분석 패턴을 이용한 객체지향 시스템 개발방법론*

盧 尙 奎**

《目 次》

요 약

객체 지향은 코드, 디자인, 분석 요소의 재활용을 통해 정보시스템 개발 생산성을 비약적으로 향상시킬 수 있는 가능성을 가지고 있다. 이런 가능성을 실현하기 위해 본 논문에서는 유사한 검색과 갱신 기능을 가지는 도메인 독립적인 객체 패턴에 기반한 시스템 개발 방법론을 제안하고 이 방법론이 시스템 분석과 구현에 어떠한 영향을 미치는가 평가하였다. 이 방법론에서는 우선 패턴을 이용하여 개념적 객체모델을 작성하고 패턴을 지원하는 개발도구를 사용하여 모델을 구현 한 후 시스템을 수정 및 보완한다. 이 방법론의 사용자들은 방법론이 시스템개발의 생산성을 향상시키는 것으로 평가하였다.

Ⅰ. 본 연구의 배경 및 필요성

객체 지향은 코드와 디자인 요소의 재활용을 통해 정보시스템 개발 생산성을 비약적으로 향상시킬 수 있는 가능성을 가지고 있다. 그러나 객체지향을 활용함에 따라 예상되는 이익은 완전히 실현되지는 않았는데, 그 이유는 재활용 가능한 객체의 구현, 재활용될 객체의 결정, 객체를 재활용하는 방법의 학습 등에 있어서 난점을 가지고 있기 때문이었다. [Curtis,

* 본 연구는 서울대학교 경영대학 경영연구소의 연구비 지원에 의하여 수행되었음.

** 서울대학교 경영대학 기금전임강사

1989; Izakowitz and Kaufman, 1995].

설계 패턴은 높은 수준의 설계와 코드 재활용을 실현해주는 기술이다. [Budinsky et al., 1996; Gamma et al., 1995; Pree, 1995; Schmidt et al., 1996]. 설계 패턴은 반복되는 설계 문제에 대한 검증된 해결책을 구조적으로 표현하고, 해결책의 적용가능성, 상호 반작용 관계, 결과 등을 표현한다. 이는 또한 Smalltalk이나 C++과 같은 프로그래밍 언어에서 해결책을 구현하는 방법을 포함할 수도 있다. 따라서 설계패턴은 시스템 개발자에게 시스템 구현에 대한 검증된 해결책과 가이드 라인을 제공하게 된다. 그러나 이런 패턴들은 분석보다는 소프트웨어 설계에 적용된다.

본 논문에서는 개념적 객체 모델의 설계와 구현에 패턴의 개념을 적용하였다. 유사한 갱신 기능을 가지는 객체 패턴을 정의하고, 표준 템플릿을 개발하였다. 본 논문에서는 일곱 개의 패턴(독립, 외부식별자, 재귀, 서브클래스, 부자, 완전교차, 순환제약)을 제안하고, 각각의 패턴에 대한 표준 템플릿을 개발함으로써 이런 패턴들의 효익을 예시하고자 한다. 개발자는 이런 패턴을 이용하여 개념적 객체 모델을 설계하고, 메타클래스를 재활용함으로써 시스템 구현 시간을 단축할 수 있다.

비록 모든 부분을 완전하게 지원하는 것이 아니지만, 이런 패턴의 활용은 객체 모델링의 학습을 쉽게 하며, 개념적으로 더욱 풍부한 객체 모델의 기초가 된다. 또한 패턴의 활용은 설계와 코드 구성요소의 재활용을 가능하게 함으로써 개발자의 노력을 감소시킬 수 있다. 본 논문의 개발 도구는 코드를 생성하는 것이 아니라 메타클래스를 상속함으로써 애플리케이션이 만들어진다는 점에서 대부분의 코드 생성 도구와 차별성이 있다. 애플리케이션 수정도 더욱 용이하게 이루어질 수 있는데, 그 이유는 생성된 코드를 수정하는 것이 아니라 메소드를 재정의(overloading)함으로써 수정되기 때문이다. 또한 메타클래스의 개선이 도구를 사용하여 개발된 애플리케이션에 반영되기 때문에 애플리케이션의 점진적 개선을 가능하게 해준다.

이러한 방법론의 효과를 평가하기 위하여 설문을 통해 방법론의 효과성, 패턴이 분석에 미치는 영향, 개발도구가 구현에 미치는 영향 등에 대해 조사하였다. 그 결과 시스템개발의 생산성을 어느 정도 향상시키는 것으로 나타났다.

이 논문의 나머지 부분은 다음과 같이 구성되어 있다. 다음 장은 설계 패턴 및 분석 패턴에 대한 이전 연구에 대해서 논의한다. 다음 장은 예제 객체 모델을 이용하여 분석패턴을 이용한 객체지향시스템 개발 방법론에 대해 설명한다. 그 다음 장에서는 방법론에 대한 평가결과에 대해 논의한다. 마지막 장은 본 연구의 한계점과 향후 연구방향에 대해 논의한다.

Ⅱ. 이전의 연구

시스템 개발의 생산성을 향상시키기 위한 기술로서의 설계 패턴과 객체지향 시스템 개발 프레임웍에 대한 관심은 점차적으로 증가되어왔다. 설계 패턴이란 반복되는 설계 문제에 대한 일반적인 해결책을 구조적으로 설명하는 것이다.[Alexander, 1979; Alexander et al., 1977] 프레임웍은 "소프트웨어의 특정 클래스에 대해 재활용 가능한 설계를 구성하는 상호 협력하는 클래스들의 집합"으로 정의된다. [Gamma, et. al., 1995, p. 26]

Gamma et al. [1993, 1995]는 객체지향 설계를 표현하는 메카니즘으로 설계 패턴을 제안하였다. 그들의 패턴은 반복되는 설계문제에 대한 해결책과 이에 대한 이론적 근거, 구현을 위한 시사점, C++이나 Smalltalk 예제 코드 등을 설명한다. 그들은 패턴을 "전체 시스템 구조에 공헌하는 재활용 가능한 하부 구조"와 "설계를 위한 공통적인 언어"로 바라본다. Schmidt [1995]는 상업적인 통신 소프트웨어 개발에서 설계 패턴을 적용한 경험을 설명했다. 그는 객체지향 프로그래밍 언어에 대한 확장으로 패턴 언어가 형성되고, 궁극적으로 패턴의 시스템을 형성하는 프레임웍으로 패턴이 통합될 것이라고 예견하였다.

Budinsky et al. [1996]는 설계 패턴의 구현에 있어서 주의해야 할 점에 대해서 논의했다. 즉 설계 패턴이 각각 다른 시간에 개발되고 적용되기 때문에, 그리고 패턴의 적용에 있어서 고려해야 할 상호 반작용 관계가 다수 존재하기 때문에 발생하는 문제에 대해서 논의한 것이다. 이런 문제를 해결하기 위해 그들은 Gamma, et. al. [1993, 1995]에 의해 정의된 패턴을 구현하는 클래스들의 선언과 정의를 생성해 주는 툴을 개발하였다. 그들의 툴은 애플리케이션에 대한 특정 정보와 설계 상의 상호 반작용 관계에 대한 선택 등을 요구한다.

Coad [1992]는 객체지향 분석 및 설계의 영역에서 패턴의 개념에 대해 연구하였다. 객체지향 패턴은 클래스와 관계의 집합으로 구성되는 것으로 객체지향시스템을 구축하기 위한 벽돌과 같은 것이다. 그가 제시한 대부분의 패턴은 특정 상황에 부합하는 개념적 객체 모델을 구성하는데 클래스 집합이 어떻게 활용될 수 있는가를 설명한다. Fowler [1997]는 분석 패턴이라 명명되어지는 개념적 수준에서 패턴을 제시하였다. 분석 패턴은 하나 이상의 도메인에서 적용가능한, 특정한 경영 상황을 표현하는 클래스와 관계의 집단이다.

Rho and March [1997]는 분석 패턴[Fowler, 1997]과 프레임웍의 아이디어를 통합하여, 도메인 독립적인 분석 패턴에 기반한 사용자 인터페이스 개발법을 제안하였다. 그들은 개념적 객체 모델을 분석하여 유사한 갱신 기능과 인터페이스를 가지는 객체 패턴을 파악하

였다. 이런 패턴들은 개념적 수준이지만 도메인 독립적이라는 측면에서 Fowler의 패턴 [1997]과는 다르다. 또한 SOODAS (Semantic Object-Oriented Data Access System) [March and Rho, 1996a, 1996b]내에 이런 패턴을 구현하는 프레임웍을 개발하였다.

