컴포넌트 프레임웍의 실용적 참조 모델
(A Practical Reference Model of Component Frameworks)

허 진 선 † 김 수 동 ††
(Jin Sun Her) (Soo Dong Kim)

요 약 컴포넌트 기반 소프트웨어 공학(CBSE)은 재사용 가능한 소프트웨어 부품을 이용하여 어플리케이션을 개발하는 새로운 패러다임이다. 그러나 소프트웨어 부품을 조립하는 과정에 부품들 간에 불일치 문제가 발생하여 CBSE를 수행함에 있어 부담이 되고 있다. 따라서, 컴포넌트의 재사용 단위보다는 큰 프레임웍(Framework)이 요구되고 있으며 프레임웍은 관련 있는 컴포넌트와 커넥터들 그리고 도메인에 특화된 아키텍처로 구성된다. 프레임웍은 컴포넌트보다 많은 장점을 가지고 있지만, 프레임웍을 구성하는 핵심 요소들과 내부 구조는 명확히 정의되지 않고 있다. 즉, 현재까지 제시된 대부분의 프레임웍 모델들은 실용적이지 못한 개념적인 수준에 머물러 있다. 본 논문에서는 실용적인 수준의 프레임웍 참조 모델을 제안하고 프레임웍의 핵심 요소들을 정의한다. 프레임웍의 구체적인 구성요소들, 컴포넌트와 프레임웍의 상호 관련성, 커넥터의 확장된 개념들, 그리고 프레임웍 내에서의 특화된 아키텍처의 의미에 대해서 명확히 식별한다. 본 논문에서 제안된 프레임웍은 Java, J2EE, CORBA Component Model (CCM) 그리고 .NET과 같은 객체지향 기반 미들웨어에서 실용적으로 구현될 수 있을 것이다.

키워드 : 컴포넌트 프레임웍, 범용 아키텍처, 인터페이스, 컴포넌트, 커넥터, CBSE

Abstract Component-Based Software Engineering (CBSE) is widely accepted as new paradigms for building applications with reusable assets. Mismatch problems occur while integrating the assets and make a burden in CBSE. Therefore, a larger-grained reuse unit than a component such as a framework is required, and it consists of relevant components, connectors and domain-specific architecture. The benefits of frameworks over components are commonly known, however, the key elements and internal structure of framework has not been clearly defined. We believe most of the framework models proposed in recent works remain at conceptual level. In this paper, we propose a practical-level framework reference model, and define key elements of frameworks. The research goal is to precisely identify concrete elements of a framework, inter-connection of components within a framework, extended notion of connectors, and tailored meaning of architecture in a framework. Hence, the proposed framework can be practically implemented in common object-oriented languages and with middleware such as Java, J2EE, CORBA Component Model (CCM) and .NET.

Key words : Component framework, generic architecture, interface, component, connector, CBSE

1. 서 론

CBSE는 재사용 가능한 컴포넌트를 이용하여 소프트웨어 어플리케이션을 개발하기 위한 패러다임이다[1]. CBSE를 통해 컴포넌트를 재사용하는데 있어 어려운 점은 컴포넌트 사용자가 적합한 컴포넌트의 집합을 선택하여 연관 관계와 의존 관계를 이용하여 상호운영이 잘 되게 끔 구성해야 한다는 것이다. 이것은 컴포넌트 사용자에서 상당한 부담을 안겨주며 비성공적인 컴포넌트 선택을 야기시키기도 한다. 또한, 사용자가 필요로 하는 기능성을 적합하게 제공해주는 컴포넌트를 재대로 찾기 힘들다. 종종 Commercial Off-The-Shelf(COTS) 컴포넌트는 요구하는 기능의 일부분만을 제공하거나 필요로 하지 않는 기능을 제공하게 된다. 따라서, 신중히 고려하여 선택한 컴포넌트의 집합이라 할지라도 서로 간에 상호운영이 성공적으로 이루어지기는 힘들다[2]. 이러한 불일치의 문제는 CBSE를 실용적으로 적용하는데 있어 한계점을 가져다 준다.

이러한 문제점들은 Product Line Engineering(PLE)

· 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음
† 학생회원 : 숭실대학교 컴퓨터학과
jsher@otlab.ssu.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@ssu.ac.kr
논문접수 : 2003년 7월 21일
심사완료 : 2006년 4월 5일

의 프레임웍(Framework)을 통해 해결할 수 있다[3]. 프레임웍은 반 구현된 어플리케이션으로서, 프로덕트 멤버들 사이에서의 공통적인 기능을 제공하게 되며 멤버들 사이의 가변적인 부분을 설정하기 위한 미완성된 부분인 핫 스팟(Hot Spot)을 갖는다. 이 핫 스팟은 프레임웍을 이용하여 특정 어플리케이션을 만드는 어플리케이션 공학 과정에서 채워지게 된다. 그러나, 최근까지 발표된 프레임웍 모델들은 개념적인 수준에 머물러 있다. 프레임웍 자체가 어떻게 구성되어 있는지, 그리고 각각의 구성 요소가 가진 의미는 무엇이며 어떻게 구현되는지에 대한 정의가 추상적인 수준에 그치고 있다. 또한, 컴포넌트 간의 불일치 문제를 해결하기 위한 장치에 대한 제안이 부족한 실정이다. 이로 인해, OO 프로그래밍 언어와 미들웨어를 통해 프레임웍을 구현하는 방법과 프레임웍으로부터 어플리케이션을 인스턴스화(Instantiate)하는 방법이 명확하지 않아 구현 가능한 실용적인 프레임웍 모델에 대한 제시가 우선적으로 요구되고 있다. 따라서, 본 논문에서는, 실용적인 수준의 프레임웍 참조 모델을 제안하고 프레임웍의 주요 요소들을 정의한다. 본 논문의 목표는 프레임웍의 구체적인 구성 요소들, 프레임웍 내의 컴포넌트 간의 관계, 커넥터의 확장된 의미, 그리고 프레임웍 내에서 범용 아키텍처가 가지는 역할을 명확히 식별하는 것이다. 따라서, 제안된 프레임웍은 Java, J2EE, CCM, 그리고 .NET과 같은 객체지향 언어와 미들웨어를 통해 실용적으로 구현될 수 있다.

본 논문의 2장에서는 최근까지의 프레임웍 모델에 대한 연구 내용을 살펴보고, 3장에서는 본 논문에서 제안하는 컴포넌트 프레임웍에 대한 참조 모델을 제시하고 4장에서는 각각의 구성 요소들에 대해 상세히 기술한다. 5장에서는 제안된 프레임웍 참조 모델을 기반으로 한 사례 연구를 수행하고 6장에서는 각 구성요소들을 J2EE 환경에서 어떠한 구현 장치를 이용해서 구현할 수 있는지에 대한 적용 지침을 제시하며 7장에서는 제안된 프레임웍 모델에 대한 평가를 수행한다.

2. 관련 연구

Fayad의 연구: OO 프레임웍은 상속이라는 구현 장치를 이용하여 프레임웍 내에 정의된 여러 가지 클래스들을 특수화(Specialize)함으로써 어플리케이션을 구축할 수 있는 클래스의 집합이다[4]. OO 프레임웍에서 사용되는 상속이라는 구현 장치가 파라미터화 메커니즘으로써 여러 가지 장점을 제공하지만 프레임웍 내의 클래스를 특수화하여 만든 클래스들은 프레임웍에 상당 부분 의존성을 가지게 되어 다른 환경에서는 사용될 수 없다는 문제점 또한 가지게 된다. 이러한 상속을 통해 특수화된 코드와 제한된 재사용 단위의 문제점은 컴포넌트 프레임웍을 통해 해결할 수 있다. 컴포넌트 프레임웍은 OO 프레임웍과 비슷하지만 특수화된 코드와 일반적인(Generic) 코드라는 차이점이 있다. 컴포넌트 프레임웍은 또한 다양한 형태의 재구성으로 더욱 큰 단위의 재사용성을 제공할 수 있다. 즉, OO 프레임웍에서 고려하고 있는 재사용의 단위는 컴포넌트 프레임웍과는 상당한 차이가 있으며 구성요소가 정확히 무엇인지, 그리고 구성요소 간의 불일치의 문제를 해결하기 위한 장치의 제시가 부족하다. 따라서, 재사용 단위가 큰 컴포넌트 프레임웍이 무엇인지, 무엇으로 구성되어 있는지 등의 기반을 세우는 것이 중요하다.

Atkinson의 연구: KobrA 컴포넌트는 KobrA의 접근 방법에 따라 문서화된 논리적인 컴포넌트를 의미하는데, Composition, Clientship, Ownership, Containments와 같은 컴포넌트 조합을 위한 메커니즘을 가지고 있다[3]. 이 모델에서의 컴포넌트의 의미는 타입과 런타임 요소가 아닌 명세 수준으로 제한하고 있다. 또한, KobrA 컴포넌트는 특정 요소들을 담기 위한 컨테이너로서 작용하는 모듈과 비슷한 성격을 가진다. 이와 같이 KobrA 모델은 컴포넌트의 의미를 제한하고 있으며 컴포넌트 프레임웍의 구체적인 구성 요소에 대한 제시가 부족하다.

