論文
2-4-4

비즈니스 컴포넌트의 상호운용성을 위한 *Intermediator* 패턴
(*Intermediator* Pattern for Interoperability of Business Components)

이 창 목* 유 철 중** 장 옥 배*** 문 윤 호****
(Chang-Mog Lee) (Cheol-Jung Yoo) (Ok-Bae Chang) (Yun-Ho Moon)

요 약

기존의 비즈니스 컴포넌트들간의 의사소통은 Façade 패턴을 따르는 방식이었다. 그러나 Façade 패턴은 대표객체 하나만으로 소속객체 밖의 모든 객체들과의 의사소통을 전달하는 방식으로서 기존의 Interface 패턴과 거의 비슷하며, 반드시 대표객체를 통해서만 의사소통이 가능하므로 여러 객체들의 동시 접근시 과부하로 인하여 결과 반환에 속도저하의 단점을 가지고 있다. 따라서 본 논문에서 제안한 *Intermediator* 패턴은 이러한 과부하 현상으로 인한 저효율성의 컴포넌트 의사소통 방식을 개선하여 여러 인터페이스를 통한 속도저하의 단점을 보완하였고, *Intermediator* 객체내의 메소드 구체화를 통하여 실제 행위만을 관리하는 *Intermediator* 객체를 두어 모든 행위는 *Intermediator*에서 이루어지도록 하였다. 그렇기 때문에 본 논문에서 제안한 *Intermediator* 패턴은 객체간의 결합도를 줄일 수 있고 또한 응집력은 높일 수 있으며, 다른 여러 객체들의 참조에도 여러 인터페이스를 통하여 유연하게 대처할 수 있다는 장점이 있다.

ABSTRACT

Software design patterns are reusable solutions to recurring problems that occur during software development. As programmers gain experience, they recognize the similarity of new problems to problems they have solved before. With even more experience, they recognize that solutions for similar problems follow recurring patterns.

In this paper, as one of these reusable design-patterns, the *Intermediator* pattern for the efficient communication between business component is designed and implemented. Existent business components were way that follow Façade pattern. But Façade pattern is almost like the existing Interface pattern in that the communication of all the object outside the set is done by only one representative object, and this causes the traffic system to be overloaded. Therefore, the *Intermediator* pattern supplements the traffic overloads by improving the inefficient system of business components' communication through multi-interface, and make all communication behavior between objects done by *Intermediator* object which controls all actual behavior by way of *Intermediator* method implementation.

Consequently, the *Intermediator* pattern is designed and implemented in this paper can bring down the level of coupling and raise the cohesion among objects, and refer to many of other object flexibly through several interfaces.

* 학생회원 : 전북대학교 컴퓨터통계정보학과 박사과정
** 종신회원 : 전북대학교 컴퓨터과학과 조교수
*** 정회원 : 전북대학교 컴퓨터과학과 교수
**** 정회원 : 전주기전여자대학 부교수

논문접수 : 2001. 4. 4.
심사완료 : 2001. 4. 20.

1. 서론

컴퓨터 발전과 인터넷의 급성장은 소프트웨어공학의 발전과 함께 이루어졌다. 1970년대 말부터 많은 관심이 되었던 소프트웨어공학의 발전은 80년대 후반부터 객체지향 방법론에 의해 가속화되었으며, 90년대에 이르러서는 컴퓨터 운영체제에서부터 제어기, 편집기에 이르기까지 대부분의 소프트웨어 개발에 사용되어졌다

객체지향 방법론을 소프트웨어 개발에 효과적으로 활용하기 위해서는 소프트웨어 객체에 대한 실질적인 개념이 요구되어지며 기존의 프로시저 기반의 절차적 개발방법론과는 다른 규격화된 패턴이 절실히 필요하다. 프로그램의 실제 구현 전에 전체적인 로직과 알고리즘을 확고히 구축해야함은 물론, 상황에 따라 올바른 객체 솔루션의 선택과 적용이 요구된다[1].

소프트웨어 개발에 있어서의 설계 패턴은 이러한 객체지향적 설계 개념을 이론적으로 제시하는 데 그치지 않고 실제 소프트웨어 개발에 적용이 가능하도록 구체적으로 규격화한 것을 말하며 객체지향 방법론의 가장 큰 장점인 재사용성(Reusability)을 극대화시켜 실제 구현 과정에서의 솔루션으로 제시 가능하도록 한 것이다. 재사용성이나 확장성을 고려하지 않은 설계를 바탕으로 소프트웨어를 제작한다면 변동사항이나 추가 요구사항을 처리할 때 다시 설계하는 최악의 경우가 발생한다[2]. 따라서 가장 이상적인 경우는 개발자가 설계했던 설계 구조를 다시 수행할 솔루션에 최대한 적용하는 것이다. 설계 패턴은 특수한 설계 문제를 해결하는 동시에 객체지향 설계를 보다 유동적이고 재사용 가능한 형태로 만들어 준다 [3, 4].