본 연구에서는 Rho and March [1997]의 분석패턴을 확장, 개선하고 분석패턴의 적용범위를 시스템분석, 설계 및 구축으로 확장하여 분석패턴에 기반한 시스템 개발방법론을 제안하고 기초적인 평가를 하고자 한다.

Ⅲ. 도메인 독립적인 분석 패턴을 이용한 시스템 개발

도메인 독립적인 분석 패턴에 기반한 객체지향 시스템 개발 방법론에 따르면, 먼저 개발자는 마치 레고의 블록처럼 패턴들을 활용하여 개념적 객체 모델을 구축한다. 일단 객체 모델이 만들어지면 패턴을 지원하는 개발 도구를 이용, 모델을 구현한다. 시스템과의 상호 작용에 기반하여 개발자는 객체 모델을 수정하고, 다시 구현할 수도 있다. 시스템의 요구사항이 충족되면, 개발자는 시스템을 보다 정교하게 수정할 수 있다.

비록 모든 부분을 완벽하게 지원하는 것은 아니지만, 이런 패턴의 활용은 객체 모델링의 학습을 쉽게 하며, 결과적으로 더욱 풍부한 의미(semantics)를 가지는 객체 모델을 구축하게 해준다. 또한 설계와 코드의 구성요소 재활용을 가능케 함으로써 개발자의 노력을 크게 줄여줄 수 있다. 본 연구의 개발 도구는 코드의 생성을 최소화하고 대부분의 기능이 메타 클래스를 상속함으로써 구현된다는 점에서 대부분의 코드 생성도구와 차별성이 있다. 애플리케이션 수정도 더욱 용이하게 이루어질 수 있는데, 그 이유는 생성된 코드를 수정하는 것이 아니라 메소드를 재정의함으로써 수정되기 때문이다. 또한 메타클래스의 개선이 도구를 사용하여 개발된 애플리케이션에 반영되기 때문에 애플리케이션의 점진적 개선을 가능하게 해준다. 다음 장에서는 개발 방법론에 사용된 패턴에 대해서 설명한다.

1. 분석 패턴

클래스(class), 속성(attribute), 관계(relationship), 서브클래스(subclass)는 객체 표현에 있어서 일반적인 개념이다[Fowler and Scott, 1997]. 본 연구에서는 이런 개념을 사용하는 개념적 객체 모델에서 여러 도메인에 걸쳐 반복되는 패턴을 정의하였는데, 이는 독립(Independent), 외부식별자(External Identifier), 재귀(Recursive), 서브클래스(Subclass),

부자(Parent-Child), 완전교차(Full Intersection), 순환제약(Cyclic Constraint) 패턴이다. 이 패턴들은 Rho and March [1997]의 패턴을 확장, 향상시킨 것이다.

1) 독립 (Independent)

(1) 무제약적 독립 (Unconstrained Independent)

이 패턴은 클래스 인스턴스가 시스템 내의 다른 인스턴스와 독립적으로 존재하는 (추가되거나 수정될 수 있는) 단일 클래스 패턴이다. 이 패턴의 클래스는 식별자(identifier)로 속성(변수)들만을 가지며, 모든 관계가 자신에 대한 최대 카디낼러티(cardinality) 값으로 1, 관련된 다른 클래스에 대한 최소 카디낼러티 값으로 0을 가진다. 이 패턴의 클래스가 가지는 관계는 자신에 대한 최소 카디낼러티 값으로 0 또는 1, 다른 클래스에 대한 최대 카디낼러티 값으로 1 또는 many를 가질 수 있다. 클래스의 인스턴스들은 일 대 다 관계(해당 클래스에 최소 카디낼러티 1을 가지는)에 대한 참조 대상으로서의 역할을 수행하는 경우가 있다. 이럴 경우에는 (a) 관련된 인스턴스를 삭제할 것인지(cascade delete) 아니면 (b) 삭제를 허락하지 않을 것인지를 삭제 기능을 결정해야할 필요가 있다.

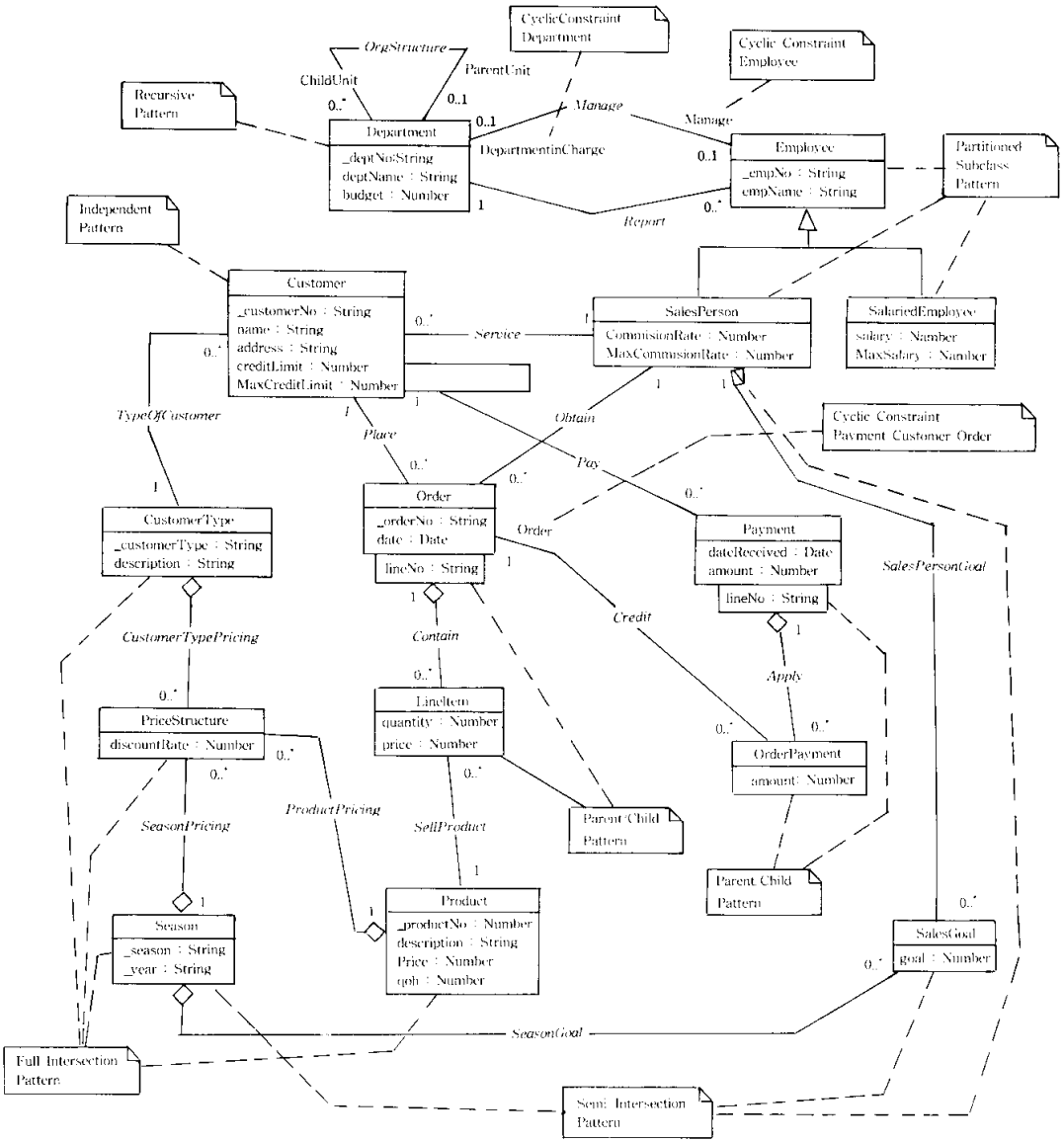
(2) 제약적 독립 (Constrained Independent)

이 패턴은 클래스 인스턴스가 최소한 하나 이상의 다른 클래스 인스턴스를 참조하는 단일 클래스 패턴이다. 이 패턴의 클래스는 식별자로 속성(변수)들만을 가지며, 자신에 대해 최소 카디낼러티 0, 관련된 다른 클래스에 대해 최대 카디낼러티 1인 관계를 최소한 하나 이상 가진다. 만약 다른 클래스에 대해 최소 카디낼러티가 1이면 참조무결성(referential integrity) 제약조건이 존재한다. 즉 이 패턴에 속하는 클래스의 인트턴스들은 다른 클래스의 관련된 인스턴스 없이 존재할 수 없다. 참조무결성 제약조건을 위배하는 추가, 관리 기능은 허락되지 않는다.

