설계패턴을 이용한 객체지향 방법론에 관한 연구

김 치 수† · 임 경 미†† · 권 민 주††

요 약

융통성 있는 소프트웨어 개발을 위해 많은 객체지향방법론들이 제공되어 지고 있으나, 이러한 객체지향방법론들은 자동적으로 재사용을 제공하거나 건실한 소프트웨어 시스템을 만들지 못하고 있다. 따라서 이러한 객체지향 방법론들은 개발자에게 상세한 설계지침을 제공하지 못하고 있으며, 분석·설계자들도 이런 문제점들을 인정하고 방법론들을 향상시켜 나가려고 노력하고 있다. 이와 같이 분석·설계자들의 노력을 줄여 주면서 설계단계의 정보를 재사용할 수 있도록 제공하고자 하는 것이 설계패턴의 개념이며, 이러한 설계패턴과 객체지향 방법론은 상호 보완적인 관계를 가지면서 시스템을 개발하는데 유용한 설계 길잡이의 역할을 한다.

이에 본 논문에서는 객체지향 방법론 중에서 가장 많이 다루어지고 있는 OMT방법론과 Gamma가 제안한 설계패턴 중 Facade, Mediator, Observer의 세 가지 설계패턴에 대한 상호작용을 보여주고, OMT방법론 내에 설계패턴을 적용함으로써 구체적인 설계지식과 재사용에 대한 명확한 해법을 제안한다.

A Study on the Object-Oriented Methodology on the Basis of Design Patterns

Chi-Su Kim† · Kyoung-Mi Im†† · Min-Ju Kwon††

ABSTRACT

The various kinds of object-oriented methodology were provided for the development of flexible software. However, they do not automatically make the reusable and robust Object-Oriented software systems. The Object-Oriented mechanism is simply used as the means to obtain the aim. We have heard many complaints from the developers that methods did not give concrete design guidance. Methodologists recognize these problems and have been improving their methods. These design patterns reduce methodologists' effort and make reusable information in the design phase.

The relation between the Object-Oriented methods and the design patterns is mutually complementary, and they play a role as a valuable design guidance in the development of the Object-Oriented system.

This paper shows the interaction between OMT which is most well-known in the Object-Oriented Methodology and Facade, Mediator and Observer designed by Gamma. We suggest the clear solution for concrete design knowledge and reusability of them by applying design patterns to Object-oriented Methodology.

1. 서 론

소프트웨어의 사회적 중요도가 인식되고 방대해짐에 따라 소프트웨어 개발을 위한 유지보수비용은 점점 증가하고 있으며, 사용자의 요구사항 변경 또한 유동적으로 늘어나고 있는 추세이다. 이에 따라 융통성 있는 소프트웨어 개발을 위해 많은 객체지향방법론들[1,2,3,4,5]이 제공되어지고 있다. 그러나 이러한 객체지향방법론들이 자동적으로 재사용을 제공하고 건실한 소프

† 종신회원 : 공주대학교 교육과학연구소
†† 준 회 원 : 공주대학교 멀티미디어연구소
논문접수 : 1998년 10월 30일, 심사완료 : 1999년 4월 2일

트웨어 시스템을 만들지는 않으며, 단순히 목적달성을 위한 수단으로만 쓰여지고 있다. 따라서 기존 객체지향방법론들은 설계상의 많은 문제점을 수반하게 되며, 방법론자들은 이런 문제점들을 인정하고 방법론들을 향상시켜 나가려고 노력하고 있다.

또한, 코딩단계에서 생성된 부품들을 재사용할 때는 그것을 자신의 개발환경에 맞게 수정해서 사용해야 하는데 이는 설계단계에서 변경이 발생되면 코드자체에 많은 수정을 해야하며, 추가비용을 부담해야하기 때문에 재사용 효율성을 떨어뜨릴 수 있다[6]. 이와 같이 방법론자들의 노력을 줄여 주면서 설계단계의 정보를 재사용하는 개념이 설계 패턴이다. 설계패턴은 기존 객체지향방법론들에 구체적인 설계지식을 제공하며, 반복적으로 발생하는 설계상의 문제들과 재사용에 대한 명확한 해법을 제공한다.

