http://dx.doi.org/10.5392/JKCA.2011.11.12.597

SQL을 이용한 메모리 데이터 조작

Manipulation of Memory Data Using SQL

나영국*, 우원석**
서울시립대학교 전자전기컴퓨터학부*, 이화여자대학교 국제학부**

Young-Gook Ra(ygra123@gmail.com)*, Won-Seok Woo(wwoo@ewha.ac.kr)**

요약

데이터베이스 응용 프로그램 개발에서 데이터는 메모리 공간과 디스크 공간에 공존한다. 메모리 공간의 데이터를 조작하기 위하여 일반 프로그래밍 언어를 사용하고 디스크 공간의 데이터 조작을 위하여 SQL을 사용한다. 특히 메모리 데이터를 조작하기 위해 사용되는 절차적 언어는 SQL등의 선언적 언어보다 작성 및 유지보수가 어렵다. 이에 본 논문은 특수한 형태 즉, 트리 구조의 메모리 데이터는 선언적 언어인 SQL로 조작이 가능함을 보인다. 특히 UI (user interface)의 모델 데이터는 트리 구조로 표현 될 수 있기 때문에 예외적인 계산을 제외하고는 대부분의 메모리 데이터 조작은 SQL로 가능하다. 예외적인 계산은 도움 클래스 (helper class)로 처리하면 된다. 본 논문이 제시하는 SQL 메모리 데이터 조작은 예외적인 계산이 적은 데이터베이스 응용 프로그램 개발에 특히 적합하다.

■ 중심어 : | SQL | 데이터베이스 응용 프로그램 개발 | 선언적 언어 | 절차적 언어 | 사용자 인터페이스 | 모델 |

Abstract

In database application developments, data coexists in memory and disk spaces. To manipulate the memory data, the general programing languages are used and to manipulate the disk data, SQL is used. In particular, the procedural languages for the memory manipulation are difficult to create and manage than declarative languages such as SQL. Thus, this paper shows that a particular structure of memory data, tree structured, can be manipulated by SQL. Most of all, the model data of the user interfaces can be represented by a tree structure and thus, it can be processed by SQL except non set computations. The non set computations could be done by helper classes. The SQL memory data manipulation is more suited to the database application developments which have few complex computations.

■ keyword : | SQL | Database Application Development | Declarative Language | Procedural Language | User Interface | Model |

I. 서 론

데이터베이스 애플리케이션 작성자는, 메모리 공간에서 사용되는 데이터 조작하기 위해서 C++, C# 또는 Java 등의 절차적 언어(Procedural Language)를 사용하여 데이터베이스 애플리케이션을 코딩한다. 또한 데이터베이스 애플리케이션 작성자는 데이터베이스 공간에서 사용되는 데이터를 조작하기 위해서 SQL(Structured

* 이 논문은 2010년도 서울시립대학교 연구년교수 연구비에 의하여 연구되었음.
접수번호 : #111103-002
접수일자 : 2011년 11월 03일
심사완료일 : 2011년 12월 05일
교신저자 : 나영국, e-mail : ygra123@gmail.com

Query Language) 등의 선언적 언어(Declarative Language)를 사용하여 데이터베이스 애플리케이션을 코딩한다. 즉 데이터베이스 애플리케이션 작성자는 데이터베이스 애플리케이션의 업무 로직, 데이터 접근 로직을 구현하는 경우는 절차적 언어를 이용하여 코딩을 수행하여야 하며, 또한 데이터베이스 내의 동작을 위해서는 선언적 언어를 이용하여 코딩을 수행해야 하므로, 절차적 언어와 선언적 언어 모두에 대해서 숙지하고 있어야 한다.

기본적으로 데이터베이스 어플리케이션은 폼 중심이다. 그리고 비즈니스 로직뿐만 아니라 데이터 접근 로직도 SQL을 포함하는 절차적 언어로 작성하고 있다. 본 논문은 비즈니스 로직은 절차적 언어로 작성하고 데이터 접근 로직을 순수한 SQL로 작성하는 대안을 제시하고 있다.

따라서 데이터베이스 애플리케이션을 개발하기 위해서는 많은 시간이 걸리는 단점이 있다. 또한 데이터베이스 애플리케이션 작성자는, 메모리 공간에서 사용되는 데이터를 조작하기 위해서 해당 데이터의 클래스(Class)에 대한 메소드(Method)를 작성 하여야 하나, 이러한 메소드는 데이터베이스 공간 내의 데이터와 대비를 수행하여야 하므로, 메소드를 작성할 많은 시간이 걸리는 단점도 있다. 특히 메소드의 테스트를 위해서 많은 시간이 필요한 단점이 있다.

기본적으로 데이터베이스 어플리케이션은 폼 중심이다. 그리고 비즈니스 로직뿐만 아니라 데이터 접근 로직도 SQL을 포함하는 절차적 언어로 작성하고 있다. 본 논문은 비즈니스 로직은 절차적 언어로 작성하고 데이터 접근 로직을 순수한 SQL로 작성하는 대안을 제시하고 있다.

위의 기술적 과제를 달성하기 위하여, 본 논문은 (a) 데이터베이스의 테이블과 관련하여 데이터베이스 애플리케이션에 대한 계층적 데이터 구조의 클래스(class)의 정의를 입력받는 단계와, (b) 이 클래스에 대해서 메소드(method)의 정의 및 위의 메소드에 대한 선언적 언어 형태의 본체를 입력받는 단계와, (c) 위의 선언적 언어 형태의 본체를 가지는 메소드를 미리 지정된 변환 로직을 이용하여 절차적 언어 형태의 본체를 가지는 메소드로 변환하는 단계와, (d) 위의 계층적 데이터 구조를 디스플레이하고, 상기 절차적 언어 형태의 본체를 가지는 메소드를 실행하여 데이터베이스 애플리케이션에 대한 테스트를 수행하는 단계를 포함하는 계층적 데이터 구조 기반 데이터베이스 애플리케이션 - 계층적 데이터 구조를 MVC (Model View Controller) 패턴의 모델로 사용하는 데이터베이스 어플리케이션 - 생성 방법을 제공한다.

위의 방법은 데이터베이스 애플리케이션 개발하기 위해서 소요되는 시간을 감소시킬 수 있으며, 절차적 언어를 이용하여 메소드를 정의함으로써 데이터베이스 애플리케이션 개발자의 업무 부담을 최소화하여 개발 속도를 높이고 데이터베이스 애플리케이션의 가독성(Readability)을 높일 수 있으며, 또한 메소드를 실행하면서 데이터 구조와 실시간 비교가 가능한 장점이 있다.

다음 장은 관련 연구를 소개하고 비교 평가 하였다. 3 장은 본 시스템을 사용하는 방법을 설명하였다. 4 장은 본 시스템의 특징을 추상적인 레벨에서 제시하였다. 5 장은 시스템을 상세히 설명하였다. 6 장은 시스템을 요약하였다.

II. 관련연구

이 연구는 이전의 프리뷰 (preview) 논문 [1]들을 계승, 확장한 연구로 특히 OR (Object Relational) 매핑 분야에서 많은 진척이 있었다.

데이터베이스와 연동한다는 점에서 본 논문은 오브젝트-관계형 매핑 (OR Mapping)을 제공하는 하이버네이트 (Hibernate)[2]와 아이바티스 (iBatis)[3]와도 어느 정도 관련이 있다. 하이버네이트는 광범위하게 사용되는 오브젝트-관계형 매핑 프레임워크로 데이터베이스의 테이블과 자바 클래스를 매핑 설정하여 프로그래머는 매핑 설정 이후에 데이터베이스를 의식하지 않고 해당 자바 클래스만으로 데이터베이스 데이터를 갱신하고 검색할 수 있게 하여 프로그래머의 생산성을 높인다. 아이바티스는 프로그램에서 사용되는 모든 SQL (structured query language)를 XML로 설정하여 프로

그래머의 생산성을 높이고자 한다. 위의 도구들이 메모리와 디스크 공간의 괴리를 특수한 데이터베이스 객체를 제공함으로 해결하려 하는데 비하여 본 논문은 메모리 내의 대부분의 데이터 조작을 선언적 언어인 SQL로 수행할 수 있음을 보인다.