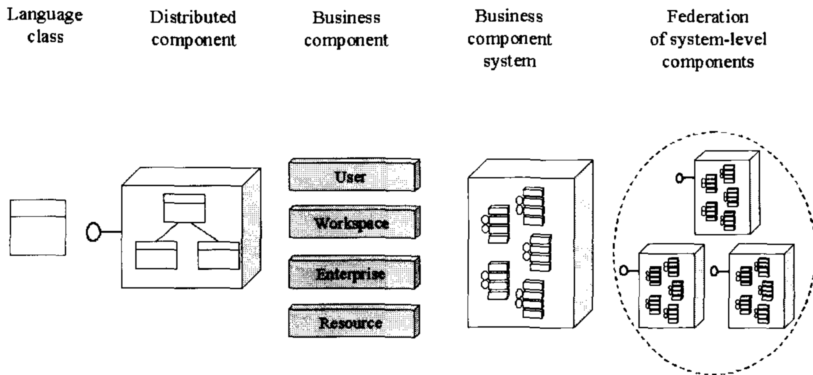
설계 패턴은 설계뿐만 아니라 나아가 시스템 구조의 재사용성을 향상시킬 수 있다. 설계 패턴의 장점으로 개발 결과에 따른 산출물들을 보다 향상시킬 수 있으며, 불명확한 클래스의 기능 및 객체간의 부적절한 연관 관계 등을 제거해 현존하는 시스템에 대한 유지보수도 용이하게 할 수 있다. 각 패턴은 자신의 특성을 잘 표현하는 이름을 갖고 있어 개발을 진행하거나 문서화 작업을 하면서 개발자간의 의사소통에 도움을 준다. 그리고 패턴은 특정 설계 형태나 구현 코드 자체를 제시하는 것은 결코 아니다. 그러나 다양한 솔루션으로 적용 가능한 일종의 템플릿 형태로 되어 있어 설계에 대한 추상화된 상위 단계의 솔루션, 즉 클래스와 객체에 대한 배치(arrangement) 등을 제시하는 역할을 한다[5]. 그러나 패턴이 장점만 있는 것은 아니다. 패턴을 익히는데 시간과 노력이 필요하며 패턴을 잘못 적용할 경우 오히려 설계와 구현을 어렵게 할 수도 있기 때문이다[6]. 하지만 패턴을 정확히 이해하고 적절히 사용한다면 객체지향 설계의 장점인 재사용을 기반으로 개발 시스템의 유연성, 확장성 및 이식성을 향상시킬 수 있다. 따라서 본 연구에서는 컴포넌트 상호 운용성을 위한 여러 패턴 중에서 Façade 패턴을 개선한 *Intermediator* 패턴을 제안하고 이를 실제 구현 및 적용하여 컴포넌트간의 상호운용성에 효율성을 증가시키는데 그 목적이 있다.

2. 관련 연구

2.1 컴포넌트의 단위별 구성 분류

컴포넌트는 구성되는 범위에 따라서 여러 가지 단계로 분류가 가능하다. 각 분류된 컴포넌트들간의 상호운용에 필요한 다양한 패턴이 존재할 수 있다. 따라서 본 절에서는 컴포넌트의 다양한 구성원리에 대하여 알아보고자 한다.

컴포넌트의 단위별 구성을 분류해보면 '랭귀지 클래스(Language class)', '분산 컴포넌트(Distributed component)', '비즈니스 컴포넌트(Business component)', '비즈니스 컴포넌트 시스템(Business component system)', '시스템 레벨 컴포넌트 연합(Federation of system-level component)' 등으로 구분을 할 수 있다[7]. 작은 단위의 '랭귀지 클래스'로부터 크게는 시스템 단위의 컴포넌트 연합까지 다양한 컴포넌트 구조를 가질 수 있는데 이러한 다양한 컴포넌트의 구성에 따른 그들간의 여러 가지 상호운용 패턴이 존재할 수 있다. 본 논문에서는 이러한 다양한 컴포넌트의 구성 중에서 비즈니스 컴포넌트 시스템간의 상호운용 패턴을 제안한다. [그림 1]은 이러한 컴포넌트의 단계별 분류를 보여준다.

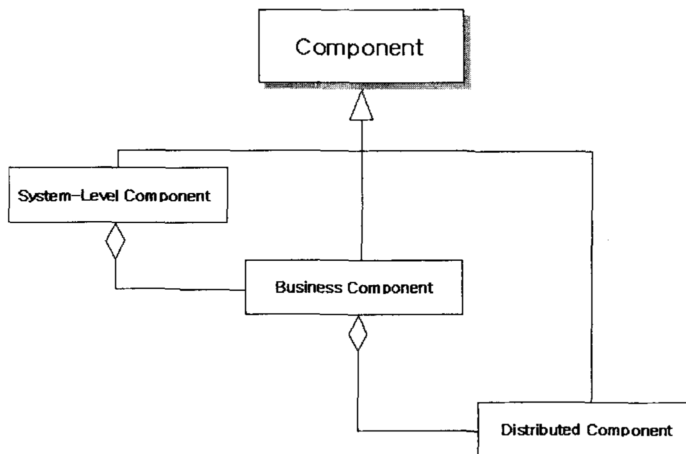


[그림 1] 컴포넌트의 단계별 분류

[Fig. 1] Step classification of component

[그림 1]에서 잘 나타나 있듯이 '랭귀지 클래스'는 객체지향 언어에서 공통적 성질을 가진 객체들의 집합의 클래스를 의미한다. '분산 컴포넌트'는 이러한 클래스들이 모여 최소한의 컴포넌트개념을 갖는 단위이다[8]. '비즈니스 컴포넌트'는 클래스들의 모임인 분산컴포넌트들이 다수의 상호 관계를 가지며 모인 컴포넌트이다. 이러한 비즈니스 컴포넌트는 또한 'User', 'Workspace', 'Enterprise', 'Resource'의 4가지의 계층 구조를 가지는데 'User' 층은 사용자 인터페이스에서 이벤트를 발생시키는 계층을 의미하고, 'Workspace' 층은 발생된 이벤트의 결과를 처리하여 나타내는 인터페이스를 의미한다. 또한, 'Enterprise' 층은 비즈니스 로직에 관련된 계층이며 마지막으로, 'Resource' 층은 데이터를 나타내는 데이터베이스를 의미한다.