Latchem의 연구: Latchem은 컴포넌트 인프라스트럭처 모델을 제시하고 있는데, 컴포넌트들 사이에 불필요한 결합도가 생기지 않도록 컴포넌트들의 역할이 설계되어야 함을 강조한 모델이다[2]. Latchem의 연구에서의 인프라스트럭처는 본 논문에서 소개하고 있는 컴포넌트 프레임웍과 맥락을 같이 한다. Latchem이 제시한 인프라스트럭처 모델은 컴포넌트 스테레오타입이라는 형식으로 컴포넌트의 기능적인 역할을 분류하여 정의한다. 그러나, 컴포넌트 프레임웍에 대한 종합적이고 체계적인 메타 모델에 대한 명시적인 제시가 명확하지 않고 구성 요소의 역할이나 의미에 대한 제시 또한 충분치 않다.

Wills의 연구: Wills는 컴포넌트를 통합하여 컴포넌트 인프라스트럭처를 만드는데 있어 커넥터의 사용을 강조한다[5]. Wills의 연구에서의 컴포넌트 인프라스트럭처는 본 논문에서 소개하고 있는 컴포넌트 프레임웍과 맥락을 같이 한다. 컴포넌트와 커넥터를 잘 재구성하면 여러 형태의 컴포넌트 인프라스트럭처를 만들 수 있어 컴포넌트의 재사용성을 높일 수 있기 때문에 커넥터에 대한 조심스러운 설계를 강조한다. Wills의 연구에서는 커넥터의 정의와 타입이 존재함을 암시한다. 그러나, 가정하고 있는 컴포넌트 인프라스트럭처에 대한 총체적인 참조 모델이나 커넥터의 구성 요소 및 타입에 대한 명확한 언급은 하고 있지 않다.

3. 컴포넌트 프레임웍 참조 모델

본 장에서는 컴포넌트 프레임웍에 대한 참조 모델을 제안한다. 컴포넌트 프레임웍은 범용 아키텍처, 인터페이스, 컴포넌트, 컴포넌트 사이의 관계, 커넥터로 구성된다. 그림 1은 이러한 컴포넌트 프레임웍의 추상적 개념도를 보여주고 있다. 그림 1에서 범용 아키텍처는 컴포넌트 프레임웍에 투영되어 있으며 범용 아키텍처가 어떻게 구성되는가에 따라 컴포넌트의 종류가 나뉠 수 있다. 그림에서는 시스템(System) 컴포넌트와 비즈니스(Business) 컴포넌트로 나뉘어 있는 예를 보여주고 있다[6]. 그리고, 본 모델에서는 각각의 컴포넌트마다 세 종류의 인터페이스를 가지고 있을 것을 제안한다. 또한, 프레임웍 내에는 컴포넌트들 사이의 관계로 의존 관계와 연관 관계가 있으며 컴포넌트들 사이의 불일치 문제를 해결하기 위한 커넥터가 정의되어야 함을 정의한다. 아래 그림의 좌측 커넥터들은 커넥터가 같은 역할을 수행하는 다른 커넥터들로 대체 가능함을 나타낸다. 일반

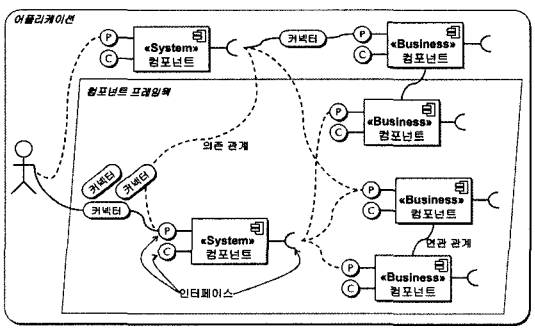

그림 1 컴포넌트 프레임웍 참조 모델

적으로 제품 계열 아키텍처에서는 범용 아키텍처 및 컴포넌트를 대표적인 구성요소로 고려한다. 이를 기반으로 본 논문에서는 상세한 인터페이스의 종류 및 커넥터, 컴포넌트 사이의 관계 등 이러한 모든 구성요소에 대한 총체적인 모델을 제시한다.

그림 2는 컴포넌트 프레임웍에 대한 메타 모델을 보여준다. 컴포넌트 프레임웍은 하나의 범용 아키텍처를 가지지만 하나의 범용 아키텍처는 복수개의 컴포넌트 프레임웍에 포함될 수 있다. 이는 기능적 요구사항이 다를지라도 비기능적 요구사항이 같을 경우 범용 아키텍처가 같기 때문이다[7]. 범용 아키텍처는 아키텍처 뷰타입(ViewType)의 관점에서 설계되며 하나의 뷰타입에는 여러 개의 스타일(Style)이 존재한다[8]. 프레임웍은 여러 개의 인터페이스와 인터페이스에 명시된 기능성을 준수하는 컴포넌트들로 구성된다. 커넥터는 두 개 이상의 인터페이스들 사이에 존재하게 되며 프레임웍에서는 이러한 커넥터를 여러 개 포함할 수 있다. 프레임웍 내에는 기본적으로 두 개 이상의 컴포넌트가 존재하고 컴포넌트들 사이에서 워크플로우를 제공하기 때문에 여러 개의 의존 관계를 가지게 된다. 컴포넌트들 사이의 연관 관계는 의존관계보다 적지만 프레임웍 내에는 연관 관계가 0개 이상 존재할 수 있다.

본 논문에서 제안하고 있는 프레임웍은 가변성 개념을 지원한다. 가변성은 패밀리 멤버들 사이에서의 차이점을 의미하며 어떠한 한 부분을 멤버들이 특정 환경에 맞게 자유롭게 바꾸거나 커스터마이즈할 수 있는 능력을 제공한다. 가변성은 PLE에 재사용성과 적용성을 높여주는 중요한 메커니즘이다. 그림 3은 프레임웍 가변성에 대한 메타 모델을 설명하고 있다. 가변성은 여러 개

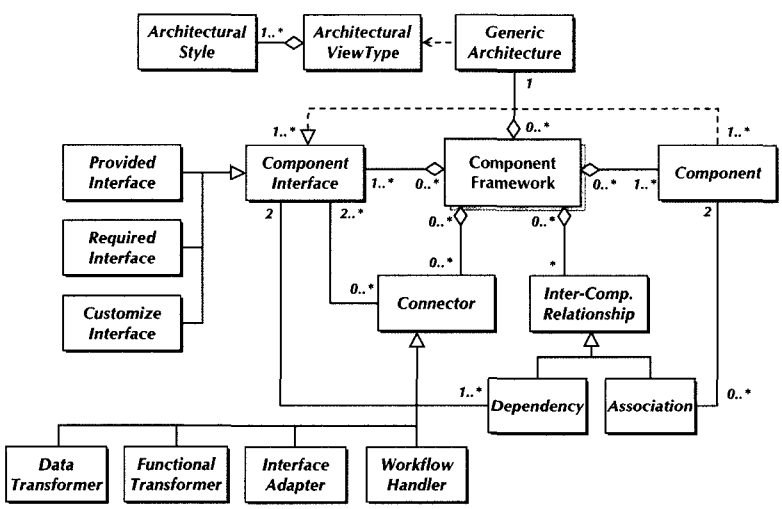

그림 2 컴포넌트 프레임웍 메타 모델

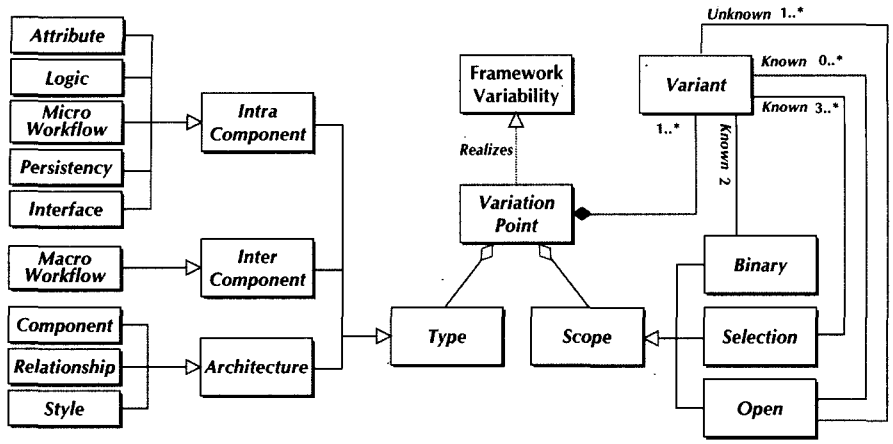

그림 3 프레임웍 가변성에 대한 메타 모델

의 가변점(Variation Point)에 의해 구현된다. 가변점은 변화가 발생될 지점을 의미하며 커스터마이즈된 프레임웍을 생성하기 위한 가변성 메커니즘이 적용될, 명시적으로 설계된 지점이다. 가변치(Variant)는 가변점에 대한 가능한 솔루션을 의미한다. 일반적으로 하나의 가변점에는 여러 개의 가변치가 존재하게 된다[9].

