomatic complexity)를 이용하고 있으며, 다른 세 개의 메트릭과는 달리 구현 완료 후에 적용 가능한 메트릭이다.

본 논문에서는 CPC와 CSC에 기반하고, 컴포넌트 구성 클래스들의 합성 멤버 클래스 및 부모 클래스들까지 포함하여 새 컴포넌트 복잡도 메트릭을 정의하였다. 본 복잡도 메트릭은 컴포넌트의 구조적이고 추상적인 특성에 초점을 둔 것으로 다음과 같다.

$$COX(C_i) = \sum_{c_u \in SET(C_i)} COX(c_u)$$

여기서

$$SET(C_i) = CSET(C_i) \cup \sum_{c_j \in CSET(C_i)} GEN(c_j)$$

$$COX(c_i) = \sum_{a \in TYPE(c_i)} \begin{cases} w_{pri}, & \text{if } a \text{ is primitive type} \\ w_{abs}, & \text{if } composition(c_i, a) \text{ is false} \\ COX(a) * w_{abs}, & otherwise \end{cases}$$

$TYPE(c_i) = SIGLIST(MSET(c_i)) \bigcup ATTRSET(c_i)$

4.2.2 클러스터링 과정

본 절에서는 상호 관련된 컴포넌트들을 더 큰 단위의 컴포넌트들로 클러스터링하며, 이때 유사성(similarity) 척도로서 CS와 COH를 이용한다. 두 컴포넌트간에 CS가 클수록 그들이 더 강하게 결합되어 있는 것이다. 클러스터링 시작 전에, 이전 단계에서 나온 기본 컴포넌트에 포함되지 않은 각 클래스들도 하나의 컴포넌트로 간주한다. 즉 클러스터링의 대상이 되는 시스템은 최소 한 개 이상의 클래스들로 구성된 컴포넌트들의 집합이다. 시스템 S'의 응집도와 연결강도 메트릭을 이용하여 중간 단계 컴포넌트 시스템의 품질 $Q(S')$를 정량적으로 측정한다. $Q(S')$는 변경 전 시스템인 S와 변경 후 시스템인 S'에 대한 COH와, 그들 내부 컴포넌트들 간의 CS 값을 이용하여 측정한다. 즉 전체 시스템의 응집도에 해당하는 $COH(S')$는 S'의 컴포넌트들에 대한 각 컴포넌트들 $COH(C)$의 평균을 이용해서 구하며, 전체 시스템의 결합도에 해당하는 $CSS(S')$는 각 시스템의 내부 요소들 간의 연결강도 $CS(C_i, C_j)$들의 합으로 구한다. 클러스터링 프로세스의 목표는 $Q(S')$를 최대화하는 것이다. 현 컴포넌트들 중에서 가장 높은 CS값을 갖는 컴포넌트의 쌍을 결합하여 새 컴포넌트를 생성하되, 새 컴포넌트의 복잡도는 정해진 임계치를 넘을 수 없도록 한다. 컴포넌트의 복잡도가 높아질수록 이해도는 낮아지고 재사용성은 떨어지게 된다. 이 클러스터링 과정은 Q(S')가 더 이상 향상되지 않거나 모든 컴포넌트의 복잡도가 임계치를 넘길 때 종료된다.

$$Q(S') = w_1 * \frac{COH(S')}{COH(S)} + w_2 * \frac{CSS(S)}{CSS(S')}$$

1. 가중치 w_{pri}, w_{abs}, w_c, w_g, w_u, w_1, w_2 와 복잡도 임계치 MAXCOX를 결정한다.
2. S'=S로 두고 $Q(S)$를 계산한다.
3. 이전 단계에서 $Q(S')$값을 향상시키지 않은 쌍을 제외한 모든 $<C_i, C_j>$쌍에 대해, $CS(C_i, C_j)$를 최대로 하는 쌍을 찾는다. 단, $COX(C_i)$와 $COX(C_j)$는 MAXCOX보다 작아야 한다. 만일 모든 $COX(C_i)$ ($1 \le i \le k$) 가 MAXCOX보다 크거나 모든 컴포넌트들에 대해 이 프로세스가 수행되었다면 프로세스를 종료한다.
4. $COX(C_{i\cup j})$가 MAXCOX보다 크다면 단계3으로 간다.($C_{i\cup j}$는 C_i와 C_j를 병합시킨 새 컴포넌트이다.)
5. $COH(S')$와 $CSS(S')$를 이용하여 $Q(S')$를 계산한다. 만일 $Q(S')$가 $Q(S)$보다 크다면 S를 S'로 대체한다. 이때 $COH(S')$, $CSS(S')$, $Q(S')$는 각각 $COH(S)$, $CSS(S)$, $Q(S)$가 된다.
6. 단계3으로 간다.

