능동적 객체간 의미 제약성을 제공하는 자바 기반 객체-관계 랩퍼의 구현

김규백, 김홍기, 현순주
한국정보통신대학원대학교 공학부

Implementation of a Java-based Object-Relational Wrapper Supporting Active Inter-object Semantic Constraints

Gyubaek Kim, Hongki Kim, SoonJoo Hyun
School of Engineering, Information and Communications University(ICU)

요 약

복잡해지고 있는 응용 프로그램의 스키마를 객체형 모델링 기법을 통해 효과적으로 정의함으로써 풍부한 의미관계를 유지하면서, 보편적인 하부 저장시스템 솔루션으로 제공되는 관계형 DBMS를 활용할 수 있는 객체-관계 접근 방법은 현실적인 선택으로 각광을 받고 있다. 이러한 접근 방법은 객체-관계 랩퍼(Object-Relational Wrapper)의 사용을 통해 가능할 수 있다. 본 논문에서는 기존의 객체-관계 랩퍼들의 문제점들을 지적하면서 보다 능동적으로 객체간의 의미 관계를 유지시켜 주는 새로운 객체-관계 랩퍼의 설계와 구현에 대해 설명한다. 자바로 개발된 객체-관계 랩퍼는 DBMS와 플랫폼에 독립적인 시스템으로 제공되며, 객체간 상속의 무결성 계약 조건을 유지하기 위해 메터 정보를 활용하는 특성을 가지고 있다.

1. 서 론

응용 프로그램들이 점차 대규모, 복잡화해짐에 따라 이를 효과적으로 모델링을 하기 위해 객체지향 설계 기법들이 도입되었다. 객체형 모델링 기법은 특히 의미와 관계를 표현함에 있어 강력한 특징을 가지고 있어서 복잡한 응용 프로그램의 스키마 설계시 널리 쓰이고 있다. 그러나 이러한 스키마 모델링 이후에 객체 형태로 설계된 데이터가 저장, 관리를 위한 보편적인 솔루션으로 제공되고 있는 관계형 DBMS에 저장되어야 하는 상황에 놓이는 경우가 많다. 객체-관계 접근 방법은 이러한 두 가지 형태의 상이한 특성과 구조의 불일치 문제를 해결하며 서로의 장점을 살릴 수 있는 설득력 있는 방안이기 때문에 각광을 받고 있다. 많은 상용 관계형 DBMS 제품들이 SQL3를 기반으로 하여 객체의 상속, 추상화 데이터 타입 등의 기능을 가지는 객체-관계형으로 제공되고 있고, 관계형 DBMS를 이용한 응용 프로그램 개발자에게 객체형 인터페이스를 제공해 주는 객체-관계 랩퍼도 활발히 개발되고 있다.

이처럼 관계형 DBMS를 객체의 관점으로 활용할 수 있게 하는 여러 기법들은 응용 프로그램의 특성을 보다 잘 이해해야 하는 차세대 DBMS의 요구사항을 만족시킬 수 있을 것으로 기대된다 [1]. 이를 통해 응용 프로그램과의 인터페이스의 추상화, 고차원적인 모델기반의 툴 제공 등이 가능할 것이다.

본 논문에서는 객체-관계형 DBMS에서 제공되는 부분적인 객체 모델링 기능의 한계점을 인식하면서, 객체 지향 언어를 통해 제공될 수 있는 객체간의 풍부한 의미 관계를 모두 표현하면서 상용 관계형 DBMS와 연동을 할 수 있도록 해주는 객체-관계 랩퍼의 설계와 구현에 관련된 내용을 설명하고자 한다. 자바로 개발된 객체-관계 랩퍼는 자바 인터페이스를 통해 DBMS와 플랫폼에 독립적인 시스템으로 제공되며, 객체간 상속의 무결성 계약 조건을 유지하기 위해 메터 정보를 활용하는 특성을 가지고 있다.

본 논문의 구성은 다음과 같다. 제 2장에서는 객체-관계 랩퍼의 핵심 기술인 두 모델간의 매핑 기술에 대해 살펴본다. 제 3장에서는 객체-관계 랩퍼의 구현과 관련된 세부 사항을 설명한다. 제 4장에서는 개발된 랩퍼를 이용하는 응용 프로그램의 개발 과정을 보여 준다. 그리고 제 5장에서 결론을 맺는다.

