객체지향 방법론에서의 데이터 모델링의 역할

Subtitle :-

Access method of Data Modeling from Object-Oriented Modeling

(주)엔코아 정보 컨설팅

대표컨설턴트 이 화식

객체지향 소프트웨어 개발 방법론에서 사용하는 클래스(class)는 데이터 아키텍처의 데이터 영역이나 데이터 클래스, 엔터티, 서브타입과 매우 유사한 개념이다. 클래스와 데이터 모델 간의 대비를 통해 객체지향 방법론에서는 데이터 모델링의 역할이 어떻게 이루어져야 할 것인지를 알아보기로 하겠다.

객체지향 방법론은 클래스의 계층구조를 이용하여 매우 강력한 클래스 재사용을 지원한다. 하위 클래스를 생성한다는 것은 소프트웨어 재사용 측면에서 볼 때, 두 가지의 중요한 의미를 갖는다. 첫째 기존 클래스에 대한 수정을 매우 유연하게 지원하게 한다. 둘째 기존의 클래스에 새로운 기능 추가가 매우 용이하다는 것이다. 객체지향 방법론이 소프트웨어 재사용에 도움을 주는 것은 프레임워크(Framework)를 이용한 모듈의 재사용이다.

객체지향 설계의 결과는 결국 클래스의 계층구조라고도 할 수 있겠다. 각 클래스는 제어와 데이터 구조가 함께 포함된 모듈이다. 우리의 대상 영역은 객체와 거기에 관련된 메소드(method)들의 모임으로서 좀더 자연스럽고, 사실적으로 관찰될 수 있다. 객체(Object)는 객체지향 설계의 초기 요소이자 가장 작은 독립적인 존재가치를 가지는 개체이다. – 사물(Thing)을 말하며 고유성과 상태, 행동을 갖음 – 이러한 낱개의 객체들은 점차 그들간의 공통점이 인식되면서 결합되어, 같은 공통점을 갖는 객체들의 집합인 클래스를 만들게 된다. 우리는 또한 이들 클래스를 좀더 추상화시켜 추상 클래스(Abstract Class)를 형성할 수 있다. – 공통적인 객체들의 집합 – 이와 같은 추상화 작업의 가장 상위 단계는 프레임워크이다.

객체지향 설계는 마치 기존의 시스템에 적용된 프로토타입(Prototype) 접근방법과 같이 순환적이다. 프로그래머는 클래스의 집합에서 설계를 시작해서, 그것을 확장 시키고, 고쳐서 하나의 응용 프로토타입으로 꾸며 맞춘다. 주로 최종 사용자와의 협의를 통하여 프로토타입의 개선 방안을 도출하고, 설계자나 프로그래머는 다시 분석/설계 작업을 반복하여 발전시켜 나간다. 이러한 재작업을 통하여 클래스는 개선되어 재구성되며, 만일 이렇게 개선된 클래스가 추후의 응용에 재사용될 수 있을 만큼 일반적이라면 그것을 표준 클래스 라이브러리(Standard Class Library)에 추가시킨다. 프레임워크 설계자는 응용 프로그램간의 유사성을 발견하여, 미래의 비슷한 문제에 대한 프로그래밍 해결책으로 유용한 프레임워크를 개발한다.

객체지향에서 말하는 클래스는 '데이터 속성과 메소드'를 같이 가지고 있다는 것이 데이터 모델의 엔터티와 가장 큰 차이점이다. 물론 데이터 속성만 가지고 있을 수도, 메소드만 가지고 있을 수도 있기 때문에 속성만 가지고 있는 클래스는 엔터티와 개념적으로는 동일하다고 할 수 있다. 이 말은 곧 객체지향 방법론에서는 별도의 데이터 모델링을 하지 않

고 클래스 정의만 제대로 하면 된다고 해도 이론적으로는 누구도 부정할 수 없다. 이 말을 '구조적 방법론'에 그대로 대입시켜 본다면 DFD 를 그릴 때 데이터 저장소를 제대로만 정의하면 데이터 모델링을 할 필요 없다고 할 수 있는 것과 다를 것이 없다.

그러나 문제는 데이터 모델링을 하지 않고서도 이상적인 데이터 구조를 데이터 저장소에 정의하였다면 누구도 시비를 걸지는 않았을 것이다. 그렇지만 '안 되는 것은 아니지만 잘 되는 것은 아니다'라는 것은 현실적으로 보더라도 매우 중요한 일이 아닐 수 없다.

모든 것을 종합하여 보았을 때 결국 데이터와 기능은 존재하지 않을 수는 없다. 단지 DFD 에서는 기능을 버블로, 데이터를 데이터저장소로 표현했고, 객체지향 방법론에서는 이를 모두 클래스라는 개념으로 통합했을 뿐이다. 그렇다면 역시 객체지향 방법론에서도 '제대로'만 클래스를 만든다면 별 문제가 없겠지만 데이터 모델링을 무시하고서는 오류에 빠질 수 있을 확률이 충분하다는 것을 간과해서는 안 된다.

그것은 클래스를 도출하고 결정하는 기준이나 시각이 데이터의 본질적 성격과 얼마간의 차이가 분명히 있기 때문이다. 객체지향 방법론이 지향하는 것은 깊이 숨어 있는 본질적인 집합이 아니라 겉으로 드러난 실용적인 집합이라 할 수 있다. 영화를 예로 들어보자. 영화에는 '배역'이라는 의미와 '배우'라는 일견 비슷해 보이는 개념이 있다. 그러나 좀더 구체적으로 따져보면 이들 두 개의 집합은 결코 동일하지 않다는 것을 발견할 수가 있다.