그 다음 컴포넌트의 분류로는 이러한 비즈니스 컴포넌트가 다수의 상호 관계를 가지며 소규모 시스템을 구성하는 '비즈니스 컴포넌트 시스템'이 있으며 마지막으로 이러한 소규모 '비즈니스 컴포넌트 시스템'이 다수의 상호 관계를 가지며 연동을 하는 '시스템 레벨 컴포넌트 연합'을 구성한다. [그림 2]는 컴포넌트의 구성 단위별 계층을 UML 표기법에 의해 표현한 그림이다[9, 10].



[그림 2] 컴포넌트의 단위별 구성 원리

[Fig. 2] Unit composition principle of component

2.2 Facade 설계 패턴

Façade 패턴은 대표객체 하나를 두어 다른 그 외 모든 객체들과 대화할 수 있도록 함으로써 관련 객체 집합에 접근을 간단하게 한다[11]. 본 절에서는 컴포넌트간 의사소통에 사용되는 Façade 패턴의 특징과 그에 따른 단점에 대해 알아본다.

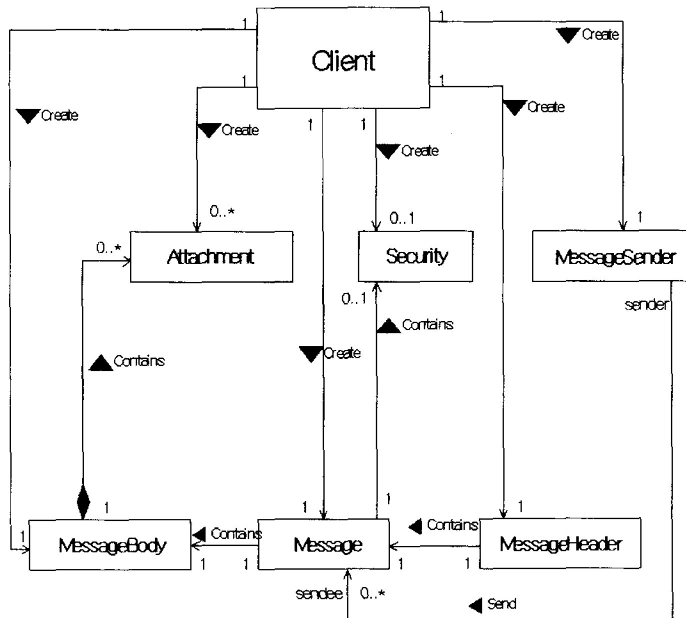
2.2.1 Façade 패턴의 구조

Façade 패턴을 설명하기에 전에 먼저 E-mail 생성 다이어그램을 통해 기존의 복잡성을 인식하고 Façade 패턴의 필요성을 알아보기로 한다.

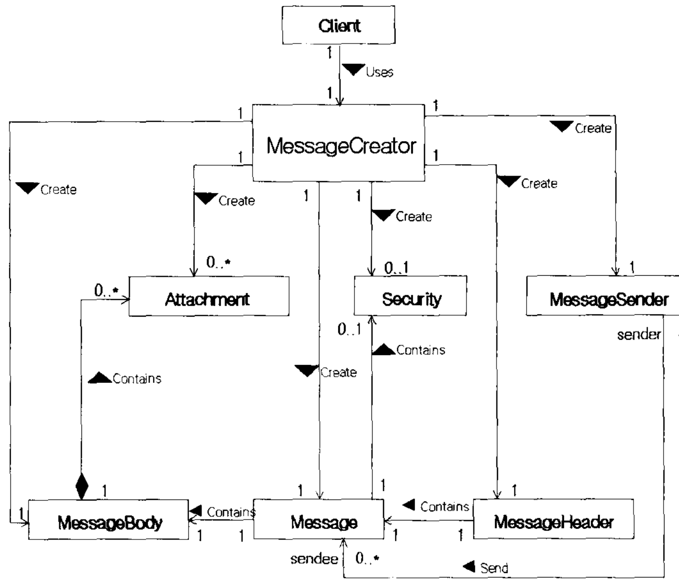
[그림 3]은 E-mail 생성 다이어그램이다[4]. 'MessageBody' 클래스는 메시지의 내용을 포함하는 인스턴스를 소유하는 클래스이다. 'Attachment' 클래스는 메시지 내용 객체에 첨부될 수 있는 메시지 첨부를 포함하는 인스턴스를 소유하는 클래스이다. 'MessageHeader' 클래스는 E-mail 메시지에 대한 header 정보(~에게, ~로부터, 제목 등등)를 포함하는 인스턴스를 소유하고있는 클래스이다. 'Message' 클래스는 'MessageHeader' 객체와 'MessageBody' 객체를 함께 연결하는 인스턴스를 소유하는 클래스이다. 'Security' 클래스는 메시지에 전자 서명을 추가하는데 이용되는 인스턴스를 소유하는 클래스이다. 마지막으로 'MessageSender' 클래스는 E-mail을 목적지나 또 다른 서버에 전달하는 것을 담당하는 서버에 'Message' 객체 보내는 것을 담당하는 인스턴스를 소유하고있는 클래스이다. [그림 3]은 이러한 클래스들과 'Client' 클래스의 관계를 보여주고 있다. 그러나 이러한 구조는 'Client' 클래스가 적어도 여섯 개의 클래스를 알고있어야 한다는 점과 재사용이 어렵다는 단점이 있다.