2. 객체-관계 매핑

근본적으로 객체 패러다임은 객체의 데이터와 행동(Behavior)을 이용하여 응용 프로그램을 구성한다. 그러나, 관계형 패러다임은 저장되어 있는 데이터에 근거한 것이다. 이러한 차이점은 기존 데이터에 접근을 하는 경우에 그 차이점을 명백히 알 수 있다. 객체 패러다임에서 사용자는 여러 객체들의 관계를 통해 객체의 정보를 유출할 수 있는 반면, 관계형 패러다임에서는 테이블에 저장되어 있는 데이터들을 조인(Join)하여 중복 데이터를 생성해야 하는 단점이 있다. 따라서, 관계형 패러다임에 의한 시스템에서 객체 지향적 개발을 위해서는 객체를 관계형 데이터베이스에 매핑하는 설계 원칙이 필요하다.

기존의 랩퍼에서 객체를 관계형 데이터베이스에 저장하기 위해 사용하는 매핑 과정에서 고려해야 할 사항은 상속(Inheritance), 집합(Aggregation), 그리고 객체간의 관계이다. 그 중에서 다음은 객체간의 상속을 관계형 데이터베이스에 매핑하는 세 가지 방법을 설명한 것이다.

첫번째 방법은 상속관계를 나타내는 상속 트리(Inheritance Tree)내의 각각의 클래스들이 단일 테이블로 매핑되는 것이다. 즉, 부모 클래스와 이를 상속하는 모든 하위 클래스의 속성들이 하나의 테이블에 저장된다. 이 방법은 단일 질의어를 사용하여 다중의 클래스로부터의 객체 검색을 지원하나 정규화에 어긋나는 단점이 있다. 이와 같은 기법을 이용한 시스템의 예로는 INRIA의 ObjectDRIVER [2]가 있다.

두 번째 방법으로는 하위 클래스만이 테이블에 매핑되며, 이때 부

모 클래스의 모든 속성을 포함하여 테이블에 저장된다 이 방법은 부모 클래스의 속성이 함께 테이블에 저장되어 있으므로, 하위 클래스의 인스턴스에 접근할 때, 하나의 테이블만이 필요하다

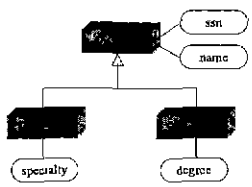
세 번째 방법에서는 각각의 클래스들이 해당하는 테이블에 매핑되는 것이다. 객체에 부여된 객체 식별자와 속성들이 해당 테이블에 저장되는 방식으로, 이러한 접근은 객체지향 개념에 가장 가까우며, 다형성의 지원, 그리고 부모 클래스와 새로운 하위 클래스간의 관계 설정을 쉽게 할 수 있는 장점을 제공한다 반면, 데이터베이스 내에 많은 수의 클래스가 생성되는 점과 데이터 연산을 위해서 많은 수의 테이블에 접근해야 하는 단점을 가진다.

Objectmatter의 Visual BSF [3]는 위의 세가지 방법을 모두 지원하는 시스템의 예이다

PowerTier, DBConnect, DBTools.h+-, GemConnect, Java Blend, Integrator, Cocobase, JRB와 CRB, JDX [4] 등 이외의 많은 랩퍼들은 일반적인 매핑의 원칙들을 준수하고 있으나 상속 관계를 나타내는 매핑 전략에 있어서는 선택을 달리하고 있다. 비록 각각의 기법들이 서로 장단점을 가지고 있으나, 객체지향 개념을 유지함에 있어 첫 번째와 두 번째의 기법은 문제점을 가지고 있다. 그래서 의미 관계 유지를 중요시 하는 시스템에서는 세 번째 방법이 실제 모델링 된 객체를 보다 잘 나타낸다 [5]

그러나 세 번째 방법에서 각각의 클래스마다 생성된 테이블간의 관계를 유지하기 위한 유일한 수단으로 사용되는 외부 키(Foreign Key)는 개념적으로 정규화된 테이블간의 관계를 표시하는 방법이긴 하나 객체간의 의미를 능동적으로 유지하기 위한 방법이 아니다. 반면 이러한 정보를 메타 정보를 통해 관리하면 하위 시스템에서의 제약 조건을 관리하지 않아도 상위 응용 프로그램에서 이를 제어할 수 있다. 따라서 본 논문에서 보고되는 랩퍼에서는 상속을 관리하는 세 번째 기법에서 외부 키를 없애고, 필요한 정보를 메타 정보화 하여 관리한다.