이에 본 논문에서는 이러한 설계패턴을 기존 객체지향 방법론 중 가장 많이 다루어지고 있는 OMT방법론[4]에 적용시킬 수 있는 효과적인 방법을 제안한다. Rumbaugh에 의해 개발된 OMT방법론은 다양한 도구와 개념들이 명확하게 규정되어 있으며, 데이터베이스 설계와 밀접한 관련을 가지고 있다. 그러나, 사용자의 요구나 문제를 완벽하게 표현하지 못하고, 3가지 분석 모델간의 상호관련 표시와 이해가 어렵다. 또한 설계단계가 상대적으로 취약하여, 기능모델이 구현되기 어렵다는 단점이 있다. 이러한 설계문제들을 해결하기 위해 방법론내에 자연스럽게 설계패턴을 적용시키는 방법을 제시한다.

본 논문에서는 Gamma가 제안한 설계패턴 카탈로그[7] 중에서 Facade, Mediator, Observer 이 세 가지 설계패턴만을 사용한다. OMT방법론에서 서브시스템과 인터페이스가 이루어질 경우, 내부객체들이 서브시스템의 public 인터페이스의 일부분이 되어야 한다는 것에 대한 구체적인 규칙을 제공하지 못하는 문제점을 해결하기 위해 Facade 설계패턴을 OMT방법론에 적용시키는 방법을 제안한다. 또한, Mediator의 적용으로 OMT방법론에서 구현과 연관된 협력을 설계하는 방법을 제시하며, OMT가 dependsOn의 의미를 가진 객체들과의 특별한 연관성을 가지고 있지 않다는 것을 감안하여, Observer를 적용시킴으로써 객체들간의 dependsOn 연관성을 포함하고, 객체 연관성을 모델화하여 이러한 문제점을 해결할 수 있는 방안을 제시한다.

2. 관련 연구

2.1 OMT방법론

2.1.1 개념 및 특징

OMT방법론은 Rumbaugh와 Michael Blaha 등이 구성한 것으로 분석, 설계, 개발까지의 일련의 과정을 지원하며, 전체적으로 깊이 있고 잘 편성된 방법론으로, 다른 방법론의 표기법보다는 월등할 뿐만 아니라 매우 다양한 도구와 개념들을 제시하며, 사용되는 도구 및 개념들이 명확하게 규정되어 있다. 또한, 데이터베이스 설계와 밀접한 관련이 있으며, 시스템의 다른 국면들을 나타낼 수 있는 세 가지 분석 모델을 지원한다. 즉, 이 방법은 문제영역을 객체모델과 동적모델, 기능모델로 나누어 분석하며, Ivar Jacobson의 OOSE[3]와 연계 사용할 때 현재까지의 방법론 중 가장 효과적인 것으로 평가된다.

2.2 설계패턴

최근, 유연성과 재사용성을 지닌 객체지향시스템 설계방법에 관한 새로운 연구들[8,9,10]이 많이 이루어지고 있다. 설계패턴은 설계상에 반복적으로 발생하는 문제들에 대한 명확한 해법과 재사용을 제공한다. 설계패턴은 많은 사람들의 경험을 통해 인정된 세부 설계과정의 해결책으로, Christopher Alexander는 "각 패턴은 개발환경 내에서 반복적으로 발생하는 문제점들을 기술하고, 그 다음에 문제에 대한 핵심적인 해법을 기술한다"[11]라고 정의하였다.

설계패턴은 독립적인 영역으로 프레임워크의 성공적인 경험과 클래스 라이브러리의 개발로부터 얻어진 재사용 가능한 설계 생성물이다. 이것은 객체보다 좀더 큰 시스템설계 블록으로서 합성적, 또는 협동적 행위를 표현하고 있다. 각 패턴은 서브시스템 인터페이스 설계방법, 객체들간의 복잡한 협동 설계방법 등과 같은 명확한 설계를 위한 논제를 설명하고 있다. 각 패턴은 문제에 대한 구체적인 해법을 제공하기 위해 설계, 그들 사이의 관계, 확실성, 협동을 만들어 줄 요소들을 묘사하고 있다. 해법 자체가 완벽한 수행을 하지는 않지만, 객체들 사이에서의 일반적인 규약으로 표현되어질 수 있다.