가변점의 종류에 따라 가변성의 타입이 결정되는데, 본 프레임웍에서는 가변성의 종류를 크게 컴포넌트 내부(Intra-Component), 컴포넌트 외부(Inter-Component), 아키텍처(Architecture) 수준으로 분류한다. 컴포넌트 내부 수준의 가변성으로는 속성, 논리, 워크플로우, 영구[10], 인터페이스[11] 가변성이 있으며, 컴포넌트 외부 수준의 가변성으로는 컴포넌트들 간의 워크플로우 상에서 발생하는 매크로 워크플로우 가변성이 있으며 아키텍처 수준의 가변성으로는 컴포넌트, 컴포넌트간의 관계, 스타일이 있다. 이와 같이, 프레임웍에서 가변성의 타입을 분류하여 제안하는 것은 각각의 타입에 따라 설계 기법이 달라지기 때문이다. 그러나 본 논문의 목적은 프레임웍의 개념 및 구성요소를 제안하는데 있으므로 상세한 설계 기법에 관한 내용은 본 논문의 범위에 해당하지 않는다.

4. 구성 요소 별 상세 정의

4.1 범용 아키텍처

범용 아키텍처는 여러 어플리케이션에 공통적으로 사용될 핵심 부분을 정의한 아키텍처이다. 반면, 어플리케이션 아키텍처는 특정한 하나의 어플리케이션에만 적용되는 아키텍처를 의미한다. 아키텍처는 비기능적 요구사항을 설계하기 위해 프레임웍 요구사항 가운데 공통된 비기능적 요구사항은 범용 아키텍처를 통해 설계 가능하다. 범용 아키텍처는 대상 컴포넌트 프레임웍의 기본 골격이 되며 프레임웍이 사용될 대상 어플리케이션에 그대로 적용되기 때문에 프레임웍에 설계된 비기능적 요구사항이 어플리케이션에 반영된다. 이러한 프레임웍에서 정의한 아키텍처는 특정 어플리케이션에 적용할 때, 인스턴스화하여 특정 어플리케이션에 맞게 특화하여 사용한다.

소프트웨어 아키텍처는 요소(Element), 요소 사이의 관계, 그리고 외부에서 관찰 가능한 특성(Property)으로 구성된다[8]. 그러나, 컴포넌트 프레임웍의 구성요소로서의 범용 아키텍처의 구성 요소는 컴포넌트와 컴포넌트 간의 관계가 된다. 범용 아키텍처 설계 시에는 [8]에서 제안한 뷰타입과 스타일을 적용할 수 있으며 그 중 특별히 Module과 Component-and-Connector(C&C) 뷰타입을 통해 아키텍처의 정적인 측면과 동적인 측면을 설계할 수 있다. Allocation 뷰타입은 하나의 특정 어플리케이션 종속적인 측면을 고려하기 때문에 범용 아키텍처 설계에 적용하기는 적합하지 않다.

범용 아키텍처를 특정 어플리케이션에 맞게 인스턴스화하기 위해서는 아키텍처가 가변성을 수용할 수 있어야 한다. 범용 아키텍처는 컴포넌트와 컴포넌트 간의 관계로 구성되기 때문에 각 구성 요소 상에 가변성이 발생할 수 있다. 즉, 범용 아키텍처의 가변점은 컴포넌트와 컴포넌트 간의 관계가 되며 컴포넌트 상에 가변성이 발생할 경우에는 대부분 컴포넌트 간의 관계에 가변성을 발생시킬 것이다. 또한 어플리케이션 마다 중요시 하는 비기능적인 요구사항에 따라 스타일이 달라질 수 있다[8]. 스타일이 변경될 경우에는 컴포넌트 및 컴포넌트 간의 관계에 변경이 생기며 동일한 스타일이라 할지라도 멤버마다의 요구사항에 따라 컴포넌트 및 컴포넌트 간의 관계에 변화가 생길 수 있다.

4.2 인터페이스

인터페이스에는 컴포넌트가 제공해야 하는 기능성이 명시되어 있다. 각각의 인터페이스에는 클라이언트에 의해 호출될 수 있는 하나 또는 그 이상의 서비스를 포함하게 된다. 각각의 인터페이스는 공통 목적을 제공하는 연관된 서비스들로 묶여진다[12]. 인터페이스는 시그너처와 시맨틱(Semantics)의 형태로 제공되는 함수들로 구성된다. 함수의 시맨틱에는 선조건(Precondition), 후조건(Postcondition), 불변조건(Invariants), 그리고 미치는 영향으로 표현된다.

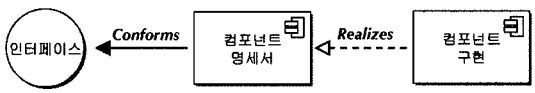

그림 4 인터페이스 vs. 컴포넌트 명세 vs. 컴포넌트 구현

본 논문의 참조 모델은 그림 4와 같이 인터페이스, 컴포넌트 명세서(Component Specification) 그리고 컴포넌트 구현(Component Implementation)에 대한 분리를 강조한다. 인터페이스는 컴포넌트에 의해 제공되는 외부에 드러나는 기능성을 나타내며 이는 클라이언트와 컴포넌트 사이의 규약으로 작용한다. 따라서, 인터페이스에는 클라이언트가 알아야 하는 모든 기능성을 정의하고 있어야 한다. 컴포넌트 명세서는 컴포넌트가 무엇을 하는지에 대한 명세이기 때문에 인터페이스에 정의된 모든 기능성을 준수해야 한다. 컴포넌트 명세서에는 내부에 구현되어야 하는 클래스, 지원하는 인터페이스, 구현하는데 있어서의 제약 사항 등이 기술되어 있다. 컴포넌트 구현물은 컴포넌트 명세서를 준수하여 구현한 런타임 요소의 컴포넌트이다. 이렇게 분리를 함으로써 컴포넌트를 조합할 때, 컴포넌트 사이의 결합도를 낮추고 의존성 관리를 용이하게 함으로써 컴포넌트 대체나 변화가 일어나도 클라이언트에게 미치는 영향을 줄일 수 있게 된다. 인터페이스, 명세서, 구현의 분리를 통해 클라이언트 코드에 미치는 영향을 최소화하면서 컴포넌트 명세서와 컴포넌트 구현의 유지보수성을 향상시킨다.

SEI[13], Fayad[4], Latchem[2], Atkinson[3]의 연구에서처럼 인터페이스는 컴포넌트 프레임웍의 기본적인 구성 요소가 된다. 그러나, 본 모델에서는 인터페이스를 그림 5와 같이 Provided, Required, Customize의 3가지 타입으로 나누어 제안한다.

Provided 인터페이스: Provided 인터페이스는 함수의 형태로 컴포넌트에 의해 제공되는 응집력 높은 서비스의 집합을 정의한다. 컴포넌트 프레임웍 수준의 워크플로우는 Provided 인터페이스의 함수들을 호출함으로써 실행되며 이러한 함수들은 실행 시간에 대체로 빈번히 호출된다. Provided 인터페이스 상에 가변성이 발생할 수 있으며, 그 경우는 다음과 같이 분류할 수 있다. 첫째, 클라이언트가 레가시 코드이어서 기존의 호출 코드를 수정하기 힘들 경우, 둘째, 클라이언트 마다 사용하는 네이밍 규칙(Convention)이 달라서 같은 기능에 대해 다른 오퍼레이션 시그너처를 요구할 경우이다. 이러한 인터페이스상의 가변성은 Customize 인터페이스나 4.5 절에서 소개할 커넥터를 사용하여 설정할 수 있다.

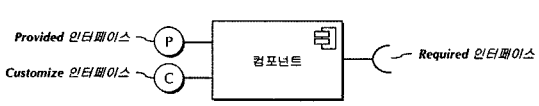

그림 5 3 가지 타입의 인터페이스

Required 인터페이스: Required 인터페이스는 컴포넌트가 사용하는 외부의 서비스, 즉 현재 컴포넌트가 호출하는 외부 함수 시그너처의 집합을 의미한다. Required 인터페이스는 Provided, Customize 인터페이스와는 달리 실행 시간에 사용되지 않는다. Required 인터페이스는 컴포넌트에 의해 사용되는 외부 함수들의 집합을 요약해 놓은 명세 수준의 정보이다. 따라서, Required 인터페이스의 실제적인 구현은 다른 컴포넌트에 있게 된다. 컴포넌트를 사용하는 클라이언트는 Required 인터페이스를 통해 컴포넌트가 실행되기 위해 필요한 외부 함수들을 한 눈에 알 수 있으며 컴포넌트 간의 의존 관계를 알 수 있다. Required 인터페이스 상에 가변성이 발생할 수 있으며, 그 경우는 다음과 같이 분류할 수 있다. 첫째, Required 인터페이스를 구현한 외부 컴포넌트의 Provided 인터페이스 상에 가변성이 발생할 경우, 둘째, 같은 기능에 대해 서로 다른 오퍼레이션을 호출해야 하는 경우이다.

Customize 인터페이스: 컴포넌트의 형태가 본 모델에서 지향하는 블랙 박스 형태일 경우, 컴포넌트 내부를 직접 수정할 수 없기 때문에 가변성을 설정하기 위해 외부로 드러난 인터페이스가 있어야 한다. 이러한 목적을 가진 인터페이스가 Customize 인터페이스이며 가변점에 가변치를 설정하기 위해 호출되는 함수들의 집합을 포함하고 있다. 외부에서 호출할 수 있는 인터페이스라는 측면에서 Provided 인터페이스와 비슷하나, Provided 인터페이스는 컴포넌트가 외부에 제공하는 서비스를 의미하는 반면 Customize 인터페이스는 가변성 설정을 위해 일반적으로 배치 시간에 특화하기 위해 한 번 호출되는 함수들의 집합을 의미한다.