하이버네이트[2]는 데이터베이스 테이블을 메모리 내의 객체로 매핑하는 툴이다. 하지만 실제 개발에서는 테이블에 일대일 대응 하지 않지만 데이터베이스로부터 매핑되는 객체들이 필요하다. 조회 객체가 대표적이지만 이 외에 갱신이 필요한 객체들도 있다. 조회 객체는 관계형 데이터베이스에서 뷰로 정의하면 하이버네이트로 메핑이 가능하다. 하지만 갱신 필요 객체 중의 데이터베이스 테이블과 일대일 대응되지 않는 객체는 뷰로 정의하여도 뷰가 갱신 불가능하기 때문에 하이버네이트로는 매핑이 어렵다. 하지만 본 연구는 SQL을 이용하여 메모리 객체로부터 다수의 데이터베이스 테이블들로 매핑을 하기 때문에 모든 메모리 객체를 갱신 가능하게 할 수 있다.

MVC 프레임워크에서 데이터 구조를 비쥬얼라이제이션 (visualization) 하는 기술은 교육과 테스팅 목적으로 연구를 하여왔다[4]. 이 연구는 기존의 MVC 패턴에 지시자 (indicator) 객체를 추가하여 비쥬얼라이제이션을 가능하게 하였다. 이 논문은 더 나아가 리스트 등의 순차적 알고리즘 (sequential algorithm) 객체를 패턴에 추가하여 이를 가시화하는 객체 모듈을 소개한다.

계층적 데이터 구조를 비쥬얼라이제이션하는 접근 방식을 소개하는 연구도 발견할 수 있다[5]. 이 논문은 데이터베이스 어플리케이션의 MVC에서 가장 자주 발견되는 계층 구조를 비쥬얼라이제이션하는 효율적인 알고리즘을 제시한다. 계층적인 데이터 구조는 벤다이어그램 (Venn Diagram), 중첩 트리 맵 (Nested Tree Map), 그리고 트리 맵 (Tree Map) 방식 중 트리 맵이 계층 데이터를 비쥬얼라이제이션하는 데 있어서 가장 효율적인 알고리즘임을 보인다. 특히 이 접근 방식은 복잡한 데이터 구조에서 유효하다.

본 논문은 MVC 모델을 비쥬얼라이제이션하는 소프트웨어 모듈을 소개한다. MVC 모델은 계층적 구조를 가지기 때문에 위의 연구[5]에서 제시한 트리 맵 알고리즘이 특히 모듈 구현에 유용하게 사용되었다. 테스팅 목적으로 본 논문의 실행 모듈은 연구[4]의 알고리즘을 차용할 수 있음을 확인하였다. 특히 마스터슬레이브 UI에서 발견할 수 있는 순차적인 리스트 디스플레이 알고리즘은 지시자 객체에 ms 덧붙여서 하나의 모델 패턴을 구성하였다.

이 논문의 모델 비쥬얼라이제이션 기술은 웹 프로그래밍과 클라이언트/서버 (client/server) 프로그래밍에 같이 적용될 수 있다. 이러한 MVC 프로그래밍 프레임워크는 스트러츠[6], 스프링[7], MVC 프레임워크[8], 루비온레일즈 (ruby on rails)[9]가 있다. MVC 프레임워크는 마이크로소프트 제품으로 비주얼 스튜디오에 통합되어 있다. 이는 모델, 뷰 및 컨트롤러 구성 요소를 분리한다. 그리고 URL 라우팅과 포스트백 대신에 컨트롤러 클래스를 사용하는 등 프레임워크는 MVC의 컨트롤러를 지원하는데 초점을 두고 있다. 스트러츠는 가장 성숙한 웹 프레임워크로 이는 jsp의 모델2를 구현하였다. 이는 URL 요청을 액션 (action)으로 매핑 하는 설정을 가능하게 한다. 스트러츠는 주로 컨트롤러를 제공하고 뷰 템플릿 제공하여 뷰 작성을 원활하게 한다. 스프링은 경량 프레임워크의 선두 주자로 제어의 반전 (inversion of control)과 관점 지향 프로그래밍 (aspect oriented programming)을 주요 특징으로 한다. 제어의 반전은 시스템의 흐름이 반전되어 자신의 표준 설정 방법에 따라 의존성 정보를 확인하고 객체 생성 및 초기화를 대신한다. 관점 지향 프로그래밍은 공통 관심 사항을 구현한 코드를 핵심 로직을 구현한 코드 안에 삽입한다는 것이다. 루비온레일즈는 데이터베이스와 웹 애플리케이션 개발에 있어 가볍고 유연한 환경을 제공한다. 이는 매우 빠른 웹 개발을 가능하게 한다. 이는 설정 보다는 이름을 기준으로 url 요청을 처리하는 컨트롤러를 제공한다.

위의 프레임워크들은 주로 MVC의 컨트롤러 구현을 제공한다는 점에서 본 논문과 다르다. 본 논문은 MVC에서 모델에 초점을 맞추어 비즈니스 로직을 삽입하고 이 로직이 데이터베이스와 연동하면서 즉각적으로 테스팅해볼 수 있는 테스트 베드[10]를 제공한다. 이의 산출물로 모델을 구현하는 자바 빈 클래스들과 비즈니스

로직을 구현하는 메소드 코드를 들 수가 있다. 그러므로 본 논문의 프레임워크는 컨트롤러를 구현하는 위의 프레임워크들과 공존하여 사용될 수 있다.

MVC 패턴은 본 논문의 기본 가정이다[8]. 왜냐하면 MVC 패턴은 모델을 뷰와 분리하여 독립적인 객체를 구성할 수 있게 하기 때문이다. [11]은 작은 규모 (10 개의 테이블과 19 웹 페이지)의 프로젝트를 수행하면서 개발 도중의 요구 사항 변경에 MVC 패턴이 얼마나 잘 대응하는지를 경험적으로 보여주고 있다. 이 논문의 예에서 대부분의 요구사항 변경이 트리거를 통해서 반영될 수 있음을 보인다. 또한 이 연구 [11]는 MVC의 장점으로 (1) 적은 커플링 (coupling) (2) 높은 응집력 (cohesion) (3) 뷰의 유연성 (flexibility) (4) 디자인명확성 (clarity) (5) 유지보수 용이 등을 들고 있다. 하지만 이 연구는 하나의 소규모 프로젝트에서 얻은 경험을 기술하고 있어 여러 프로젝트에서 다양한 요구사항을 MVC 패턴이 처리할 수 있는 지는 불명확하다. 본 연구는 이에 모델 비쥬얼라이제이션을 통하여 직관적인 시스템 이해가 가능하기 때문에 요구사항 변경에 더 적극적으로 대처할 수 있다.

모델이 독립적인 객체이므로 계층 구조를 강제할 수 있으며 이러한 특성으로 인해 MVC 모델이 관계형 뷰를 가질 수 있다. 그러므로 MVC 모델의 데이터를 SQL로 조작할 수 있다. 더욱이 데이터베이스 어플리케이션의 MVC 모델은 계층 구조로 구현할 수 있으며 이는 기존의 연구[5]에 의해 트리-맵을 이용하여 모델 비쥬얼라이제이션이 가능하게 되었다.