〈그림1〉에서 Customer는 제약적 독립 객체이다. 이 클래스는 자신의 식별자로 customerNo라는 속성만을 가지고 있다. 이 클래스는 Service와 TypeOfCustomer, 두 개의 관계에 참여하고 있는데, 각각 자신에 대해 최소 카디낼러티 0, SalesPerson과 CustomerType에 대해 최소 카디낼러티 1을 가지고 있다. 즉 Customer는 SalesPerson과 CustomerType에 대한 참조 무결성 제약조건을 가지고 있다. Customer 인스턴스는 관련된 SalesPerson 인스턴스와 CustomerType 인스턴스 없이 존재할 수 없다. Customer는 Pay와 Place라는 또다른 관계에도 참여하고 있는데, 이 관계에 대해서는 참조 대상 클래스이다.

이 패턴에 속하는 클래스들은 〈그림 2.a〉와 같은 검색 윈도우와 〈그림 2.b〉와 같은 갱신 윈도우를 이용하여 관리될 수 있다. 갱신 윈도우는 1: many 관계를 관리하기 위한 구성요소(예

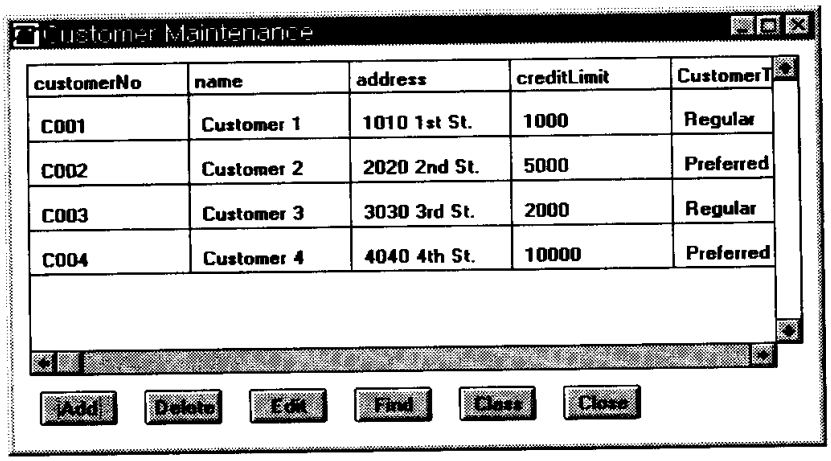
(그림 1) 개념의 객체모델



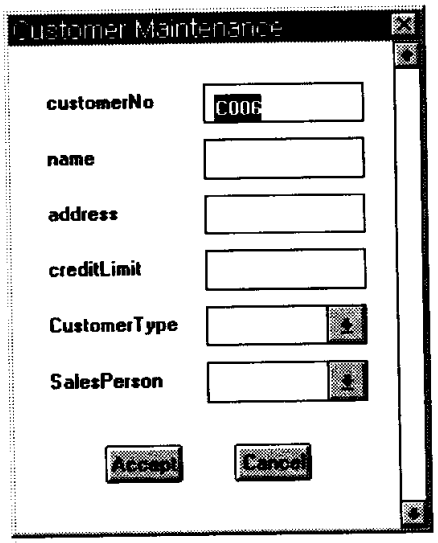
를 들어, 콤보 박스)를 가지고 있다. 이 구성요소는 foreign key 개념과 유사하게 관계의 다른 쪽에 있는 클래스의 외부 식별자(external identifier)를 사용하여 관계에 합당한 값들만을 표시한다. 참조무결성 제약조건을 가지는 관계를 관리하는 구성요소는 관련된 클래스의 외부 식별자를 표시하고, 사용자는 이 중의 한 값을 선택할 수 밖에 없다. 참조무결성 제약조건이 없는

〈그림 2〉 Customer관리를 위한 기본 사용자 인터페이스

a. 기본 검색 윈도우



b. 기본 인스턴스 갱신 윈도우



관계를 관리하는 구성요소는 관계에 참여하지 않는 인스턴스를 위해 널(null) 값을 포함한다.

2) 외부식별자 (External Identifier)

이 패턴은 클래스 자신과 서브클래스의 인스턴스들 사이에서 자신의 값을 구별할 수 있는

하나 이상의 속성 또는 관계로 이루어진다. 객체 시스템이 객체 자신을 식별하는 객체 식별자(object identifiers, OIDs)를 가지고 있지만, 이는 사용자에 의해 활용될 수 없다.

따라서 대부분 개념적 수준의 객체는 외부 식별자를 가지고 있다. 또한 외부 식별자의 대부분은 자동적으로 순서가 정해지는 숫자이다. 인스턴스를 추가할 때는 클래스의 다른 인스턴스와 동일한 외부 식별자 값을 가질 수 없다.

〈그림 1〉에서 orderNo는 Order의 외부 식별자이고, 자동적으로 번호가 정해진다.

3) 재귀 (Recursive)

이 패턴은 클래스 인스턴스가 자신과 같은 클래스의 인스턴스를 참조하는 단일 클래스 패턴이다. 이 패턴은 인스턴스가 자기자신이나 자신의 하위 인스턴스(방향에 따라서는 상위 인스턴스가 될 수도 있다.)를 참조할 수 없다는 것을 제외하면 위에 설명한 독립 패턴과 유사하다.

〈그림 1〉에서 Department는 재귀클래스이며, OrgStructure라는 재귀적 관계를 갖는다. Department의 인스턴스는 자신이나 하위 부서의 ChildUnit이 될 수 없다.

이 패턴에 속하는 클래스들은 독립클래스와 같은 윈도우를 사용하여 관리될 수 있다. 관계는 구성요소(콤보 박스)를 통해 관리되기 때문에 각 재귀적 관계를 관리하는 구성요소는 관계에 타당한 값만을 표시한다.

4) 서브클래스 (Subclass)

(1) 분할서브클래스 (Partitioned Subclass)

이 패턴은 서브클래스를 가지는 클래스로 구성된다. 이 패턴에 속하는 어떤 클래스의 인스턴스는 자신의 서브클래스 중 하나에 속한다. 클래스 인스턴스를 추가할 때는 반드시 서브클래스 중의 하나로 할당되어야만한다.

〈그림 1〉에서 Employee와 서브클래스들은 분할서브클래스 패턴을 형성한다. 이는 각 employee은 salaried employee이나 salesperson 중에 하나이어야만 한다는 것을 의미한다. 객체지향언어 용어에 따르면 Employee는 추상 클래스이다. 즉, Employee는 어떤 인스턴스도 가지지 않는다. Employee 클래스의 모든 인스턴스는 자신의 서브클래스에만 속해 있는데, 이때 SalariedEmployee나 SalesPerson가 수퍼클래스인 Employee를 분할한다고 한다.

클래스의 디스플레이 윈도우는 서브클래스의 모든 인스턴스를 포함하고 있다. 이 윈도우는 각 인스턴스가 속한 서브클래스를 나타내는 추가적인 열을 가지고 있다. 또한 한 인스턴스가

추가될 때는 어떤 서브클래스에 속할 것인지를 사용자가 선택할 수 있는 리스트가 나타난다. 각 서브클래스의 갱신 화면은 서브클래스가 가지는 모든 속성과 many 측면의 관계를 포함한다.

(2) 배타적서브클래스 (Exclusive Subclass)

이 패턴 또한 하나의 클래스와 서브클래스로 구성된다. 이 패턴은 분할서브클래스 패턴과 유사하지만, 인스턴스가 추가될 때, 그 클래스 자체나 서브클래스에 할당될 수 있다는 점에서 차이가 있다.

5) 부자 (Parent-Child)

이 패턴은 부모(Parent)와 자식(Child)로 불리는 두 개의 클래스로 구성된다. 부모 클래스는 자식 클래스와 1:many 관계를 가지고 있는데, 자신 쪽에 최소, 최대 카디널러티가 1이다. 관계는 자식의 외부 식별자의 부분이 된다(즉, Qualified Relationship). 전형적으로 자식의 인스턴스는 부모 인스턴스의 "부분"으로 간주된다. 참조무결성 제약조건과는 달리, 자식 인스턴스는 하나의 부모 인스턴스를 가진채 남아야한다. 부모와 자식은 다른 관계에도 참여할 수 있다. 부자 패턴은 배타적인 구성 객체의 위계를 가지는 복합(composite) 객체의 개념과 유사하다 [Kim et al., 1987]. 자식의 인스턴스는 부모의 인스턴스를 통해서만 입력될 수 있고, 부모 인스턴스 없이 존재할 수 없다. 자식의 인스턴스는 부모 인스턴스가 삭제됨에 따라 같이 삭제된다.

