컴포넌트 프레임워크 설계를 위한 실용적인 커넥터 패턴

(Practical Connector Patterns for Designing Component Frameworks)

민 현 기 [†] 김 수 동 [††]

(Hyun Gi Min) (Soo Dong Kim)

요 약 학계와 산업계 모두 효율적인 재사용 기술인 Component-based Development(CBD)로의 전환을 받아들이고 있다. Product Line Engineering(PLE)에서의 컴포넌트 프레임워크는 컴포넌트, 커넥터와 그들의 시멘틱(Semantic)들의 집합으로 구성된다. 단순한 컴포넌트 조립 보다 반완성 어플리케이션인 컴포넌트 프레임워크가 재사용의 잠재력이 더 크다. 그러나, 어플리케이션 개발을 위해 여러 업체에서 획득된 COTS 컴포넌트들의 연관관계, 의존관계가 서로 완벽하게 일치하지 않아, 컴포넌트 조립이나 컴포넌트 대체시 구현을 변경해야 하는 문제가 발생된다. 그러므로 커넥터는 관련된 컴포넌트들간의 상호작용 관리 뿐만 아니라, 조립될 수 없는 컴포넌트들간의 문제를 보완하여 연결한다. 아직 실용적으로 사용할 수 있는 커넥터에 관한 연구 및 구체적인 해결 방안이 미흡하다.

본 논문에서는 커넥터를 정의 하기 위한 메타모델을 제시하고, 커넥터를 실용적으로 설계하고 구현할 수 있는 5개의 주요 패턴을 제시한다. 제시된 주요패턴들은 설계 지침 및 문제 해결방안을 제공하여 이를 통해 컴포넌트 프레임워크 기반의 실용적이고 효율적인 커넥터를 구성할 수 있도록 한다. 또한, 주요 커넥터 패턴의 적용 기법 및 구현을 통한 적용 사례를 제시하여 컴포넌트의 활용성과 재사용성이 증가됨을 보인다.

키워드 : 커넥터, PLE, CBD, 소프트웨어 아키텍처, 컴포넌트 프레임워크

Abstract Component-based development(CBD) has acquired a substantial acceptance in both academia and industry as an effective inter-organizational reuse technology. A component framework in product line engineering(PLE) which consists of related components, connectors and their semantics has a greater potential for reusability than components. In frameworks, components are glued with association, dependency and connections. Problems occur like affecting the implementation code of components when they are glued and replaced because the association and dependency relationships between COTS components which are acquired for application development do not match exactly. Especially, a connector may not only connect related components, but also make partially-matched COTS components fit together. However, little has been studied to date about connectors that can be used practically.

In this paper, we present a meta-model for connectors and show how a connector can be designed and implemented in practice. We propose five main patterns of connectors. Proposed major patterns provide design guidelines for practical and efficient connector configuration based on component framework. And also, applying techniques and applied case studies of the major patterns show greatly increased applicability and reusability of the component without component modification.

Key words : Connector, PLE, CBD, Software Architecture, Component Framework

1. 서 론

학계와 산업계 모두 효율적인 재사용 기술인 CBD를 가치 있게 받아들이고 있다. COTS 컴포넌트는 생산자와 잠재적 사용자들을 가지고 있다. 컴포넌트를 사용하는 사용자는 소프트웨어 개발 비용을 줄이고, 짧은 시간

· 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음
† 비 회 원 : 숭실대학교 컴퓨터학과
hgmin@otlab.ssu.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학부 부교수
sdkim@ssu.ac.kr
논문접수 : 2003년 11월 8일
심사완료 : 2004년 1월 14일

에 시스템 구축을 하기를 희망한다. 또한, 사용자들은 좋은 컴포넌트를 찾기 위해 컴포넌트를 획득하기 전에 COTS 컴포넌트를 평가하기를 희망한다[1]. 그러므로 COTS 컴포넌트들의 품질을 평가하는 것은 성공적인 컴포넌트 기반의 어플리케이션 개발에 있어서 미리 수행되어야 할 중요한 요소이다[2].

이러한 노력에도 불구하고, CBD에서 일반적인 문제점 중 하나는 요구하는 모든 기능을 제공하는 COTS 컴포넌트를 얻는 것이다. 컴포넌트를 선택하기 위한 후보 컴포넌트는 요구되는 기능의 일부분만을 제공하는 경우가 있다. 컴포넌트 미들웨어 기반의 구성(Configuring)이나 배치(Deploying)를 위한 많은 종류의 전략이 필요하게 되어 더욱더 이러한 문제를 악화시켰다. 따라서 개발자들은 불만족스러운 전략과 컴포넌트들의 선택과 관련하여 많은 시간을 디버깅 문제로 보내고 있다[3]. 심지어 오랜 노력에 의해 획득된 컴포넌트들도 서로서로 잘 맞지 않는 문제가 생긴다[4].

이러한 문제점들을 해결하기 위해 PLE의 컴포넌트 프레임워크는 컴포넌트들의 집합과 그것들을 미리 조합할 수 있는 컴포넌트 간의 관계, 불일치 문제를 중재해 주는 커넥터를 포함한다[5]. 그러나, 발표된 논문들의 커넥터는 개념적인 수준에 머물러 있고, 이러한 커넥터를 실용적으로 사용 할 수 있게 유형을 분석한 후 실체화 한 연구는 부족하다.

컴포넌트 프레임워크는 컴포넌트보다 재사용성에 관한 잠재력이 더 크다. 특히 커넥터는 컴포넌트 사이에서 요구사항에 부분적으로 맞지 않는 부분을 조율하는 역할까지 가능하다. 본 논문에서는 커넥터를 위한 메타모델과 프레임워크를 만들 때 커넥터가 어떻게 사용되는지를 보인다. 우리는 커넥터의 5개의 주요 패턴을 제시한다. 각각의 패턴을 설명하고 커넥터 패턴 적용기법을 제시한다.

본 논문을 위해서 2장에서는 기반 연구를 서술하며, 3장에서는 커넥터의 요소에 관서 기술하며, 제안한 커넥터의 주요 패턴을 4장에 설명하며, 5장에서는 본 논문에서 재시한 커넥터 패턴 적용하여 보이고, 6장에서는 본 논문에서 제시한 커넥터 패턴을 평가하고, 본 논문의 결론은 7장에서 내린다.

2. 기반 연구

2.1 인터페이스

CBD에서 인터페이스는 컴포넌트와 분리된다. 그리고 인터페이스는 사용자와의 규약(Contract)들을 정의한다. 컴포넌트는 그림 1처럼 3가지 타입의 인터페이스를 가지고 있다. 'P'는 Provide 인터페이스, 'R'은 Required 인터페이스를 표현하며, 'U'는 Uses 인터페이스를 나타낸다.

*Provide 인터페이스*는 컴포넌트에 의해 제공되는 서비스의 집합을 정의한다. 이러한 서비스는 함수라는 형태로 클라이언트에게 제공된다. 컴포넌트의 주된 성격을 결정하는 응집력 높은 동종의 기능성은 Provide 인터페이스에 의해 정의된다. 컴포넌트 프레임워크 수준의 워크플로는 Provide 인터페이스의 함수들을 호출함으로써 실행되며 이러한 함수들은 실행 시간(Runtime)에 빈번히 호출된다.