4.3 컴포넌트

컴포넌트는 기능성을 구현한 독립적으로 배포 가능한 소프트웨어 단위로서 내부 설계와 구현을 캡슐화하여 가지고 있다[12]. 어떠한 참조 모델들은 명세 수준 또는 구현 수준의 컴포넌트만을 의미하고 있지만, 본 논문의

참조 모델은 명세 수준의 컴포넌트뿐 아니라 인스턴스화가 가능한 구현 수준의 컴포넌트를 의미한다. 컴포넌트는 그림 6과 같이 클래스, 클래스들 사이의 워크플로우, 가변성으로 구성된다. 패밀리 멤버들 사이의 공통성이 컴포넌트 내부에 구현되어 있으며 가변성을 통해 컴포넌트의 커스터마이제이션이 가능하다.

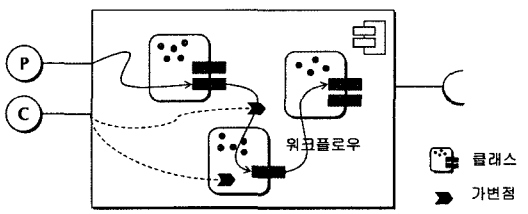

그림 6 컴포넌트 모델

클래스: 컴포넌트는 관련된 클래스의 집합을 통해 동종의 기능성을 제공한다. 컴포넌트는 하나의 클래스보다 더 큰 재사용성을 제공하며 이상적으로는 글래스(Glass) 박스 컴포넌트가 사용하기에 효율적이다. 글래스 박스 컴포넌트는 내부의 수정은 허용하지 않고 안에서 어떠한 로직의 흐름이 있는지 알 수 있게 하기 때문이다. 그러나, 각 기업 마다 고유 업무 노하우와 지식을 공개하지 않으려 하기 때문에 이러한 형태의 컴포넌트를 개발하는 데는 한계가 있다. 따라서, 본 논문에서는 블랙 박스 형태의 컴포넌트를 가정한다.

워크플로우: 컴포넌트가 제공하는 기능성을 실행하기 위한 클래스들 사이의 일련의 메시지 흐름을 워크플로우라 하며 특별히 컴포넌트 내부의 워크플로우를 마이크로 워크플로우라고 한다. 마이크로 워크플로우는 비즈니스 컴포넌트의 비즈니스 함수를 실행하기 위한 일련의 함수 호출이다. 마이크로 워크플로우는 하나의 Use-case의 워크플로우와 대응되며 상호작용 다이어그램으로 설계된다. 가변점은 이러한 워크플로우 상에 생길 수 있으며 이를 워크플로우 가변성이라 한다.

컴포넌트 내부 가변성: 그림 6에서와 같이 컴포넌트 내부에 있는 클래스 안에 가변점이 발생할 수 있다. 가변점이 발생할 수 있는 위치를 더욱 상세히 분류하면 어트리뷰트 상의 가변점과 오퍼레이션 로직 상의 가변점이 된다. 또한, 영구적으로 저장해야 하는 어트리뷰트를 2차 저장소에 저장할 때, 멤버마다 2차 저장소의 종류가 달라질 수 있으며 같은 저장소라 할지라도 데이타 스키마가 달라질 수 있다. 이러한 영구 데이타를 저장하는데 있어 발생될 수 있는 가변성이 영구 가변성이다.

4.4 컴포넌트 간의 관계

컴포넌트 프레임웍 내에서는 컴포넌트들이 어플리케이션을 위한 시스템 수준의 서비스를 제공하기 위해 상호작용한다. 이러한 상호 작용을 통해 컴포넌트들 사이에는 연관 관계(Association)와 의존 관계(Dependency)의 관계(Relationship)가 생긴다. 컴포넌트 간의 관계는 기존의 모델들에서 암시적으로 포함시키고 있지만, 본 논문에서는 이를 명시적으로 구별하여 정의한다.

연관 관계(Association): 연관 관계는 컴포넌트의 인스턴스들 사이의 영구적인 상호 의존 관계를 의미한다. 컴포넌트들 사이에 결합도를 낮추는 것이 컴포넌트 설계의 중요 기준이 되지만, 컴포넌트의 크기, 역할 등에 따라 다른 컴포넌트의 인스턴스와 영구적인 관계를 유지해야 하는 경우가 발생한다. 명세 수준의 컴포넌트만을 제공하는 참조 모델은 이러한 형태의 관계를 제공하지 않지만, 본 논문에서는 컴포넌트의 인스턴스화를 허용하기 때문에 컴포넌트의 인스턴스들 간의 영구적인 링크 관계를 허용한다. 연관관계는 의존관계보다 강한 결합도의 형태를 가지며 의존 관계와의 주된 차이점은 영구성에 있다.

의존 관계(Dependency): 의존 관계는 컴포넌트들 사이의 메시지 호출로 정의될 수 있다. 인스턴스화와 같은 개념이 꼭 필수적인 것은 아니고 두 개의 연관된 컴포넌트들 사이에 메시지를 보낼 수 있다는 것이 의존 관계의 핵심적인 의미이다. 의존 관계는 영구적이지 않으며 약한 형태의 결합도를 가진다. 의존 관계는 Required 인터페이스의 명세서를 통해 검사할 수 있다. Required 인터페이스에 명시된 함수에 대한 구현 부분을 제공하는 컴포넌트가 바로 현재 컴포넌트가 의존하는 컴포넌트가 된다.

워크플로우는 이러한 의존 관계들의 일련의 집합을 의미한다. 하나의 컴포넌트 범위 내에서의 워크플로우를 마이크로 워크플로우라 부르는 반면 컴포넌트 외부에서 컴포넌트들 사이의 워크플로우를 매크로 워크플로우라 부른다. 매크로 워크플로우는 시스템 수준의 기능성을 수행하기 위해 여러 개의 컴포넌트들 간의 일련의 메소드 호출을 의미한다. 매크로 워크플로우는 일반적으로 여러 개의 Use-case들 간의 워크플로우에 대응되기도 한다. 그러나 프레임웍을 사용하여 개발하는 특정 어플리케이션 마다 워크플로우는 다양할 수 있어 워크플로우 상에 가변점이 발생할 수 있다. 이렇듯, 각 어플리케이션 마다 희망하는 워크플로우로 변경하여 프레임웍을 사용하게 되면 재사용성을 높일 수 있게 된다.

4.5 커넥터

컴포넌트 통합 시에 여러 불일치의 문제로 인해 컴포넌트 통합에 현실적인 어려움이 있다. 따라서, 이러한 문제점을 해결하기 위해 본 모델에서는 커넥터라는 요소를 제안한다. 소프트웨어 아키텍처에서 커넥터라는 개

념이 사용되고 있지만[8], 본 논문에서는 기존의 커넥터의 의미를 특화하여 새로운 용도 및 역할, 그리고 세부적인 타입들을 제안한다.

역할: 만약 획득된 블랙 박스 형태의 바이너리 컴포넌트가 요구되는 환경과 상당 부분은 일치하지만 정확하게 맞지 않을 경우 어떻게 할 것인가. 컴포넌트들이 원활히 상호운영 되도록 새로운 컴포넌트를 만들 것인가. 커넥터가 이러한 문제를 해결해줄 수 있다. 커넥터는 현재 상태의 컴포넌트에 대한 변경 없이 컴포넌트를 재사용할 수 있게 해준다. 따라서, 커넥터는 하나의 새로운 컴포넌트를 만드는 것에 비해 많은 생산 비용을 절감할 수 있다는 이점이 있다. 커넥터를 사용함으로써 컴포넌트를 커스터마이즈하는데 드는 노력을 감소시키고 컴포넌트와 프레임웍의 적용성을 증가시킨다.

커넥터 사용의 이점 중 하나는 컴포넌트들 사이의 관계에 있어서 비교적 낮은 결합도를 제공한다는 것이다. 커넥터를 통해 상호 작용하는 컴포넌트들은 서로 간에 직접적인 의존성을 가지고 있지 않기 때문에 컴포넌트들 사이의 의존성 문제에 해결책을 제공할 수 있다. 커넥터의 중요한 역할은 컴포넌트가 사용될 환경에 적합하도록 컴포넌트에 기능을 약간 추가하는 것이다. 컴포넌트 사용자가 원하는 기능성을 제공하도록 컴포넌트 외부에 추가적인 기능을 더하는 것은 클라이언트 코드와 컴포넌트 내부 코드의 변화를 최소화하는 범위 내에서 재사용 가능하도록 한다.

커넥터 사용의 또 다른 이점은 프레임웍 수준의 가변성이 지원될 수 있다는 것이다. 특별히 매크로 워크플로우 상의 가변점은 커넥터에 의해 처리될 수 있으며 컴포넌트 프레임웍 내의 구성 요소의 유연한 재구성을 가능하게 한다. 또한, 컴포넌트 내에 존재하게 되는 가변성을 외부화시켜서 커넥터를 통해 해결할 수 있다[15]. 따라서, 바이너리 형태로 제공되는 컴포넌트의 커스터마이제이션을 외부에서 할 수 있게 함으로써 용이한 재사용성을 제공한다.