그림 1에서 Order와 LineItem은 부자패턴을 형성하고 있다. 이 두 클래스 사이에는 1:many 관계가 존재하며, LineItem의 외부 식별자에 Contain이라는 관계가 포함되어 있다. LineItem 인스턴스는 하나의 Order 인스턴스의 부분으로 간주된다. 하나의 Order 인스턴스와 이와 연관된 LineItem 인스턴스들의 집합은 복합 객체를 구성한다. Order와 LineItem 모두 다른 관계에 참여한다.

이 패턴에 속한 클래스들은 부모 인스턴스에 대한 상세 화면으로 자식 인스턴스들이 표시되는 통합된 인터페이스를 이용함으로써 관리될 수 있다.

6) 완전교차 (Full Intersection)

(1) 완전교차

이 패턴은 2개의 진입(Entry) 클래스와 1개의 완전교차 클래스로 이루어진다. 각 진입 클래스는 완전교차 클래스와 1:many 관계를 가지는데, 이 관계는 진입 클래스 측에 최소 카디널러티 1, 완전교차 클래스 측에 최소 카디널러티 0을 가진다. 진입 클래스들과 가지는 관

계는 완전교차 클래스의 외부식별자가 된다. 완전교차 클래스는 각 진입 클래스 인스턴스의 쌍에 대응하는 인스턴스를 하나씩 가지게 된다. 어떤 진입 클래스의 인스턴스를 하나 추가하는 것은 다른 진입 클래스의 기존 인스턴스들에 대해 교차(Intersection) 클래스 인스턴스를 하나씩 추가하는 것을 요구한다. 어떤 진입 클래스 인스턴스를 삭제하면 연관된 모든 교차 클래스 인스턴스도 삭제된다. 교차 클래스의 인스턴스는 진입 클래스 인스턴스를 통해서만이 추가되거나 삭제될 수 있다. 이 패턴은 두 개 이상의 진입 객체를 가지는 교차패턴으로 일반화될 수 있다.(즉 N개 이상의 관계를 표현할 수 있다.)

〈그림 1〉에서 CustomerType, PriceStructure, Season, Product는 완전교차 패턴을 형성한다. CustomerType, Product, Season은 진입 클래스이고, PriceStructure는 완전교차 클래스이다. PriceStructure의 외부식별자는 CustomerType, Product, Season과의 관계로 정의된다. 각 CustomerType, Product, Season 인스턴스의 조합에 대해 할인율을 정의하는 PriceStructure 인스턴스가 반드시 하나 존재해야 한다.

이 패턴에 속하는 클래스들은 부자 패턴과 유사한 인터페이스를 통해 관리될 수 있는데, 각 진입 클래스에 대한 상세화면에서 교차 클래스 인스턴스들이 표시된다.

(2) 부분교차 (Semi-Intersection)

이 패턴은 진입 클래스 인스턴스를 추가할 때, 다른 진입 클래스의 기존 인스턴스에 대응하는 교차 클래스 인스턴스를 생성하지 않는다는 점을 제외하면 완전교차 패턴과 유사하다.

〈그림 1〉에서 Season, SalesPerson, SalesGoal은 부분교차 패턴을 형성한다. Season, SalesPerson은 진입 클래스이다. SalesGoal은 부분교차 클래스이다. 이 패턴은 SalesPerson 인스턴스가 추가될 때, 기존 Season 인스턴스에 대응하는 SalesGoal 인스턴스가 추가될 필요가 없다는 점에서 완전교차 패턴과 다르다.

7) 순환제약 (Cyclic Constraint)

이 패턴은 제한적 관계 루프(constrained relationship loop)에 속하는 클래스의 집합으로 구성된다. 제한적 관계 루프는 하나의 클래스가 다른 클래스로 2개의 관계 경로(many:1관계의 집합)를 가지는데 전자의 한 인스턴스에서 시작하면 각 경로의 끝에 존재하는 후자의 동일한 인스턴스에 도달하게 된다.

예를 들면 부자 패턴의 자식 클래스가 부모를 통한 시간 의존적 관계 루프일 때 이런 패턴이 존재한다. 이 경우 부모를 통하는 경로는 기본적인 함수적 의존관계(functional dependency)를 표현하는 관계경로 상의 인스턴스들을 생성하기 전에 결정되는 transitive

dependency를 표현한다. 결과적으로 관계 루프 상에 존재하는 자식 클래스의 다른 관계에 대해 입력 가능한 인스턴스들은 부모 경로를 통해 제한된다.

〈그림 1〉에서 부자 패턴인 Payment와 OrderPayment가 순환제약 패턴을 형성한다. 어떤 고객이 대금을 지불할 때, Payment 인스턴스가 생성되고, Pay라는 관계를 경유하여 고객(Customer)과의 관계가 생성된다. 이 때 지불은 같은 고객에 의해 발주된 주문에 할당되어야만 한다. OrderPayment 인스턴스는 Order 인스턴스에 대한 Payment 인스턴스의 할당을 포함한다. 순환제약은 Payment 인스턴스가 자신과 연관된 Customer 인스턴스, Customer 인스턴스와 연관된 Order 인스턴스에만 할당될 수 있다는 사실을 표현한다. 이 클래스들과 정의된 관계를 이용하여 제약조건을 표시하면 다음과 같다. (여기에서 anOrderPayment는 OrderPayment의 인스턴스이다.)

anOrderPayment Apply Pay = anOrderPayment Credit Place.

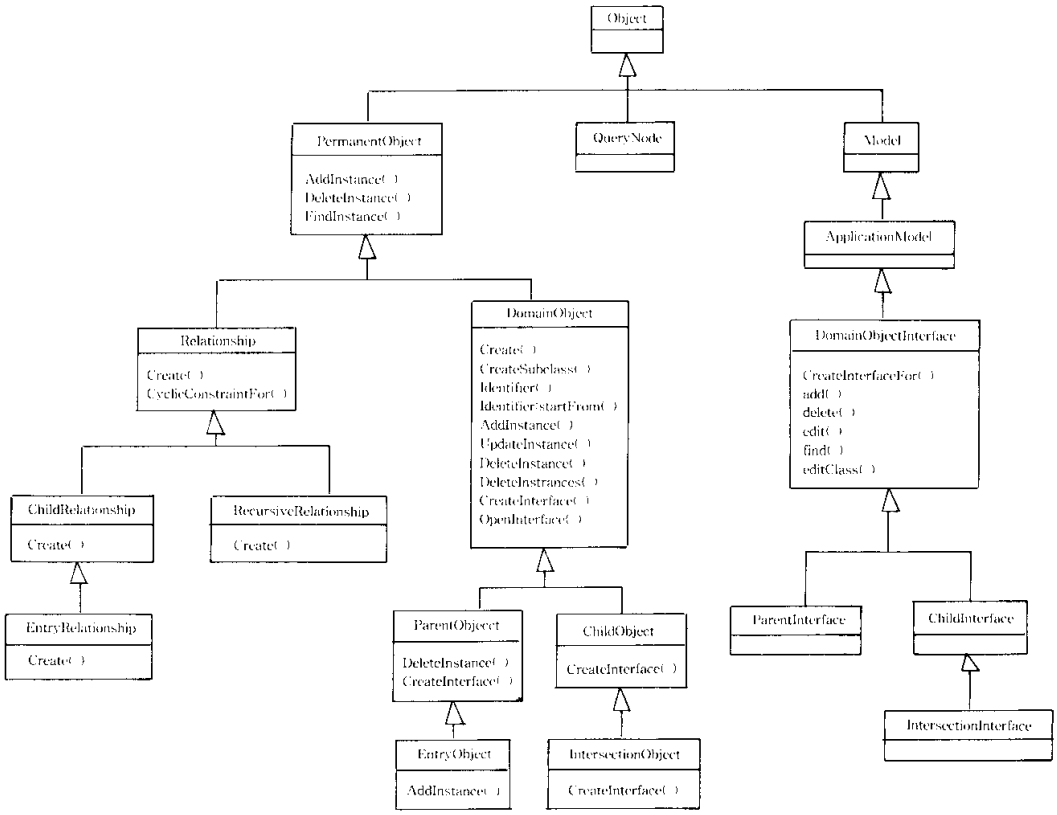
여기에서 Apply, Pay, Credit, Place는 관계를 검색하는 메시지이다. SOODAS에서는 데이터베이스가 정의될 때 이에 상응하는 메소드가 자동으로 생성된다. [March and Rho, 1996a, 1996b].