*Required 인터페이스*는 컴포넌트가 외부에 요청하는 서비스를 의미한다. 이러한 의미는 컴포넌트가 요청하는 외부의 컴포넌트나 소프트웨어 단위에 있는 함수를 의미하거나 컴포넌트의 가변성을 설정하기 위해 외부에 요청한다. 이러한 두 가지 의미를 명백히 구분하기 위해 Uses와 Required 인터페이스로 나눈다. 따라서, Required 인터페이스는 가변점(Variation Point)[6]에 가변치(Variants)[6]를 설정하는 함수들의 집합으로 정의된다. 컴포넌트 내부에 존재하는 가변성은 Required 인터페이스에 의해 설계되며 특정 어플리케이션에 맞게 특화할 때 사용된다.

*Uses 인터페이스*는 컴포넌트가 사용하는 외부의 서비스, 즉 현재 컴포넌트가 호출하는 외부 함수의 집합을 의미한다. Uses 인터페이스는 위의 두 개 인터페이스와는 달리 실행 시간에 사용되지 않는다. Uses 인터페이스는 실제 구현되는 인터페이스가 아니고, 컴포넌트에 의해 사용되는 외부의 함수들의 집합을 요약해 놓은 명세 수준의 정보이다. Uses 인터페이스의 실제적인 구현은 다른 컴포넌트의 Provide 인터페이스에서 구현한다. Uses 인터페이스를 통해 컴포넌트가 호출하는 외부 함수들을 알 수 있으며, 또한 컴포넌트간의 의존 관계를 알 수 있다.

2.2 컴포넌트 프레임워크

컴포넌트 프레임워크는 그림 1과 같이 관련된 컴포넌트들의 연관성, 커넥터와 제약사항들의 집합으로 이루어진 대형의 재사용 단위이다. 컴포넌트 프레임워크는 컴포넌트보다 재사용성을 위한 잠재력이 더 크다. 어플리

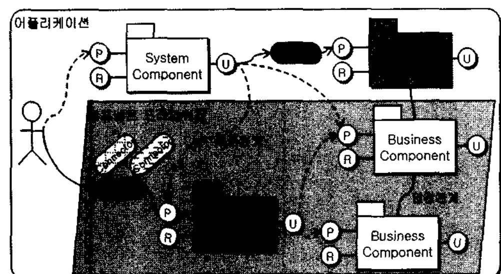

그림 1 컴포넌트 프레임워크

케이션 개발은 모두 컴포넌트들을 조립함으로 이루어지는 것이 아니라, 반완성된 어플리케이션인 컴포넌트 프레임워크에 필요한 컴포넌트들을 추가하여 구축한다.

컴포넌트 프레임워크는 책임과 역할을 분리하여 설계되며, 컴포넌트들 사이의 논리적인 연결들은 불필요한 결합도를 갖지 않아야 한다. 그러므로 변경 없이 재설치할 수 있는 컴포넌트의 프레임워크를 만들기 위해 컴포넌트들간의 상호 작용과 연관관계를 체계적으로 설계해야 한다[4].

2.3 커넥터

커넥터는 컴포넌트 프레임워크 기반에서의 컴포넌트 간의 연결에 유용하다. 커넥터는 컴포넌트들간의 상호작용을 표현할 수 있다. 커넥터는 커넥터가 연결되는 포트에 역할 및 특정한 제약 사항을 부과하며, 컴포넌트간의 연결행위(Joint Action)을 구현하는 특정한 상호작용 프로토콜로 정제된다[7]. 상호작용을 설계하기 위해 커넥터를 사용하는 것은 교체단위가 있는 컴포넌트 프레임워크 설계를 위해 유용한 기술이다. 어플리케이션을 위해 불완전하게 만족하는 컴포넌트의 경우 그림 2와 같이 커넥터를 사용하여 컴포넌트가 어플리케이션을 완벽히 지원할 수 있도록 도와준다.

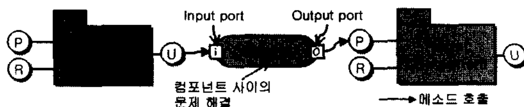

그림 2 컴포넌트와 커넥터의 상호 관계

3. 커넥터의 구성 요소

3장에서는 커넥터의 메타 모델(Meta-Model)을 제시한다. 커넥터는 입력과 출력 포트를 가지고 있는 작은 소프트웨어 부품이다. 이러한 포트는 그림 2와 같이 컴포넌트를 연결하기 위해 사용된다. 그림 2처럼 소스 컴포넌트(Source Component)는 커넥터를 통해 타겟 컴포넌트에 메시지를 전달한다. 이러한 메시지는 소스 컴포넌트의 Uses 인터페이스에서 커넥터의 입력 포트(Input Port)를 통해 전달되고, 그리고 커넥터는 컴포넌트 연결 및 컴포넌트간의 불일치를 해결하고, 출력 포트(Output Port)를 통해 타겟 컴포넌트(Target Component)로 메시지를 전달한다. 이때 타겟 컴포넌트의 Provide 인터페이스가 호출된다.

Uses 인터페이스는 해당 컴포넌트가 다른 컴포넌트를 호출하는 외부의 서비스들을 명세한 후에 입력 포트는 Uses 인터페이스와 연결되고, 출력 포트는 호출되는 컴포넌트의 Provide 인터페이스와 연결된다. 커넥터의 메

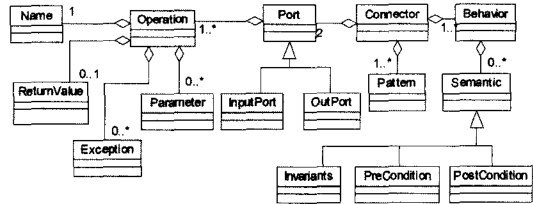

그림 3 커넥터의 메타 모델

타 모델은 그림 3과 같이 UML의 클래스 다이어그램[8]으로 표현한다.

본 논문에서 제안한 커넥터의 모델에서의 커넥터는 컴포넌트들 사이의 연결을 지원할 뿐만 아니라 소스 컴포넌트에 의해서 기대되는 요구사항과 타겟 컴포넌트에 의해 제공되는 기능의 차이점을 변환하는 중요한 역할을 수행한다. 따라서 커넥터는 선택된 컴포넌트의 부분적으로 불일치(Mismatch)되는 문제점을 해결한다. 이러한 불일치 항목은 데이타의 값, 데이타의 포맷, 기능성, 인터페이스의 시그니처(Signature) 또는 워크플로가 된다.

4. 커넥터 패턴 유형

4장에서는 표 1처럼 5개의 주요한 커넥터 패턴을 정의한다. 커넥터 패턴은 동시에 여러 개의 패턴을 복합적으로 적용할 수 있다. *데이타 변형(Data Transformer) 커넥터*는 불일치한 데이타의 포맷이나 데이타 값의 의미를 적절히 변환하는데 사용된다. 따라서, 불일치 된 데이타 타입 또는 데이타 값의 의미는 커넥터에 의해 중재될 수 있다.

*기능 변환(Functional Transformer) 커넥터*는 소스 컴포넌트의 요구기능과 타겟 컴포넌트에서의 제공기능 사이의 기능적 불일치를 중재하기 위해 사용된다. 커넥터는 소스 컴포넌트로부터 요청을 받은 후 기능을 추가하거나 수정한 후에 타겟 컴포넌트로 서비스를 요청한다.