이 연구의 모델 비쥬얼라이제이션은 특히 테스팅 분야에서 많은 진척을 이루었다[1]. 이전 프리뷰는 하나의 폼 화면에 여러 개의 데이터베이스 테이블을 관련시키기 때문에 폼 로딩 (loading)과 저장 (save)로 매우 복잡하였다. 그러므로 빠른 시간과 적은 노력으로 폼 디자인을 수정하면서 비즈니스 로직 추가와 데이터베이스 디자인 수정으로 어려웠다. 다시 말하면 하나의 폼에 여러 개의 자바빈 객체가 있으면 이를 하나의 폼에서 데이터를 검색하거나 변경하는 내부 로직이 복잡해진다. 이 결점을 보완하기 위하여 본 논문은 하나의 폼에 하나의 갱신 가능한 트리 구조[12]를 대응시킬 것을 제안한다.

갱신 가능한 트리 구조는 뷰의 패턴을 정의하는 것으로 트리 구조의 뷰[12] 연구에서는 이를 SPJN으로 명명, 에는 애매하지 않게 그 갱신을 정의할 수 있다. 이에 대한 알고리즘은[12] 논문에서 자세히 기술하였다. 부연설명하면 셀렉트와 프로젝트로만 구성된 뷰는 뷰 정의 시 갱신 시맨틱 (semantic)을 애매하지 않게 정의할 수 있으며 하나의 테이블이 트리의 루트 (root)가 되고 다른 테이블들은 주키 외래키 관계를 에지 (edge)로 하여 루트의 자식이 되는 트리 구조의 조인 뷰도 역시 정의 시에 애매하지 않게 시맨틱을 정의할 수 있음을 보인다[12].

객체 관계 매핑의 사용은 모델의 사용을 권장하여 현대 웹 어플리케이션에 특히 유용하다[13]. 이는 매핑 툴이 데이터베이스 접근에서 레이스 컨디션 (race condition)을 감안하여 트랜즈액션 (transaction) 처리를 대행하여 기존의 방법에서 세심한 관심이 필요한 부분을 쉽게 하여 준다. 본 연구는 트랜즈액션을 포함하지 않고 이를 응용 프로그램 개발자에 맡긴다. 이에 미래 연구가 필요하다.

객체와 관계형 기술은 서로 다른 패러다임 (paradigms)에 존재하여 이 둘이 혼합될 때 임피던스 (impedance) 불일치가 발생한다[14]. 이 연구는 이러한 불일치를 네 종류로 분류하고 이를 해결하는 프레임워크를 제시한다. 본 논문은 이러한 불일치를 해결하는 하나의 프레임워크로 주요한 객체들에 관계형 뷰를 제공할 것을 제안한다. 이러한 방법은[14]의 프레임워크와 동일한 목적을 갖지만 본 연구는 솔루션으로 구현 가능하지만[14] 연구는 추상적인 레벨에서의 프레임워크로 솔루션으로 구현할 수 없으며 임피던스 불일치를 이해하고 해결할 수 있는 하나의 방법론을 제공한다.

III. 시스템 사용 방법

본 논문에 따른 계층적 데이터 구조 기반 데이터베이스 애플리케이션 생성 방법에 있어서, 첫 단계는 트리 (tree) 형태로 지정된 계층적 데이터 구조의 노드를 입

력받는 단계와, 이 노드를 클래스 정의로 변환하는 단계와, 이 클래스에 대한 인스턴스(instance) 변수를 입력받는 단계, 위의 노드 및 인스턴스 변수 중 적어도 하나를 입력받기 위한 GUI(Graphical User Interface)를 디스플레이하는 단계를 포함한다. 이때 위의 계층적 데이터 구조의 클래스는 루트 클래스 (트리 구조의 루트 객체의 클래스) 및 루트 클래스에 대한 하나 이상의 자식 클래스 (루트를 제외한 노드 객체의 클래스) 를 포함할 수 있다. [그림 1] 순서도의 첫 번째 단계에 해당된다. 두 번째 단계로 위의 계층적 데이터 구조에 대한 관계형 모델 뷰를 표 하고, 기 입력된 클래스에 대한 주키(PK)를 입력받는 단계가 포함된다. 또한 위의 메소드의 정의에서, 호스트 클래스 정보, 클래스 메소드 여부, 메소드 이름, 파라미터 타입, 리턴 타입이 포함된다.

또한 위의 계층적 데이터 구조의 클래스는 루트 클래스 및 루트 클래스에 대한 하나 이상의 자식 클래스를 포함하는 것이고, 위의 파라미터 타입은 기본형 또는 루트 타입 중 어느 하나일 수 있다. 위의 메소드 리턴 타입은 기본형으로 제한된다. 더욱이 이 단계는 계층적 데이터 구조의 관계형 모델 및 위의 테이블 중 적어도 하나를 디스플레이하는 단계를 더 포함할 수 있다. 이는 [그림 1] 순서도에서 2, 3 단계에 해당된다.

세 번째 단계는 선언적 언어 형태의 본체를 가지는 메소드가 변환 로직을 이용하여 절차적 언어 형태의 본체를 가지는 메소드로 변환 가능한 지 여부를 미리 확인하는 단계와 변환 가능한 것으로 확인되면, 상기 선언적 언어 형태의 본체를 가지는 메소드를 변환 로직을 이용하여 절차적 언어 형태의 본체를 가지는 메소드로 변환하고, 변환 가능하지 않은 것으로 확인되면, 에러 메시지를 출력하는 단계이다.

또한 이 단계는 마스터슬레이브 방식 또는 트리 식을 이용하여 계층적 데이터 구조를 디스플레이하는 것과 클래스의 객체를 참조하는 로컬 변수와 클래스를 디스플레이하는 단계와, 절차적 언어 형태의 본체를 가지는 메소드를 실행하는 것에 따른 로컬 변수의 변경을 디스플레이하는 단계를 포함할 수 있다. 이는 [그림 1] 순서도의 4, 5 단계에 해당된다.

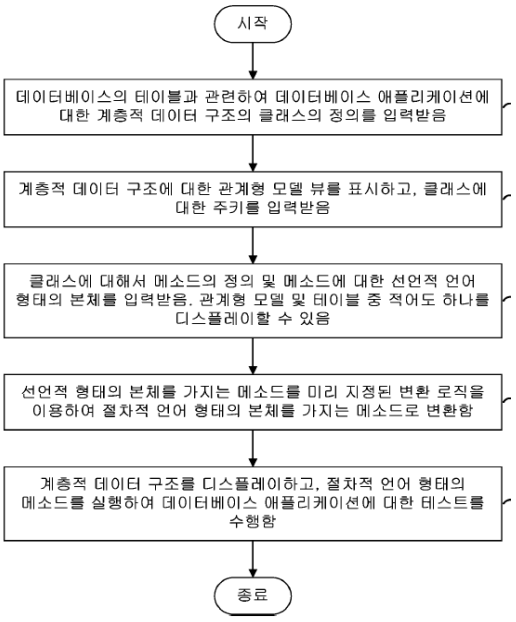

그림 1. 시스템 사용 순서도

IV. 특징

본 논문은 데이터베이스 애플리케이션을 작성하는 경우, 생산성을 높이기 위해서 데이터베이스 애플리케이션 작성자에게 다음의 두 가지 특징을 제공한다. 하나의 특징은 메소드의 "SQL 정의 가능"이고, 다른 특징은 메모리 내의 데이터 구조와 메소드의 "가시성(Visibility)"이다. 이러한 특징을 제공하기 위해 본 논문은 메모리 내의 데이터 구조를 계층적인(Hierarchical) 구조로 한정한다.

SQL 정의 가능은 데이터베이스 애플리케이션에서 비즈니스 로직과 데이터 접근 로직을 담당하는 메소드를 기존의 절차적 언어, 예컨대 Java, C++ 또는 C# 등으로 정의하지 않고, 선언적 언어인 SQL로 정의하는 것을 의미한다. 메소드가 SQL 등의 선언적 언어도 정의되면, 시스템은 이를 절차적 언어로 변환시킨다. 따라서 데이터베이스 애플리케이션 작성자는 절차적 언어가 아닌 선언적 언어를 이용하여 대부분의 작업을 수행할 수 있어서 데이터베이스 애플리케이션 작성 속도를 향

상시킬 수 있고 가독성을 높일 수 있다.