여기서

w_1, w_2 : 각각 COH와 CSS 의 중요도에 관한 가중치

$$CSS(S') = \sum_{C_i, C_j \in S'} CS(C_i, C_j)$$

$$COH(S') = \frac{\sum_{C \in S'} COH(C)}{|S'|}$$

클러스터링 수행 과정은 다음과 같다.

4.2.3 메트릭의 검증

수학적 개념에 기반 하여 특정 소프트웨어에 국한되지 않는 메트릭 프레임웍이 제안되었다[16]. 이 프레임웍은 크기(size), 길이(length), 복잡도, 응집도, 결합도 메트릭이 가져야 하는 속성을 정의하여, 새로운 메트릭을 정의하기 위한 기준을 제공해 주고 있다.

본 논문에서는 연결강도, 응집도, 복잡도 메트릭을 정의하였다. 여기서, 연결강도는 두 요소들 간의 결합 강도로, 이들의 연결강도의 합이 요소를 포함하고 있는 상위요소의 결합도에 해당한다. 예를 들어, 한 컴포넌트의 결합도는, 해당 컴포넌트를 구성하는 클래스들 간의 연결강도의 합으로 정의 가능하다.

메트릭 검증은 실제 정제 프로세스 수행에 중요한 척도가 되는 시스템의 결합도 및 응집도 메트릭과 컴포넌트 생성시 임계치 역할을 하는 컴포넌트 복잡도에 대해 [16]에서 제안한 속성을 이용하여 수행한다. 단, [16]에서의 모듈과 모듈화 시스템을 본 연구의 컴포넌트와 컴포넌트 기반 시스템으로 가정한다.

a. 연결강도

본 논문에서 제안한 결합도의 타당성을 검증하기 위해 [16]에서 제안한 결합도 성질을 이용하여 평가한다.

[결합도 성질1] 비음수성(Nonnegativity)

$CSS(S)$는 음수가 아니다.

[결합도 성질2] 널 값(Null Value)

S의 내부 컴포넌트들 간에 정보교환이 없으면 $CSS(S)$는 0이다.

[결합도 성질3] 단조성(Mononicity)

S의 내부 컴포넌트들 간에 정보교환이 추가되면 $CSS(S)$는 감소하지 않는다.

[결합도 성질4] 요소 병합(Merging of Elements)

S의 내부의 어느 두 컴포넌트가 병합된 후의 $CSS(S')$은 $CSS(S)$보다 크지 않다.

[결합도 성질5] 서로 소인 요소 병합(Disjoint Element Additivity)

S의 내부의 정보교환이 없는 두 컴포넌트가 병합된 후의 $CSS(S')$은 $CSS(S)$와 같다.

$CSS(S)$가 각 결합도 성질을 만족하는 것을 다음과 같이 보인다.

• $CSS(S)$는 S의 내부 컴포넌트 C_i, C_j 간의 $CS(C_i,$

C_j)의 합이다. CS(C_i, C_j)는 결국 C_i와 C_j의 각 구성 클래스간의 호출관계에 기반한 값이므로, 호출관계가 한번이라도 존재하면 그 값은 늘 0보다 크고, 호출관계가 없다면 0이다. 따라서 CSS(S)는 비음수성을 가진다.

- C_i와 C_j 내부 클래스들 간에 호출관계가 없다면 C_i와 C_j간의 정보 교환이 없는 것으로 CS(C_i, C_j)가 0이다. 만일 모든 C_i, C_j ($\in$ S) 간에 정보교환이 없다면 CSS(S)는 0이다.
- S 내부의 어느 두 컴포넌트 C_i와 C_j간에 정보교환이 추가된다는 것은, C_i의 한 구성클래스와 C_j의 한 구성클래스 간에 호출관계가 추가된다는 것으로 CS(C_i, C_j)의 값은 그만큼 증가한다. 따라서 정보교환이 추가된 CSS(S)는 이전에 비해 감소하지 않는다.
- S 내부의 두 컴포넌트 C_i, C_j가 병합되었다고 하면 CSS(S')의 값은 CSS(S)에서 CS(C_i, C_j)를 뺀 값이 된다. 따라서 CSS(S')은 CSS(S)보다 크지 않다.
- S 내부의 정보교환이 없는 두 컴포넌트 C_i, C_j의 CS(C_i, C_j)는 0이다. 따라서 이들 두 컴포넌트가 병합된 후의 CSS(S)는 CSS(S)에서 CS(C_i, C_j)를 뺀 값이 되므로 결국 CSS(S)와 CSS(S')은 같다.

b. 응집도

본 논문에서 제안한 응집도의 타당성을 검증하기 위해 [16]에서 제안한 응집도 성질을 이용하여 평가한다.