*인터페이스 어댑터(Interface Adapter) 커넥터*는 컴포넌트들 사이에 불일치한 인터페이스 시그니처를 중재하기 위해 사용한다. 인터페이스 어댑터 커넥터에서 지

표 1 커넥터의 주요 패턴

커넥터 패턴	컴포넌트 사용시 문제 발생 상황
데이타 변형	데이타의 타입과 데이타 값 의미의 불일치
기능 변형	요구 및 지원 기능의 불일치
인터페이스 어뎁터	인터페이스 시그니처의 불일치
워크플로 핸들러	워크플로의 불일치
예외처리 어뎁터	예외처리의 불일치

원하는 불일치는 메소드의 이름과 파라메터의 순서를 중재자 패턴(Mediator Pattern)[9,10]을 사용해서 보정할 수 있다. 또한 소스 컴포넌트로부터 전달 받은 하나의 메소드 호출을 몇 개의 메소드를 변환해서 타겟 컴포넌트로 전달한다.

*워크플로 핸들러(Workflow Handler) 커넥터*는 변경된 소스 컴포넌트에서 타겟 컴포넌트로의 워크플로를 추가 및 수정하고, 워크플로 변경에 따른 문제를 보정하는 역할을 수행한다. 이 커넥터는 워크플로의 가변성을 컴포넌트 외부에서 지원할 수 있다.

*예외처리 어뎁터(Exception Adepter) 커넥터*는 컴포넌트들 사이에 불일치한 예외처리 방법을 중재하기 위해 사용한다. 예외처리 어뎁터 커넥터에서 예외처리 불일치 발생시, 소스 컴포넌트가 원하는 적절한 예외사항을 발생시킨다. 따라서 예외처리 방법에 대한 중재를 지원한다. 그러므로 패밀리 멤버별로 예외 처리를 컴포넌트 내부가 아닌 컴포넌트 외부인 커넥터에서 관리할 수 있다.

4.1 데이타 변환 커넥터 패턴

데이타 변환 커넥터는 그림 4와 같이 소스 컴포넌트의 데이타가 타겟 컴포넌트에서 요구하는 데이타 타입과 맞지 않거나 데이타의 의미가 달라 사용할 수 없는 데이타를 사용 가능하도록 변환한다. 이 데이타 변환 패턴은 컴포넌트들 사이에서 생기는 데이타의 문법, 의미적 문제를 해결한다.

소스 컴포넌트로부터 적용될 수 없는 데이타 타입 또는 데이타의 시멘틱이 타겟 컴포넌트로 전달될 때, 이 데이타 변환 커넥터 패턴이 사용된다. 컴포넌트 마켓 플레이스에서 패밀리 멤버의 요구사양을 부분적으로 만족하는 컴포넌트들은 많다. 그러나 COTS 또는 직접 개발한 컴포넌트는 컴포넌트들간의 표준 인터페이스, 표준 오퍼레이션과 표준 시멘틱을 가지고 있지 않다. 따라서 컴포넌트가 교체 되거나 업그레이드 될 때 이러한 문제점이 언제나 발생될 수 있다.

데이타 변환 패턴은 데이타의 타입 불일치와 데이타의 의미 불일치를 해결하는 경우로 나눈다. 만약 타겟

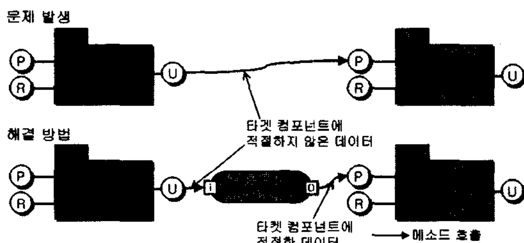

그림 4 데이타 변환 패턴의 구조

컴포넌트의 오퍼레이션에서 사용하는 파라메터의 타입이 String형에서 타겟 컴포넌트 생산자에 의해 int형으로 바뀐다면, 소스 컴포넌트와 타겟 컴포넌트의 연동을 위해 타겟 컴포넌트를 적절한 데이타 타입으로 변경하기 위해 구현을 변경해야 하는 문제가 발생한다. 다른 경우는 소스 컴포넌트에서 타겟 컴포넌트를 호출한 오퍼레이션의 데이타 타입은 서로 같지만, 데이타가 내포하고 있는 의미가 다른 경우이다.

데이타 변환 패턴을 이용하여 컴포넌트간의 데이타 불일치 해결 방법은 첫째, 소스 컴포넌트에서 전달한 데이타가 타겟 컴포넌트에서 요구하는 데이타의 컨텐츠는 같지만, 데이타 타입만 다른 경우는 커넥터에서 소스 컴포넌트에서 전달한 데이타로 타겟 컴포넌트가 요구하는 데이타를 생성시킨 후 타겟 컴포넌트로 전달한다. 둘째, 데이타의 타입은 같지만, 그 컨텐츠의 의미가 다른 경우는 타겟 컴포넌트가 원하는 의미의 컨텐츠값으로 변경하여 타겟 컴포넌트로 전달한다.

셋째, 커넥터에서 데이타 변환 시, 타겟 컴포넌트에서 요구하는 데이타 보다 소스 컴포넌트에서 제공하는 컨텐츠량이 더 많을 경우는 타겟 컴포넌트가 원하는 내용만을 포함시킨 데이타형으로 변환한다. 이 경우와 달리, 넷째, 소스 컴포넌트에서 제공하는 데이타가 타겟 컴포넌트에서 원하는 데이타보다 그 데이타의 컨텐츠량이 적은 경우는 부족한 정보를 기능 변환 패턴을 이용하여, 추가 정보를 획득한 후 소스 컴포넌트에서 제공한 기존 정보에 추가하여 타겟 컴포넌트가 원하는 데이타형으로 전달한다. 다섯째, 타겟 컴포넌트에서 요구하는 데이타를 추가적으로 획득하지 못할 경우에는 두 컴포넌트간의 조립은 불가능하므로 적절한 컴포넌트로의 교체가 필요하다.

데이타 변환 커넥터를 사용하여 컴포넌트간의 의존도를 낮추기 때문에, 만약 소스 컴포넌트의 Uses 인터페이스나 타겟 컴포넌트의 Provided 인터페이스가 변경되더라도 소스 컴포넌트 또는 타겟 컴포넌트를 변경할 필요가 없다. 따라서 이 패턴을 사용하게 되면, 데이타 타입의 불일치를 위해 인터페이스의 시그니처를 변경하거나 데이타의 시멘틱의 불일치를 위해 컴포넌트의 구현을 변경할 필요가 없다.

4.2 기능 변환 커넥터 패턴

기능 변환 커넥터 패턴은 그림 5와 같이 컴포넌트의 불충분하고 적절하지 않은 기능을 적절한 컴포넌트의 기능으로 변환하는 역할을 한다. 기능 변환 패턴은 컴포넌트들 사이의 시멘틱 불일치 문제를 해결한다.

소스 컴포넌트의 Uses 인터페이스에서 타겟 컴포넌트의 Provide 인터페이스로 메시지를 보냈지만, 타겟 컴포넌트에서 제공하는 기능이 소스 컴포넌트가 완벽하게

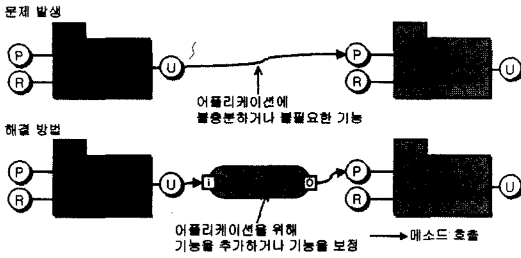

그림 5 기능 변환 패턴의 구조

요구하는 사항이 아니거나, 메시지를 받는 타겟 컴포넌트 입장에서 메시지를 받기 전에 선행 되어야 할 기능이 부족할 수 있다.