설계패턴은 다른 전문가들이 작성한 코드나 설계의 패턴들을 모방하는 것을 추상화한 개념으로, 넓게 해석하면, 소프트웨어 개발영역에서 특정한 임무를 수행

하기 위해 자주 사용되는 규칙들의 집합이다[6]. 또한 설계패턴들은 재사용 가능한 객체지향 설계들을 생성하기에 유용한 일반적인 설계구조를 추상화함으로써, 설계 단계의 재 사용방법을 도입할 수 있다. 현재, Gamma가 제안한 설계패턴 카탈로그의 개념[7]이 객체지향분야에 가장 많은 영향을 주고 있다.

2.2.1 Facade

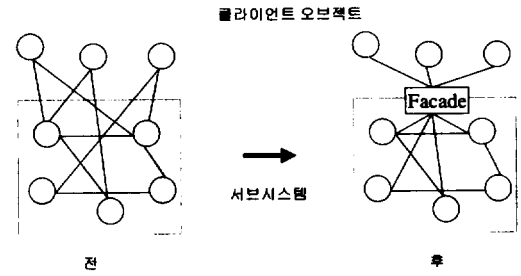
Facade는 외부클라이언트에 대한 서브시스템간의 인터페이스 설계문제를 다룬다. Facade의 목적은 서브시스템을 좀더 쉽게 사용할 수 있도록 상위레벨 인터페이스를 정의하는 것이다. 또한, 서브시스템의 내부구조로부터 클라이언트들을 보호한다. 그 때문에 서브시스템의 세부항목상의 클라이언트의 의존적인 수행을 감소시킨다.

Facade 객체는 서브시스템간의 인터페이스 역할을 수행한다(그림 1). 이것은 서브시스템 수행 서비스들에 대해 상위레벨 인터페이스를 제공하고, 외부 클라이언트와의 관계에 있어 가장 밀접한 서브시스템 인터페이스가 된다.

이것은 서브시스템 안에 있는 객체들이 요구사항을 수행하고, 클라이언트에게서 결과 값이 되돌아오기 전에, 클라이언트로부터 요구사항을 받는다. 서브시스템 객체들은 이에 반해서 서브시스템 서비스를 수행하고, facade 객체에 의해 할당된 작업을 처리한다.

서브시스템의 외부클라이언트는 서브시스템의 facade 객체에 대한 두 가지 요구사항을 가질 수 있다. 하나는 facade 객체에 의해 정의된 일반적인 서비스들에 대한 요구이다. 다른 하나는 내부 객체들에 대한 처리에 관한 요구이다. facade 객체가 일반적인 서비스에 관한 요구를 받았을 때, 이것은 요구사항을 수행할 내부객체에게 요구사항을 발송하고, 외부 클라이언트에게 결과 값을 받는다. 내부객체에 대한 요구사항을 받았을 때, 이것은 외부 클라이언트에 대해 처리된 것을 되돌려 준다.

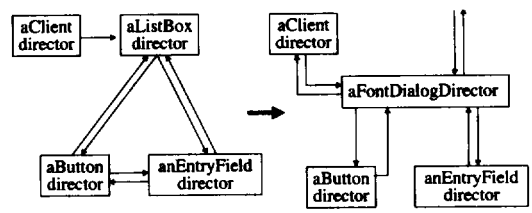
Facade는 서브시스템 인터페이스에 대해 무엇을 해야 하는 지와 어떻게 설계해야 하는지를 결정할 수 있게 해준다. Facade 접근법을 사용할 때 개발자들은 서브시스템이 제공하는 서비스들은 무엇이고, 외부클라이언트가 필요로 하는 공통서비스들은 무엇인지를 고려해야 한다. 다시 말해서, 개발에 있어 많은 인터페이스를 갖는 객체들은 효율적이지 못하므로, 하나의 facade 객체에 의해 공통 서비스들을 단순화시킨다. facade 객체는 실행목적을 위한 것도 아니고, 기능성을 추가하려는 목적도 아니다. 다만 설계 목적을 위한 것이다. facade 객체를 추가함으로써 서브시스템의 내부구조를 캡슐화하고, 클라이언트를 위한 공통 인터페이스를 쉽게 사용하도록 한다.