갱신 측면에서 보면 어떤 인스턴스에 연관될 수 있는 인스턴스들의 집합은 주어진 제약조건에 의해 제한된다. 예를 들면 OrderPayment 인스턴스에 연관될 수 있는 Order 인스턴스의 리스트는 OrderPayment 인스턴스와 연관된 Payment 인스턴스, Payment 인스턴스에 연관된 Customer 인스턴스와 연관된 것으로 제한된다.

2. 분석패턴을 위한 프레임웍

위의 분석 패턴을 지원하기 위하여 다음과 같은 메타클래스를 정의하였다: ***ParentObject***, ***EntryObject***, ***ChildObject***, ***IntersectionObject***, ***RecursiveRelationship***, ***ChildRelationship***, ***EntryRelationship***, ***DomainObjectInterface***, ***ParentInterface***, ***ChildInterface***와 ***IntersectionInterface*** (그림 3 참조). 메타클래스는 SOODAS [March and Rho, 1996a, 1996b]의 일부로 정의되었다. SOODAS는 객체지향언어 Smalltalk에 ER 개념(semantics)을 추가한 시스템 개발도구로써 클래스를 자신의 서브클래스로 구현하는 ***DomainObject***, 관계를 정의하고 관리하는 ***Relationship***, 지속성을 지원하는 ***Permanent Object***, 그리고 다중클래스 질의를 지원하는 ***QueryNode***, 네 개의 메타클래스로 구성되어 있다.

〈그림 3〉 메타클로스 구조



*DomainObject*와 *Relationship*은 독립패턴, 서브클래스, 외부식별자 및 순환제약패턴을 지원한다. *RecursiveRelationship*은 재귀패턴을 지원하며 이는 *Relationship*의 서브클래스로서 재귀관계 제약조건을 강제한다.

ParentObject, *ChildObject* 및 *ChildRelationship*은 부자 패턴을 지원한다. *ParentObject*는 *DomainObject*의 서브클래스로 정의되며 *DeleteInstance*: 클래스 메소드가 부모클래스 인스턴스가 삭제될 때 자식클래스 인스턴스도 삭제되도록 재정의되었다. *ChildObject*와 *ChildRelationship*은 각각 *DomainObject*와 *Relationship*의 서브클래스로 정의되었다.

EntryObject, *IntersectionInterface* 및 *EntryRelationship*은 완전교차 패턴을 지원한다. *EntryObject*는 *ParentObject*의 서브클래스로 정의되며 *AddInstance*: 클래스 메소드는 진입클래스 인스턴스 뿐만 아니라 관련된 완전교차클래스 인스턴스들도 삽입하도록 재정의되었다. *IntersectionObject*와 *EntryRelationship*은 각각 *ChildObject*와 *ChildRelationship*

의 서브클래스로 정의된다.

DomainObjectInterface, ***ParentInterface***, ***ChildInterface*** 및 ***IntersectionInterface***는 사용자 인터페이스의 템플릿(template)으로써 ***DomainObject***와 그 서브클래스들의 사용자 인터페이스를 구축하는 데 사용된다[Rho and March, 1997]. ***ParentInterface***는 부모 및 진입 클래스의 템플릿이고 ***ChildInterface***와 ***IntersectionInterface***는 각각 자식과 교차클래스의 템플릿이다. 분석 패턴들은 각각 다음과 같은 방법으로 구현된다(부록 1은 그림 1의 객체모델을 구현하는 방법이다).

1) 독립 (Independent)

독립패턴은 ***DomainObject***에 Create:attributes:classAttributes:under: 메시지를 보내고 독립클래스가 참여하는 관계마다 ***Relationship***에 Create:between:withMin:max:andMin:max: 메시지를 보냄으로써 구현된다. 예를 들어 Customer와 그 관계(이 경우 CustomerType과의 관계)를 구현하는 방법은 다음과 같다.

```
DomainObject Create: #Customer attributes: 'customerNo String name String
        address String creditLimit Number' classAttributes: 'MaxCreditLimit
        Number' under: 'OrderEntry'.
Relationship Create: 'TypeOfCustomer' between: CustomerType and: Customer
        withMin: 1 max: 1 andMin: 0 max: Many.
```

DomainObject는 인스턴스변수로 customerNo, name, address 및 creditLimit을 가지고, 클래스변수로 MaxCreditLimit을 가지는 Customer라는 서브클래스를 생성한다. ***Relationship***은 Customer와 CustomerType과의 관계인 'TypeOfCustomer'를 Relationship의 인스턴스로 생성한다. 독립클래스의 사용자 인터페이스는 생성된 클래스에 CreateInterface 메시지를 보냄으로써 구현된다. Customer의 인터페이스는 다음과 같은 방법으로 생성된다.

```
Customer CreateInterface.
```

그 결과 CustomerInterface라는 DomainObjectInterface의 서브클래스가 생성된다(그림 2 참조).

2) 외부식별자 (External Identifier)

외부식별자는 클래스에 Identifier: 메시지를 보냄으로써 구현된다. Customer의 외부식별자는 다음과 같이 구현된다.

```
Customer Identifier: 'customerNo'.
```

만약 외부식별자의 값이 자동으로 생성되기를 원하는 경우는 Identifier:startFrom:메시지를 사용한다.

3) 재귀 (Recursive)

재귀클래스를 구현하는 방법은 독립패턴과 거의 같으며 다만 재귀관계의 정의시 Recursive Relationship에 메시지를 보낸다. 예를 들어 Department는 다음과 같이 정의된다.

```
DomainObject Create: #Department attributes: 'deptNo String deptName
        String budget Number' under: 'OrderEntry'.
RecursiveRelationship Create: 'OrgStructure' between: Department and:
        Department withRoleName: 'ParentUnit' min: 0 max: 1 andRoleName:
        'ChildUnit' min: 0 max: Many.
Department Identifier: 'deptNo' startFrom: 1.
Department CreateInterface.
```

4) 서브클래스 (Subclass)

서브클래스는 생성된 수퍼클래스에 CreateSubtype:attributes:classAttributes: 메시지를 보냄으로써 구현된다. 분할서브클래스패턴의 경우 수퍼클래스에 Partition: true 메시지를 보내고 배타적서브클래스인 경우 메시지를 보내지 않는다. Employee와 그 서브클래스들은 다음과 같이 구현된다.

```
DomainObject Create: #Employee attributes: 'empNo String empName String'
        under: 'OrderEntry'.
Employee CreateSubtype: #SalariedEmployee attributes: 'salary Number'
        classAttributes: 'MaxSalary Number'.
Employee CreateSubtype: #SalesPerson attributes: 'commisionRate Number'
```

```
        classAttributes: 'MaxCommisionRate Number'.
Employee Partition: true.
Employee Identifier: 'empNo'.
Employee CreateInterface.
```

5) 부자 (Parent-Child)

부모와 자식 클래스는 각각 ParentObject와 ChildObject에 메시지를 보냄으로써 구현하고 두 클래스간의 관계는 ChildRelationship를 이용하여 생성한다. 자식클래스의 외부식별자는 부모와의 관계명(또는 역할명)를 포함한다. 예를 들어, Order와 LineItem은 다음과 같이 구현된다.

```
ParentObject Create: #Order attributes: 'orderNo String date Date' under:
      'OrderEntry'.
ChildObject Create: #LineItem attributes: 'lineNo String quantity Number
      price Number' under: 'OrderEntry'.
ChildRelationship Create: 'Contain' between: Order and: LineItem withMin: 1
     max: 1 andMin: 0 max: Many.
Order Identifier: 'orderNo' startFrom: 1.
LineItem Identifier: 'Order lineNo'.
Order CreateInterface.
LineItem CreateInterface.
```

6) 완전교차 (Full Intersection)

진입클래스와 교차클래스는 각각 ***EntryObject***와 ***IntersectionObject***의 서브클래스로 구현하고 두 클래스간의 관계는 ***EntryRelationship***을 이용한다. 예를 들어 완전교차패턴인 Product/CustomerType/Season/PriceStructure는 다음과 같이 구현한다.

```
EntryObject Create: #Product attributes: 'productNo String description String
      price Number qoh Number' under: 'OrderEntry'.
EntryObject Create: #CustomerType attributes: 'customerType String description
      String' under: 'OrderEntry'.
EntryObject Create: #Season attributes: 'season String year String' under:
      'OrderEntry'.
```

```
IntersectionObject Create: #PriceStructure attributes: 'discountRate Number'
    under: 'OrderEntry'.
EntryRelationship Create: 'CustomerTypePricing' between: CustomerType and:
    PriceStructure withMin: 1 max: 1 andMin: 0 max: Many.
EntryRelationship Create: 'ProductPricing' between: Product and: Price
    Structure withMin: 1 max: 1 andMin: 0 max: Many.
EntryRelationship Create: 'SeasonPricing' between: Season and: Price
    Structure withMin: 1 max: 1 andMin: 0 max: Many.
Product Identifier: 'productNo' startFrom: 1.
CustomerType Identifier: 'customerType'.
Season Identifier: 'season year'.
PriceStructure Identifier: 'CustomerType Product Season'.
Product CreateInterface.
CustomerType CreateInterface.
Season CreateInterface.
PriceStructure CreateInterface.
```

부분교차패턴의 경우에는 ***EntryObject*** 대신 ***ParentObject***를 이용하여 진입클래스를 생성한다. 예를 들어 SalesPerson은 ParentObject의 서브클래스로 구현한다(부록 1 참조).