만약 타겟 컴포넌트의 기능이 부족한 경우가 아니라, 소스 컴포넌트에서 원치 않는 불필요 기능이 타겟 컴포넌트에 포함되어 있다면, 커넥터는 그 불필요한 기능의 부작용을 보정하는 역할까지 수행한다. 또한 기능을 수정하고 추가하는 기능을 통해 컴포넌트 외부에서 여러 패밀리 프로덕트(Family Product)를 생산하기 위한 피처(Feature) [8]를 수용하기 위해 가변성을 적용할 수 있다.

기능 변환 패턴을 이용하여 컴포넌트간의 기능 불일치 해결 방법은 첫째, 소스 컴포넌트에서 요구하는 기능이 타겟 컴포넌트에서 지원하는 기능보다 많을 경우는 그 부족한 기능을 커넥터에서 추가 구현하여, 소스 컴포넌트를 지원해준다. 둘째, 소스 컴포넌트에서 요구하는 기능보다 타겟 컴포넌트에서 지원하는 기능이 더 많은 경우는 타겟 컴포넌트에서 불필요한 기능을 수행하는 것이다. 그러므로 커넥터에서 타겟 컴포넌트에서 수행하는 그 불필요한 기능을 억제하는 역할을 한다.

셋째, 불필요한 기능을 억제시 그 기능이 다른 API등과 의존관계가 있어, 다른 API에 영향을 주어 부작용(Side Effect)이 발생된다면, 그 부작용을 보정하는 기능을 커넥터에 추가한다. 만약 소스 컴포넌트가 원하는 기능과 타겟 컴포넌트가 제공하는 기능이 맞지 않다면 컴포넌트를 교체한다.

4.3 인터페이스 어뎁터 커넥터 패턴

인터페이스 어뎁터 커넥터 패턴은 그림 6과 같이 소스 컴포넌트에서 예상한 인터페이스와 일치하지 않는 타겟 컴포넌트의 인터페이스를 수용할 수 있도록 하는 패턴이다. 도메인별 표준 인터페이스 스펙이 없는 상황에서 획득된 소스 컴포넌트와 타겟 컴포넌트들 사이의 인터페이스는 불일치가 많다. 어뎁터 패턴은 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환한다. 어뎁터 패턴은 호환성이 없는 인터페이스 때문에 함께 사용할 수 없는 클래스를 사용할 수 있도록 한다[9,10]. 이와 같이 인터페이스 어뎁터 커넥터는 인터페이스의 이름이나 파라메터 타입등의 변경으로 시그니처 오류를 인터페이스 어뎁터로 해결한다.

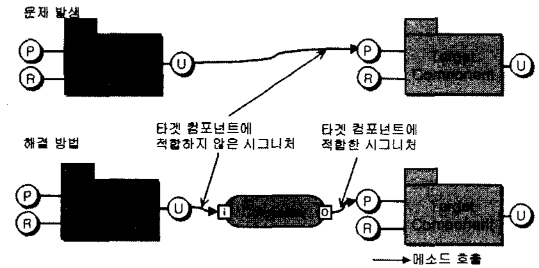

그림 6 인터페이스 어뎁터 패턴의 구조

소스 컴포넌트는 타겟 컴포넌트의 인터페이스와 의존관계를 갖는다. 만약 타겟 컴포넌트가 교체되거나 수정될 때, 소스 컴포넌트는 타겟 컴포넌트에 영향을 받는다. 인터페이스 어뎁터 패턴은 소스 컴포넌트의 변경 없이 타겟 컴포넌트가 가지고 있는 변경된 인터페이스를 수용할 수 있다.

소스 컴포넌트에서는 하나의 오퍼레이션을 호출했지만, 타겟 컴포넌트의 인터페이스가 변경되어, 이 오퍼레이션을 처리하기 위해 2개 이상의 타겟 컴포넌트의 오퍼레이션이 필요한 경우가 있다. 반대로 소스 컴포넌트로부터 전달 받은 2개 이상의 오퍼레이션을 하나로 병합하여 타겟 컴포넌트로 전달 할 수 있다. 그러므로 호출한 메시지를 분리 및 병합하여 인터페이스 어뎁터가 가능하다. 또한 오퍼레이션이 하나의 파라메터가 다형성에 의해 여러 개의 파라메터로 분리되거나 병합되는 경우도 데이타 변환 패턴과 병행하여 인터페이스 불일치도 해결 가능하다.

인터페이스 어뎁터 패턴을 이용하여 컴포넌트간의 인터페이스 불일치 해결 방법은 첫째, 소스 컴포넌트에서 타겟 컴포넌트에 기대하는 오퍼레이션명이 불일치 할 경우에는 커넥터에서 소스 컴포넌트로부터 데이타를 전달 받아, 그 정보와 함께 타겟 컴포넌트의 적절한 오퍼레이션을 호출한다. 둘째, 소스 컴포넌트에서 타겟 컴포넌트로 전달한 파라메터의 순서가 타겟 컴포넌트의 오퍼레이션에서 원하는 파라메터의 순서와 맞지 않으면, 커넥터에서 그 파라메터를 재배열 하여 전달한다.

셋째, 단지 파라메터의 순서 재배열이 아니라, 재배열시 데이타형이 변경되거나 데이타형이 분리되거나 통합돼야 한다면, 데이타 변환 커넥터를 추가 적용하여 타겟 컴포넌트를 호출한다. 넷째, 재배열시 타겟 컴포넌트가 원하는 누락된 데이타가 있다면, 기능 변환 커넥터를 사용하여 타겟 컴포넌트에서 원하는 파라메터로 작성한

후 전달한다.

다섯째, 소스 컴포넌트에서는 타겟 컴포넌트로 메시지 호출 한번으로 하나의 업무가 끝날 것으로 예상했지만, 타겟 컴포넌트가 변경되면서 해당 기능을 제공하는 오퍼레이션이 여러 개의 오퍼레이션으로 분리 되었다면, 커넥터에서는 분산된 타겟 컴포넌트의 여러 API를 호출하면서, 소스 컴포넌트로부터 전달받은 파라메터를 전달하면 된다. 여섯째, 해당 기능을 수행하기 위해 소스 컴포넌트에서는 타겟 컴포넌트에 여러 API를 호출했지만, 타겟 컴포넌트가 업그레이드되어, 한번에 업무를 수행할 수 있는 API로 변경될 경우, 커넥터는 소스 컴포넌트로부터 파라메터들을 모아서 그 정보와 함께 타겟 컴포넌트의 해당 API를 호출한다.

따라서 인터페이스 어뎁터 패턴은 타겟 컴포넌트의 인터페이스가 변경되었을 때 소스 컴포넌트나 타겟 컴포넌트 모두를 수정할 필요가 없다. 인터페이스 어뎁터 패턴은 컴포넌트와 컴포넌트 프레임워크의 실용성을 증가시킨다.

4.4 워크플로 핸들러 커넥터 패턴

워크플로 핸들러 커넥터 패턴은 소스 컴포넌트의 워크플로가 패밀리 멤버가 요구하는 어플리케이션의 워크플로와 불일치 할 때 사용한다. 워크플로 핸들러 패턴은 타겟 컴포넌트를 호출하는 새로운 워크플로를 추가하거나 워크플로의 순서를 변경이 필요할 때 사용된다. 또한 타겟 컴포넌트의 특정 메소드의 호출 순서가 서로 종속적으로 사용되어야 하는 경우에 소스 컴포넌를 변경하지 않고, 커넥터에서 이를 반영할 수 있다.