2.2.2. Façade 패턴의 적용 예

[그림 4]는 이러한 모순을 보완한 Façade 패턴의 구조를 나타내고 있다[4]. 'Client' 클래스는 'MessageCreator' 클래스만 알면 된다. 게다가 'MessageCreator' 클래스의 내부로직은 어떤 특정 주문으로 E-mail 메시지의 일부를 생성해야만 하는 것으로부터 'Client' 클래스를 보호할 수 있다. 그러나



[그림 3] E-mail 생성 다이어그램
[Fig. 3] E-mail creation diagram



[그림 4] 재사용 가능한 E-mail 생성 다이어그램
[Fig. 4] Reusable E-mail creation diagram

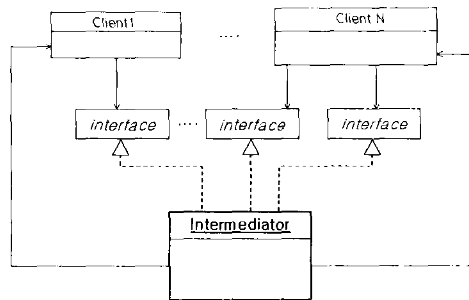
Façade 패턴은 반드시 대표객체를 통해서만 다른 객체와 대화가 가능하므로 Interface 패턴과 거의 유사하며 대표객체는 다른 객체들에 대한 정보만 가지고 있을 뿐 구체적인 행위(메소드)는 가지고 있지 않다. 따라서 객체간의 대화로부터 결과를 얻는데 반드시 대표객체를 통해서만 가능하므로 클라이언트가 다수 존재하여 동시에 객체 참조시 결과 반환시간이 길어져 비효율성을 초래한다.

3. *Intermediator* 패턴 설계

제 2장 관련연구에서 Façade 패턴의 특징과 단점에 대하여 기술하였다. 이러한 특징을 토대로 Façade 패턴이 갖는 단점을 개선한 *Intermediator* 패턴의 구조도는 [그림 5]와 같다.

3.1 *Intermediator* 패턴의 구조

본 논문에서 제안한 *Intermediator* 패턴의 목적은 비즈니스 컴포넌트간의 상호운용을 위해 최적화 된 패턴으로서 다수의 클라이언트의 요청에도 다수의 인터페이스를 이용하여 대표객체를 이용해 컴포넌트들간의 상호운용시보다 요구사항 처리시간에 있어 보다 더 효율적으로 처리하기 위하여 제안하였다.



[그림 5] Intermediator 패턴의 구조도
[Fig. 5] Structure of Intermediator pattern

[그림 5]는 다수의 비즈니스 컴포넌트들이 존재하여 상호운용을 하는데 있어 보여질 수 있는 구조도를 나타내고 있다. 다수의 비즈니스 컴포넌트에는 다수의 interface가 존재하여 이러한 interface를 구현하고있는 *Intermediator* 클래스에서는 각각의 비즈니스

컴포넌트(Client 1, Client 2, …, Client N)들의 인스턴스들을 등록시킨다. 이렇게 함으로써 각각의 비즈니스 컴포넌트들에 대한 정보가 *Intermediator* 클래스에 모두 등록이 되는 것이다. 이렇게 등록된 정보는 각각의 비즈니스 컴포넌트들로 하여금 서로의 정보를 알 필요가 없도록 한다. 이렇게되면 각각의 비즈니스 컴포넌트들은 서로에 대하여 비종속적이 될 수 있다. 따라서 하나의 비즈니스 컴포넌트가 또 다른 비즈니스 컴포넌트를 호출할 때 이미 모든 비즈니스 컴포넌트들에 대한 정보를 보유하고있는 *Intermediator* 클래스를 참조하면 된다. 이러한 방법은 객체지향의 기본 원리인 각 객체들간의 관계가 독립적으로 유지될 수 있다는 특징을 지닌다. 다음절은 이러한 Intermediator 패턴의 세부 내용에 대하여 자세히 알아본다.

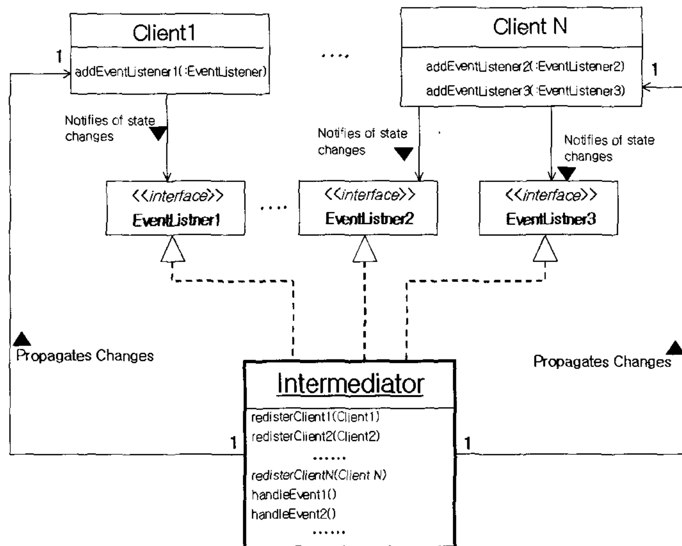
3.2 Intermediator 패턴의 특성

본 절에서는 *Intermediator* 패턴의 클래스 다이어그램과 객체관계도 비교, *Intermediator* 패턴의 프로토타입에 대하여 알아본다.