7) 순환제약 (Cyclic Constraint)

순환제약패턴은 Relationship클래스에 CyclicConstraintFor:of:relationshipPath: 메시지를 보냄으로써 구현한다. Credit관계의 순환제약은 다음과 같이 구현한다.

```
Relationship CyclicConstraintFor: 'Order' of: OrderPayment relationshipPath:
    'Payment Customer Order'.
```

Ⅳ. 개발 방법론에 대한 탐색적 평가

앞에서 제안한 시스템개발 방법론의 유용성을 평가하기 위해 서울에 소재한 대학교의 대학원 과정 객체지향프로그래밍 강좌를 수강한 학생을 대상으로 설문조사 및 인터뷰를 실시하였다. 이 강좌에서는 한 학기동안 객체지향 모델링 및 분석패턴 그리고 객체지향언어인

Smalltalk 및 SOODAS등을 다루었으며 기말 프로젝트를 수행한 후 학생들의 방법론의 유용성에 대한 인식을 조사하였다. 좀더 엄정한 평가방법으로는 통제집단과 실험집단을 이용한 실험이 있지만 우선 이 평가가 탐색적 성격을 가지고 있다는 점과 교육여건(예를 들어, 한 강좌를 통제집단과 실험집단으로 구분하기 어려운점) 등을 고려하여 설문조사 방법을 택하였다.

방법론의 평가척도로는 인식된 유용성(Perceived Usefulness) (이후, 유용성)을 선택하였다. 유용성척도는 Davis [1989]에 의해 개발되었으며 이후 많은 연구에 의해 검증된 척도이다. 물론, 유용성척도는 최종사용자의 정보기술 채택연구에서 활용되는 것이지만 본 평가의 탐색적 목적에도 적절하다고 사려되어 평가척도로 선택하였다. 이 평가에서는 유용성을 3가지 측면에서 측정하였다(부록 3 참조). 첫째는 정보요구사항 분석 및 이해에 있어서 분석패턴의 유용성이고(문항 1-6), 둘째는 응용시스템 프로그래밍에 있어서 분석패턴을 지원하는 프레임웍의 유용성이고(문항 7-14), 세째는 시스템 개발 프로젝트 수행에 있어서 방법론의 유용성이다(문항 15-21). 그 외에 사용자의 시스템개발 교육정도 및 경력에 관한 문항도 포함하였다.

방법론에 대한 교육은 강좌 전반에 걸쳐 강의 및 논문을 통해 이루어졌으며 튜토리얼을 통해 사용자들이 방법론을 실습할 기회를 주었다. 학기말에는 4인이 한 조가 되어 프로젝트를 수행하였으며 프로젝트로 학생들이 쉽게 접할 수 있으면서도 현실적인 수강신청시스템 개발을 선정하였다(부록 2 참조).

설문결과의 신뢰도를 분석한 결과 유용성척도는 신뢰할 수 있는 것으로 나타났으므로 (Cronbach's alpha는 각각 0.90, 0.98, 0.95였음) 각 유용성의 평균 및 유용성간의 상관관계를 분석하였다. 분석결과는 표 1과 2에 요약되어 있다.

〈표 1〉 변수의 기본통계치

	Mean	Std. Deviation	N
분석패턴의 유용성	5.1077	0.7612	13
프레임웍의 유용성	4.8692	1.4641	13
방법론의 유용성	5.0000	1.1283	13

분석패턴의 유용성은 5.1로 시스템 분석에 있어서 분석패턴은 유용한 것으로 나타났다(표 1 참조). 이 값은 중간값인 4보다 큰 것으로 나타났다($p=.000$). 프레임웍의 유용성은 4.9로 애플리케이션의 구현에 있어서 프레임웍은 유용성이 약간 떨어지는 것으로 나타났다

(p=.054). 방법론의 유용성은 5.0으로 시스템개발 전반에 있어서 분석패턴에 기반한 개발 방법론은 유용한 것으로 나타났다(p=.008). 프레임웍의 유용성이 떨어지는 것은 객체지향 프로그래밍의 높은 학습곡선에 있다 하겠다. 한 학기라는 짧은 기간에 객체지향언어인 Smalltalk과 그에 기반한 프레임웍을 충분히 학습한다는 것이 어려웠던 것으로 설문조사 결과 나타났다. 한 사용자에 의하면 "생산성 등에 효과를 가져오기까지는 상당한 수련이 필요할 것으로 생각되며, 처음에는 오히려 혼란스럽기까지"한 것으로 나타났다. 또 다른 이유로는 프레임웍으로 생성된 시스템을 수정하기가 어려웠다는 점을 들 수 있다. 한 사용자는 "프레임웍을 개발자가 완벽하게 알고 있지 못 하면 [프로그램을] 고치기 힘들다는 점이 있다"라고 하여 수정의 어려움을 표현하였다. 하지만 대부분의 사용자들이 패턴을 이용한 개발방법론이 상당한 가능성이 있는 것으로 생각하고 있는 것으로 나타났다. 한 사용자는 그의 생각을 다음과 같이 표현했다: "개발과 발전의 가능성이 무한히 있어 보인다. 다만 분석패턴이 어떻게 실제로 구현되고 있는지 상세히 설명되거나 쉽게 이해될 수 있어야 할 것 같다. 그래야 좀 더 나은 시스템이 만들어 질 것 같다. 분석패턴만으로는 요구수준에 맞는 프로그램을 작성하기는 힘들 것이기 때문에."

〈표 2〉 변수간의 상관관계 (Pearson Correlations)

	분석패턴의 유용성	프레임웍의 유용성	방법론의 유용성	수강강좌
프레임웍의 유용성	.557 (.048)*			
방법론의 유용성	.664 (.013)	.843 (.000)		
수강강좌	-.439 (.133)	-.252 (.406)	-.069 (.822)	
경력	-.362 (.224)	-.260 (.391)	-.182 (.552)	.526 (.065)

* significance

각 척도간의 상관관계를 분석한 결과[1]

방법론의 유용성은 분석패턴의 유용성보다는 프레임웍의 유용성과 상관관계가 높은 것으로

1) 수강강좌 및 경력은 a,b,c,d 대신 0,1,2,3으로 코딩하였다. 물론 문제점이 없는 것은 아니지만 탐색적분석이라는 점을 감안하면 무방하다 하겠다.

나타났다. 이는 프로젝트의 성격이 애플리케이션 구현에 초점이 맞추어져 있었기 때문인 것으로 사려된다. 좀더 많은 시간을 투자한 애플리케이션 구현에 있어서의 유용함이 방법론의 전반적인 유용함에 큰 영향을 미치는 것은 당연하다 하겠다.

한 가지 흥미로운 결과는 사용자의 시스템분석 및 개발 관련 과목 수강강좌 수나 개발경력은 유용성과 상관관계가 없다는 것이다. 특히 통계학적으로 유의하지는 않지만 역의 상관관계를 가지고 있다는 것이다. 이는 객체지향 시스템 개발법이 과거의 방법론과 다른 패러다임이라는 것을 다시 한 번 증명하는 것이라 하겠다. 분석패턴에 의한 개발방법론이 별로 유용하지 않다고 한 사용자 중 하나는 "ER 모델링에 의한 관계형 데이터베이스개발이 객체지향 모델에 의한 것보다 훨씬 데이터를 유지하고 관리하기 쉽다"라고 하여 전통적 방법론이 새로운 방법론을 학습하는 데 방해가 되고 있음을 간접적으로나마 표현하였다.