워크플로는 여러 개의 메시지 호출로 구성된다. 만약 커넥터를 사용하지 않고, 타겟 컴포넌트가 교체되거나, 또는 도메인의 요구사항이 변경된다면, 소스 컴포넌트가 가지고 있는 워크플로는 수정되기 때문에 타겟 컴포넌트등의 외부 환경에 영향을 받는다.

그림 7(a)와 같이 소스 컴포넌트내에 워크플로를 가지고 있는 경우와, 그림 7(b)와 같이 소스 컴포넌트내에 워크플로를 가지고 있지 않은 경우가 있다. 패밀리 멤버별로 많은 워크플로의 가변적 요소를 지원하기에는 비

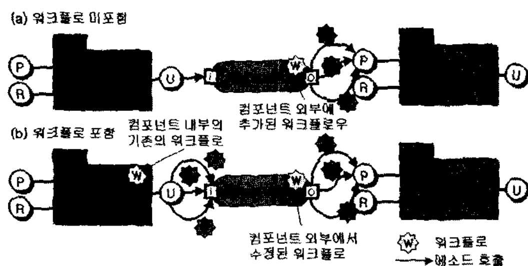

그림 7 워크플로 핸들러 커넥터의 패턴

즈니스 케이스(Business Case)[4]에 맞지 않아 컴포넌트 프레임워크 설계부터 패밀리 멤버에 맞게 커넥터를 교체하여, 가변적 워크플로를 지원하도록 커넥터에 위임하여 설계하는 경우이다.

가령 워크플로내에 두 개의 오퍼레이션이 있고, 이 순서를 변경할 때 커넥터는 먼저 전달 받은 메소드를 보관하고, 두번째 오퍼레이션을 수행하고, 먼저 입력 받은 오퍼레이션을 나중에 타겟 컴포넌트로 전달하면 된다. 하지만, 워크플로를 변경할 때 호출되는 메소드들간의 호출되는 순서나 메시지 호출을 위해 전달될 파라메터를 제공하기 위해 선행되어야 하는 메소들이 있다. 이렇게 메시지간 종속관계가 있는 경우에는 간단히 메소드의 호출 순서만을 변경한다고 그 문제가 해결되지 않는다.

워크플로 핸들러 패턴을 이용하여 소스 컴포넌트의 워크플로 불일치 해결 방법은 첫째, 소스 컴포넌트가 워크플로를 가지고 있지 않다면, 커넥터에서 어플리케이션 구성에 적절한 타겟 컴포넌트를 호출하기 위한 워크플로를 구현한다. 둘째, 소스 컴포넌트가 타겟 컴포넌트로 워크플로에 따라 메시지를 전달했지만, 그 순서가 어플리케이션에 적절하지 않을 경우에는 커넥터에서는 소스 컴포넌트에서 전달한 메시지와 파라메터를 임시 보관한 후 커넥터에서 어플리케이션 구성에 적절한 순서에 의거 타겟 컴포넌트에 메시지를 전달한다.

셋째, 호출되는 타겟 컴포넌트들간에 의존관계가 있는 경우는 타겟 컴포넌트 호출 순서가 정해지는 경우가 있다. 그러므로 호출 순서가 변경되어 발생된 부작용이나, 변경된 순서 때문에 다음 메시지 호출에 필요한 정보를 추출하는 등의 보정작업을 커넥터에서 수행한 후에 적절한 메시지 순서로 호출한다.

이러한 패턴은 워크플로가 변경되더라도, 소스 컴포넌트 내부에 있는 워크플로를 변경하기 위해 소스를 수정하지 않아도 된다. 그러므로 워크플로 핸들러 커넥터는 외부 환경이나 컴포넌트의 변경되더라도, 다른 컴포넌트에 영향을 미치지 않도록 한다.

4.5 예외처리 어뎁터 커넥터 패턴

예외처리 어뎁터 커넥터 패턴은 그림 8와 같이 컴포넌트들 사이에 예외처리 방법이 서로 다른 경우에 예외처리를 적절하게 변경시키는 역할을 수행한다. 타겟 컴포넌트에서는 예외 발생시 예외(Exception) 객체를 전달해주지만, 예외 객체를 받을 수 없는 소스 컴포넌트가 있다면, 문제가 발생된다.

컴포넌트 획득을 할 때, 참조하는 컴포넌트 스펙은 컴포넌트의 오퍼레이션, 파라메터 그리고 그것들의 설명으로 구성된 기능과 문법적인 사항을 포함하고 있다. 하지만, 예외처리에 관련된 구체적인 명세를 얻기 어렵다. 또한 다른 컴포넌트와의 상호작용 테스트를 통한 컴포

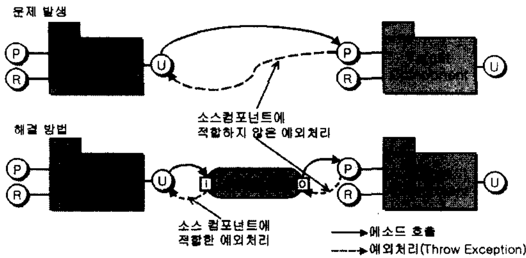

그림 8 예외처리 어뎁터 커넥터의 패턴

넌트 획득보다는 컴포넌트의 단위 테스트를 통해서 컴포넌트를 테스트 후 획득하고 있다. 그러므로 예외처리 사항이 소스 컴포넌트와 타겟 컴포넌트들 사이에 서로 불일치할 수 있다.

소스 컴포넌트는 타겟 컴포넌트의 예외처리 방법과 의존관계를 갖는다. 만약 타겟 컴포넌트가 교체되거나 수정될 때, 소스 컴포넌트는 타겟 컴포넌트의 예외사항에 영향을 받는다. 그림 8과 같이 예외처리 어뎁터 패턴은 소스 컴포넌트의 변경 없이 타겟 컴포넌트의 변경된 예외처리 방법을 수용할 수 있다.

다른 경우로는 타겟 컴포넌트에서 커넥터로부터 받은 요구사항을 서비스를 하고, 예외 발생시 예외 객체를 넘겨주는 것이 아니라, Boolean, String이나 Integer형의 에러 코드를 반환하는 타겟 컴포넌트가 있을 수 있다. 이러한 경우에는 커넥터에서 반환값을 분석해서 소스 컴포넌트에서 요구하는 예외 객체로 만들어서 전달하는 방법이 있다. 다른 경우로는 타겟 컴포넌트에서는 예외 객체를 넘겨 주었지만, 소스 컴포넌트에서 try-catch문으로 예외 사항을 처리 하지 않고, 반환값에 의해 에러를 체크하는 컴포넌트인 경우에는 커넥터에서 소스 컴포넌트에 맞는 반환값으로 변경하여 전달한다.

예외처리 어뎁터 패턴을 이용하여 예외처리 방법의 불일치 해결 방법은 첫째, 소스 컴포넌트가 예외처리를 try-catch문을 사용하지 않고 반환값에 의해 에러 유무를 확인하지만, 타겟 컴포넌트는 적절한 반환값이 아닌 예외객체를 전달한다. 이러한 경우에 커넥터에서 try-catch문으로 타겟 컴포넌트에서 전달한 예외사항을 분석하여, 소스 컴포넌트가 기대하는 반환값으로 변환하여 반환한다.

둘째, 소스 컴포넌트가 예외처리를 try-catch문을 사용하여 예외사항 등의 에러 처리를 하지만, 타겟 컴포넌트는 적절한 예외 객체를 던지지 않고, 에러에 해당하는 반환값을 반환한다. 이러한 경우에는 커넥터에서 타겟 컴포넌트의 반환값을 분석하여, 소스 컴포넌트가 희망하는 예외객체를 생성하여 소스 컴포넌트로 전달한다.