커넥터의 구성 요소: 커넥터는 네 가지 요소로 구성된다; In-port, Out-port, Behavior, 그리고 시맨틱. 그림 7에서와 같이 In-port와 Out-port는 소스 컴포넌트와 타겟 컴포넌트와 상호작용하기 위한 접점이 된다. 소스 컴포넌트는 타겟 컴포넌트로부터 어떠한 기능을 요구하는 컴포넌트이다. In-port는 소스 컴포넌트의 Required 인터페이스에 명시된 함수로부터 데이타를 받게 되고 Out-port는 그러한 데이타를 커넥터로부터 읽어서 타켓 컴포넌트에게 전달하게 된다.

Behavior는 소스 컴포넌트가 타겟 컴포넌트에게 요구하는 추가적인 기능 자체를 의미한다. 추가적인 기능의 주된 목적은 소스 컴포넌트와 타겟 컴포넌트 사이의 구문론적, 의미론적, 데이타 타입, 그리고 워크플로우 불일치 문제들을 해결하는 것이다. In-port를 통해 메시지와 데이타를 받게 되면 커넥터가 필요한 로직을 수행하여 Out-port를 통해 처리된 메시지와 데이타를 타겟 컴포넌트에게 보낸다. 그런 후, 타겟 컴포넌트는 결과를 Out-port에 보내고 Out-port가 In-port를 통해 소스 컴포넌트에게 결과를 보낸다.

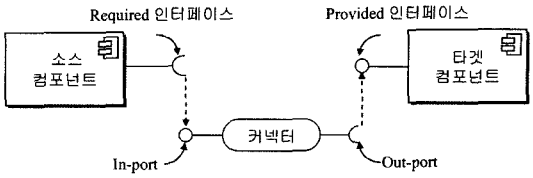

그림 7 커넥터의 in-port와 out-port

커넥터의 시맨틱은 선조건, 후조건, 그리고 불변 조건으로 구성된다. 선조건은 In-port의 메소드를 실행시키기 위해 미리 만족되어 있어야 하는 논리적인 조건을 의미한다. 후조건은 메소드가 호출된 다음에도 참이어야 하는 논리적인 조건을 의미한다. 불변 조건은 커넥터가 항상 참이라고 가정하는 포트와 관련된 논리적인 조건이다[16].

커넥터는 컴포넌트 통합 시에 발생하는 불일치의 유형에 따라 가변성을 수반한다. 컴포넌트 통합 시에 소스 컴포넌트와 타겟 컴포넌트 간에 데이타 타입이 맞지 않거나 인터페이스 시그너처가 맞지 않거나 미세한 기능성의 차이로 인해 맞지 않을 경우 등에 따라 커넥터의 내부 설계가 달라진다. 이러한 커넥터의 가변성이 커넥터 타입 분류의 근간이 된다.

커넥터의 타입: 커넥터의 종류에는 데이타 변환(Data Transformer), 인터페이스 어댑터(Interface Adaptor), 기능성 변환(Functional Transformer), 그리고 워크플로우 핸들러(Workflow Handler)의 대표적인 네 가지 종류가 있다.

데이타 변환 커넥터 타입은 컴포넌트들 사이에 데이타 타입에 대한 불일치가 일어날 경우 유용하게 사용된다. 이 커넥터는 불충분하고 상호 호환되지 않는 데이타 타입을 상호 호환되며 의미가 충분한 데이타 타입으로 변환시키는 역할을 한다. 데이타 변환 커넥터는 데이타 타입과 관련된 시그너처와 의미상의 불일치를 모두 해결하는 역할을 한다.

인터페이스 어댑터 커넥터 타입은 Application Program Interface(API)에 대한 시그너처상의 불일치가 발생할 때, 유용하게 사용된다. 이 커넥터는 소스 컴포넌트의 Required 인터페이스에 명시된 오퍼레이션의 의미와 타겟 컴포넌트의 Provided 인터페이스에 명시된 오퍼레이션의 의미가 일치한다는 것을 가정한다. 구문론적인 불

일치는 오퍼레이션 이름, 파라미터의 개수에 대해 발생할 수 있다. 획득된 COTS 컴포넌트나 대체된 컴포넌트가 API 신텍스 상의 관점에서 상호 호환되지 않을 경우에 인터페이스 어댑터 커넥터가 효율적으로 사용된다.

기능성 변환 커넥터 타입은 상호 작용하는 컴포넌트들 사이에 기능에 대한 불일치가 발생할 때 유용하게 사용된다. 타겟 컴포넌트가 클라이언트가 요구하는 기능 가운데 일부분만을 제공한다면 기능성 변환 커넥터를 사용하여 불충분한 기능성을 추가할 수 있다. 비록, 소스 컴포넌트의 API 신텍스와 타겟 컴포넌트의 API 신텍스가 일치한다 할지라도, 각 API의 의미, 즉, 실제 처리 로직이나 API 실행 전후의 결과가 다를 수 있다. 타겟 컴포넌트에서 제공하는 기능성과 소스 컴포넌트에서 요구하는 기능성 간에 교집합이 없을 경우, 커넥터에 의해 불일치의 문제가 해결될 수 없다.

워크플로우 핸들러 커넥터 타입은 매크로 워크플로우 상에 새로운 워크플로우를 더하거나 워크플로우에 대한 변화를 주어야 하는 등의 가변성이 발생될 경우에 유용하게 사용된다. 워크플로우 상에 가변성이 자주 발생되는 시스템 컴포넌트의 경우에는 이러한 커넥터를 항상 두어서 클라이언트로 하여금 커넥터를 통해서 핫 스팟을 채우게끔 한다. 타겟 컴포넌트를 추가하거나 대체할 경우에 소스 컴포넌트에 새로운 워크플로우를 추가해야 할 경우가 생긴다. 하나의 오퍼레이션이 두 개 이상의 오퍼레이션으로 분리되기도 하며 소스 컴포넌트상의 여러 오퍼레이션이 하나의 오퍼레이션으로 합쳐지기도 한다. 이러한 모든 경우의 변화에 대해 워크플로우 핸들러 커넥터를 통해 해결책을 구할 수 있다.

마지막으로 위의 네 가지 경우에 대한 복합적인 상황이 발생할 수도 있다. 이러한 경우에 여러 가지 종류의 커넥터를 하나로 합칠 수 있다. 그러나, 클라이언트가 요구하는 범위에서 구문론적인 측면으로나 의미적인 측면으로나 모두 다 맞지 않을 경우에는 커넥터를 통해 문제를 해결하기 어렵다.

5. 사례연구: 뱅킹 컴포넌트 프레임웍

본 장에서는 제시된 프레임웍 모델을 국내의 컴포넌트 산업 육성 프로그램의 일환인 뱅킹 프레임웍 개발에 적용한 과정과 설계를 나타낸다[17]. 뱅킹 프레임웍의 각 구성요소 별 Platform Independent Model(PIM) 수준의 설계를 통해 본 프레임웍 모델의 적용성을 확인한다.

5.1 범용 아키텍처 명세

뱅킹 시스템의 범용 아키텍처는 SEI에서 개발한 Module 뷰의 Decomposition과 Layered 스타일을 적용하여[8] 그림 8과 같이 설계하였다. Layered 스타일을 이용하여 표현(Presentation) 계층에서 주로 접근하는 시스템 서비스(System Service) 계층과 영구적인 데이타를 관리하고 여러 시스템 서비스 계층에서 재사용하는 비즈니스 객체(Business Object) 계층으로 나누었다. 표현 계층이 클라이언트 측이 되며 시스템 서비스 계층과 비즈니스 객체 계층이 합쳐져서 서버측을 구성하게 된다. Decomposition 스타일을 적용해서 고객, 수신, 여신, 상품, 일계의 다섯 개 서브시스템으로 분할하였다. 책임(Responsibility)별로 분할하였기 때문에 서브시스

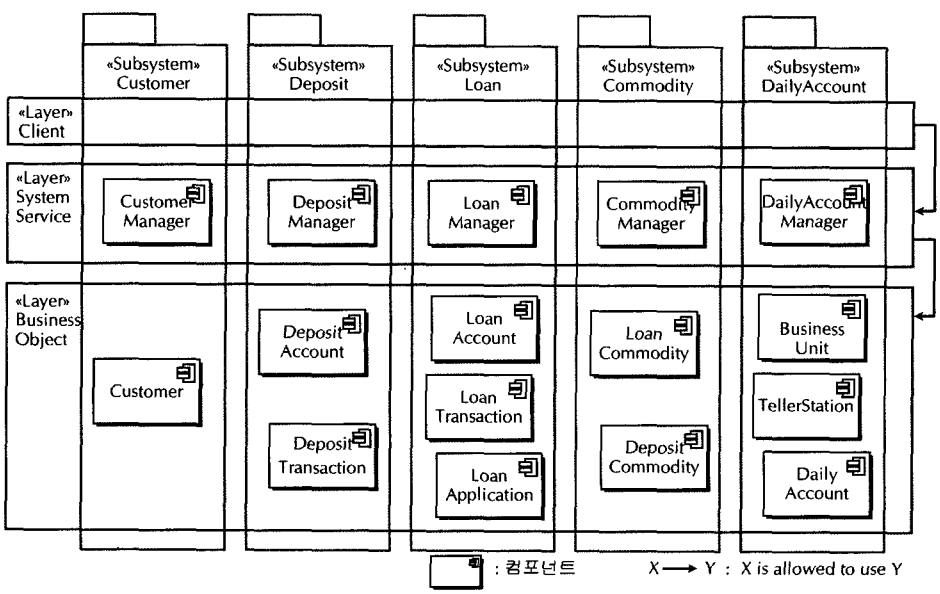

그림 8 뱅킹 컴포넌트 프레임웍의 범용 아키텍처

템의 수정 능력(Modifiability)이 향상되어 유지보수가 수월해진다.