3.2.1 *Intermediator* 패턴의 클래스 다이어그램

[그림 6]은 본 논문에서 제안하는 *Intermediator* 패턴의 클래스 다이어그램이다. 각 클래스의 기능과 역할은 다음과 같다.

먼저 Client 클래스는 *Intermediator* 클래스에 종속적이다. 모든 Client 클래스는 <<interface>> 클래스를 통하여 *Intermediator* 클래스에서 구체화되어 결과를 돌려 받게 된다. 그러므로 Client 클래스는 *Intermediator* 클래스에 대해 두 가지 종류의 종속성을 가지게 된다. 첫째는 어느 한 클래스는 다른 클래스의 정보를 인식할 필요가 있기 때문에 다른 클래스의 승인을 얻기 위해 *Intermediator* 클래스와 종속관계를 유지할 필요가 있으며, 둘째는 다른 클래스들은 자신의 상태 변화를 다른 객체들에게 알릴 필요성이 있어 이러한 정보를 *Intermediator* 클래스에게 전달하기 때문에 *Intermediator* 클래스와 종속관계를 유지한다. *Intermediator* 패턴의 핵심 클래스인 *Intermediator* 클래스 내부에는 각 클래스의 인스턴스들을 등록시킴으로써 각각의 클래스에 대한 정보를 유지할 수 있게 된다. 따라서 *Intermediator* 클래스에는 각 클래스의 정보가 등록되어 있다. 또한 *Intermediator* 클래스는 클라이언트들이 <<interface>>를 통하여 요청한 메시지



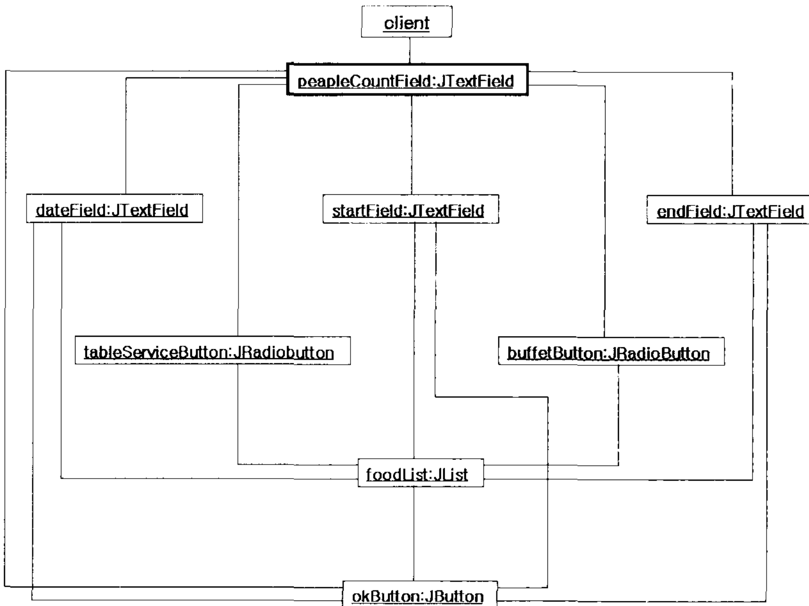
[그림 6] Intermediator 패턴의 클래스 다이어그램
[Fig. 6] Class diagram of Intermediator pattern

를 받아 구체적 행위를 실행할 수 있도록 실체화되어 있는 부분이므로 처리된 결과를 곧바로 클라이언트들에 전달된다.

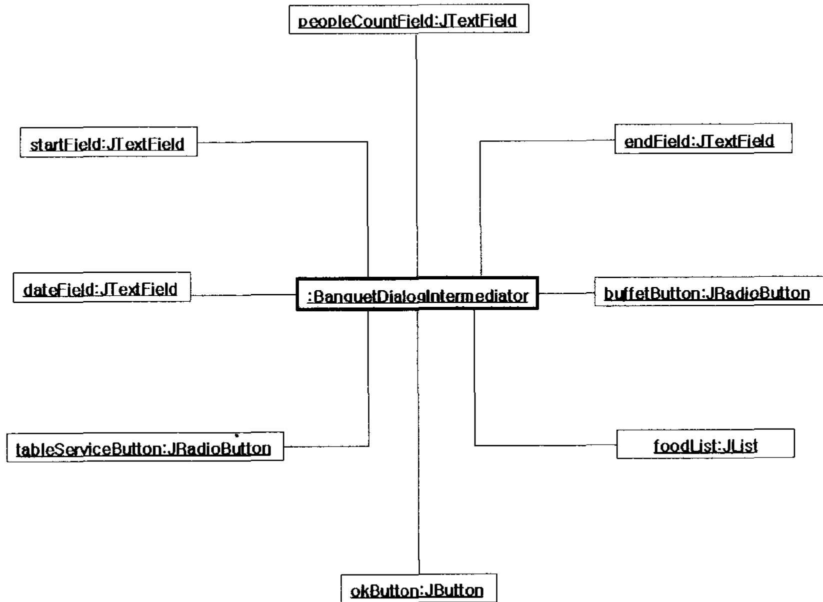
3.2.2 Facade 패턴과 *Intermediator* 패턴의 객체 관계도 비교 분석