[응집도 성질1] 비음수성과 정규화(Nonnegativity and Normalization)

COH(S)는 최소 0, 최대 1 값을 갖는다.

[응집도 성질2] 널 값(Null Value)

컴포넌트 내부 클래스들 간에 관계가 없으면 COH(S)는 0이다.

[응집도 성질3] 단조성(Monotonicity)

컴포넌트 내부 클래스들 간에 관계가 추가 되면, COH(S)의 값은 감소하지 않는다.

[응집도 성질4] 응집력 있는 컴포넌트(Cohesive Components)

S 내에 상호 관계가 없는 두 컴포넌트가 병합된 COH(S')는 COH(S)보다 크지 않다.

COH(S)는 COH(C)의 평균값이다. COH(S)가 각 응집도 성질을 만족하는 것을 다음과 같이 보인다.

- COH(C)의 정의가 C의 내부 클래스들 간의 관계 가중치의 합으로 정의하였으므로 비음수성을 가지며, 분모는 C의 내부 클래스들의 모든 쌍의 개수이고, 분자는 최소 0에서 최대 내부 클래스 쌍의 개수 사이의 값을 가지므로 COH(C)는 0과 1 사이의 값으로 정규화 된다. 따라서 COH(S) 역시 0과 1 사이의 값으로 정규화 되며, 비음수성을 가진다.
- C의 내부 클래스들 간에 관계가 없으면 COH(C)는 0이다. 그리고 모든 컴포넌트가 이와 같은 성질을 가지면 COH(S)는 0이다.
- C의 내부 클래스들 간에 관계가 추가 되면 해당 관계의 종류에 따라 해당 가중치만큼 더해지므로 COH(C)의 값은 감소하지 않는다. 따라서 COH(C)가 감소하지 않으므로 COH(S)도 감소하지 않는다.
- 상호 관계가 없는 두 컴포넌트 C_1, C_2가 병합된 컴포넌트를 $C_{1\cup2}$라고 할 때, COH($C_{1\cup2}$)의 값은 COH(C_1)와 COH(C_2)의 합보다 작아진다. C_1과 C_2가 병합되면, 두 컴포넌트 내부 클래스들 간에 상호 관련 없는 쌍의 개수가 $|C_1|\times|C_2|$만큼 늘어나므로 결국 COH($C_{1\cup2}$)의 분자는 병합되기 전과 같으나 분모가 더 커지게 되기 때문이다. 따라서 병합된 후의 시스템 응집도 COH(S')는 병합전 시스템의 응집도인 COH(S)보다 크지 않다.

c. 복잡도

본 논문에서 정의한 복잡도는 기존의 컴포넌트 복잡도 메트릭([14])을 기반으로 정의하여 구성 요소 자체의 복잡도와 요소들간의 관계를 모두 고려하였으나 [16]에서는 구성 요소들 간의 관계만을 복잡도의 대상으로 한다. 본 논문에서 정의한 복잡도의 타당성을 검증하기 위해 [16]의 복잡도 성질을 이용한다.

[복잡도 성질1] 비음수성(Nonnegativity)

COX(C)는 음수가 아니다.

[복잡도 성질2] 널값(Null Value)

컴포넌트 C의 내부 클래스 간에 관계가 없으면 COX(C)는 0이다.

[복잡도 성질3] 대칭성(Symmetry)

COX(C)는 C의 내부 클래스 간의 관계(R)를 나타내는 표현기법과 무관하다. 관계의 표현기법은 능동(Active)인 관계(R)와 수동(Passive)인 관계(R^{-1}) 형태로 표현 가능하다. 즉 C=<E, R>, C^{-1}=<E, R^{-1}>이라고 할때 COX(C)는 COX(C^{-1})와 같다. 여기서 E는 C의 구성 클래스, R은 클래스간의 관계이다.

[복잡도 성질4] 단조성(Monotonicity)

컴포넌트 C의 내부 클래스 간에 관계가 추가되면 COX(C)는 감소하지 않는다.

[복잡도 성질5] 서로 소인 요소 병합(Disjoint Element Additivity)

서로 관계가 없는 두 컴포넌트 C_1, C_2가 병합된 후의 COX($C_{1\cup2}$)는 COX(C_1)과 COX(C_2)의 합과 같다.

COX(C)가 각 결합도 성질을 만족하는 것을 다음과 같이 보인다.

- COX(C)의 정의가 C의 내부 클래스들과 그들의 상속, 합성 클래스들의 복잡도의 합으로 정의되었고, 클래스

복잡도는 클래스 내 멤버들의 자료형에 대한 가중치의 합으로 정의되었으므로 COX(*C*)는 0보다 크다. 즉 비음수성을 가진다.