결론적으로 본 연구에서 제시한 방법론은 사용자들이 유용한 것으로 평가하였으며 상당한 가능성이 있는 것으로 나타났다. 이를 위해서는 패턴을 지원하는 프레임웍이 시스템을 구현한 후 수정하기가 쉽도록 향상되어야 할 것이다.

V. 결론 및 향후 연구 과제

본 논문에서는 개념적 객체모델에서 자주 나타나는 유사한 갱신요구사항과 사용자 인터페이스를 가진 7가지 객체패턴에 기반한 시스템 개발방법론은 제안하였다. 이러한 분석 패턴으로는 독립, 외부식별자, 재귀, 서브클래스, 부자, 완전교차, 순환제약을 제안하였다. 시스템 개발자는 분석패턴을 이용하여 개념적 객체모델을 설계하고 패턴을 지원하는 프레임웍을 이용하여 시스템을 구현한 후 수정 및 보완을 하여 요구사항에 맞는 시스템을 개발한다. 이러한 방법론의 유용성을 평가하기 위해 방법론의 사용자를 대상으로 설문조사를 한 결과 유용한 것으로 평가되었으며 상당한 가능성이 있는 것으로 나타났다. 그러나 이 설문조사는 탐색적인 성격을 지닌 것으로 방법론의 유용성을 단정적으로 입증하지는 못 한다.

향후의 연구는 두 가지 방향으로 진행될 것이다. 첫째, 개발 방법론을 개선할 것이다. 우선 패턴을 실제 시스템개발에 활용함으로써 좀더 많은 패턴을 개발할 것이다. 현재의 패턴도 상당히 적용가능성이 높지만 유용한 패턴들이 더 존재할 것으로 기대되는 만큼 그러한 패턴을 파악하는 데 노력할 것이다. 그리고 프레임웍을 개선할 것이다. 메타클래스의 구조 및 메소드들을 이해하기 쉽게 구성하여 궁극적으로 프레임웍으로 구현된 시스템을 수정, 보완하기 용이하도록 할 것이다. 둘째, 방법론의 유용성을 좀 더 체계적으로 실증할 것이다. 우선 조사

방법론에 있어 단순 설문조사가 아닌 실험을 이용할 것이다. 또한 실험대상도 학생이 아닌 시스템분석가를 활용할 계획이다.

참 고 문 헌

Alexander, C., *The Timeless Way of Building*, Oxford University Press, New York, 1979.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S., A Pattern *Language: Towns, Buildings, Construction*, Oxford University Press, New York, 1977.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S., "Automatic Code Generation from Design Patterns," *IBM Systems Journal*, Vol. 35, No. 2, 1996.

Coad, P., "Object-Oriented Patterns," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 152-159.

Coad, P., North, D., and Mayfield, M., *Object Models: Strategies, Patterns, and Applications*, Yourdon Press, Englewood Cliffs, NJ, 1995.

Curtis, B., "Cognitive Issues in Reusing Software Artifacts," in Biggerstaff, T. J. and Perlis, A. J. (eds), *Software Reusability, Volume II*, Addison-Wesley, 1989, pp. 269-287.

Davis, F. D., "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology," *MIS Quarterly*, Vol. 13, No. 3, September 1989, pp. 319-340.

Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, MA, 1997.

Fowler, M. and Scott, L., *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, Reading, MA, 1997.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *Proceedings of ECOOP '93 Conference*, 1993.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Goldberg, A. and Robson, D., *Smalltalk-80*, Addison-Wesley, Reading, MA, 1989.

Isakowitz, T. and Kauffman, R. J., "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June 1996, pp 407-423.

Johnson, R., "Documenting Frameworks Using Patterns," *Proceedings of OOPSLA '92*, Vancouver, Canada, October 1992, pp. 63-76.

Kim, W., Banerjee, J., Chou, H.-T., "Composite Object Support in an Object-Oriented Database System," *Proceedings of OOPSLA '87*, October 4-8, 1987, pp.118-125.

March, S. T. and Rho, S., "Object Support for Entity Relationship Semantics," *Proceedings of Workshop on Information Technologies and Systems*, December 1996a, pp. 1-10.

March, S. T. and Rho, S., "A Semantic Object-Oriented Data Access System," MISRC working paper, WP96-05, University of Minnesota, 1996b.

Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Publishing Company, Wokingham, England, 1995.

Rho, S. and March S. T., "Object-Oriented Development: Patterns, Interfaces, and Update Semantics," *Seoul Journal of Business*, 1997.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object -Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

Schmidt, D. C., "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Communications of the ACM*, Vol. 38, No. 10, October 1995, pp. 65-74.

Schmidt, D., Fayad, M., and Johnson, R., "Software Patterns," *Communications of the ACM*, Vol. 39, No. 10, October 1996, pp. 36-40.

VisualWorks, *User's Guide*, ParcPlace Systems, Inc., 1994.

부록 1. 그림 1의 객체모델을 구현하기위한 SOODAS 스크립트

```
"Define the domain objects."
DomainObject Create: #Customer attributes: 'customerNo String name String
        address String creditLimit Number' classAttributes: 'MaxCreditLimit
        Number' under: 'OrderEntry'.
DomainObject Create: #Department attributes: 'deptNo String deptName
        String budget Number' under: 'OrderEntry'.
ParentObject Create: #Employee attributes: 'empNo String empName String'
        under: 'OrderEntry'.
ParentObject Create: #Order attributes: 'orderNo String date Date' under:
        'OrderEntry'.
ChildObject Create: #LineItem attributes: 'lineNo String quantity Number
        price Number' under: 'OrderEntry'.
ParentObject Create: #Payment attributes: 'checkNo String dateReceived
        Date amount Number' under: 'OrderEntry'.
ChildObject Create: #OrderPayment attributes: 'lineNo String amount
        Number' under: 'OrderEntry'.
EntryObject Create: #Product attributes: 'productNo String description String
        price Number qoh Number' under: 'OrderEntry'.
EntryObject Create: #CustomerType attributes: 'customerType String
        description String' under: 'OrderEntry'.
EntryObject Create: #Season attributes: 'season String year String' under:
        'OrderEntry'.
IntersectionObject Create: #PriceStructure attributes: 'discountRate Number'
        under: 'OrderEntry'.
IntersectionObject Create: #SalesGoal attributes: 'goal Number' under:
        'OrderEntry'.

"Define the subclasses."
Employee CreateSubtype: #SalariedEmployee attributes: 'salary Number'
        classAttributes: 'MaxSalary Number'.
Employee CreateSubtype: #SalesPerson attributes: 'commisionRate Number'
        classAttributes: 'MaxCommisionRate Number'.
Employee Partition: true.
```

"Define the relationships."

Relationship Create: 'TypeOfCustomer' between: CustomerType and: Customer withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'SellProduct' between: Product and: LineItem withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Place' between: Customer and: Order withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Manage' between: Department and: Employee with RoleName: 'DepartmentInCharge' min: 0 max: 1 andRoleName: 'Manager' min: 0 max: 1.

Relationship Create: 'Report' between: Department and: Employee withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Obtain' between: SalesPerson and: Order withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Service' between: SalesPerson and: Customer withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Pay' between: Customer and: Payment withMin: 1 max: 1 andMin: 0 max: Many.

Relationship Create: 'Credit' between: Order and: OrderPayment withMin: 1 max: 1 andMin: 0 max: Many.

RecursiveRelationship Create: 'OrgStructure' between: Department and: Department withRoleName: 'ParentUnit' min: 0 max: 1 andRole Name: 'ChildUnit' min: 0 max: Many.

ChildRelationship Create: 'Contain' between: Order and: LineItem withMin: 1 max: 1 andMin: 0 max: Many.

ChildRelationship Create: 'Apply' between: Payment and: OrderPayment withMin: 1 max: 1 andMin: 0 max: Many.

EntryRelationship Create: 'CustomerTypePricing' between: CustomerType and: PriceStructure withMin: 1 max: 1 andMin: 0 max: Many.

EntryRelationship Create: 'ProductPricing' between: Product and: PriceStructure withMin: 1 max: 1 andMin: 0 max: Many.

EntryRelationship Create: 'SeasonPricing' between: Season and: PriceStructure withMin: 1 max: 1 andMin: 0 max: Many.

EntryRelationship Create: 'SeasonGoal' between: Season and: SalesGoal withMin: 1 max: 1 andMin: 0 max: Many.