(그림 1) Facade

2.2.2 Mediator

Mediator 패턴은 복잡한 객체 협력 설계방법에 관한 문제를 설명한다. 객체지향은 시스템의 성능을 최대화하기 위해 협력하는 객체집합을 분리하여 신뢰성을 촉진시킨다. 그렇지만, 객체들 사이에서 결합도가 높은 경우, 다른 어플리케이션이나 개발환경에서 개별적으로 재사용되는 것은 어렵다. 협력 객체들 사이에서 결합도를 제거시키는 일은 중요한 설계상의 문제이다.



(그림 2) Mediator

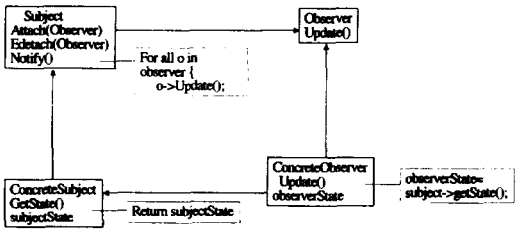
이때, Mediator 패턴은 좋은 해결방법을 제공하는데, 복잡한 객체들을 통합하기 위해 mediator 객체를 사용한다. Mediator 패턴은 mediator 객체를 통해 간접 협동을 가지고 객체들 사이에서 직접 협동을 교환한다. 이를 통해 재사용성과 확장성을 촉진시킬 수 있다.

Mediator는 유즈케이스를 추상화하고 유즈케이스에 구체적인 접근법을 제공한다. 다중제어객체들로 유즈케이스를 설계할 때, 경험적 Mediator는 제어객체를 정의하는 것을 도와준다. 경험적 Mediator는 제어객체(mediator 객체)로써 협력행위를 설계하고 확인하도록 해준다. 객체들 사이에서 강결합하는 결과를 보일 때,

다른 어플리케이션이나 개발환경에서 개별적으로 재사용되는 것이 어려울 경우, Mediator 패턴은 좋은 해결방법을 제공한다. Mediator 패턴은 협력객체들을 분리함으로써 각각의 객체에 대해 재사용성과 확장성을 촉진시키고, 객체들 사이에서 협력을 추상화시킨다.

2.2.3 Observer

Observer는 dependsOn관계를 수행하기 위한 한 가지 방법이다. Observer패턴은 객체들 사이에서 1대 다 종속 설계방법에 대한 문제를 설명한다. 연관성내의 관계자는 subject객체와 observer객체로 구분할 수 있는데, subject객체는 다른 객체들의 상태변환에 영향을 미치는 객체이다. 설계단계에서 제어역할을 수행하며, 어떤 일이 발생하면 모든 것을 observer객체들에게 통보한다. observer객체들은 다른 객체의 상태변환에 영향을 받는 객체이다. observer객체들은 변환 즉, 확실한 행위의 수행, subject가 가진 상태와 일치시키기 위한 상태변환 등에 대한 응답방법을 준비한다.



(그림 3) Observer

3. 설계패턴이 적용된 OMT방법론

3.1 OMT방법론에 적용한 Facade 설계패턴

OMT방법론에서 서브시스템 인터페이스는 보편적으로 사용되는 유형적 클래스들, 일반적인 관계, 속성들, 오퍼레이션들의 집합으로서 정의되는 서브시스템의 public view와, 나머지는 개별적 클래스들과 public클래스 중에서 개별적인 부분인 서브시스템의 private view로 구분된다.

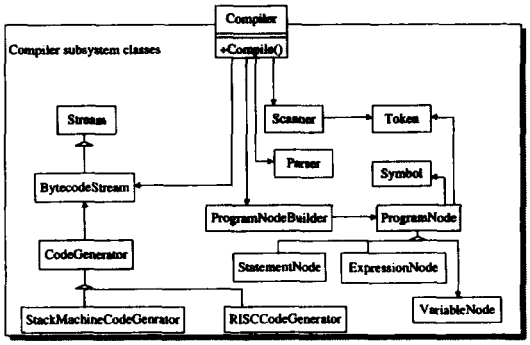
OMT 서브시스템 인터페이스는 두 부분으로 구성되는데, 서브시스템의 public view에 있는 public인터페이스와 외부 클라이언트가 서브시스템의 public객체들을 액세스하는 것을 가능하도록 해주고, 서브시스템의 내부객체들에 대한 처리를 되돌려 주는 dominant 객체이다[12]. 여기서, public객체들은 서브시스템의 서비스들을 제공한다. 예를 들어, 컴파일러 서브시스템은 컴파일러를 수행하는 Scanner, Parser, ProgramNode, BytecodeStream, ProgramNodeBuilder와 같은 클래스들을 포함한다(그림 4). 컴파일러 클래스들은 대개 컴파일러 서브시스템으로부터 기대되는 일반적인 기능들을 제공하는 인터페이스를 정의한다. 이런 인터페이스는 만일 일반적인 서비스들을 액세스할 필요가 있을 경우 Scanner, Parser 등으로부터 클라이언트들을 보호한다. 한편, public 인터페이스는 클라이언트 객체들을 이용할 수 있는 다른 기능들을 제공하는 Parser, Scanner 등을 포함한다.