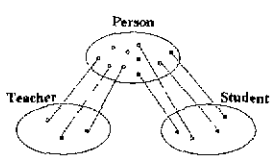
메타 정보의 내용은 그림 1-a 와 그림 1-b 에서 보듯이 클래스간의 관계, 인스턴스들간의 관계로 이루어져 있고, 이들 역시 테이블에 저장이 된다 그러나 데이터베이스의 연산이 이루어지는 과정에서 메타 정보가 참조되는 것이 아니라, JDBC를 통한 연결이 초기화 되는 시점에서 미리 불러와 메모리에 상주해 매번 데이터베이스와의 입출력을 해야 하는 문제점을 해결하였다



[그림 1-a] 클래스간의 관계(Intensional Diagram)

3. 객체-관계 랩퍼의 구현

3.1. 객체간 의미 유지를 위한 제약 조건들



[그림 1-b] 인스턴스간의 관계(Extensional Diagram)

객체지향 스키마에 내포된 의미들은 데이터베이스의 연산 과정 중에 다음과 같은 규칙을 준수하며 유지될 수 있다.

(1) 데이터베이스 입력

지속성을 가지는 저장 시스템에 새로운 객체 데이터가 입력될 때는 유일성을 가지기 위해 객체 식별자(Object Identifier OID)를 생성해 주어야 한다 개발된 시스템에서 사용하는 객체 식별자는 클래스 식별자(Class Identifier CID)와 인스턴스 식별자(Instance Identifier IID)로 구성이 된다. 인스턴스 식별자는 시스템이 생성 시간을 이용하는 타임스탬프(Timestamp)값을 이용한다. 이러한 객체 식별자 값도 관계형 테이블에서 하나의 컬럼으로 입력이 되고, 또한 주 키(Primary Key)로서 역할을 하게 된다.

객체들간의 스키마에서 부모 클래스만의 인스턴스로 데이터 입력이 이루어 질 수 있다 그림 1-b 에서도 보듯이 Teacher와 Student에는 속하지 않고 Person 클래스에만 존재하는 인스턴스들이 있다

그러니 하위 클래스의 인스턴스로 데이터 입력이 이루어지는 경우는 부모 노드와 1:1의 인스턴스간의 대응 관계가 존재해야 한다 예를 들어 Teacher 클래스의 인스턴스 객체가 입력될 때는 부모 클래스인 Person에서도 해당 인스턴스가 생성되어야 한다 그러므로 Person의 테이블의 다른 컬럼에는 널(null) 혹은 다른 기본값(Default)을 채워 넣을 필요가 있다 만약 널을 입력하고자 한다면 테이블을 정의할 때 속성값 뒤에 "not null" 제약 조건이 없어야 한다 그리고 생성된 인스턴스는 결국 동일한 사람을 가리켜야 한다는 조건은 Teacher쪽과 Person쪽에 입력되는 각각의 객체 식별자에서 동일한 시간에 생성된 인스턴스 식별자 비교를 통해 유지될 수 있다

(2) 데이터베이스 삭제

부모 클래스의 인스턴스가 삭제되면 해당하는 하위 클래스의 인스턴스도 함께 삭제 되어야 한다

그러나 하위 클래스의 인스턴스가 지워져도 부모 클래스의 해당 인스턴스가 함께 삭제되어서는 안된다. 예를 들어, Student라는 신분이 변경되어 삭제가 이루어졌어도, Person으로서의 기록은 남아 있어야 되는 경우를 생각해 볼 수 있다.

객체-관계 랩퍼에서 많이 이용하는 외부 키는 이러한 의미를 유지시키기에는 한계점을 가지고 있다. 기본적으로 외부 키가 가리키는 해당 튜플이 테이블에서 삭제가 될 때 연산을 제한하는 것에 그치는 기존의 방식으로는 보다 강력한 의미를 유지할 수 없다. 그러므로 이를 위해서는 앞서 설명한 능동적인(Active) 연산 적용이 필요하다. 이를 가능하게 하는 것이 객체간의 의미를 포함하는 적절한 메타 정보들이다.