선언적 언어는 실행 속도가 10 배 정도 느린 반면에 코드 사이즈는 절차적 언어의 반 정도 이다[15]. 이는 대형 프로그램의 유지 보수에 유리한 점이다. 개발 속도는 절차적 언어가 2 배 정도 빨랐으며 일반적으로 프로그램의 가독성 (Readability)이 높았다.

데이터 구조와 메소드의 "가시성"은 메모리에 있는 데이터 구조를 GUI를 이용한 화면에서 디스플레이할 수 있고, 이 화면에서 메소드를 실행할 수 있는 것을 의미한다. 본 논문에서 이러한 화면은 "실행 화면"라고도 지칭된다. 메모리의 데이터 구조를 디스플레이함으로써, 데이터베이스 애플리케이션 작성자는 데이터 구조의 값들을 실시간으로 확인할 수 있으며, 또한 이 데이터 구조에 적용 될 수 있는 메소드도 실행 화면에 디스플레이되기 때문에, 데이터베이스 애플리케이션 작성자는 메소드를 인터액티브(Interactive)하게 실행할 수 있다. 따라서 데이터와 메소드를 실시간으로 보면서 작성 및 수정을 수행할 수 있어 데이터베이스 애플리케이션 작성자의 인지적(Cognitive) 부담을 크게 줄여주며 직관적으로 파악할 수 있게 한다. 또한 메소드를 여러 가지 환경에서 실행시켜 확인할 수 있기 때문에, 데이터베이스 애플리케이션의 효과적인 테스트가 가능하다.

이러한 특징을 구현하기 위해서, 계층적 데이터 구조 기반 데이터베이스 애플리케이션 생성 방법은 메모리에 정의되는 모델 객체의 데이터 구조를 "계층적 데이터 구조"로 한정한다. "계층적 데이터 구조"는 여러 개의 루트(Root)를 가질 수 있으며, 루트 객체는 그 자신의 컬렉션(Collection) 객체를 통하여 여러 개의 자식 객체를 가질 수 있다. 또한 자식 객체 역시 다른 컬렉션 객체를 통하여 자식의 자식 객체를 가질 수 있다. 이러한 계층적 데이터 구조는 재귀적(Recursive)으로 정의될 수 있으며, "계층적 데이터 구조" 안의 모든 원소 객체는 단지 하나만의 객체를 가질 수 있다.

다시 말하면 본 논문은 어플리케이션의 MVC 구조에서 모델을 계층적 데이터 구조를 갖도록 강제하였다. 이 제약은 데이터베이스 어플리케이션이 통상적으로 마스터슬레이브 (master-slave) 또는 트리 (tree) 화면으로 구현되기 때문에 대부분의 데이터베이스 어플리케이션에 적용된다.

데이터 구조가 "계층적"이 아닌 예외적인 객체의 계산은 SQL 조작이 불가능하다. 이러한 객체의 계산 및 조작은 Java등의 절차적인 언어로 작성되어야 한다. 본 논문은 "계층적" 구조의 객체의 메소드들을 SQL로 입력하면 시스템이 이를 절차적 언어로 된 메소드들로 변환시켜준다. 그러므로 예외적인 계산을 포함하는 메소들은 표현 불가능하여 직접 절차적인 언어로 작성되어야 한다. 이 두 가지 종류의 메소드들은 이벤트 핸들러 (event handler) 에서 호출되어 실행되어 어플리케이션을 동작한다. 하지만 이러한 예외적인 메소드들은 데이터베이스 어플리케이션에서 작은 부분을 차지하기 때문에 메소드들을 SQL로 표현하는 본 논문의 접근 방식은 개발자의 생산성을 높여준다.

V. 시스템 설명

1. 대표적인 예

이해를 돕기 위해서 예시적인 데이터베이스 애플리케이션을 참조로 하여 설명한다. [그림 2] 내지 [그림 3]은 계층적 데이터 구조 기반 데이터베이스 애플리케이션의 생성 방법을 설명하기 위한 예시적인 데이터베이스 애플리케이션을 도시하는 그림이다.

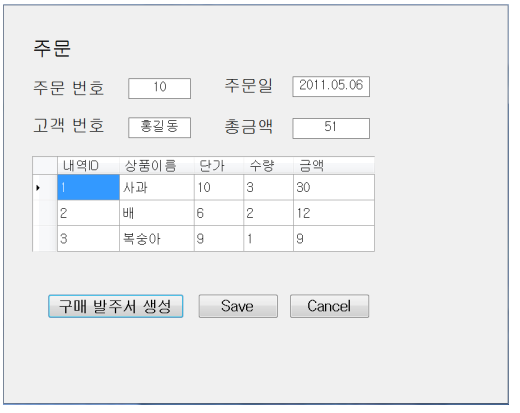

그림 2. 주문 화면

[그림 2]를 참조하면, 주문을 받고 그 내역을 기록하

기 위한 사용자 화면, 즉 주문 화면이 표시된다. 예컨대 데이터베이스 애플리케이션의 사용자는 주문 번호, 주문일, 고객 이름 등을 입력하고, 테이블에 상세 주문 내역을 기록한다.

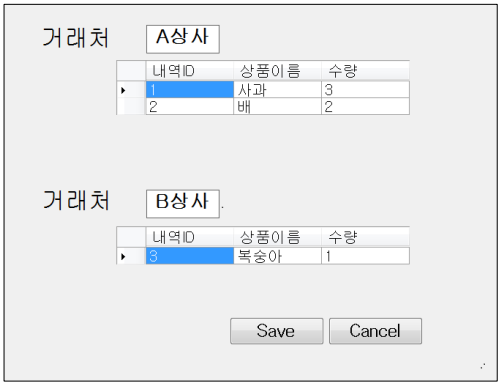

그림 3. 구매발주서 생성

[그림 3]을 참조하면, [그림 2]의 주문 화면에서 "구매발주서 생성" 버튼을 누르는 경우 표시되는 사용자 화면, 즉 구매 발주서를 확인하고 수정하기 위한 구매발주서 화면이 표시된다. 사용자는 예컨대 구매 발주서 화면에서 생성되는 구매 발주서를 확인하고, 각각에 대해서 발주 수량을 수정하거나 또는 발주 상품을 추가 또는 제거할 수 있다. 예컨대 상품 중 사과와 배는 A상사에서 발주하며 복숭아는 B상사에서 발주할 수 있음을 알 수 있다.

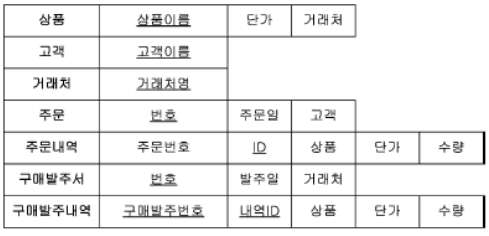

그림 4. 데이터베이스 스키마

[그림 4]는 예시적인 데이터베이스 애플리케이션에 대응하는 데이터베이스 테이블의 예를 나타내는 화면이다. 상품 테이블에는 상품 이름, 단가뿐만 아니라 거래처 필드가 저장되어 있다. 즉 상품 테이블을 참조하면 상품에 따라서 거래처가 결정되는 것을 알 수 있다. 따라서 [그림 3]의 구매 발주서 화면에서 거래처는 자동으로 지정되어 표시될 수 있다.

2. 클래스 정의의 계층화

[그림 5]는 클래스의 정의를 입력받는 단계를 예시적으로 구현한 흐름도 이다.