- 컴포넌트 내부 클래스 간에 관계가 없어도 내부 클래스들은 적어도 하나 이상의 멤버를 가지게 되므로 복잡도는 항상 0보다 크다. 따라서 널 값의 성질을 만족하지 않는다.
- COX(*C*)에서 고려하는 관계는 상속과 합성 관계이다. 컴포넌트 *C*의 내부 클래스 *a*의 부모 클래스가 *pa*라고 할때, "*pb*가 *a*에게 상속한다"라는 관계(*R*)가 "*a*가 *pb*에게 상속받는다"관계(R^{-1})로 바뀌는 경우, 복잡도 산정은 재사용 하는 멤버들을 대상으로 하므로 복잡도는 변하지 않는다. 또 *C*의 내부 클래스 *b*가 *c*를 합성 멤버 클래스로 갖는 경우, "*b*가 *c*를 합성 멤버 클래스로 갖는다"라는 관계(R)가 "*c*가 합성 멤버 클래스로서 *a*에 포함된다"라는 관계(R^{-1})로 바뀌는 경우에도, R과 R^{-1}모두 포함되는 합성 멤버 클래스 *c*의 복잡도가 고려되므로 복잡도에 영향을 미치지 못한다. 따라서 대칭성을 만족한다.
- *C*의 내부 클래스 간에 상속 및 합성 관계가 추가되면, 추가된 클래스의 복잡도는 상속받은 멤버와 합성 멤버 클래스를 포함하여 계산되므로 COX(*C*)는 증가하게 된다. 그리고 클래스간의 관계 중 사용관계는 복잡도의 고려대상이 아니므로 사용관계 추가시 COX(*C*)는 변함이 없다. 즉 관계가 추가되면 COX(*C*)는 감소하지 않는다.
- 서로 관계가 없는 두 컴포넌트 C_1, C_2가 병합되었다고 하면 C_{1U2} 의 내부 클래스 간에 관계가 추가되지 않고, C_{1U2} 의 구성 클래스는 C_1과 C_2의 내부 클래스들의 합이 되므로 COX(C_{1U2})는 COX(C_1)과 COX(C_2)의 합과 같다.

복잡도 메트릭의 경우, 복잡도 성질 2를 만족시키지 못했다. 본 논문의 복잡도는 클래스 자체의 복잡도와 그들간의 정적인 관계들로부터 정의한 기존의 컴포넌트 복잡도인 CPC, CSC([14])에 기반하여 정의되었으나, 검증 프레임웍에서는 요소 자체의 복잡도는 고려하지 않고 요소들 간의 관계들만을 복잡도 산정에 포함 시켰기 때문이다. 그러나, 구성 요소들간에 관계가 없다고 하더라도 그들 자체의 복잡도 때문에 전체 복잡도가 0이 되지 않을 수 있다는 관점을 언급하고 있으며([16]), 본 논문의 복잡도는 요소 자체의 복잡도를 고려하지 않는다는 가정을 제외하면 검증 프레임웍의 복잡도 성질을 만족한다.

4.3 인터페이스

하나의 컴포넌트는 클래스들과 인터페이스들로 구성된다. 이전의 단계에서는 한 컴포넌트에 포함되는 클래스들을 식별하는 과정을 기술하였고, 본 절에서는 인터페이스의 생성에 관해 기술한다. 컴포넌트 인터페이스는 컴포넌트가 제공하는 서비스를 명세 한다. 컴포넌트 내의 모든 메서드들 중에, 외부 다른 컴포넌트나 클래스들에게 서비스를 제공해 주는 것은 public 메서드이다. 본 논문에서는 한 컴포넌트 내의 클래스들마다, 해당 public 메서드를 인터페이스화 하였다.

예를 들어, 네 개의 클래스 *a, b, c, d*가 있다고 가정하자. 이들간의 관계 및 이들이 가지고 있는 멤버 함수와 멤버 변수를 포함하여 UML 표기법을 이용한 다이어그램으로 나타내었다 (그림 4). 그림 4에서 '-, +, #'은 멤버에 대한 접근제어자 로서, '-'는 '*private*', '+'는 '*public*', 그리고 '#'은 '*protected*'를 나타낸다. 그리고 각 함수의 인자 및 결과값의 자료형은 편의상 생략하였다.