(3) 데이터베이스의 변경

중복을 통해 상속을 유지하는 기법에서는 부모 클래스의 데이터베이스의 변경이 하위 클래스에 모두 반영되어야 한다는 조건이 있지만, 클래스 하나에 각각의 테이블을 유지하는 방식에서는 해당 클래스를 나타내는 테이블에만 변경된 내용을 반영시키면 된다.

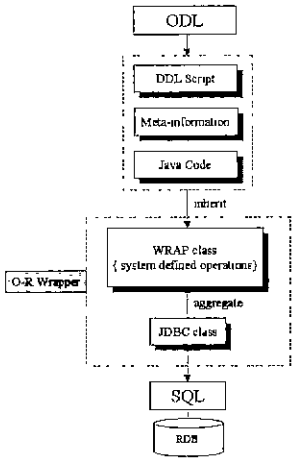
3.2. 특징

일반적인 랩퍼로서 적용하기 위해 DBMS에 독립적이어야 하는 조건을 하위 관계형 DBMS와의 연결 방법인 JDBC 기법들을 메소드 오버로딩을 통해 제공함으로써 만족하였다 또한 자바로 개발함으로써 플랫폼에도 독립적인 성격을 유지한다 개발된 랩퍼는 Linux상의 Oracle8i와 Window NT상의 IBM DB2 Universal Database Version 5, UNIX상의 mSQL 2.0 버전에서 테스트 되었다. 그리고 풍부한 객체간 의미 정보를 메타 정보로 유지함으로써 능동적인 연산 적용을 할 수 있다는 특징을 가지고 있다 시스템에서 기본적으로 제공하는 create, store, remove, change, retrieve(System defined operations: 그림 2 참조) 외에 개발자가 원하는 형태로의 API는 이미 개발된 메

소드를 오버라이드함으로써 가능하다

4. 랩퍼를 이용한 응용 프로그램의 개발

개발된 랩퍼를 이용하는 전체 시스템의 구성도는 아래 그림과 같다.



[그림 2] 전체 시스템 구성도

그림은 또한 전체 시스템과 랩퍼와의 상호 작용을 보여주고 있다. ODMG의 ODL은 언어에 중립적인 객체간 데이터정의를 통해 필요한 스키마 정보의 테이블 생성을 위한 스크립트 정보를 생성키기고, 이후 응용 프로그래머에게 자바로 바인딩(Binding)된 코드를 생성시킨다. 생성된 초기 코드는 자바의 패키지 형태로서 정의한 스키마를 이용해 쉬게 응용 프로그램을 개발할 수 있도록 도와준다

전체 시스템의 구성 요소들이 이용되는 과정을 순서대로 설명하고자 한다. 예제에 활용되는 스키마는 그림 1-a 이다

(1) 사용자는 ODL을 통해 데이터베이스의 스키마를 정의한다 각각의 클래스와 대응되는 테이블 생성을 위한 스크립트 정보는 랩퍼를 통해 create table 명령어로 변환되어 DBMS에서 실행된다

(2) 또한 ODL에서 표현된 클래스들간의 관계와 속성의 타입, 그리고 이후 데이터베이스 연산을 통해 유지되어야 하는 인스턴스간의 관계들을 메타 정보화 하여 데이터베이스에 저장한다.

(3) 그리고 ODL은 각각의 클래스들을 자바 클래스로 바인딩하여 코드를 생성시킨다 (1)에서 (3)까지의 과정은 모두 ODL을 파싱하는 과정 중에 일어난다 그림 1-a의 스키마의 경우 세 개의 자바 클래스가 생성되는데, 각각 Person, Teacher, Student이다 이 중 이들의 부모 클래스에 해당 되는 Person은 랩핑 메커니즘을 포함하고 있는 WRAP 클래스를 상속받는다 또한 WRAP 클래스는 사용자에게 JDBC를 통한 연결의 부담을 덜어주기 위해 JDBC라는 미리 정의된 클래스를 활용한다. 따라서 Teacher와 Student도 이런 기능들을 상속받게 된다. 이후 생성된 클래스들은 패키지화 하여 제공된다 현재까지의 과정을 통해 생성된 코드는 아래의 예와 같다.

```
import University *;
class Example {
    public static void main(String argv[]) {
        Teacher t = new Teacher();
        t.begin();
        t.specialty = "database";
        t.store(),
        t.end();
    }
}
```