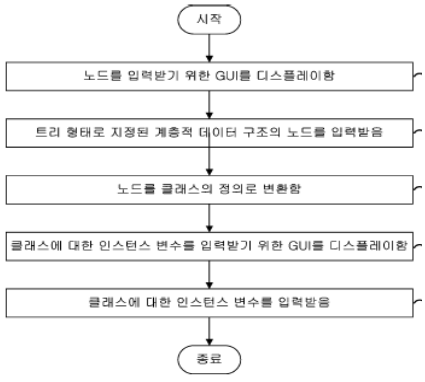

그림 5. 클래스의 정의를 입력받는 단계

우선 트리(tree) 형태로 지정된 계층적 데이터 구조의 노드를 입력받는다. 계층의 모든 노드는 예컨대 Java의 경우, 객체일 것을 요구한다. 이후 단계를 통하여 입력된 노드를 클래스의 정의로 변환한다. 이후 단계를 통하여 변환된 클래스에 대한 인스턴스(instance) 변수를 입력받는다.

[그림 6]은 계층적 데이터 구조 기반 데이터베이스 애플리케이션 의 생성 방법에 있어서, 클래스의 정의를 입력받는 사용자 인터페이스의 예를 나타내는 화면이다. [그림 6]의 사용자 인터페이스를 사용하여, 사용자는 클래스가 각 노드인 트리 형태의 계층을 정의한다. 예컨대 사용자는 노드를 계층적으로 배치하고 노드의 이름을 입력한다. [그림 6]의 예를 참조하면, 계층 전체를 표현하는 클래스는 "구매발주서" 클래스이다. 가장 윗부분의 클래스(거래처)는 루트 클래스로 지칭된다. 루트 클래스는 단 하나만 존재한다. [그림 6]을 참조하면, list1과 list2는 루트의 컬렉션 객체로 자식 객체를 보유할 수 있다. 자식 객체 역시 같은 방법으로 자식의 자식 객체를 보유할 수 있다.

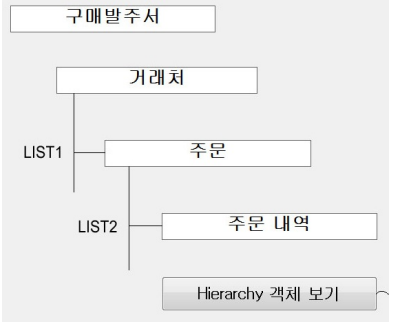

그림 6. 계층 정의

[그림 6]에서 정의된 구조는 다음과 코드의 클래스로 변환된다.

```
class 구매발주서{
    class 거래처{ List<주문>
  list1;}
    class 주문{ List<주문내역>
  list2;}
    class 주문내역 { }
}
```

한편, [그림 6]에서 "hierarchy 객체 보기" 버튼을 누르면, [그림 7]의 화면이 표시된다. [그림 7]은 계층 데이터 구조의 예를 나타내는 화면이다. [그림 7]의 계층 데이터 구조의 예를 나타내는 화면을 참조하면, 루트는 "거래처" 타입 객체들의 컬렉션이고, 컬렉션의 각 객체는 그 자신의 컬렉션 객체들(list1, list2)을 통하여 자식 객체들을 보유할 수 있다. 또 [그림 7]의 예에서 루트의 각 객체들은 "주문" 타입의 자식들을 list1을 통하여 보유하며 "주문" 타입의 객체들은 "주문내역" 타입의 객체를 list2를 통하여 보유한다.

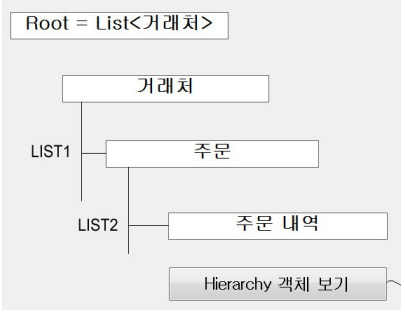

그림 7. 계층 데이터 구조

[그림 6] 또는 [그림 7]에서, "주문" 노드를 클릭하면, [그림 8]이 표시된다. [그림 6] 또는 [그림 7]에서 정의된 list2는 List 타입 변수로 자동 정의되고 그 외의 변수는 기본형만 그 타입으로 가질 수 있다.

[그림 8]은 인스턴스 변수 정의 화면의 예를 나타내는 화면이다. [그림 8]의 인스턴스 변수 정의 화면을 이용하여, 사용자로부터 인스턴스 변수를 입력받으면, 다음과 같은 절차적 인어, 예컨대 Java 형태의 코드가 생성된다

3. 메소드 본체를 SQL로 정의

[그림 9]는 계층 데이터 구조의 테이블 뷰의 예를 나타내는 화면이다. [그림 9]를 참조하면, 예컨대 [그림 7]의 메모리 내의 계층적 데이터 구조를 관계형 모델의 테이블 뷰로 표시한다. 예컨대 [그림 7]에서 "hierarchy 관계형 모델 보기" 버튼을 누르는 경우, [그림 9]의 테이블 뷰가 표시된다. 그러나 [그림 7]의 객체가 직접 관계 테이블로 변환되어 하드 디스크 등의 저장 장치에 저장되는 것은 아니다. 계층적 데이터 구조의 객체는 주키(PK)를 가지지 않기 때문에, 사용자는 주키(PK)를 관계형 모델에 지정하여야 한다.

```
class 구매발주서{
    class 거래처{ List<주문> list1;
  }
    class 주문{
      List<주문내역> list2;
      int 주문번호;
      Date 주문일;
       String 고객;}
    class 주문내역 { }
}
```

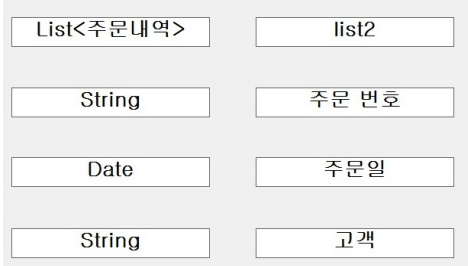

그림 8. 인스턴스 변수 정의

[정의 1] 계층적 구조의 메모리 데이터는 클래스의 인스턴스를 노드로 가지면 노드 인스턴스는 하위 노드를 원소로 하는 컬렉션 타입의 참조형 멤버 변수만 가질 수 있다.

[정리 1] 계층적 구조의 메모리 데이터는 주키만 정의하면 관계형 모델 뷰를 갖기 때문에 SQL로 조작이 가능하다.

[증명] 이는 트리의 각 노드에 클래스를 대응시키고 트리의 에지를 외래키에 대응시킬 수 있기 때문이다. 이때 정의 1의 계층적 데이터 구조 정의에 따르면 이 클래스들은 참조형 멤버 변수를 가질 수 없다. 그러므로 컬렉션 타입의 참조형 변수 이외의 모든 멤버 변수는 기본형 타입만 가질 수 있고 이는 관계형 모델의 제 1 노말 폼 (1st normal form)을 만족시킨다[증명끝].

그림 9. 관계형 모델 뷰

[그림 10]을 참조하면, 사용자는 "주문" 클래스의 메소드 "generatePOs"를 정의한다. 메소드의 리턴 타입은 기본형, 예컨대 void, int 등으로 제한되고 파라미터타입은 여러 개 정의 할 수 있다. 파라미터 타입은 전술하듯이 기본형 또는 이미 정의된 모든 계층의 루트 클래스의 리스트, 예컨대 표시되듯이 "List<구매발주서> "로 제한된다.

또한 [그림 10]을 참조하면, SQL 등의 선언적 언어를 이용하여 메소드의 본체를 입력할 수 있는 SQL 입력 영역이 표시된다. 사용자는 SQL 입력 영역을 통하여 단지 SQL 등의 선언적 형태의 언어로만 메소드의 본체를 입력할 수 있다. [그림 10]의 "데이터 모델 보기" 버튼을 누르면 그림 10에 표시된 메소드에 필요한 데이터 모델이 표시된다[그림 11].