각 클래스를 3장에서 정의한 대로 다음과 같이 명세 한다.

a = <*ID_a*, {<*int, private*>, <*char, private*> <*b, private*> <*c, private*>}, {<*FID_f1, sig_f1, body_f1, public*>, <*FID_f2, sig_f2, body_f2, public*>}, ∅, {*b, c*}, ∅}>

b = <*ID_b*, {<*int, private*>}, {<*FID_g1, sig_g1, body_g1, public*>}, ∅, ∅, ∅>

c = <*ID_c*, {<*char, private*>}, {<*FID_h1, sig_h1, body_h1, private*>}, {d}, ∅, ∅>

d = <*ID_d*, {<*float, private*>}, {<*FID_i1, sig_i1, body_i1, protected*>, <*FID_i2, sig_i2, body_i2, public*>}, ∅, ∅, ∅}

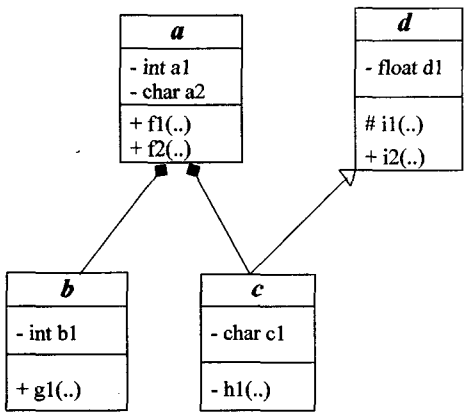


그림 4 예제 클래스 다이어그램

이때, 클래스 *a, b, c*는 합성관계로 하나의 컴포넌트 *comp_a*가 되며(단계1), *d*는 컴포넌트 구성 클래스 *c*의 부모 클래스이므로 *comp_a* 내부로 복제된다(단계2). *comp_a*를 3장에서 정의한 대로 다음과 같이 명세 한다.

comp_a=<*CID_comp_a*, *INTERFACES*, {*a*, *b*, *c*, *d*}, ∅>

INTERFACES={<*FID_f1*, *sig_f1*>, <*FID_f2*, *sig_f2*>, <*FID_g1*, *sig_g1*>, <*FID_i2*, *sig_i2*>}

*INTERFACES*는, *comp_a*의 인터페이스들의 집합으로 *comp_a*의 구성 클래스들의 메서드중 접근 제어자가 *public*인 메서드들의 합집합이다. 실제 인터페이스 구현에 대한 예는 다음 5장에서 보인다.

컴포넌트 설계자가 개발자들이 메트릭에 속하는 가중치나 임계 복잡도를 결정하는 데 도움을 줄 수 있다. 본 4.2절에서 기술된 정제 프로세스를 통해 고품질의 컴포넌트를 생성할 수 있을 것으로 기대되며, 관련된 클래스들을 그룹화 함으로써 비교적 큰 단위의 컴포넌트를 이용하여 시스템에 대한 보다 높은 수준의 뷰를 가질 수 있다. 따라서 소프트웨어 엔지니어들이 시스템을 재사용하고 유지보수 하는 데 도움을 줄 수 있을 것이다.

5. 실험 및 토의

본 장에서는 [15]에 소개된 소스코드를 바탕으로 한 책·비디오테이프 대여 시스템(BMRS)에 대해 4장에서 정의한 변환 프로세스를 적용한 것을 보인다. 이 BMRS 시스템을 수정, 확장하여 C++로 구현하였다. BMRS는 17개의 클래스와 약 1500 라인의 코드로 이루어 졌다. 본 시스템을 시스템에서 제공해 주는 'string' 클래스를 제외하여 UML 다이어그램으로 표현하였다(그림 5).

그림 5에서 일반적인 화살표는 사용 관계를 나타낸다. 즉 그림 5에서 *Rental*에서 *Book*으로 가는 화살표는, 클래스 *Rental*이 클래스 *Book*을 사용한다는 것을 가리킨다. 4.1절에서 기술된 프로세스를 그림 5에서 표현된 시스템에 적용하여 나온 중간 단계의 시스템이 그림 6이다. 그림 6에서는 기본 컴포넌트들로 구성된 BMRS 시스템을 표현하고 있다. 그림 6에서 나타난 점선 화살표는 컴포넌트간의 의존 관계를 나타낸다. 즉 컴포넌트 *C_Customer*는 컴포넌트 *C_Book*과 *C_Movie*가 제공해 주는 서비스를 이용하고 있다.

다음, 4.2절에서 기술된 정제 프로세스를 그림 6에 나타난 기본 컴포넌트에 적용시켰다. w_{pri}와 w_{abs} 가중치는 각각 0.3, 0.7로 주었다. 일반적으로 사용자 정의형이 기본 자료형보다 좀더 복잡하기 때문이다. w_u, w_g, w_c는

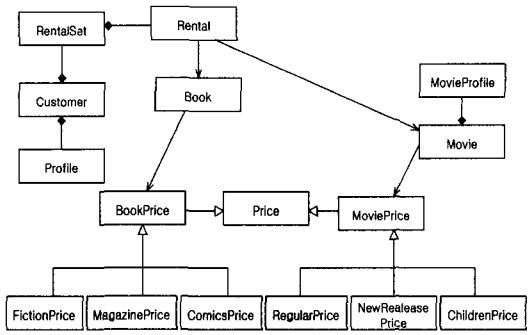


그림 5 BMRS(객체지향)