셋째, 소스 컴포넌트가 예외처리를 try-catch문을 사용하여 에러처리 등의 예외사항을 처리를 하고, 타겟 컴포넌트도 예외 발생시 예외 객체를 던지지만, 그 예외 객체가 소스 컴포넌트에서 기대하는 예외객체가 아닌 경우이다. 이러한 경우에 커넥터에서 try-catch문으로 타겟 컴포넌트에서 전달한 예외사항을 분석하여, 소스 컴포넌트가 희망하는 예외객체를 생성하여 소스 컴포넌트로 전달한다.

따라서 예외처리 어뎁터 패턴은 컴포넌트의 예외처리 방법이 변경되었을 때 소스 컴포넌트나 타겟 컴포넌트를 수정할 필요가 없이 커넥터에서 지원이 가능하다. 예외처리 어뎁터 패턴은 컴포넌트와 컴포넌트 프레임워크의 실용성을 증가시킨다.

5. 커넥터 패턴 적용 사례 연구

5.1 전자상거래 시스템의 데이타 변환 커넥터 적용

전자상거래 컴포넌트 프레임워크에서 배송관리 시스템 컴포넌트와 배송관리 비즈니스 컴포넌트가 서로 의존 관계를 갖도록 설계되었다. 시스템 컴포넌트는 시스템의 외부에 기능을 제공하기 위한 퍼사드(façade) 역할을 수행하고, 비즈니스 컴포넌트는 핵심 비즈니스 정보, 규칙들을 관리한다[11]. 배송관리 시스템 컴포넌트에서는 고객 아이디, 이름, 주소 등의 배송정보를 java.util.Hashtable을 사용한다. 하지만, 획득된 배송관리 비즈니스 컴포넌트는 DeliveryVO객체를 사용하여 배송정보를 등록한다. 그러므로 소스 컴포넌트와 타겟 컴포넌트간에 데이타 타입의 불일치가 발생되었다. 이 문제를 해결하기 위해 커넥터는 java.util.Hashtable 타입의 배송정보를 DeliveryVO로 변환하여, 타겟 컴포넌트로 전달해야 한다.

데이타 불일치 문제를 가지고 있는 EJB기반의 컴포넌트를 사용하여, 이러한 데이타 불일치를 해결하는 커넥터에 관한 구체적인 구현 예를 보이면, 소스 컴포넌트와 타겟 컴포넌트를 연결하기 위하여 소스 컴포넌트는 커넥터에 메시지를 전달한다. 소스 컴포넌트의 빈에서 사용하는 데이타 구조에 의해 타겟 컴포넌트로 전달할 데이타를 생성한다. 그림 9와 같이 EJB기반의 소스 컴

```
public class DeliveryCtrlBean implements SessionBean {
  . . .
  public void makeDelivery(String id, String name, . . . ) {
    Hashtable deliveryInfo = new Hashtable();
    deliveryInfo.put("id", id);
    deliveryInfo.put("name", id);
    . . .
    //커넥터 연결
    DeliveryConnector deliveryConnector = new DeliveryConnector();
    //커넥터에 메시지 전송
    deliveryConnector.makeDelivery(deliveryInfo);
  }
}
```

그림 9 데이타 변환 커넥터의 소스 컴포넌트의 구현

```
  public DeliveryConnector() {
    Context context = getInitialContext();
    Object ref = context.lookup("deliveryBean");
    deliveryHome = (DeliveryMgrHome) PortableRemoteObject.narrow(...);
  }
  //데이터 수신
  public void makeDelivery(Hashtable deliveryInfo){
    //데이터 변환
    DeliveryVO deliveryVO = transform(deliveryInfo);
    //데이터 전송
    DeliveryMgr delivery = deliveryHome.create(deliveryVO);
  }
  private DeliveryVO transform(Hashtable deliveryInfo){
    DeliveryVO deliveryVO = new DeliveryVO();
    deliveryVO.id = (String) deliveryInfo.get("id");
    deliveryVO.name = (String) deliveryInfo.get("name");
    . . .
    return deliveryVO;
  }
  private Context getInitialContext() throws Exception {
     . . .
     return new InitialContext(properties);
  }
}
```

그림 10 데이타 변환 커넥터의 구현

포넌트에서 생성된 데이타 구조를 커넥터에 전달하기 위해 커넥터에 연결하고, 생성된 커넥터로 java.util.Hashtable 타입의 데이타를 메시지를 전달한다.

데이타 변환 커넥터는 그림 10과 같이 구현된다. 커넥터는 소스 컴포넌트로부터 메시지를 전달 받을 포트 역할의 메소드를 구현한다. 그리고 소스 컴포넌트로부터 전달 받은 데이타를 타겟 컴포넌트의 자료 구조에 종속적인 데이타로 변환을 하는 로직을 구현한다.

커넥터에서는 소스 컴포넌트로부터 전달 받은 데이타를 타겟 컴포넌트의 자료 구조에 적합하도록 변경한 후에 타겟 컴포넌트의 EJB 홈 인터페이스 또는 컴포넌트 인터페이스를 통해 메시지를 전달한다.

5.2 기상예보 시스템의 데이타 변환 커넥터 적용

기상 예보 컴포넌트 프레임워크에서 온도 수집 컴포넌트와 기상 예측 컴포넌트가 의존관계를 가지고 있다. 온도 수집 컴포넌트와 기상 예측 컴포넌트 모두가 float 타입으로 온도를 관리 하지만, 온도 수집 컴포넌트는 섭씨(C)를 사용하는 국가에서 생산되어, 온도를 섭씨로 관리하고, 기상 예측 컴포넌트는 화씨(F)를 사용하는 국가에서 생산되어 온도는 화씨로 관리된다.

이럴 경우 온도 수집 컴포넌트에서 기상 예측 컴포넌트가 가지고 있는 setTemperature (temperature : float) 오퍼레이션을 호출하므로 조립하는 과정에서 시그니쳐 불일치의 문제가 발생되지 않지만, 데이타의 의미에 문제가 생겨 예측치 못한 문제가 발생된다. 이러한 경우가 발생되면, 커넥터는 'F = 1.8 * C + 32' 공식에 의하여 온도수집 컴포넌트로부터 전달 받은 섭씨 온도를 화씨 온도로 변환하여 기상예측 컴포넌트로 전달한다.

5.3 호텔관리 시스템의 기능 변환 커넥터 적용

호텔관리 컴포넌트 프레임워크에서 예약관리 컴포넌트와 고객관리 컴포넌트가 의존 관계를 가지고 있다. 호텔관리 어플리케이션을 만들기 위해 사용되는 고객관리 컴포넌트는 협력 여행사의 고객관리 시스템을 사용한다. 예약 관리 컴포넌트는 고객의 주민등록 번호와 호텔 객실의 정보 등으로 예약을 수행한다. 이때 예약 관리 컴포넌트는 내부적으로 고객 확인 로직이 없기 때문에, 고객의 주민등록 번호 체크 비트 확인 및 호텔의 고객 평가 등급에 이상이 없는 고객의 주민등록 번호 입력만을 요구한다.