이와 같이 뱅킹 시스템을 위한 범용 아키텍처를 설계함으로 시스템을 구성하는 계층과 서브 시스템 및 컴포넌트로 표현되는 독립적인 기능성들을 설계할 수 있다. 이러한 범용 아키텍처 설계를 각각의 특정한 뱅킹 어플리케이션에 적용함으로 아키텍처를 매번 새롭게 설계하는 부담을 덜어줄 수 있다.

5.2 인터페이스 명세

그림 9는 뱅킹 컴포넌트 프레임웍의 DepositManager 컴포넌트의 인터페이스를 보여주고 있다. Provided 인터페이스로는 IpDepositManager가 있으며, 계좌등록 기능인 registerAccount( ), 계좌해지 기능인 closeAccount( )등이 있다. Required 인터페이스로는 IrCommodity와 IrCustomer가 있으며, 수신 상품의 최대 최소 금액을 조회하기 위한 기능인 getContractMinAmount( )와 getContractMaxAmount( ) 등이 있다. Customize 인터페이스로는 IcDepositManager가 있으며 이자계산 방식을 결정하기 위한 오퍼레이션인 setVariabilityOfAccountId( )와 계좌번호 생성 방식을 결정하기 위한 오퍼레이션인 setVariabilityOfInterestWay( )가 있다. IpDepositManager내의 오퍼레이션 중 일부에 속성 가변성이 존재하여 이를 {VP = AV}라는 tagged value를 사용하여 표기하였다.

이와 같이 세 종류의 인터페이스를 스테레오 타입 등 확장 장치를 이용하여 명세함으로써, 각각의 컴포넌트가 외부와 갖는 의존성의 종류를 구별하여 관리할 수 있다. 각각 컴포넌트에서 제공하는 기능성, 각각의 컴포넌트에서 요구하는 기능성 및 특화시키기 위해 사용해야 하는 기능성 명세는 컴포넌트들 간의 통합 및 컴포넌트 호출 시 이용된다.

5.3 컴포넌트 명세

본 프로젝트에서는 서브시스템 별로 비즈니스 로직을 담당하는 5개의 System 컴포넌트와 10개의 Business 컴포넌트가 존재한다. 이렇게 추출된 컴포넌트 안에는 상호 관련성이 높은 클래스들을 가지고 있으며 독립적인 단위로 배포되어 재사용될 수 있다. 여기서 정의한 컴포넌트를 다른 프레임웍이나 어플리케이션을 개발할 시에 재사용할 수 있다. 그림 10은 Customer 컴포넌트 내부에 대한 정적 설계를 보여주고 있다. Entity 클래스인 Customer, Individual, Corporation, Accident, CustomerValidation와 이들을 중재하는 CustomerCtrl 클래스, 그리고 4개의 value object[18], CustomerInfo, CIndividualInfo, CCorporationInfo, AccidentInfo로 구성되어 있다.

이와 같이 컴포넌트 내부에 대한 정적 설계 외에도 동적 설계 및 가변성 설계를 명세하여 각 컴포넌트의 기능성을 제공하기 위해 내부를 어떻게 구현해야 하는지 설계할 수 있으며 KJ 은행뿐 아니라 다른 은행에서도 이러한 컴포넌트 설계를 특화하여 바로 구현에 사용할 수 있다.

5.4 컴포넌트 간의 관계 명세

뱅킹 컴포넌트 프레임웍내의 컴포넌트 간의 관계는 그림 11과 같이 설계되었다. 고객과 고객에 대한 해당 수신 계좌, 수신 계좌당 기록되는 트랜잭션 정보 간에는 영구적인 관계로 설계되기 때문에 Customer와 DepositAccount, DepositAccount와 DepositTransaction 컴포넌트 사이의 연관 관계로 표기하였다. 일계 계산 기능을 수행하기 위해서는 DailyAccountManager 컴포넌트에서 LoanTransaction, DepositTransaction 컴포넌트들에게 조회하기 위한 메시지를 보낸 후에 DailyAccount 컴포넌트에 일계 등록 메시지를 보내야 하는데, 이러한 관계는 의존 관계로 표기하였다.