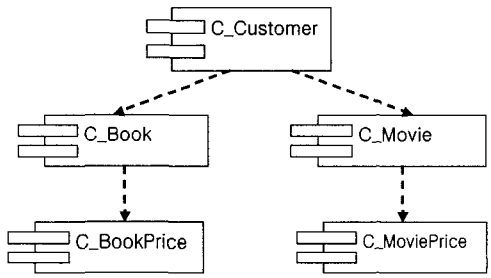


그림 6 BMRS(기본컴포넌트)

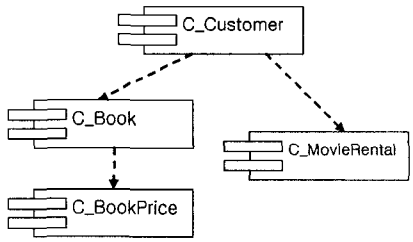


그림 7 BMRS(컴포넌트 기반)

각각 0.25, 0.35, 0.4로, CS와 COH의 가중치는 각각 0.5로 주었으며 MAXCOX는 시스템의 평균 복잡도 값으로 주었다.

표 1은 기본 컴포넌트간의 CS값들을 보여준다. *C_Movie*와 *C_MoviePrice*간의 CS값이 가장 크므로 이들을 클러스터링 하여 *C_MovieRental* 컴포넌트를 새로 생성한다. 그림 7은 BMRS에 적용된 정제 프로세스의 결과를 나타낸다. 표 2에서는 정제프로세스 적용전(그림 6)과 적용후(그림 7)에서의 COH, CS, Q 값을 각

표 1 기본 컴포넌트들 간의 CS 값

컴포넌트	C_Book C_BookPrice	C_Movie C_MoviePrice	C_Customer C_Book	C_Customer C_Movie
CS 값	1.7	2.7	1.0	2.0

표 2 컴포넌트 정제 전 후의 각 메트릭의 값

시스템	COH	CS	Q
컴포넌트 정제전(그림 6)	0.38	7.40	1.00
컴포넌트 정제후(그림 7)	0.36	4.70	1.27

각 보여준다. CS는 36%, Q는 27% 향상되었으나 COH는 약간 감소하였다.

개발자들은 그들이 갖고있는 도메인에 관한 지식과 경험들을 이 프로세스에 적용하여 더 나은 결과를 얻을 수 있다. 예를 들면 컴포넌트 *C_Book*은 오직 한 클래스로만 이루어져 있으므로 그 COH 값은 1이 된다. 이런 경우 *C_Book*이 다른 컴포넌트와 병합될 때 그 COH 값은 늘 감소하게 되므로 다른 컴포넌트와의 병합이 어려워진다. 따라서 개발자들은 그들의 지식과 경험을 통해 *C_BookPrice*와 *C_Book*을 클러스터링 하도록 결정할 수 있을 것이다.

최종 시스템은 다음과 같이 명세 된다.

UCOMP = {*C_MovieRental, C_Customer, C_BookPrice, C_Book*}

C_MovieRental = <*IDC_MovieRental, I_MovieRental*, {*Movie, MovieProfile, ChildrenPrice, NewReleasePrice, RegularPrice, MoviePrice, Price*}, ∅ >

C_Customer = <*IDC_Customer, I_Customer*, {*Rental, RentalSet, Customer, Profile*}, {*C_Book, C_MovieRental*} >

C_BookPrice = <*IDC_BookPrice, I_BookPrice*, {*ComicsPrice, FictionPrice, MagazinePrice, BookPrice, Price*}, ∅ >

C_Book = <*IDC_Book, I_Book*, {*Book*}, {*C_BookPrice*} >

I_Book, I_Customer, I_BookPrice, I_MovieRental 은 각 컴포넌트의 인터페이스이다. 각 컴포넌트들이 제공해주는 서비스를 외부에 원활히 제공하도록 하기 위해 각 컴포넌트들은 그들의 컴포넌트 클래스를 가진다. 컴포넌트 클래스는 일종의 래퍼(wrapper)로서, 구 객체지향 언어로 생성된 컴포넌트를 실제 구현하도록 하기 위한 개체 클래스이다. 외부 호출자로부터 인터페이스에 대한 요청을 받으면, 호출된 컴포넌트 내의 컴포넌트 클래스는 그 요청을 받아 적절한 클래스의 메서드로 호출을 넘기고 필요시 결과를 호출자에게 넘겨준다.