이런 OMT접근방법의 이점은 서브시스템의 외부 클라이언트들이 public객체에 액세스하기 위해 정적참조를 할 필요가 없다는 것이다. 즉, 재번역없이 실행시간에 public객체들과 서브시스템들이 치환될 수 있도록 해준다.

반면에, 외부클라이언트들이 이들을 참조할 필요가 없는데도, 외부클라이언트들은 public객체들, 그들간의 인터페이스, 직접 액세스방법 등을 알고 있어야 한다. 외부클라이언트의 설계는 서브시스템의 public객체들에 의존하도록 만든다. 따라서, 내부객체들이 서브시스템의 public인터페이스의 일부분이 되어야 한다는 것에 대한 구체적인 규칙들을 제공하지 못한다. 개발자들은 public인터페이스로써 외부클라이언트에 의해 요구되는 명확한 서비스들을 제공하는 내부객체들은 무엇이든지 사용하게 되는데, 이것은 개발자들이 외부클라이언트에 대해 서브시스템의 내부설계와 수행구조를 들어내는 것과 같다.

이런 경우, Facade방법을 이용해 OMT접근방법을 정제할 수 있다. 서브시스템의 외부클라이언트들은 dominant객체에 대해 두 가지 요구사항을 보낸다. 먼저, dominant객체가 공통서비스에 대한 요구를 받았을 경우, 요구사항을 수행할 내부객체에게 요구사항을 전달하고, 외부클라이언트에게 직접 결과를 되돌려준다. 두 번째, 내부객체의 처리에 대한 요구를 받았을 경우, 외부클라이언트에게 처리를 되돌려 준다.

결과적으로, 외부사용자들이 몇 가지 공통서비스들을 완수하기 위해 서브시스템을 원할 때, dominant객체에게 직접 요구사항들을 보낼 수 있다. 다른 객체들에 대한 어떠한 사항들도 알 필요가 없으므로, 이러한 설계는 서브시스템 내부에 있는 객체들에 의존하지 않다. 이러한 설계는 dominant객체들과 서브시스템의 가장 견실한 부분인 public인터페이스에 의존할 필요가 있다.



(그림 4) Facade 적용 예

3.2 OMT방법론에 적용한 Mediator 설계패턴

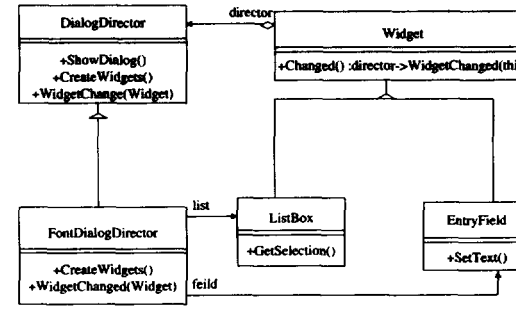
OMT방법론에서 시스템 구축을 위해 도메인, 인터페이스, 컨트롤러, 뷰, 장치 및 내부객체들과 같은 몇 가지 유형의 객체들이 사용된다. 여기서, 컨트롤러 객체들은 어플리케이션에 대한 제어위치에 따라 행동하고, 시스템과 외부 액터들간의 상호작용을 관리하기 위해 사용한다. 그리고, 사용자들, 또는 외부장치들과 같은 외부 액터들로부터 사건을 받는다. 컨트롤러 객체는 view, 또는 subject들과 같은 다른 객체들의 연산들로 해석하며, 고도로 문맥 의존적이며, 많은 정책결정, 가설, 설계 또는 전개하는 동안 변환될 수 있는 디폴트값들을 포함한다. 또한, 사용자 제어 옵션과 같은 어플리케이션내의 문맥 의존적인 결정을 구체화한다. 컨트롤러 객체들은 수정할 필요 없이 융통성 있는 방법내에서 재사용할 수 있는 컴포넌트들을 함께 설치하고, 다른 방법에서 실시되는 동일한 view들과 같은 고도의 동적 지식을 구체화한다.