클래스의 인스턴스를 생성하는 코드는 부모 클래스에서 JDBC 연결을 위한 메소드를 호출하고, 해당 클래스와 관련된 메타 정보를 미리 메모리에 상주시킨다

소스 코드에서 begin과 end 메소드는 트랜잭션 개념을 지원하기 위한 설정이다. 또한 위의 소스 코드를 통해 알 수 있듯이 개발된 랩퍼는 처음부터 끝까지 객체 지향 언어 프로그래밍 스타일로 제공되어 응용 프로그램 개발자에게 별도의 부담을 제거할 수 있는 특징을 가지고 있다

(4) 프로그래밍 코드에서 입력한 값을 저장하기 위해 WRAP 클래스로부터 상속받은 store라는 메소드를 호출한다.

(5) 호출된 store 메소드는 데이터베이스 입력 연산으로서 먼저 객체 식별자를 생성한다 그런 후 메타 정보에서 클래스간의 관계를 조사한다. 예제의 스키마에서 Teacher와 Person은 부모와 자식의 관계가 있음을 알 수 있다. 따라서 store 메소드는 메모리에 있는 Teacher 객체의 값을 읽어 오고 제 3.1절에 설명한 것과 같이 능동적인 객체간 의미 제약성 유지를 위해 동일한 인스턴스를 Person 클래스를 나타내는 테이블에도 입력하게 된다 이때 Teacher에 입력되는 객체식별자는 {Teacher + 타임스탬프} 이고, Person 클래스에 입력되는 객체식별자는 {Person + 동일한 타임스탬프} 이다 이들은 결국 동일한 시점에 생성된 인스턴스들로서 동일한 타임스탬프를 나타내는 인스턴스 식별자를 통해 구분될 수 있다 또한 인스턴스간 관계를 나타내기 위해 메모리에 상주하는 메타 정보에 생성된 객체 식별자와의 관계를 입력해 놓는다

(6) end 메소드를 통해 메모리상에 존재하는 메타 정보들을 다시 테이블에 저장하고, 데이터베이스와의 연결을 해제한다

5. 결론

강력한 객체 모델링 기법을 사용하면서 상용 관계형 데이터베이스와 연동하기 위한 API를 제공해 주는 객체-관계 랩퍼는 다양한 응용 프로그램을 개발하고자 할 때 점차 실특력 있는 시스템 혹은 라이브러리가 되고 있다. 개발된 랩퍼에는 객체 지향 모델이 가지는 의미적 제약 사항들을 모든 데이터베이스 연산의 과정에서 엄격히 준수하도록 설계하였으며, 이를 위해 풍부한 메타 정보를 이용해 능동적으로 관리한다. 향후 객체형 데이터 질의를 위해 ODMG에서 표준으로 정하고 있는 OQL을 지원하는 랩퍼로서 기능을 확장해 나갈 것이다 또한 IBM의 Garlic 프로젝트 [6]와 같이 랩퍼가 폭넓은 응용을 지원할 수 있는 방향으로 개발이 계속해서 이루어질 것이다.

참고 문헌

[1] P Bernstein, M Brodie, S.Ceri, and et. al., "The Asilomar Report on Database Research," ACM SIGMOD Record, Vol 27, No.4, December 1998, pp.74-80

[2] F.Lebastard, S Demphlous, V Agulera, and et. al, "ObjectDRIVER Reference Manual," Version 1.1, INRIA

[3] Objectmatter Inc, "Visual BSF Mapping .Tool Guide," Version 2.0

[4] J. Duhl, "Storing Objects in a Database," July 1999, http://www.kinetica.com/ootips/persistent-objects.html

[5] S. Ambler, "Mapping Objects To Relational Databases," An AmbySoft Inc. white paper, February 1999

[6] L.Haas, R.Miller, B.Niswonger, and et. al, "Transforming Heterogeneous Data with Database Middleware. Beyond Integration," http://www.almaden.ibm.com/cs/garlic/homepage.html