그러나 타사의 시스템과 연동되는 고객 관리 컴포넌트는 주민등록 번호 비트 확인 기능 및 호텔 객실 예약에 이상이 없는 고객인지를 확인하는 평가 기능을 포함하지 않았다. 따라서 커넥터는 기능 변환 커넥터를 사용하여 고객 컴포넌트로부터 전달 받은 고객의 주민등록 번호를 이용하여, 주민등록 번호의 체크 비트를 확인하고, 호텔의 고객평가 시스템을 통해 객실 예약이 가능한 고객일 경우에만 예약관리 컴포넌트로 메시지를 전달한다.

고객 관리 시스템으로부터 전달 받은 고객 정보로 호텔 객실을 예약 할 때 주민번호가 맞지 않거나, 호텔 객실을 사용할 수 없는 고객이라면, 예외사항을 발생시킨다. 이때 타사의 고객 관리 시스템에 적절한 예외 사항을 발생시키기 위해서 예외처리 어뎁터 패턴을 병행해서 커넥터를 구성한다.

5.4 호텔관리 시스템의 인터페이스 어뎁터 커넥터 적용

호텔관리 컴포넌트 프레임워크에서 어플리케이션 계층이 예약관리 컴포넌트와 의존 관계를 가지고 있다. 호텔 예약 어플리케이션은 예약관리 컴포넌트 인터페이스의 makeReserve (memberID : String, name : String, SSN : String) 오퍼레이션을 사용하여 호텔을 예약한다. 그러나 예약관리 컴포넌트가 교체되면서 예약관리 컴포넌트의 오퍼레이션이 createReservation(memberID : long)로 변경되었다.

이러한 인터페이스 불일치 상황이 발생되면, 인터페이스 어뎁터 커넥터는 예약 어플리케이션으로부터 make-Reserve 메시지와 아이디, 이름, 주민등록번호를 전달 받아, 변경된 예약관리 컴포넌트에 createReservation 메시지와 전달 받은 파라메터들 중에 String 타입의 memberID를 데이타 변환 패턴을 병행 사용하여 long 타입으로 변환하여 예약관리 컴포넌트로 전달한다.

5.5 전자상거래 시스템의 워크플로 핸들러 커넥터 적용

전자상거래 컴포넌트 프레임워크에서 배송관리 컴포넌트와 결재관리 컴포넌트가 의존 관계를 가지고 있다. 요구사항 명세서에 고객이 제품을 구매한 후에 결재가 완료되면 그 다음에 배송 받을 고객의 이름, 주소지 등을 입력 받아 배송관리를 하게 된다. 그리고 이러한 순서에 의해 만들어진 배송 컴포넌트는 해당 배송지 입력이 마무리되면, 주문 컴포넌트로 해당 주문이 완료되었

다는 메시지를 전달한다.

하지만, 요구사항 명세서의 변경으로 인해, 제품을 구매하는 방법이 배송 받을 고객의 정보가 완벽한 경우에만, 결재처리 시스템을 가동하는 순서로 변경되었다. 이러한 워크플로 불일치 상황이 발생되면, 워크플로 핸들러 커넥터는 결재처리 요청 정보를 보관하고, 요청 받은 배송 정보를 배송관리 컴포넌트로 전달하고, 보관하고 있던 결재정보를 이용하여 결재관리 컴포넌트로 메시지를 전달한다.

이때 배송관리 컴포넌트로 전달하면, 배송 관리 컴포넌트 내부에서 주문이 완료된 상태로 변경이 되어, 결재처리시 이미 결재 완료 상태로 인식 한다. 그러므로 결재관리 컴포넌트를 호출하기 전에 주문완료 상태를 주문진행 상태로 변경하는 보정 작업을 추가한 후에 결재관리 워크플로를 수행한다.

6. 평 가

최적의 컴포넌트 프레임워크를 표현하기 위해서는 컴포넌트와 커넥터가 서로 결합되어 구성되어야 한다. 구성은 컴포넌트와 컴포넌트들과 함께 결합된 커넥터들의 집합이다[4]. 커넥터 구성은 커넥터를 직렬식 연결 또는 병력식 연결 모두 가능하다. 본 논문에서 제안한 커넥터 패턴들은 커넥터 내부에서 여러 가지의 패턴을 조합해서 사용할 수 있다. 조합된 커넥터들은 보다 복잡한 방식에 의해 다양한 서비스를 제공한다.

컴포넌트 설계자는 다른 컴포넌트들의 구체적인 내부 구조나 관계를 알지 않아도 되고, 컴포넌트는 최종 어플리케이션을 만들기 위해 다른 컴포넌트들과 다양한 가능성을 가지고 조립된다. 따라서 커넥터는 정확하게 정의되어야 하고, 컴포넌트 설계자는 그들과의 관계에 따라야 한다.

커넥터를 구성하는 기법으로는 커넥터를 식별하기 위해 패밀리 맴버의 요구사항 명세서와 컴포넌트 명세서를 통해 그들의 의존관계를 분석하여 본 논문에서 제안한 커넥터 패턴으로 커넥터를 식별한다. 커넥터를 사용하는 기법은 컴포넌트와 컴포넌트의 의존관계를 줄임으로써 컴포넌트의 재사용성, 활용성을 높일 수 있기 때문에, 컴포넌트와 컴포넌트 사이에 적합한 표준 커넥터 패

표 2 타 커넥터 타입과의 비교

문자 : 커넥터 이름, O : 직접지원, △ : 부분적 또는 간접적 지원, - : 미흡 또는 미지원

[illegible]	[illegible]	Helgo	Mehta	제안한 패턴
커넥터의 솔루션을 제공	△	△	-	O
레거시 시스템 지원	△	-	Convers.	Interface Adapter
커넥터간의 연결	Splice	Composition	-	Interface Adapter Workflow Handler
세션 기반의 연결 지원	Sessionize	-	Linkage	△
파라메터 타입 변경	Data Transfer	-	Comm., Convers., PC,Event, DA,Stream, Adaptor	Data Transformer
파라메터의 의미 변경	-	-	-	Data Transformer
파라메터의 반환값 변경	△	-	Comm., Convers., DA	Data Transformer
파라메터의 반환값의 의미 변경	-	-	-	Data Transformer
필요한 기능 추가,변경 기능	-	Interface	-	Functional Transformer
기능 사용시 발생한 부작용 해결을 위한 보정 기능	-	-	-	Functional Transformer
파라메터의 순서 적절히 배치	Splice	-	DA, Convers.	Interface Adapter
오퍼레이션 통합 및 분할 기능	Aggregate	Composition	-	Interface Adapter
새로운 인터페이스 적용 기능	Add a Role	-	-	Interface Adapter
워크플로 추가 및 수정	Aggregate	-	Coord., PC,Event, Arbitr.	Workflow Handler
예외처리 적절히 변경	-	-	-	Exception Adapter
이벤트 기반의 연결	△	O	-	△
상호작용을 위한 프로토콜 적용	-	Interface	Adaptor	Interface Adapter

턴을 식별한다. 식별된 커넥터를 분석하기 위해, 커넥터가 수행해야 할 기능적, 비기능적 요구사항을 통해서 커넥터 요구사항을 명세 한다. 커넥터의 요구사항을 만족하는 커넥터를 개발하기 위해 커넥터의 여러 패턴을 적용하여 컴포넌트에 매핑시킨다.

Latchem은 컴포넌트 인프라스트럭쳐 모델을 제시하고 있는데, 컴포넌트들 사이에 불필요한 결합도가 생기지 않도록 컴포넌트들의 역할이 설계되어야 한다[4]. 중계자(Mediator) 패턴[8,9]의 장점을 적용한 커넥터는 컴포넌트간의 상호작용을 캡슐화하여 컴포넌트들간의 의존 관계를 분리함으로써 상호작용만을 독립적으로 확대할 수 있다. 본 논문에서는 여러 도메인을 통해 컴포넌트간의 불일치 문제를 제시하고, 이 문제를 커넥터 패턴을 적용하여 컴포넌트 변경없이 효율적이고, 재사용성을 높였다.