제어 객체들은 유즈케이스들로 강력한 관계성을 갖는다. 대부분, OMT방법론은 유즈케이스에 대하여 하나의 컨트롤러를 가정하고 몇 개의 유즈케이스를 결합함으로서 출발한다. 즉, 작은 어플리케이션에 대해서는 하나의 컨트롤러 객체를 사용하고, 큰 분산시스템들에서는 많은 사용자들과 컨트롤러를 가진다.

OMT방법론은 OOSE처럼 다중 제어객체들을 사용하는 경우나, 유즈케이스를 모델화하기 위해 하나의 제어 객체를 사용할 때를 명확하게 하지 않으며, 또 몇 개의 제어 객체들로 유즈케이스의 전체적인 협력을 분리하는 방법이 분명하지 않다.

또한, OOSE처럼 OMT방법론은 구현과 관련된 협력을 설계하는 방법을 설명하지 않는다. 다시 말해서, OMT에서 컨트롤러는 구현과 관련된 협력을 모델화하기 위해 사용될 수 없다. 왜냐하면, 그들은 단지 객체들간의 논리적 협력을 모델화하기 위해 사용되는 것만을 가정하기 때문이다. 이것은 OMT가 협력들을 모델화하기 위해 내부객체 유형을 사용하고 있음을 나타낸다. OMT는 새로운 객체들을 추가할 수 있고, mediator 객체가 실제로 OMT의 객체유형과 매치되지 않기 때문에 mediator 객체는 새로운 OMT 객체 유형으로서 추가될 수 있다.

따라서, OMT내에 적용되는 Mediator는 OOSE내에 적용되는 Mediator와 거의 같다. OMT내에 적용되는 패턴 대신에 경험들은 실제로 방법론들이 다중 제어 객체들이 하나의 유즈케이스를 위해 사용될 것인지에 대한 상황을 결정하고, 다중 제어 객체들로 유즈케이스를 분리하기 위한 안내서를 제공하도록 하는데 적합하다. Mediator는 OMT방법론에서 구현과 연관된 협력을 모델화하기 위해 사용된다.



(그림 5) Mediator 적용 예

3.3 OMT방법론에 적용한 Observer 설계패턴

OMT는 dependsOn의 의미를 가진 객체들간의 특별한 연관성을 가지고 있지 않다. 그리고, Observer 패턴처럼 dependsOn을 위한 설계 해결책도 가지지 않는다. 하지만 OMT'94에서 두 가지 특수한 객체유형인 도메인 객체와 뷰 객체를 위해 Observer의 특별한 형식인 Model-View-Controller(MVC)를 제공한다[13]. 대개 도메인 객체가 몇 개의 뷰 객체들을 갖는 동안 뷰 객체는 도메인 객체를 위한 정보를 표현한다.

OMT는 MVC가 지닌 도메인 객체들과 뷰 객체들간의 연관성 설계를 제안한다. 만일, 구조가 올바로 되어 있을 경우, 새로운 뷰들은 subject 객체들 또는 subject 오퍼레이션들을 수정하지 않고도 설계과정이나 실행시간동안 추가될 수 있다. 뷰 객체들은 도메인 객체들에

영향을 주지 않고도 변환될 수 있다. 뷰들은 다른 뷰들에 대해 알 필요가 없으나, subject 데이터에 대해서는 알아야 한다. 그러나, subject들은 뷰들에 대한 어떠한 지식도 갖고 있지 않다.

따라서, OMT가 이미 도메인 객체들과 뷰 객체들간의 관계성을 설계하기 위해 MVC를 사용한다할 지라도, OMT는 여전히 Observer를 필요로 한다. 또한, OMT는 객체들간의 특별한 연관성으로써, dependsOn 연관성을 포함해야 하고, 객체 연관성을 모델화하기 위해 제공되어야 한다.