[그림 11]은 혼합 데이터 모델 뷰의 예를 나타내는 화면이다. [그림 11]을 참조하면, 혼합 데이터 모델 뷰는 계층적 데이터 구조의 객체 및 데이터베이스의 테이블의 객체를 혼합하여 표시한다. 즉 계층적 데이터 구조의 관계형 모델 및 테이블을 혼합하여 표시한다.

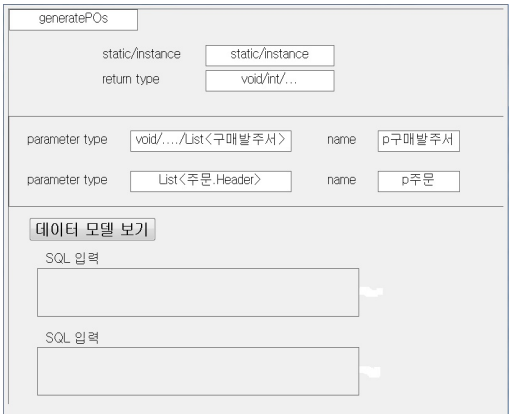

그림 10. 메소드 정의화면

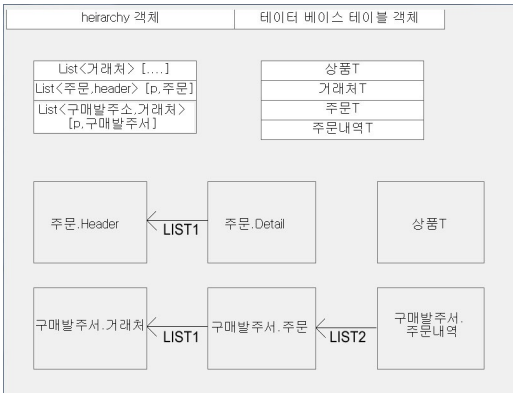

그림 11. 혼합 데이터 모델 뷰

[그림 11]에서는, 사용자가 그림 10에서 파라미터 타입으로 List<주문.Header> 루트 객체를 선택하였기 때문에 [그림 11]의 "hierarchy 객체" 메뉴(240)를 선택하면 보여지는 리스트에서 "List<주문.Header>"가 표시된다. 이때 메뉴를 선택하면 파라미터 ([그림 11]의 "p주문" 즉, 파라미터 이름이 메뉴 아이템에 표시)에 의해

도입되는 주문 계층의 주문 클래스 및 주문내역 클래스가 하단에 표시된다. 다시 말하면 노드 객체들 List<주문.Header>, List<주문.Detail>이 데이터 모델 창에 표시되고, 이전에서 정의된 list1 관계도 표시된다. 계층 객체들이 파라미터로 정의한 루트와 자식들만 데이터 모델 창에 나타나는 것에 비하여, 데이터베이스의 모든 테이블들이 "데이터베이스 테이블 객체" 메뉴 하단에 표시된다. 따라서, 혼합 데이터 모델 뷰를 통하여 계층적 데이터 구조의 관계형 모델 및 테이블 모두가 표시된다. 혼합 데이터 모델 뷰를 통하여 사용자는 용이하게 데이터 모델을 확인할 수 있고, 이에 따라서 메소드 본체를 작성할 수 있다. 메소드의 본체는 [그림 10]의 SQL 입력 영역을 통하여 단지 SQL 등의 선언적 형태의 언어로만 입력될 수 있다. 예컨대 [그림 10]의 SQL 입력 영역 중 어느 하나에 아래와 같은 선언적 언어 형태의 SQL 코드를 메소드의 본체로서 입력하면 절차적 언어의 형태의 본체를 가지는 메소드로 변환한다. 예컨대 [그림 10]의 "generatePOs" 메소드의 본체를 코드 1의 SQL로 정의하면 이를 코드 2-1과 2-2의 자바 코드로 변환시킨다. 주의할 점은 코드 2-1과 2-2의 SQL은 메모리 내의 객체의 테이블 뷰와 함께 하드 디스크의 데이터베이스 테이블들을 혼합해서 포함한다는 것이다.

4. 메소드 SQL을 Java로 전환

코드 1은 메소드 본체를 정의하는 선언적 언어 형태의 코드 예를 나타낸다. 코드 2는 코드 1의 SQL 코드로부터 자동적으로 생성되는 자바 코드의 예를 나타낸다.

코드 1로부터 자동적으로 생성되는 코드이기는 하지만 자동적으로 변환되지 않는다면 사용자가 직접 작성하여야 할 코드이기도 하다. 두 코드를 비교하면, 생성된 코드 길이가 SQL 코드에 비해서 4 배정도 길며, 또한 실제 작성 시간 역시 코드 2-1과 2-2의 경우가 코드 1의 경우에 비해서 10 배 이상 걸렸다. 이는 이 예에서만 적용될 수 있는 수치이고 좀 더 광범위하고 정밀한 실험에서는 절차적 언어의 작성 시간이 선언적 언어의 2 배라는 보고가 있다[13].

```
//Order
 generatePOs {
  Insert Into 구매발주서.거래처(거래처);
  select unique x.거래처 from 주문.Detail x join
  상품T y based on x.거래처=y.거래처;

  Insert Into 구매발주서.주문(번호, 주문일, 거
래처);
  select x.번호, x.주문일, z.거래처
  from 주문.Header x join 주문.Detail y based on
  x.번호=y.주문번호 join 상품T z based on
 y.상품=z.상품이름;

  Insert Into 구매발주서.주문내역(주문ID,
  내역번호, 상품, 단가, 수량, 거래처);
  select x.주문ID, y.내역번호, y.상품, y.수량,
  y.단가, z.거래처
  from 주문.Header x join 주문.Detail y based on
  x.번호=y.주문번호 join 상품T z based on y.상품
=
  z.상품이름;
 }
```

코드 1. 구매발주서 생성 (SQL)

```
// Order
 generatePOs (List<주문.Header> p주문,
  List<구 매발주서.거래처> p구매발주서) {
  for each e: p주문.list1 {// e 는 주문내역
  Boolean contactDuplicateFlag = false;
  String c = select 거래처 from 상품T where
  상품이름=e.상품
  for each f: p구매발주서 {
  if (f.거래처 == c)) {
  contactDuplicateFlag = true;
  break;}
  }
  if (!contactDupliacFlag) {
  구매발주서.거래처 o = new 구매발주서.거래처();
  0.거래처= c;
  p구매발주서.add(o);
  } //if
  //a는 구매발주서.거래처 객체
  for each a: p구매발주서 {
  for each e: p주문 { //e 는 주문 객체
   구매발주서.주문 o = new 구매발주서.주문();
   boolean flag=false;
   for each f: e.list1 {// f는 주문내역 객체
    int x = select count(*) from 상품T where
    상품이름=f.상품 and 거래처=a.거래처;

  if (1x>0) { flag=true; break; }
  } // 거래처 상품내역이 존재하면 flag가 true
  if (flag=true;) {
  o.주문번호=e.주문번호; o.주문일=e.주문일;
  a.list1.add(o);
  }
  } //for
 } //for
```

코드 2-1. 구매발주서 생성 (Java)