[그림 7]은 Facade 패턴을 이용한 객체들간의 관계를 보여주는 다이어그램이다. 각 객체간의 의존성을 보면 객체들간의 높은 결합력을 가지게 되며, 응집력은 낮아지게 된다. 그러면 컴포넌트의 궁극적인 목표라 할 수 있는 재사용성 또한 낮아져 이러한 컴포넌트를 재사용하기는 어렵다. [그림 7]에서 주시해야 할 사항은 각 관련 클래스간의 링크관계이다. [그림 7]에서도 알 수 있듯이 'peopleCountField: JTextField' 객체 같은 경우는 'okButton:JButton' 객체, 'dateField: JTextField' 객체, 'tableServiceButton: JRadioButton' 객체, 'startField:JTextField' 객체, 'buffetButton:JRadio-Button' 객체, 'endField:JTextField' 객체 등 여섯 객체와 의존성이 있다는 것을 알 수 있다. 바로 이 'peopleCountField:JTextField' 객체가 Facade 객체이다. 이러한 객체간의 의존성은 유지보수를 어렵게 만드는 주원인이 된다. 따라서 이러한 의존성을 줄이고 이러한 관계를 중계해줄 수 있는 중계자 하나를 두면 이러한 복잡한 의존성은 사라질 것이다.

본 논문에서 제안하는 *Intermediator* 패턴은 이러한 중계자 역할을 '*Intermediator*' 라는 객체가 대신한다. 따라서 *Intermediator* 클래스 내에는 각 Client 클래스들의 정보가 등록되어 있으며, 메소드의 구체적인 행위가 이루어지는 곳이기도 하다. [그림 8]은 [그림 7]에서 보여주었던 객체간의 적어도 두 가지 이상의 의존성을 가졌던 것에 반해 '*Intermediator*' 객체를 통해 각 객체와 연결되어 있기 때문에 각각의 객체들은 서로 다른 객체들을 알 필요가 없으므로 각각의 객체들은 의존성이 전혀 없다는 것을 그림으로부터 알 수 있다. 이러한 각각의 객체들에 관한 정보는 이미 '*Intermediator*' 객체 내에 등록이 되어있기 때문에 각각의 객체들은 유일하게 '*Intermediator*' 객체에만 의존성을 가진다. 이러한 구조는 코드 구현과 유지보수를 좀 더 쉽게 할 수 있을 뿐만 아니라 다른 여러 개발자들이 이해하는데 용이하다.



[그림 7] Facade 객체를 이용한 객체 관계도
[Fig. 7] Object relation that use Facade object



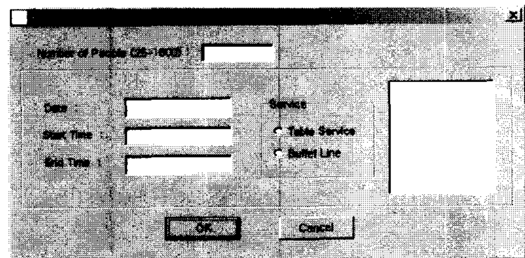
[그림 8] Intermediator 객체를 사용한 객체 관계도

[Fig. 8] Object relation that use Intermediator object

[그림 8]을 보면 ':BanquetDialogIntermediator' 객체는 'peopleCountField:JTextField' 객체와 'endField:JTextField' 객체, 'buffetButton:JRadioButton' 객체, 'foodList:JList' 객체, 'okButton:JButton' 객체, 'tableServiceButton:JRadioButton' 객체, 'dateField:JTextField' 객체, 'startField:JTextField' 객체들과 의존성을 가지고 있다. ':BanquetDialogIntermediator' 객체가 중계자 역할을 하기 때문에 나머지 다른 객체들은 서로에 대해 독립적일 수 있다.

4. 적용 및 평가

[그림 9]는 연회석 룸을 예약하기 위해 간단히 제작된 다이얼로그박스이다. 기존의 구현 방식에서는 처음 다이얼로그 박스가 나타나면 'Number of People'의 필드와 'Cancel'필드만이 활성화되고 나머지 'Date', 'Start Time', 'End Time', 'Service'필드는 사용할 수 없는 상태로 되어있다. 'Number of People'의 필드에 25에서 1000명 사이의 숫자를 입력하면 나머지 필드들이 활성화된다. 'Start Time'은 'End Time'보다 시간상으로 앞서야 하고, 이 두 가지 필드를 입력하고 나서 서비스의 라디오 버튼을 선택하면 해당 음식이 보여지게된다. 여러 메뉴 중에서 적어도 하나 이상을 선택하여야만 'OK' 버튼이 활성화된다.