이러한 이유는 dependsOn 연관성이 객체간의 일반적인 관계성을 표현하고 있기 때문이다. 만일 subject 객체가 변경된다면, observer객체는 상태를 변경하거나 무언가 일치하도록 할 필요가 있다. 연관성은 도메인 객체들과 뷰 객체들간의 관계성을 설계할 수 없고, OMT 객체들의 다른 유형들간의 관계성을 설계한다.

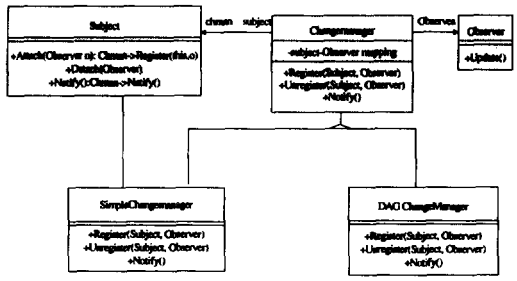
도메인 객체들은 다른 도메인 객체들에 의존한다. 스프래드시트 어플리케이션에서의 예를 보면, 셀 객체들은 그들 사이의 의존적인 관계를 가질 수 있는 도메인 객체들이다. 제어 객체들은 도메인 객체들에 의존할 수 있다.

OMT는 또한 특별한 연관성들의 개념을 채택해야 한다. 이론적으로, OMT의 일반적인 연관성은 객체/클래스들 간의 모든 관계성을 표현할 수 있다. 그렇지만, dependsOn과 같은 명확한 연관성을 되풀이하여 발생한다. 이러한 연관성들은 특별한 의미를 가질 수 있고 객체설계에 상당한 영향을 미친다. 객체유형으로서 그들은 시스템 모델링과 빌딩 블록들이며, 재사용 시스템 개발에 대한 경험을 강화시켜준다. dependsOn은 객체들간의 상태가 방법론 안에서 동시성을 가질 필요가 있음을 의미한다.

어플리케이션 모델링에서 OMT는 먼저 도메인 객체들에 대해 뷰 객체들을 정의해야 한다. OMT는 개발자들이 subject-view relation을 설계하기 위해 MVC 프레임웍크를 사용할 것을 제안한다. 한편, 이것은 OMT의 도메인 객체들과 뷰 객체들간의 dependsOn 연관성과 Observer패턴을 사용한다. dependsOn은 객체들간의 특수한 형태의 연관성이기 때문에 OMT가 정의한 곳에서 설계될 수 있고, 어플리케이션 도메인을 위해 객체 모델이 생성될 때 설계될 수 있다. OMT가 연관성 속성들을 분석할 때, OMT의 개발자들은 dependsOn 연관성의 카디날리티를 정의해야만 한다.

OMT의 객체설계단계에서 OMT가 객체 연관성을 설계하는 것을 "association design"이라고 한다. 이때, OMT는 Observer의 사용을 필요로 하며, OMT방법론 내에 알맞게 적용이 되야 한다. dependsOn연관성 설계를 위해, 개발자들은 Observer를 먼저 고려해야 하는데, 예를 들어, 1대 다의 카디날리티를 가진 dependsOn은 Observer로 설계되어야 한다.

설계단계와 수행단계에서, 몇몇 내부 설계 객체들과 구현 객체들이 생성된다. 그들은 dependsOn의 관계를 가질 수 있다. 예를 들어, 구현단계에서 하나의 객체에 대해 반복된 객체들을 생성한다. 이들 객체는 primary 객체로 동시화되어야 한다.



(그림 6) Observer 적용 예

4. 결론 및 향후 연구과제

본 논문에서는 재사용성과 유연성을 가진 객체지향 시스템 설계를 위해, 객체지향 시스템 설계방법 중에서 가장 널리 사용되는 OMT방법론에 몇 가지 설계패턴이 적용되는 방법을 제안하였다. 시스템을 개발하는데 방법론과 설계패턴이 함께 제공됨으로써 재사용성을 가지며, 건실한 객체지향 시스템을 개발하기 위해 상세한 설계지식을 제공하고, 이들 사이의 단점을 보완해주고 있음을 제시하였다.