인터페이스와 컴포넌트, 그리고 다른 클래스들을 이용하여 이 시스템의 수행을 보여주기 위해 본 논문에서는 컴포넌트 *C_BookPrice*가 어떻게 사용되는지 C++로 간단히 구현하였다. 다음 코드가 *C_BookPrice* 컴포넌트에 삽입된다.

```
#define CLASS_COMICS        Class.ComicsPrice
#define CLASS_FICTION       Class.FictionPrice
#define CLASS_MAGAZINE      Class.MagazinePrice
// This component-class CBookPrice is included in component C_BookPrice.
// ClassID is defined as string type and is a unique identifier among all classes.

class CBookPrice : public I_BookPrice {
private:
 BookPrice* price;
public:
 CBookPrice(ClassID id) {
   if( !strcmp(CLASS_COMICS, id) ) price = new ComicsPrice;
   if( !strcmp(CLASS_FICTION, id) ) price = new FictionPrice;
   if( !strcmp(CLASS_MAGAZINE, id) ) price = new MagazinePrice;
 }
 double getCharge(int daysRented) {
   return price->getCharge(daysRented);
 }
};

class I_BookPrice {
 virtual double getCharge(int daysRented) = 0;
};
```

아래 코드는 다른 클래스나 컴포넌트가 C_BookPrice를 사용하는 방식을 보여준다.

```
// Using component C_BookPrice for ComicsPrice.
CBookPrice *comp = new CBookPrice(CLASS_COMICS);
// bp points to the interface of CBookPrice and used like this form, bp->getCharge(arg).
IBookPrice *bp = comp;
```

생성자 메서드를 호출하여 컴포넌트클래스 *CBookPrice*의 인스턴스를 생성한다. 실제 *ComicsPrice* 클래스가 사용된다고 가정하면, *ComicsPrice*의 식별자인 'CLASS_COMICS'가 생성자의 인자로 삽입된다. *C_BookPrice*의 인터페이스인 *I_BookPrice*는 추상클래스로 사용된다. 만일 컴포넌트 내에 추상클래스가 없다면, 실제 사용될 클래스를 결정하는 *if* 문은 생략될 것이다.

BMRS 예에서 보듯 본 프로세스는 클래스를 대상으로 하여 수행되므로, 일반적인 객체지향 시스템에 대한 적용성이 있으며, 그 과정이 비교적 직접적으로 적용 가능하다. 객체지향 시스템을 재공학 하여, 재사용성 있는 컴포넌트를 생성하는 것에 초점을 두고 있으므로, 본 프

로세스를 적용시킨 컴포넌트 기반 시스템은 변환전의 객체지향 시스템에 비해 재사용하기 적절한 규모인 기능적 단위로 사용 가능한 컴포넌트들로 구성된다.

여타 컴포넌트 식별 방법들이 대부분 클래스 다이어그램이나 시퀀스 다이어그램 등과 같은 분석이나 설계시의 모델을 이용하고 있는데 비해([4,5,10]), 본 프로세스에서는 객체 지향 코드에서 정적으로 도출 될 수 있는 정보들을 바탕하고 있으며, [11]과 같은 도메인에 대한 전문적인 지식을 요구하지 않는다. 또한 비교적 정형적인 명세 모델을 제시하여 시스템 이해나 프로세스 수행에 정확성을 부여하였고, 컴포넌트의 품질 측정을 위한 메트릭을 제안하였다.

6. 결론 및 향후과제

본 논문에서는 기존의 객체지향 레거시 시스템을, 특정 도메인에 적합한 기능적 단위인 컴포넌트로 변환하는 방법론을 제안하였다. 기본 컴포넌트는 프로그램 소스 코드 상에서 나타난 클래스들간의 상속 및 합성 관계를 이용하여 생성하였다. 컴포넌트의 품질을 측정하고 기본 컴포넌트를 정제하기 위해 정적 관계를 기반으로 한 컴포넌트 메트릭을 정의하였으며, 이 메트릭을 클러스터링 프로세스에 적용시켜 보다 고품질의 컴포넌트 기반 시스템으로 변환 될 수 있도록 하였다. 본 프로세스에서는 객체지향 시스템에서의 단위인 클래스들간의 관계를 이용하였으므로 대부분의 객체지향 시스템에 대해 포괄적으로 적용될 수 있을 것이다. 그리고 원칙적으로 컴포넌트간의 의존관계만을 허용하였으므로 각 컴포넌트들이 특정 도메인에 한정적인 기능적 단위가 될 수 있도록 하였다.

본 논문에서는 메서드간의 호출 관계는 고려했으나 호출 횟수와 같은 시스템의 동적인 성질은 다루고 있지 않다. 이러한 동적인 측정치는 분산 환경에서의 컴포넌트 할당에서 유용할 것이다.

현재 프로세스의 부분적 자동화를 위한 도구 생성 중이며, 향후 연구는 컴포넌트 정제 단계에 초점을 맞추고 있다. 또한 적절한 휴리스틱 알고리즘을 사용하여 보다 대규모의 시스템에 본 방법론을 적용시켜 효율성을 입증 할 것이다.