[그림 9] 연회석 룸 예약 다이얼로그 창

[Fig. 9] Annual meeting room reservation dialog windows

본 논문에서 제안한 *Intermediator* 패턴은 JDK 1.2 환경에서 구현되었으며 *Intermediator* 패턴의 프로토타입은 [그림 10]과 같다.

```
private class BanquetIntermediator {
    private JButton okButton;
    private JTextComponent dateField;
    private JTextComponent startField;
....
}
```

[그림 10] Intermediator 클래스의 프로토타입
[Fig. 10] Prototype of Intermediator class

Intermediator 클래스의 프로토타입은 [그림 10]과 같다. [그림 10]에서도 알 수 있듯이 *Intermediator* 클래스는 private로 보호되어 있어 다른 외부객체로부터 쉽게 접근이 허용되어있지 않기 때문에 *Intermediator* 클래스에 허가된 객체만이 접근을 할 수 있도록 캡슐화되어 있다.

```
private ItemAdapter itemAdapter
= new ItemAdapter();
```

[그림 11] ItemAdapter 객체 생성 프로토콜
[Fig. 11] ItemAdapter object creation protocol

ItemAdapter 클래스는 private 어댑터 클래스로 선언되고 다이얼로그박스에서 라디오 버튼 두 가지로부터 이벤트를 처리하는데 BanquetIntermediator 클래스에 의해 사용된다.

```
BanquetIntermediator() {
WindowAdapter windowAdapter
    = new WindowAdapter() {
 public void windowOpened(WindowEvent e) {
    initialState();
 }
};
addWindowListener(windowAdapter);
}
```

[그림 12] BanquetIntermediator 클래스의 생성자
[Fig. 12] Constructor of BanquetIntermediator class

BanquetIntermediator 클래스의 생성자는 다이얼로그 박스가 오픈될 때 감싸진 다이얼로그 박스 객체가 보내는 이벤트들을 처리하는 익명의 어댑터 클래스를 선언하고 초기화한다.

```
private class ItemAdapter implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
            enforceInvariants();
      }
}
```

[그림 13] ItemAdapter의 구현 클래스
[Fig. 13] Implementation class of ItemAdapter

'ItemAdapter' 객체가 'ItemEvent'를 받을 때 'ItemAdapter'는 BanquetIntermediator 클래스의 'enforceInvariants' 메소드를 호출한다.

'enforceInvariants' 메소드는 다이얼로그 컴포넌트들 사이에 변치 않는 관계를 강화한다. 'enforce-Invariants' 메소드는 모든 다이얼로그 박스의 GUI(Graphic User Interface) 컴포넌트들로부터 이벤트에 대한 응답으로 호출된다.

[그림 14]에서 나타내는 메소드는 제한된 인원수를 체크한 후 허용범위이면 'dateField', 'startField', 'endField', 'buffetButton', 'tableServiceButton' 등의 메뉴 필드들을 활성화시킨다.

본 절에서는 앞서 3절에서 제안한 *Intermediator* 패턴의 구현 예로서 연회 예약을 하기 위한 예약 다이얼로그 창을 통하여 실제 비즈니스 컴포넌트간의 상호운용에 대하여 간단하게 구현해보았다. 구현언어는 Java를 사용하여 구현하였다. 앞서 설명한 바와 같이 객체들간의 참조는 참조하고자 하는 방법에 따라서 매우 다양한 관계를 가질 수 있다. 그러나 컴포넌트들간의 상호운용을 위한 객체참조의 방법은 시스템의 성능과 밀접한 관련이 있으므로 참조 방법을 정의하는 것은 매우 중요한 일이라 할 수 있다. 컴포넌트간의 상호운용을 위한 객체참조방법에는 여러 가지 설계 패턴이 존재할 수 있다. 컴포넌트간 상호운용에 있어 기존의 Façade 패턴의 대표객체를 통하여서만 참조를 하는 방법은 클라이언트의 수가 많고 복잡한 구조를 가지는 비즈니스 컴포넌트간에는 결

```
private boolean busy = false;
 ...
private void enforceInvariants() {
      if (busy)    return;
      busy = true;
      protectedEnforceInvariants();
      busy = false;
}

private void protectedEnforceInvariants() {
      boolean enable = (peopleCount != PEOPLE_COUNT_DEFAULT);
      dateField.setEnabled(enable);
      startField.setEnabled(enable);
      endField.setEnabled(enable);
      buffetButton.setEnabled(enable);
      tableServiceButton.setEnabled(enable);
      if (enable) {
          enable = (buffetButton.isSelected()
                       || tableServiceButton.isSelected());
          foodList.setEnabled(endAtLeastOneHourAfterStart());
      } else {
         foodList.setEnabled(false);
         buffetButton.setSelected(false);
         tableServiceButton.setSelected(false);
      }
      okButton.setEnabled(foodList.isEnabled()
                 && foodList.getMinSelectionIndex()>-1);
}
```

[그림 14] protectedEnforceInvariants 메소드

[Fig. 14] protectedEnforceInvariants method

과값 반환의 시간이 많이 소요되는 비효율성이 발생한다는 문제점 있다. 그에 대한 대안으로 앞서 설명한바와 같이 여러 컴포넌트들에 대한 각각의 정보를 *Intermediator* 클래스에 등록시켜 구체적인 행위는 이곳에서 이루어지도록 하는 방식인 *Intermediator* 패턴은 다른 모든 컴포넌트들간에는 서로의 정보를 알 필요가 없어 상호운용 하는데 확실한 독립상태를 가지므로 보다 좀 더 객체지향적으로 설계될 수 있다는 점을 연회예약 다이얼로그 창을 통해 알아보았다.