본 논문에서 제안한 커넥터 패턴과 다른 논문의 커넥터를 비교하면 표 2와 같다. Spitznager와 Garlan의 논문[12]에서는 커넥터의 유형을 데이타 변환(Data transform), 결합(Splice), 기능 추가(Add a role), 세션화(Sessionize)와 집합 변환(Aggregate transform)의 5개로 나누었다. 그러나 구체적인 사례 제시가 없고, 커넥터를 병렬적으로 연결하는 방법 위주로만 제시되어 있다.

Helgo와 George는 소프트웨어 아키텍처에서의 포트, 역할, 호출과 outgoing 메시지를 정의하였다[13]. Helgo는 포트와 역할들을 사용하여 복합관계(Composition)와 상속(Inheritance) 관계로 분류하여, 이를 기반으로 인터페이스를 정의한 후 JavaBeans 컴포넌트 모델[14]을 기반으로 연결하는 연구를 하였다. Helgo의 논문은 커넥터를 위한 실제 자바 코드를 사용하여 커넥션 관계를 잘 표현하였다. 그러나 Helgo의 기법들은 JavaBeans의 이벤트간의 연결에 국한되었다.

Mehta는 커넥터의 서비스 카테고리를 제안하였다[15]. 제안된 서비스 카테고리는 전달(Communication), 대등(Coordination), 변환(Conversion), 용이성(Facilitation), 절차적 호출(Procedure Call), 이벤트(Event), 데이타 처리(Data Access), 결합(Linkage), 스트림(Stream), 조정자(Arbitrator), 어댑터(Adaptor), 분배자(Distributor)로 구성된다. 그러나 Mehta의 논문은 커넥터들을 복합 사용하기 위한 각 카테고리의 구체적인 지침이 정의되어있지 않고, 제안한 여러 커넥터의 서비스 카테고리들을 컴포넌트 기반 개발(CBD)을 위한 커넥터의 역할별로 분리하면 카테고리가 중복되는 경우가 많다. 특히 파라메터 타입을 연결하는 카테고리는 7가지가 중복되고 있다.

본 논문에서 제안한 5가지의 커넥터 패턴과 커넥터 패턴 적용 기법을 통해 컴포넌트와 컴포넌트간의 의존 관계 때문에 재사용성과 활용성이 저하되는 것을 컴포넌트 프레임워크 기반의 커넥터를 사용함으로써 완벽한 컴포넌트를 획득 못하거나, 컴포넌트가 변경되어도 다른 컴포넌트에 영향을 주지 않아, 컴포넌트의 재사용성과 활용성을 높일 수 있다.

7. 결 론

CBD는 컴포넌트를 이용한 효율적인 재사용 기술임을 보였다. 하지만, 획득된 COTS 컴포넌트는 요구하는 기능의 일부분만을 제공하고, 필요하지 않은 기능이 포함되어 있으며, 인터페이스 시그니처 및 워크플로 불일치한 경우가 발생된다. 따라서, 신중히 고려하여 획득한 컴포넌트의 집합이라도 컴포넌트 상호간에 완벽히 만족되어 연결되기는 힘들다[4].

이러한 문제점들은 PLE의 프레임워크는 컴포넌트들의 집합과 그것들을 미리 조합할 수 있는 컴포넌트간의 관계, 불일치 문제를 중재해주는 커넥터를 포함한다[5]. PLE기반의 컴포넌트 프레임워크는 관련된 컴포넌트들과 그들의 관계를 관리하는 커넥터를 사용함으로써 기존 컴포넌트보다 더욱더 큰 재사용성을 갖도록 하였다.

컴포넌트 프레임워크에서 컴포넌트들은 연관관계, 의존관계의 연결로 조립된다. 특히 커넥터는 컴포넌트간의 관계를 연결 할 뿐만 아니라, 완벽하게 서로 만족하는 COTS 컴포넌트를 획득하지 못한 경우에도 보정하여 사용 가능하도록 한다.

본 논문에서는 컴포넌트와 커넥터를 조립하기 위한 패턴 및 패턴 적용 기법을 실용적으로 제시하였다. 또한 제안된 커넥터 패턴을 사용해서 컴포넌트 프레임워크 설계 시 컴포넌트간의 결합도를 낮추는 커넥터를 설계 할 수 있다. 각각의 패턴은 컴포넌트를 교체하기에 유용하고, 컴포넌트를 조립할 수 있게 하였다. 따라서 제안된 커넥터는 컴포넌트의 재사용성과 활용성을 향상시킨다.

참 고 문 헌

[1] Kim, Soo, "Lessons Learned from a Nation-wide CBD Promotion Project," Communications of The ACM, vol. 45, Issue 10, Oct. 2002.

[2] Kim, Soo and Park, Ji, "A Practical Quality Model for Evaluating COTS Components," Proceedings of International Association of Science and Technology for Development(IASTED) International Conference on Software Engineering(SE'2003), Innsbruck, Austria, 2003.

[3] Anriruddha Gokhale, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," Communication of ACM, Vol.45.No.10, October 2002.

[4] Heineman and Councill, Component-Based Soft-

ware Engineering, Addison Wesley, 2001.

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

[6] L. Geyer and M. Becker, "On the Influence of Variabilities on the Application-Engineering Process of a Product Family," SPLC2 2002, LNCS 2379, 2002.

[7] D'Souza, Wills, Objects, Components, and Frameworks whit UML, Addison Wesley, 1998.

[8] Booch and Rumbaugh, The Unified Modeling Language User Guide, Addison Wesley, 1998.

[9] Gamma, E, Helm, R, Design Patterns, Addison Wesley, 1995.

[10] Carey, Carlson, SanFrancisco™ Design Patterns, Addison Wesley, 2000.

[11] J. Chessman and J. Daniels, UML Components, Addison Wesley, 2001.

[12] B. Spitznagel and D. Garlan, "A Compositional Approach for Constructing Connectors," WICSA '01, 2001.

[13] Helgo O and George H, "Composition and Interfaces within software architecture," Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, November 1998.

[14] Sun Microsystems Inc., "JavaBeans™ Specification Ver 1.01," Sun Microsystems Inc., August, 1997.

[15] N. R. Mehta, N. Medvidovic and S. Phadke, "Towards a taxonomy of software connectors," Proceedings of the 22nd international conference on Software engineering, June 2000.

민 현 기

1999년 건양대학교 전자계산학과 공학사 2001년 숭실대학교 컴퓨터학과 공학석사 2001년~현재 숭실대학교 컴퓨터학과 박사과정. 관심분야는 소프트웨어 아키텍처, 컴포넌트 기반 개발(CBD), 모델 기반 아키텍처(MDA), 제품-라인 공학 (PLE)

김 수 동

1984년 Northeast Missouri State University 전산학 학사. 1988년/1991년 The University of Iowa 전산학 석사/박사. 1991년~1993년 한국통신 연구개발단 선임연구원. 1994년~1995년 현대전자 소프트웨어연구소 책임연구원. 1995년 9월~현재 숭실대학교 컴퓨터학부 부교수. 관심분야는 객체지향 개발 방법론, 컴포넌트 개발 방법론, 소프트웨어 아키텍처 및 MDA, Embedded S/W