본 논문에서는 OMT방법론에 Facade설계패턴을 적용시킴으로써 서브시스템의 public 인터페이스가 이루어질 경우, 내부객체들이 서브시스템의 public 인터페이스의 일부분이 되어야 한다는 것에 대한 구체적인 규칙을 제공할 수 있었으며, 또한, Mediator의 적용으로 OMT 방법론에서 구현과 연관된 협력을 자연스럽게 설계하는 방법을 제시하였고, Observer를 적용시킴으로써, OMT가 dependsOn의 의미를 가진 객체들과의 특별한 연관성을 가지고 있지 않다는 문제점을 해결할 수 있었다.

향후 연구과제는 3가지 설계패턴 이외의 설계패턴들

도 OMT방법론에 적용시켜 볼 것이다. 또한 다른 객체지향방법론들에 대해 설계패턴들이 효과적으로 적용될 수 있는 방법에 관한 연구와, 공통 설계패턴이 각 방법론들에 대해서 같은 역할과 기능들을 제공할 수 있는지에 대한 연구가 이루어 질 것이다.

참 고 문 헌

[1] Rebecca Wirfs-Brock, Brian Wilkerson and L. Wiener, "Design Object-Oriented Software," Peretice Hall, Englewood Cliffs, NJ, 1990.

[2] Grady Booch, "Object-Oriented Design with Applications," Second Edision, Benjamin Cummings, 1993.

[3] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, "Object-Oriented Software Engineering : A Use Case Driven Approach," Addison-Wesley, 1992.

[4] James Rumbaugh, M.Blaha, W.Premerlani, F.Eddy and W.Lorenson, "Object-Oriented Modeling and Design," Prentice Hall, Englewood Cliffs, NJ, 1991.

[5] Peter Coad, Edward Yourdon, "Object-Oriented Analysis," Prentice-Hall, Englewood Cliffs, NJ, 1991.

[6] 배제민, "설계패턴이 적용된 인트라넷 어플리케이션의 객체모델링", 한국정보처리학회 논문지 제4권 제8호, 1997.

[7] E.Gamma, R.Helm, R.Johnson, and J.Vlissides, "Design Pattern : Elements of Resuable Object-Oriented Software," Addison-Wesley, 1995.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal," Pattern-Oriented Software Architecture, A Pattern System," Draft 1995.

[9] Erich Gamma, Richard Helm, Ralph Jojnson and John Vlissides, "Design Patterns : Abstraction and Reuse of Object-Oriented Design," ECOOP '93.

[10] Ralph E. Johnson and Jonathon Zweig, "Delegation in C++," Journal of Object-Oriented Programming, Nov./Dec., 1991.

[11] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel, "A Pattern Language," Oxford University Press, NY, 1977.

[12] James Rumbaugh, "Object-Oriented Modeling Technology(OMT)", Tutorial, OOPSLA'94.

[13] Arj Jassksi, "OMT implementation with C++," TOOLS USA, 1996.

[14] http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/

[15] http://siesta.cs.wussi.edu/~schmidt/tutorials~patterns.html

[16] http://wwwipd.ira.uka.edu/~tichy/patterns/overview.html

김 치 수

e-mail : cskim@knu.kongju.ac.kr

1984년 중앙대학교 전자계산학과 (학사)

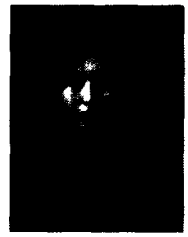
1986년 중앙대학교 전자계산학과 (석사)

1990년 중앙대학교 전자계산학과 (박사)

1990년 9월~1992년 8월 공주교육대학교 전임강사

1992년 9월~현재 공주대학교 전자계산학과 부교수

관심분야 : 소프트웨어공학, 객체지향분석·설계



임 경 미

e-mail : omnibus@kcs.kongju.ac.kr

1996년 한국방송통신대학교 전자계산학과(학사)

1998년 공주대학교 전자계산학과 (석사)

1998년 3월~현재 공주대학교 전자계산학과 시간강사

1999년 3월~현재 공주대학교 전자계산학과 박사과정

관심분야 : 소프트웨어공학, 객체지향분석·설계

권 민 주

e-mail : gaia@kcs.kongju.ac.kr

1998년 대전산업대학교 전자계산학과(학사)

1998년~현재 공주대학교 전자계산학과(석사과정)

관심분야 : 소프트웨어공학, 객체지향분석·설계