프레임웍 내의 컴포넌트 간의 관계 설계를 통해 컴포

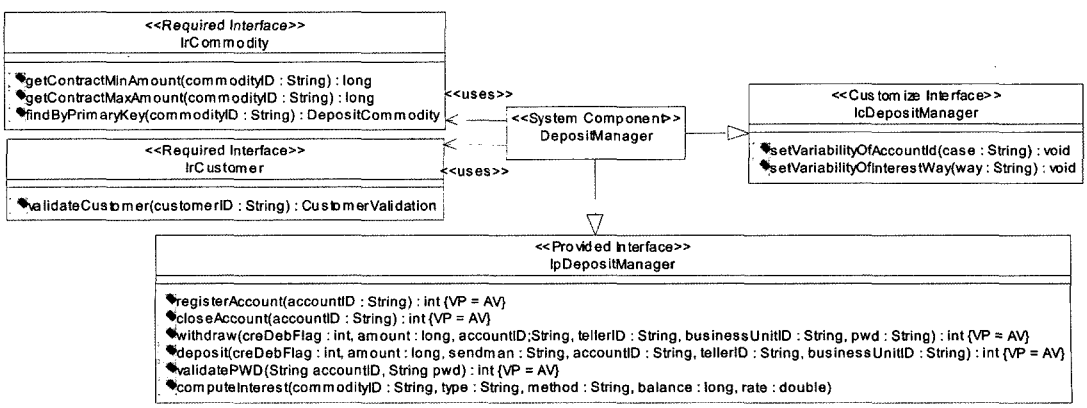

그림 9 DepositManager 컴포넌트의 인터페이스

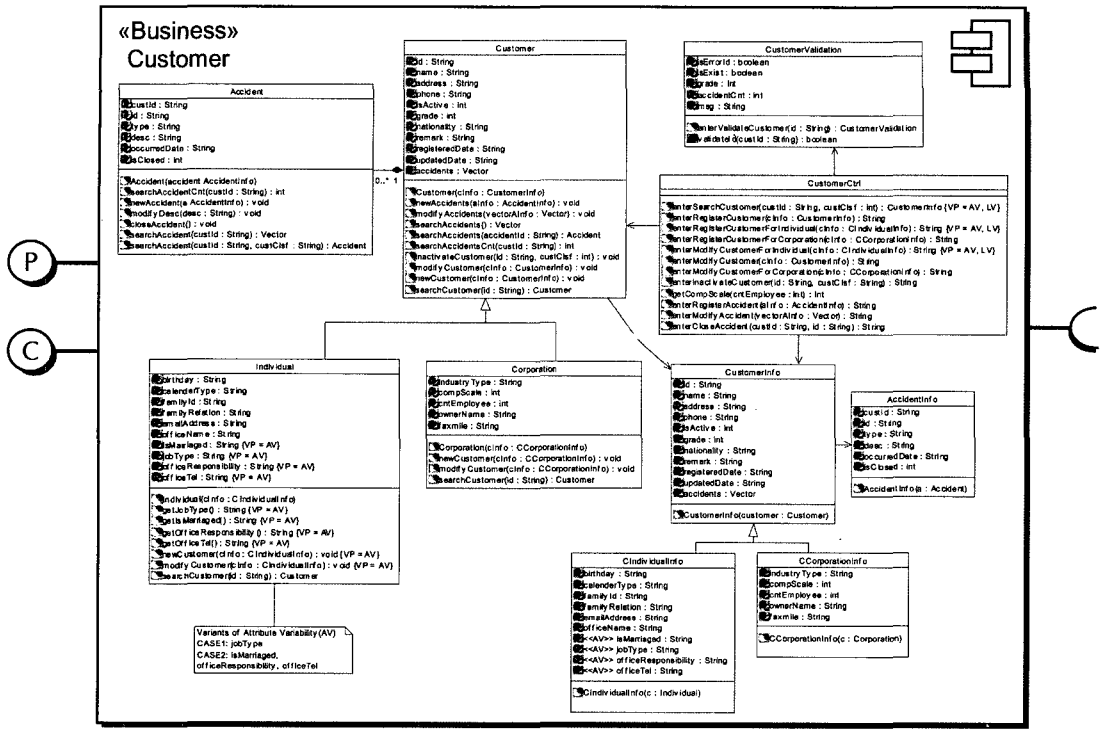

그림 10 Customer 컴포넌트의 내부 정적 모델

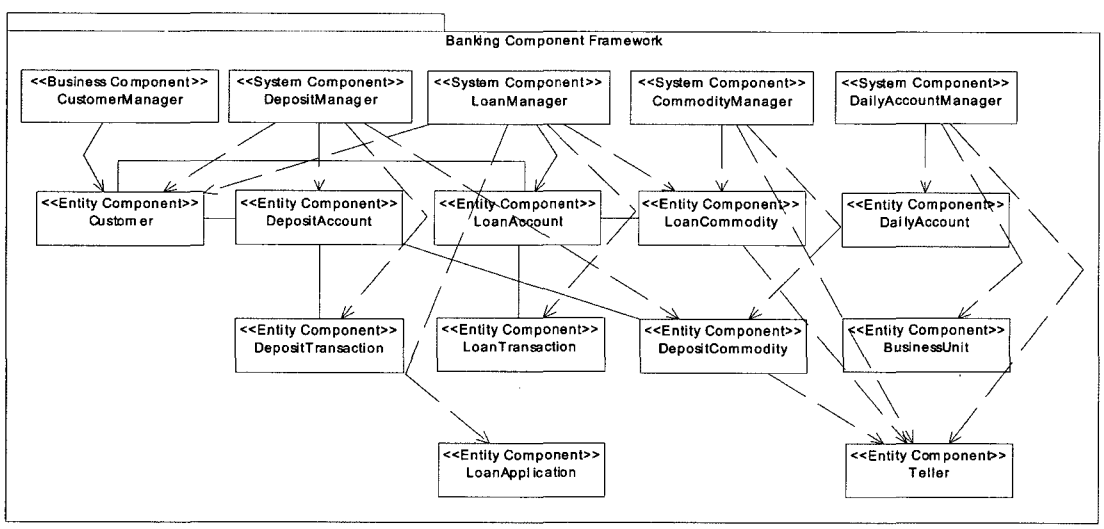

그림 11 뱅킹 컴포넌트 프레임웍의 컴포넌트 간의 관계

넌트 간의 의존성이나 호출 관계 및 나아가서는 워크플로우 설계에 도움을 준다.

5.5 커넥터 명세

본 사례 연구의 뱅킹 컴포넌트 프레임웍을 KJ 뱅킹 어플리케이션에 특화하여 사용할 때, 커넥터로 인해 새로운 컴포넌트를 처음부터 개발하는 것을 방지할 수 있었다. 계좌 이체를 위해 BankCtrl 컴포넌트가 DepositManager 컴포넌트의 기능을 사용하는데 있어, BankCtrl에서 외부 컴포넌트에게 transfer 라는 오퍼레이션을 호출한다. 그러나 뱅킹 컴포넌트 프레임웍 내의 DepositManager 컴포넌트에서는 withdraw와 deposit 오퍼레이션을 제공하여서 통합하는데 문제가 생겼다. 그

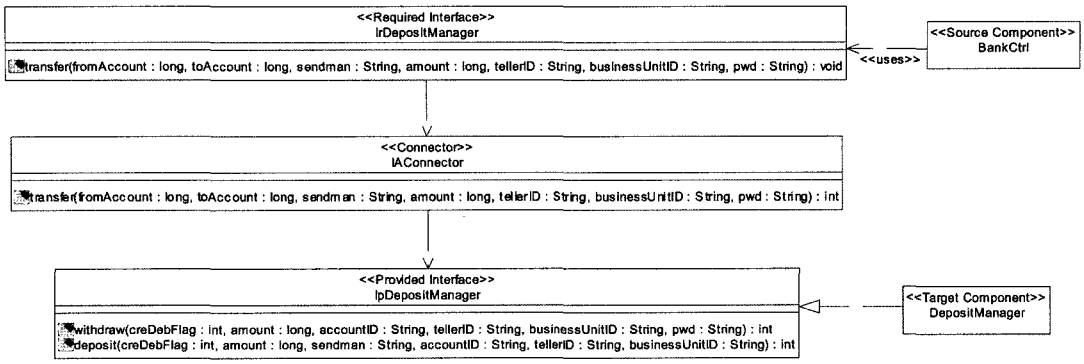

그림 12 IAConnector의 정적 설계

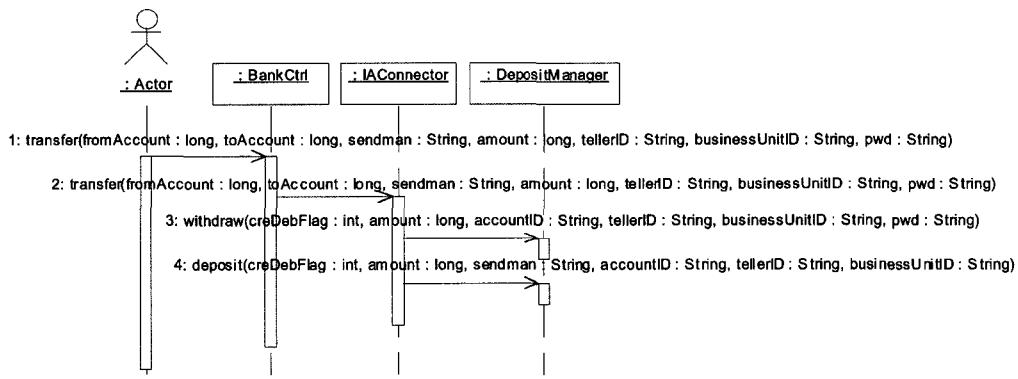

그림 13 IAConnector의 동적 설계

리고, BankCtrl에서는 계좌 번호에 대한 데이타 타입이 long형인 반면, DepositManager의 계좌 번호 타입은 String형이다. 따라서, 두 컴포넌트 간의 데이타와 인터페이스 상에 불일치가 생겨서 이를 맞춰주기 위해 IAConnector를 사용하였다. 다음 그림 12는 두 컴포넌트, BankCtrl과 DepositManager, 그리고 그 사이의 커넥터인 IAConnector에 대한 정적 설계 모델이다.

다음 그림 13은 IAConnector를 사용하기 위한 동적 설계 모델이다. 2번 메시지의 계좌번호 타입이 타겟 컴포넌트에서 요구하는 타입으로 바뀌었고, transfer 대신에 deposit, withdraw 두 개의 오퍼레이션을 커넥터에서 호출해주어 인터페이스 상의 불일치가 해결되었다.

이와 같이 커넥터를 사용함으로써 컴포넌트를 새롭게 구축하지 않고 기존의 컴포넌트를 재사용할 수 있어 경제적인 측면의 장점이 있다.

6. 프레임웍 모델의 J2EE 적용 지침

본 장에서는 제시된 프레임웍 참조모델의 실용성을 판단하기 위해서, 상용 플랫폼인 J2EE에서의 적용 지침을 살펴본다.

6.1 범용 아키텍처의 적용

범용 아키텍처를 구성하고 있는 컴포넌트들은 J2EE에서 여러 계층에 스테레오타입을 사용하여 명세한다. 시스템 아키텍처는 MVC 모델[18]에 기반하여 표현 계층, 로직 계층, 데이타 계층으로 분류된다. 컴포넌트는 표현 계층에서 웹 요소, JSP, Servlet 등으로 구현되며 로직 계층과 데이타 계층인 서버 단 계층은 EJB를 이용하여 구현할 수 있다.

6.2 인터페이스의 적용

J2EE 환경에서 프레임웍의 인터페이스는 Home Interface와 Component Interface로 구현할 수 있다. 본 모델에서의 Provided 인터페이스는 EJB에서 클라이언트가 사용할 수 있는 비즈니스 메소드를 정의한 Component 인터페이스에 의해 구현될 수 있다. 또한, 가변성 설정을 위한 Customize 인터페이스는 Home 인터페이스의 홈 비즈니스 메소드를 통해 가변성 값을 저장하기 위한 State variable을 설정함으로써 구현되거나 Deployment Descriptor에 설정해 놓고 배치시에 가변성 설정 값의 정보를 이용함으로써 구현될 수 있다. 마지막으로 Required 인터페이스는 앞에서도 기술하였듯이 구현 요소가 아닌 명세 수준의 요소이므로 특정 구현 메커니즘으로 매핑되지 않는다.

6.3 컴포넌트의 적용

J2EE 환경에서 컴포넌트는 EJB Bean(Entity, Session, Message-driven)의 묶음인 jar 단위로 구현될 수 있다. 시스템 서비스 계층의 컴포넌트는 클라이언트가 접근할 수 있어야 하고 대부분 영구적인 데이타를 관리하지 않고 비즈니스 로직을 관리하기 때문에 시스템 계층 컴포넌트의 Provided 인터페이스를 구현한 빈은 리모트 세션 빈으로 구현할 수 있다. 그리고, 비즈니스 객체 계층의 컴포넌트는 영구적인 데이타를 관리하고 클라이언트 계층에서는 접근 가능하지 않아야 하기 때문에 로컬 엔티티 빈으로 구현할 수 있다.

6.4 컴포넌트 간의 관계의 적용

연관 관계: J2EE 환경에서 연관 관계는 엔티티 빈의 컴포넌트 관리 영구성(Container Managed Persistence, CMP)과 빈 관리 영구성(Bean Managed Persistence, BMP)으로 구현될 수 있다. CMP의 경우, 컨테이너가 엔티티 빈들 간의 관계를 Deployment Descriptor의 컨테이너 관리 관계(Container Managed Relationship, CMR) 필드를 통해 관리할 수 있다. 엔티티 빈들 사이의 연관 관계를 컨테이너가 관리해준 CMP와는 달리 BMP는 빈 작성자가 직접 빈의 코드에 연관관계를 기술함으로써 관계를 유지시켜준다.