5. 결론 및 향후 연구과제

본 연구는 비즈니스 컴포넌트들간의 의사소통을 하기 위한 설계 패턴을 제안하였다. 기존의 비즈니스 컴포넌트간의 의사소통을 위한 Façade 패턴은 대표 객체(Façade)만을 통해 다른 비즈니스 컴포넌트들간의 대화를 하기 때문에 클라이언트는 참조하고자 하는 서로 다른 컴포넌트를 알 필요 없이 Façade 패턴의 프로토타입만을 정의하면 대표객체를 통하여 원하는 결과를 되돌려 받는 방식이다. 그러나 이러한 방식은 클라이언트의 수가 많고 복잡한 구조를 가지는 비즈니스 컴포넌트간에는 결과 반환의 시간이 많이 소요되는 비효율성이 발생할 수 있다. 따라서 본 논문에서는 이러한 모순을 보완하여 새로운 *Intermediator* 패턴을 제안하였다.

Intermediator 패턴은 다수의 클라이언트가 존재하더라도 그에 따른 프로토타입을 가지는 다수의 인터페이스를 통하여 '*Intermediator*' 객체에 전달하는 것이다. 그러면 '*Intermediator*' 객체는 다른 여러 클라이언트의 정보를 이미 등록시킨 상태이기 때문에 클라이언트들은 서로 참조하기 위해서 직접 호출할 필요없이 '*Intermediator*' 클래스에 메시지를 보내면 '*Intermediator*' 객체는 필요한 정보를 수행하여 요청 클라이언트에게 결과를 되돌릴 수 있기 때문에 객체간의 결합도는 줄일 수 있고 상대적으로 응집도는 높일 수 있다.

향후 연구과제로는 본 논문에서 제안한 *Intermediator* 패턴은 현재 비즈니스 컴포넌트 단위를 갖는 컴포넌트들의 상호운용을 보다 효과적으로 이루어질 수 있도록 제안되었다. 즉 비즈니스 컴포넌트라는 제한된 단위에서는 효과적일 수 있으나 비즈니스 컴포넌트들이 모여 시스템 단위의 컴포넌트들간의 상호운용에도 적용가능한지를 검증하고1 이를 비교 분석하여 비즈니스 컴포넌트보다 더욱 큰 단위의 컴포넌트에서도 효과적인지를 체계적으로 평가할 필요가 있다.

※ 참고문헌

[1] Booch, Object-Oriented Analysis and Design, 2nd ed., Benjamin/Cummings, 1994.

[2] James William Cooper, Java Design Patterns: A tutorial, Addison Wesley Longman, Inc., 2000.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriendted Software, Addison Wesley Longman, Inc., 1995.

[4] Mark Grand, Patterns in Java Vol. 1: A Catalog of Reusable Design Patterns Illustrated with UML, John Wiley & Sons Inc., 1998.

[5] Mark Grand, Patterns in Java Vol. 2, John Wiley & Sons Inc., 1999.

[6] Desmond F. D'Souza Alan C. Wills, Objects, Components, and Frameworks with UML: the caltalysis approach, 1998.

[7] Peter Herzum, Oliver Sims, Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise, John Wiley & Sons, Inc., 2000.

[8] Robert Orfali, Dan Harkey, Client/Server Programming with Java and CORBA 2nd Edition, jhon Wiley & Sons, Inc., 1998.

[9] Peter Coad, Mark Mayfield, Java Design: Building Better Apps and Applets 2nd edition, Prentice-Hall, Inc., 1999.

[10] Ed Roman, Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition, John Wiley & Sons, Inc., 1999.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns : Elements of Reusable Object-Oriented Software, Addison Wesley Longman, Inc., 1994.

이창목

1999년 2월 호원대학교 전자계산학과(학사)
2001년 2월 전북대학교 대학원 전산통계학과 석사
2001년 3월～현재 : 전북대학교 대학원 컴퓨터통계정보학과 박사과정
관심분야 : 객체지향개발방법론, 객체지향 프레임워크, 컴포넌트개발 디자인 패턴

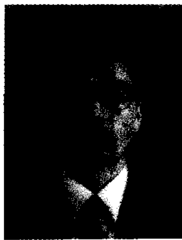
유철중

1982년 2월 전북대학교 전산통계학과 졸업(이학사)
1985년 8월 전남대학교 대학원 계산통계학과 졸업(이학석사)
1994년 8월 전북대학교 대학원 전자계산학과 졸업(이학박사)
1982년 9월～1985년 3월 전북대학교 전자계산소 조교
1985년 4월～1996년 12월 전주기전여자대학 전자계산과 부교수
1997년 1월～현재 전북대학교 자연과학대학 컴퓨터과학과 조교수
관심분야 : 소프트웨어공학, 에이전트공학, 컴포넌트기술, 분산객체기술, 지리정보시스템, 멀티미디어, 인지과학

장옥배

1966년 고려대학교 수학과 졸업
1980년 죠지아 주립대, 오하이오 주립대 박사과정 수료
1987년 산타바바라대학교 졸업(Ph.D)
1990년 영국에딘버러대학교 객원교수
1980년 4월～현재 전북대학교 컴퓨터과학과 교수
관심분야 : 소프트웨어공학, 전산교육, 수치해석, 인공지능

문윤호



1985년 2월 전북대학교 전산통계학과 (이학사)
1987년 8월 숭실대학교 대학원 전산학과 (공학석사)
1991년 3월 전주기전여자대학 컴퓨터과 전임강사
2001년 1월 숭실대학교 컴퓨터학과 (공학박사)
현재 전주기전여자대학 부교수
관심분야 : 컴퓨터구조(결함허용), 분산처리, 소프트웨어공학