참 고 문 헌

[1] I. Sommerville, Software Engineering, 6th Ed., Addison Wesley, Harlow, England, 2001.

[2] F. Huber, A. Rausch, and B. Rumpe, "Modeling Dynamic Component Interfaces," In Proceedings of Technology of Object-Oriented Languages, pp.58-70, 1998.

[3] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig, "Putting the Parts Together-Concepts, Description Techniques, and Development Process for Componentware," In proceedings of the Thirty-Third Annual Hawaii International Conference on System, Vol 8, 2000.

[4] H. Jain, "Business Component Identification - A Formal Approach," In Proceedings of Fifth IEEE International Enterprise Distributed Object Computing Conference, pp 183 -187, 2001.

[5] E. Holz, and O. Kath, "Manufacturing Software Components from Object-Oriented Design Models," In Proceedings of Fifth IEEE International Enterprise Distributed Object Computing Conference, pp. 262-272, 2001.

[6] E. Stroulia and T. Systa, "Dynamic Analysis for Reverse Engineering and Program Understanding," Applied Computing Review, ACM Press, vol 10, issue 1, pp. 8-17, 2002.

[7] M.A. Serrano, C.M. de Oca and D. L. Carver, "Evolutionary Migration of Legacy Systems to an Object-Based Distributed Environment," In Proceedings of the IEEE International Conference on Software Maintenance, pp. 86-95, 1999.

[8] A. De Lucia, G. A. Di Lucca, A. R. Fasolino, P. Guerram and S. Petruzzelli, "Migrating Legacy Systems towards Object Oriented Platforms," In Proceedings of the IEEE International Conference on Software Maintenance, pp. 122-129, 1997.

[9] H. M. Sneed, "Generation of stateless components from procedural programs for reuse in a distributed system," In Proceedings of the Fourth European Software Maintenance and Reengineering, pp. 183-188, 2000.

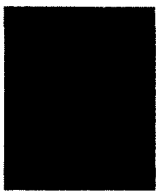
[10] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang and D. H. Ham, "Component Identification Method with Coupling and Cohesion," In Proceedings of Asia-Pacific Software Engineering Conference, pp. 79-86, 2001.

[11] M. W. Price, D. M. Needham and S.A. Demurjian, "Producing Reusable Object-Oriented Components:A Domain-and-Organization-Specific Perspective," In Proceedings of Symposium on Software Reusability, pp. 41-50, 2001.

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

[13] D. F. Bacon and P. F. Sweeney, "Fast static Analysis of C++ Virtual Function Calls," In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, San Jose, California, October pp. 324-341, 1996.

[14] E. S. Cho, M. S. Kim, and S. D. Kim, "Component metrics to measure component quality," In Proceedings of Asia-Pacific Software Engineering Conference, pp. 419-426, 2001.
[15] M. Fowler, Refactoring : Improving the Design of Existing Code, 4th Ed., Addison Wesley, Harlow, England, 1999.
[16] L. C. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," IEEE Transaction on Software Engineering, Vol. 22, No. 1, pp. 68-86, 1996



이 은 주
1997년 서울대학교 계산통계학과(학사) 1999년 서울대학교 전산과학과(석사). 1999년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 소프트웨어 재공학, 컴포넌트개발방법론, 소프트웨어 매트릭 등

신 우 창
1993년 서울대학교 계산통계학과(학사). 1995년 서울대학교 전산과학과(석사). 2003년 서울대학교 전기컴퓨터공학부(박사). 2003년~현재 서경대학교 인터넷정보학과 전임강사. 관심분야는 디자인패턴, 통합프로그래밍환경, 테스팅, 컴포넌트개발방법론 등임

이 병 정
1990년 서울대학교 계산통계학과(학사) 1998년 서울대학교 전산과학과(석사). 2002년 서울대학교 전기컴퓨터공학부(박사) 1990년 1월~1998년 1월 현대전자 소프트웨어연구소 주임연구원. 2002년 3월~현재 서울시립대학교 컴퓨터과학부 조교수. 관심분야는 소프트웨어 재공학, 소프트웨어 품질 평가, 소프트웨어 개발방법론 등

우 치 수
1972년 서울대학교 응용수학과(공학사) 1972년~1974년 한국과학기술원 연구원 1977년 서울대학교 대학원 전산학(석사) 1982년 서울대학교 대학원 전산학(박사) 1978년 영국 라퍼러대학 연구원. 1975년~1982년 울산대학교 전자계산학과 조교수, 부교수. 1985년~1986년 미국 미시간대학교 Post-doc 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 소프트웨어 공학, 프로그래밍 언어 등