또한 SQL 코드는 선언적 언어 형태이므로 이해하기 쉬어 유지보수가 용이하지만 자바 코드는 절차적 언어 형태이므로 이해하기가 훨씬 어려워 유지보수에 비용이 많이 든다. 하지만 [그림 11]에서 보듯이 데이터베이스를 다루는 load, save 메소드는 계층적 메모리 데이터 구조와 데이터베이스의 테이블들을 동시에 혼합하여 다루기 때문에 SQL 작성이 복잡해지는 단점이 있다.

```
for each e: p구매발주서 { //e는 거래처 객체\
   for each f: e.list1 { //f는 주문 객체
    for each g: p주문 {
     for each h: g.list1 {//h는 주문내역객체 {
      String x = select 거래처 from 상품T where
       상품이름=h.상품;
      if (x==e.거래처 && h.주문ID==f.주문번호)
   {
       구매발주서.주문내역 o =
       new 구매발주서.주문내역();
       o.내역ID=h.내역ID;
       o.상품=h.상품;
       o.단가=h.단가;
       o.수량=h.수량;
       f.list2.add( o );
     } //if
     } //for
    }//for
   }//for
  }//for
 }
```

코드 2-2. 구매발주서 생성 (Java)

[그림 12]는 절차적 언어 형태의 본체를 가지는 메소드로 변환하는 단계를 예시적으로 구현한 흐름도이다. [그림 12]를 참조하면, 우선 선언적 언어 형태의 본체를 가지는 메소드가 변환 로직을 이용하여 절차적 언어 형태의 본체를 가지는 메소드로 변환 가능한 지 여부를 미리 확인한다. 이후 단계에서 변환 가능한 것으로 확인되면, 선언적 언어 형태의 본체를 가지는 메소드를 변환 로직을 이용하여 절차적 언어 형태의 본체를 가지는 메소드로 변환하고, 변환 가능하지 않은 것으로 확인되면, 에러 메시지를 출력한다.

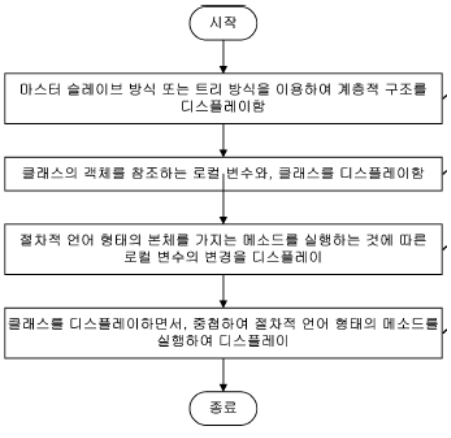

그림 12. 선언적 SQL을 절차적 Java로 변환

5. 모델 실행 및 비쥬얼라이제이션

[그림 13]은 데이터베이스 애플리케이션에 대한 테스트를 수행하는 단계를 예시적으로 구현한 흐름도이다. [그림 13]을 참조하면, 마스터슬레이브 방식 또는 트리 방식을 이용하여 계층적 데이터 구조를 디스플레이 한다. 다음으로, 클래스의 객체를 참조하는 로컬 변수와, 클래스를 함께 디스플레이 한다. 다음으로, 절차적 언어 형태의 본체를 가지는 메소드를 실행하는 것에 따른 로컬 변수의 변경을 디스플레이한다.

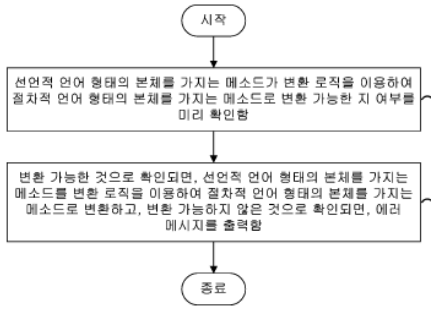

그림 13. 테스트 수행 단계

그림 14. 실행 화면

[그림 14]는 실행 화면 인터페이스의 초기 모습을 나타내는 화면이다. [그림 14]의 실행 화면 인터페이스에서, "hierarchy 루트 참조 로컬 변수" 버튼을 누르면 사용자가 정의한 모든 계층의 루트 객체의 리스트를 참조하는 로컬 변수 리스트가 표시된다. 사용자가 로컬 변

수 리스트 중에 하나를 선택하면 "루트 클래스 이름"+ "Var"의 이름을 갖는 로컬 변수가 현재 스코프 안에 생성되어 그 변수를 나타내는 화면이 표시된다 [그림 14]의 "hierarchy 클래스" 버튼을 누르면 사용자가 기정의한 계층을 구성하는 모든 클래스와 그 이너(inner) 클래스 이름들이 리스트된다. 이 리스트 중 하나를 선택하면 선택된 클래스를 나타내는 화면이 표시되고 해당 화면에는 그 클래스의 static 메소드들이 나타난다. 화면에서 create 메소드는 각 계층의 루트 클래스에 static 메소드로 자동 생성된다. create 메소드는 계층의 루트부터 출발하여 각 단말 노드까지 방문 하면서 해당 노드의 리스트에서 한 객체를 생성한다. 이 객체들은 PK 값도 갖지 않고 모든 값이 null로 지정되어 있다. create 메소드는 hierarchy의 생성자 역할을 한다. 그림 14에서 "변수 스코프 이동" 버튼을 누르면 모든 로컬변수들 즉, [그림 14]에서의 "주문Var"와 "구매발주서Var"이 소멸된다.

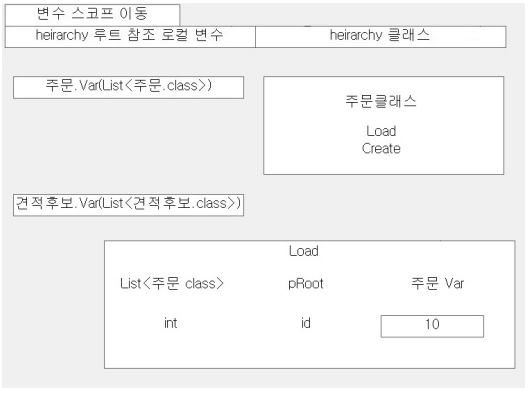

그림 15. load 메소드 실행

[그림 15]는 행 화면 인터페이스에서 주문 클래스의 생성 메소드의 실행하는 예를 나타내는 화면이다. 실행 화면 인터페이스에서 주문 클래스의 [그림 12]에서 주문 클래스의 "load" 버튼을 누르면 load 메소드의 파라미터를 입력 받는 화면이 뜬다. 첫 번째 파라미터 pRoot는 현재 화면에 떠 있는 주문Var 변수가 자동으로 지정되어 있다. 이는 하나의 스코프에서 List<주문Class> 형 변수는 오직 하나 존재할 수 있기 때문이며 해당 변수의 창이 떠 있지 않으면 pRoot 파라미터는 지정될 수 없다. 다른 int 형 변수 id는 사용자가 직접 그 값을 입력한다. 기본형 파라미터는 이와 같이 사용자가 그 값을 입력한다.

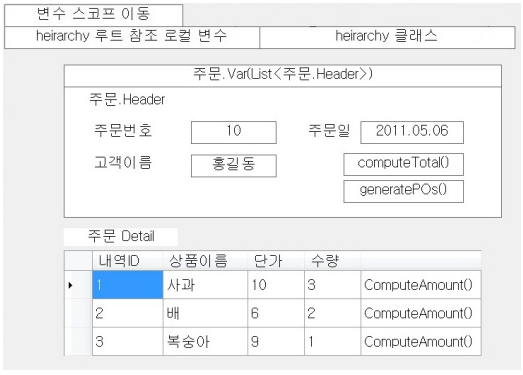

그림 16. load 메소드 실행 결과

[그림 16]은 실행 화면 인터페이스에서 load 메소드를 실행한 결과를 예시적으로 나타내는 화면이다. [그림 16]은 예컨대 load 메소드를 실행하는 경우 화면에 표시되어 있는 변수 (이 경우 주문Var)에 대해서 메소드 실행 결과를 표시한다. [그림 16]에서 computeTotal()과 generatePos()는 주문.Header 클래스의 instance 메소드 computeAmount()는 주문.Detail 클래스의 instance 메소드이다.