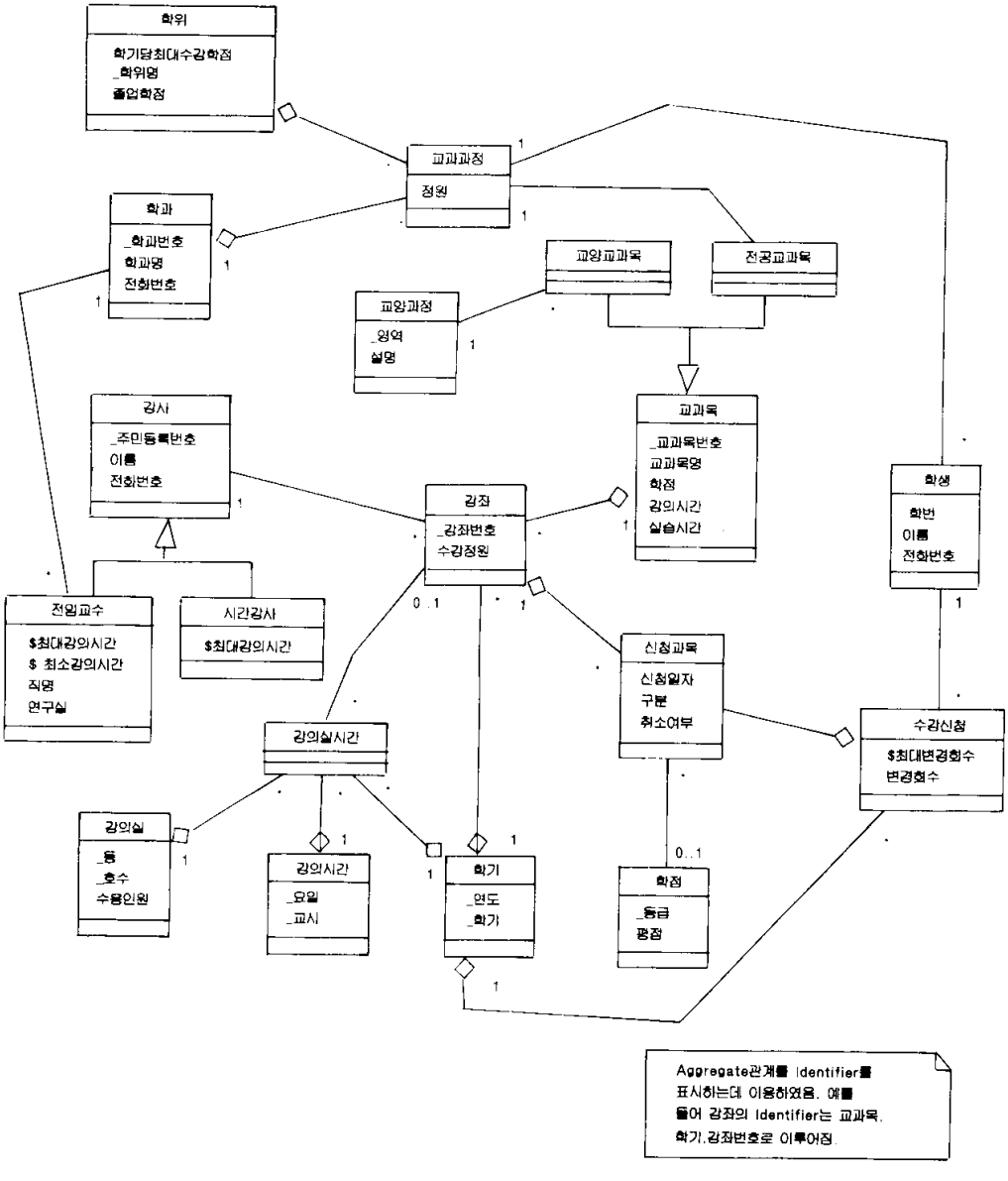
EntryRelationship Create: 'SalesPersonGoal' between: SalesPerson and: SalesGoal withMin: 1 max: 1 andMin: 0 max: Many.

```
"Define the cyclic constraints."
Relationship CyclicConstraintFor: 'Manager' of: Department relationshipPath:
        'Employee'.
Relationship CyclicConstraintFor: 'DepartmentInCharge' of: Employee relationship
        Path: 'Department'.
Relationship CyclicConstraintFor: 'Order' of: OrderPayment relationshipPath:
        'Payment Customer Order'.

"Define the external identifiers."
Customer Identifier: 'customerNo' startFrom: 1.
Department Identifier: 'deptNo' startFrom: 1.
Employee Identifier: 'empNo' startFrom: 1.
Order Identifier: 'orderNo' startFrom: 1.
LineItem Identifier: 'Order lineNo'.
Payment Identifier: 'Customer checkNo'.
OrderPayment Identifier: 'Payment lineNo'.
Product Identifier: 'productNo' startFrom: 1.
CustomerType Identifier: 'customerType'.
Season Identifier: 'season year'.
PriceStructure Identifier: 'CustomerType Product Season'.
SalesGoal Identifier: 'Season SalesPerson'.

"Create the intefaces."
Customer CreateInterface.
Department CreateInterface.
Employee CreateInterface.
Order CreateInterface.
LineItem CreateInterface.
Payment CreateInterface.
OrderPayment CreateInterface.
Product CreateInterface.
CustomerType CreateInterface.
Season CreateInterface.
PriceStructure CreateInterface.
SalesGoal CreateInterface.
```

부록 2. 방법론 평가에 사용된 프로젝트의 개념적 객체 모델



부록 3. 방법론 평가를 위한 설문지

분석패턴을 이용한 시스템 개발에 대한 평가

이 설문은 분석패턴을 이용한 시스템 개발에 대한 평가를 위한 것입니다. 분석패턴을 이용한 시스템 개발이란 분석패턴(예, parent/child)을 이용하여 정보요구사항(예, 객체모델)을 표현하고 분석패턴을 지원하는 Application Framework (예, SOODAS)을 이용하여 프로그래밍을 하는 것을 의미합니다.

1. 위에서 정의된 용어에 유의하여 다음항목을 평가하시오.

	전혀 그렇지 않다						매우 그렇다
1. 분석패턴은 요구사항을 보다 빨리 이해하게 하였다.	1	2	3	4	5	6	7
2. 분석패턴은 요구사항을 보다 명확하게 이해하게 하였다.	1	2	3	4	5	6	7
3. 분석패턴은 요구사항의 이해의 생산성을 높였다.	1	2	3	4	5	6	7
4. 분석패턴은 요구사항의 이해를 보다 효과적으로 하게 하였다	1	2	3	4	5	6	7
5. 분석패턴은 요구사항의 이해를 보다 쉽게 하였다.	1	2	3	4	5	6	7
6. 분석패턴은 요구사항의 이해에 유용하였다.	1	2	3	4	5	6	7
7. Application Framework(SOODAS)은 응용시스템을 보다 신속하게 프로그램하게 하였다.	1	2	3	4	5	6	7
8. Application Framework은 나의 응용시스템 프로그래밍 업적을 향상시켰다.	1	2	3	4	5	6	7
9. Application Framework은 응용시스템 프로그래밍의 생산성을 높였다.	1	2	3	4	5	6	7
10. Application Framework은 응용시스템 프로그래밍을 보다 효과적으로 하게 하였다	1	2	3	4	5	6	7
11. Application Framework은 응용시스템 프로그래밍을 보다 쉽게 하였다.	1	2	3	4	5	6	7
12. Application Framework은 응용시스템 프로그래밍에 유용하였다.	1	2	3	4	5	6	7
13. 분석패턴을 이용한 시스템 개발은 프로젝트를 보다 신속하게 수행하게 하였다.	1	2	3	4	5	6	7
14. 분석패턴을 이용한 시스템 개발은 나의 프로젝트 수행 업적을 향상시켰다.	1	2	3	4	5	6	7
15. 분석패턴을 이용한 시스템 개발은 프로젝트 수행의 생산성을 높였다.	1	2	3	4	5	6	7
16. 분석패턴을 이용한 시스템 개발은 프로젝트를 보다 효과적으로 수행하게 하였다	1	2	3	4	5	6	7
17. 분석패턴을 이용한 시스템 개발은 프로젝트 수행을 보다 쉽게 하였다.	1	2	3	4	5	6	7
18. 분석패턴을 이용한 시스템 개발은 프로젝트 수행에 유용하였다.	1	2	3	4	5	6	7

Ⅱ. 다음의 질문에 답하시오

1. 이 수업 이전에 정보시스템 개발관련 과목(예. 시스템 분석 및 설계, 데이터베이스관리 등)을 몇 강좌 수강하였습니까?

a. 0 b. 1 c. 2 d. 3이상

2. 정보시스템 개발 실무경험이 얼마나 있습니까?

a. 없음 b. 1년미만 c. 1-2년 d. 3년이상