의존 관계: J2EE 환경에서 의존 관계는 메시지로 구현할 수 있다. 의존 관계는 컴포넌트 들 사이의 메시지 호출 관계를 의미하는 것이므로 J2EE 뿐 아니라, OOP의 메시지 전달이라는 메커니즘을 통해 구현 가능하다. J2EE의 경우, 홈 인터페이스를 통해 컴포넌트 인터페이스를 얻어와서 그 내의 비즈니스 메소드에게 메시지를 보내는 형태로 구현된다.

6.5 커넥터의 적용

J2EE 환경에서 커넥터는 일반 자바 클래스로 구현할 수 있다. 커넥터는 영구성을 가지고 있지 않기 때문에 가벼운 모듈 형태의 클래스나 EJB의 경우, 빈으로도 구현 가능하다. 커넥터의 포트는 메소드 형태로 제공되어 소스 컴포넌트에서 이러한 메소드를 호출하여 사용하게 된다. 이러한 자바 클래스나 빈 형태의 커넥터는 다른 머신에 배치되어 있는 경우에도 JNDI 네임을 통해 쉽게 찾아서 사용 가능하다.

7. 평가

본 장에서는 컴포넌트 프레임웍의 참조모델이 가져야 하는 적합한 기준(Criteria)을 제시하고, 이 기준에 따라 프레임웍 모델들을 평가를 한다. 컴포넌트 프레임웍은 특정 도메인에서 재사용될 수 있어야 하고 범용적 기능성과 범용적 비기능성을 효율적으로 설계할 수 있어야 한다[7]. 또한, 재사용성을 높이기 위해 대량 특화가 가능해야 하며[13] 컴포넌트 통합 시의 문제를 해결할 수 있어야 한다[2]. 본 논문에서 제안한 컴포넌트 프레임웍의 참조 모델과 다른 모델을 비교한 결과는 표 1과 같다. 각 기준은 다음과 같은 기술적 이유와 타당성을 기반하여 선정되었다.

- **범용 아키텍처:** 범용적 비 기능성 및 범용적 기능성 제공을 통한 재사용성을 얻을 수 있다.
- **인터페이스:** 범용적 기능성에 대한 기준을 마련하여 재사용성을 높일 수 있고 컴포넌트와의 의존성을 낮추어 주어서 유지보수성 및 대체성을 높일 수 있다.
- **컴포넌트:** 범용적 기능성을 제공하여 재사용성을 높이는 주된 구성요소이다.
- **커넥터:** 컴포넌트 통합 시의 문제를 해결해줄 수 있으며 컴포넌트 내부의 수정 없이 외부에서 기능 수정을 하게함으로써 유지보수나 재사용성을 높을 수 있다.
- **가변성:** 대량 특화를 통해 재사용성을 높이게 해주는 주된 요소이다.
- **상세한 인터페이스 타입:** 세부적인 타입을 나눔으로써 컴포넌트가 외부와 갖는 모든 의존성을 관리할 수 있다. 컴포넌트 간의 상호작용 설계와 컴포넌트 간의 불일치 분석을 하기에 용이하다.
- **상세한 커넥터 타입:** 세부적인 타입을 나눔으로써 예상 가능한 불일치의 문제를 해결할 수 있다.
- **상세한 가변성 타입:** 하나의 컴포넌트 내의 가변성보다 프레임웍 수준의 가변성을 정의할 수 있어 본 모델의 재사용성을 증가시킬 수 있으며 특화 가능한 부분을 미리 예상할 수 있다.

비교 결과 다른 모델은 컴포넌트 간의 통합 문제를

표 1 프레임웍 모델 비교표

	Fayad's	Atkinson's	Latchem's	Wills'	제안된 모델
범용 아키텍처	-	✓	✓	✓	✓
인터페이스	✓	✓	✓	✓	✓
컴포넌트	-	✓	✓	✓	✓
커넥터	-	-	-	✓	✓
가변성	✓	✓	-	✓	✓
상세한 인터페이스 타입	-	✓	-	-	✓
상세한 커넥터 타입	-	-	-	-	✓
상세한 가변성 타입	-	-	-	-	✓

해결하기 위한 참조 모델 상에서의 지원을 해주지 않고 있었다. 또한, 상세한 인테페이스, 커넥터, 가변성 타입에 대한 정의가 없었다. 본 모델은 커넥터의 종류를 데이타 변환, 인터페이스 어댑터, 기능 변환, 그리고 워크플로우 핸들러의 네 가지로 분류하고, 인터페이스의 종류를 Provided, Required, Customize의 세 가지로 분류하였으며, 가변성의 종류를 속성, 로직, 마이크로 워크플로우, 영속성, 인터페이스, 매크로 워크플로우, 컴포넌트, 컴포넌트 간 관계, 아키텍처 스타일로 분류함으로써 효율적인 프레임웍 설계를 위한 기초를 다졌다.

8. 맺음말

CBSE와 PLE는 재사용 가능한 소프트웨어 부품을 이용하여 어플리케이션을 개발하기 위한 새로운 패러다임으로 널리 인정되고 있다. PLE의 프레임웍은 CBD의 컴포넌트보다는 큰 재사용 단위이며 관련된 컴포넌트들의 집합, 커넥터들 그리고 도메인에 특정한 아키텍처로 구성된다. PLE의 프레임웍이 컴포넌트보다 많은 장점을 가지고 있다는 것은 일반적으로 알려져 있는 사실이지만, 프레임웍의 핵심 요소들이나 내부 구조는 명확히 정의되지 않았다. 최근까지 연구된 대부분의 프레임웍 모델은 개념적인 수준에 머물러 있었다. 따라서 본 논문에서는 개념적인 수준에 그치는 프레임웍 모델이 아닌 실용적인 수준의 프레임웍 참조 모델을 제안하였으며 프레임웍의 핵심 요소들을 정의하였다.

또한, 본 논문에서는 프레임웍 모델의 구체적인 구성요소를 식별하고 프레임웍 내에서의 컴포넌트들 간의 상호 연관성, 커넥터의 확장된 의미, 프레임웍 모델에서의 범용 아키텍처의 역할 등에 대해 기술하였다. 제시된 모델은 실용적으로 적용 가능하도록 제안되었으며 실용성을 증명하기 위해 뱅킹 시스템에서의 적용 사례와 각 구성요소들이 J2EE의 어떠한 구현 메커니즘을 이용하여 구현될 수 있는지를 명시하였다. 이는 J2EE 환경 뿐 아니라 .NET과 같은 다른 구현 환경에서도 동일하게 적용 가능하다.

참 고 문 헌

[1] Kim, S. and Park, J., "C-QM: A Practical Quality Model for Evaluating COTS Components," Proceedings of the 21st IASTED SE'2003, pp. 991-996, Feb. 2003.

[2] Latchem, S., "Component Infrastructures: Placing Software Components in Context," Chapter 15 of *Component-Based Software Engineering: Putting the Pieces Together*, Heineman G. and Council, W., ed., Addison-Wesley, 2001.

[3] Atkinson, C., et al., *Component-Based Product Line Engineering with UML*, Addison Wesley, 2002.

[4] Fayad, M. and Schmidt, D., "Object-Oriented Application Frameworks," *Communications of the ACM*, Vol.40, No.10, pp. 32-38, Oct. 1997.

[5] Wills, A. C., "Components and Connectors: Catalysis Techniques for Designing Component Infrastructures," Chapter 17 of *Component-Based Software Engineering: Putting the Pieces Together*, Heineman G. and Council, W., ed., Addison-Wesley, 2001.

[6] Cheesman, J., and Daniels, J., *UML Components*, Addison-Wesley, 2000.

[7] Matinlassi, M., Niemela, E., and Dobrica, L., "Quality-driven Architecture Design and Quality Analysis Method," VTT Technical Research Centre of Finland, Espoo, 2002.

[8] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J., *Documenting Software Architecture*, Addison Wesley, 2002.

[9] Geyer, L., and Becker, M., "On the Influence of Variabilities on the Application Engineering Process of a Product Family," LNCS 2379, pp.1-14, Proceedings of SPLC2 2002, San Diego, CA, USA, August 19-22, 2002.

[10] Kim, S., Her, J., and Chang, S., "A Theoretical Foundation of Variability in Component-Based Development," *Information and Software Technology*, Vol.47, pp.663-673, 2005.

[11] Sharp, D. C.: "Containing and Facilitating Change via Object Oriented Tailoring Techniques," *Proceedings of the 1st SPLC*, Denver, Colorado, August 2000.

[12] Whitehead, K., *Component-based Development: Principles and Planning for Business Systems*, Addison-Wesley, 2002.

[13] Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[14] Crnkovic, I., and Larsson, M., Building Reliable Component-Based Software Systems, Artech House, 2002.

[15] Levi, K., and Arsanjani, A., "A Goal-driven Approach to Enterprise Component Identification and Specification," CACM, Vol.45, No.10, pp. 45-52, Oct. 2002.

[16] D'Souza, D. and Wills, A., Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison Wesley, 1999.

[17] Kim, S., "Lessons Learned from a Nation-wide CBD Promotion Project," Communications of the ACM, Vol.45, Issue.10, pp. 83-87, Oct. 2002.

[18] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley, Jan. 15, 1995.

허 진 선
정보과학회논문지 : 소프트웨어 및 응용
제 33 권 제 3 호 참조

김 수 동
정보과학회논문지 : 소프트웨어 및 응용
제 33 권 제 1 호 참조