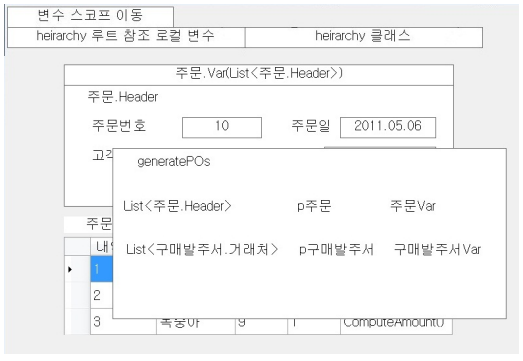

그림 17. generatePOs 메소드 실행 예

[그림 17] 내지 [그림 18]은 실행 화면 인터페이스에서 generatePOs()를 실행하는 예를 나타내는 화면이다.

[그림 17]을 참조하면, generatePOs() 메소드를 실행하면 주문Var가 p주문 파라미터 값으로 자동 지정되고, 구매발주서Var가 p구매발주서 파라미터 값으로 자동 지정된다(화면에 표시되어 있기 때문에). 이후 그 결과 값이 [그림 18]에서 구매발주서 Var 의 값으로 보여진다.

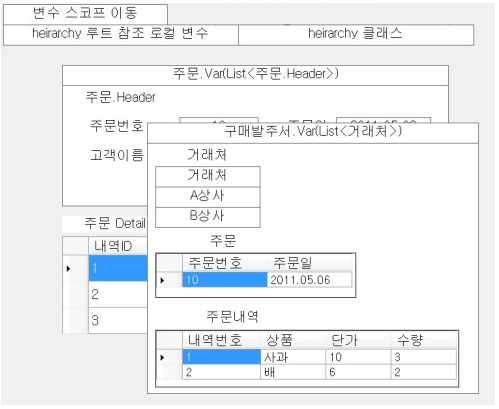

그림 18. generatePOs 실행 결과

[그림 18]은 generatePOs() 메소드 실행과 그 결과를 나타낸다. 구매발주서Var는 초기의 null 값에서 거래처 리스트를 참조하고 그 각각의 거래처 객체는 주 리스트를 가지고 있으며, 각각의 헤더는 주문내역 리스트를 가지고 있다. 거래처, 주문, 주문내역은 계층적 모델이므로 위는 마스터이고 아래는 슬레이브 관계를 각각 갖고 있다. 즉, [그림 18]에서 거래처가 A상사이면 주문은 10, 그리고 주문내역은 10번 주문 중 A상사에서 납품하는 상품만 보여진다. [그림 18]의 자동 생성된 UI는 [그림 3]의 목표로 하는 UI와 다르다. [그림 18]은 마스터 슬레이브 방식으로 동작하고 [그림 3]은 트리 (tree) 방식으로 동작하기 때문이다. 하지만 UI 후단의 데이터 구조는 동일하며 단지 다른 UI 컴포넌트를 사용하여 동작 방식과 사용자에 보여지는 뷰 형태가 다를 뿐이다. 어떤 UI 컴포넌트를 사용하여 계층적 데이터 구조를 사용자에 표시하는 지 는 전적으로 작성자에 달려 있다

VI. 결론

데이터베이스 응용 프로그램 개발에서의 메모리 데이터 조작은 코딩 및 유지보수가 어려운 일반적인 프로그래밍 언어로 이루어져 왔다. 본 논문은 이 중 대부분을 선언적 언어인 SQL로 가능함을 보인다. 일반적인 프로그램보다 폼 중심의 비즈니스 어플리케이션에서 보다 많은 메모리 데이터 조작이 SQL로 대체할 수 있다. 본 논문은 이를 '주문' 예를 통하여 직관적으로 입증하였다. 폼의 MVC[8]모델은 트리 구조를 가지고 있으며 트리 구조의 메모리 데이터는 관계형 뷰를 가지고 있음을 보였다. 이 관계형 뷰 [11]로 인해 메모리 데이터 조작을 SQL로 표현할 수 있다. 그럼으로 SQL을 일반 프로그래밍 언어인 Java로 변환하여 메모리 데이터 조작을 할 수 있다. 이러한 일련의 과정으로 폼 중심 프로그램에 있어서 대부분의 절차적 언어를 선언적 언어로 대체할 수 있게 됐다.

좀 더 이론적으로 SQL로 대체될 수 있는 연산과 그렇지 못한 연산을 분류하여 비 집합연산의 특징을 파악하는 연구를 진행할 것이다. 덧붙여 메소드의 본체를 SQL로 정의하는 것을 발전시켜 임의의 집합 코드 블록을 SQL로 정의하는 연구를 계획하고 있다. 그러기 위하여 MS (Microsoft)의 LINQ (Language Integrated Query) 를 좀 더 비교 연구할 필요가 있다. LINQ의 실패 요인으로는 어려운 문법과 데이터베이스 공간의 테이블 정의들을 통합하지 않은 것으로 분석된다.

참 고 문 헌

[1] 나영국, "자동 생성 폼과 SQL을 이용한 ERD 표현", 한국콘텐츠학회논문지, 제9권, 제5호, pp.63-75, 2009.

[2] C. Bauer, *Hibernate in Action*, Manning Pub, 2007.

[3] C. Begin, *IBatis in Action*, Manning Pub, 2007.

[4] P. L. Zhou and B. Z. Xu, "Visualization of Data Structure on MVC Framework," Tech Report,

Monash University.

[5] B. Johnson and B. Shneideman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures," Visualization '91, Proceedings of IEEE Conference on Visualization, 1991.

[6] C. Cavaness, *Programming Jakarta Struts*, O'Reilly, 2005.

[7] C. Walls and R. Breidenbach, *Spring in Action*, Manning Pub, 2005.

[8] S. Sanderson, *Pro ASP.NET MVC Framework*, APress, 2009.

[9] B. Tate, *Ruby on Rails: Up and Running*, O'Reilly, 2006.

[10] K Haller, "White-box testing for database-driven applications: a requirement analysis," DB Test Proceedings of the Second International Workshop on Testing Database Systems, 2009.

[11] D. M. Self, M. Carrillo, and M. Del Rocio Boone, "A Database and Web Application Based on MVC Architecture," Electronics, Communications and Computers, pp48-48, 2006.

[12] Arthur M. Keller, "Algorithms for Translating View Updates to Database Updates for View Involving Selection, Projections, and Joins," Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems, 1985.

[13] E. J. O'Neil, "Object/relational mapping 2008: hibernate and the entity data model (edm)," SIGMOD 2008.

[14] C. Ireland, D. Bower, M. Newton, K. Waugh, "A Classfication of Object-Relational Impendence Mismatch," First International Conference on Advances in Databases, Knowledge, and Data Applications, 2009.

[15] J. S. Gero and M. Balachandran, "A Comparison of Procedural and Declarative Programming Languages for the Computation of Pareto Optimal Solutions," Engineering Optimization, Vol.9, pp.131-142, 1985.

저 자 소 개

나 영 국(Young-Gook Ra) 정회원

- 1987년 2월 : 서울대학교 전자과 (공학사)
- 1996년 6월 : 미시간대학 컴퓨터학과(공학박사)
- 2002년 12월 ~ 현재 : 서울시립대학교 전자전기컴퓨터학부 교수

<관심분야> : 데이터베이스, 소프트웨어공학

우 원 석(Won-Seok Woo) 정회원

- 1987년 2월 : 서울대학교 경영학과(학사)
- 1990년 5월 : 미국 카네기멜론대학교 경영학 석사(MBA)
- 2001년 6월 : 미국 뉴욕주립대 경영학 박사(Ph.D.)
- 2004년 9월 ~ 현재 : 이화여자대학교 국제학부 교수

<관심분야> : 기업전략, MIS, 콘텐츠 가치평가