물론 중요한 배역에는 단 한 명씩의 배우가 캐스팅되겠지만 그렇지 않은 배역에는 한 사람의 배우가 분장을 달리하고 여러 배역을 맡을 수도 있을 것이다. 감독 입장에서는 실제 배우가 누구든 상관없이 가장 배역에 어울리는 사람을 쓰는 것에 목적이 있고, 비용을 걱정해야 하는 제작자나 소품 조감독은 배역에 문제가 없도록 하면서 최소공배수의 배우 집합(SET)을 준비하려고 할 것이다.

가장 이상적인 조화는 최소의 배우로써 최대의 배역을 무리 없이 소화해 내는 것이다. 여기서 말한 '배역'은 클래스가 될 것이고, 실제 존재해야 할 집합인 '배우'는 당연히 실체 집합을 의미하는 엔터티라고 할 수 있겠다.

이처럼 클래스만큼 데이터를 만들어도 비용을 무시할 수 있다면 고민할 필요가 없겠지만 그렇지 못하다면 우리의 클래스를 위해 필요한 최소공배수의 데이터 집합이 무엇인지에 대해 반드시 고민해 봐야 한다. 곧 객체지향 방법론에서도 올바른 데이터 구조를 정의하기 위해서는 무엇인가 좀더 체계적인 방법이 필요하다는 것을 의미하고 있다.

대부분은 관계형 데이터베이스를 사용하고 있기 때문에 이러한 문제는 당장의 현실적인 문제라는 것을 명심해야 한다. 물론 객체지향으로 접근하기 위해서는 여기서 생성되는 클래스 혹은 클래스의 계층구조는 객체지향 데이터베이스(OODB, Object Oriented Database)의 데이터 모델과 일치하므로, 객체지향 데이터베이스 관리 시스템(OODBMS)을 적용하는 것

이 가장 효과적인 것만은 틀림이 없다.

우리가 가장 많이 사용하고 있는 관계형 데이터베이스에는 여러 가지 문제점들이 존재하고 있는 것은 사실이다. 여기에 대한 해결책으로 제시된 개념이 OODB 기술이며, 소프트웨어 전반에 불고 있는 객체지향 흐름을 타고 기대와 관심이 한껏 고조되고 있다. 그러나, 현재까지의 OODB 시스템들은 객체지향 시스템으로서의 유연한 모델링 기능은 지원하는 반면, 표준의 부재, 필수적인 데이터베이스 기능의 미비 외에도 기존 사용자들의 지나친 RDB 집착 성향으로 인해 현실적으로 보편화되는데 많은 시간과 노력이 요구되고 있다. 따라서, 최근 객체관계형 데이터베이스(ORDB;Object Relational Database) 기술이 크게 부각되고 있다. ORDB 기술은 RDB 에서 사용하는 데이터를 확장하는 기술이다. 즉, 관계형 데이터베이스를 객체지향 모델링과 데이터 관리 기능을 갖도록 확장한 것으로 설명할 수 있다.

ORDB 는 기존 NDB(네트워크 데이터베이스), RDB, OODB 의 특징을 모두 가지고 있다. NDB 의 포인터 및 항해(navigation) 특성, RDB 의 유연성, 데이터 독립성, 표준 질의어 지원 특성, OODB 의 상속(inheritance) 및 캡슐화(encapsulation) 특성이 그것이다. ORDB 는 객체 데이터 모델의 특수한 하나의 경우로서 관계형 데이터 모델을 제공한다. 또한, RDB 에서 지원하는 수준의 질의어 제공, 질의 최적화, 동시성 제어, 트랜잭션 관리 기능을 제공한다. 따라서, 사용자는 기존 RDB 애플리케이션을 그대로 수행시킬 수 있다는 커다란 장점을 가지고 있다.

이로 인해 오늘날 기업의 기간업무용 시스템의 기반을 이루고 있는 관계형 데이터베이스와 객체 기술의 접목은 피할 수 없는 현실로 다가오고 있다. ORDBMS 는 업체별로 다른 시도를 통해 소개되고 있으나 중요한 사실은 관계형 데이터베이스의 장점과 객체 기술의 장점만을 제공하는 '이상적인' 데이터베이스를 찾기는 아직까지는 쉽지 않다는 것이다. 비정형 데이터 처리는 객체 기술이, 질의 성능의 최적화 및 확장성은 관계형 기술이 우월하므로 업체별 개발 방식은 그 성능이나 활용면에서 각각 다를 수 있다는 점을 지적하고 싶지만 향후 수 년내에 양 기술의 접목은 최적화 및 필연적이 될 것으로 기대된다.

중요한 것은 객체기술을 효과적으로 도입함으로써 앞으로의 수고나 비용을 줄이는 것도 중요하지만 기존의 투자분을 보호하는 것도 이에 못지 않게 중요하다는 사실이다. 객체도입의 필요성 중에 개발자들이 가장 공감을 느낄 수 있는 것은 아마도 오브젝트 컴포넌트(Component)의 재사용성 일 것이다. 오브젝트 컴포넌트의 재사용성은 복잡한 비즈니스를 모델링하는데 더 없는 장점으로 떠오르고 있다. Visual C^{++}과 Java와 같은 객체 지향 개발 환경은 컴포넌트 기반의 소프트웨어 개발에 촉진제 역할을 한다.

이러한 현실적인 환경으로 인해 애플리케이션의 개발은 객체지향인데 반하여, 그 애플리케이션을 통해 액세스되는 데이터는 관계형 데이터베이스 내에 주로 저장되고 있는 것이

현실이다. 여기서 해결해야 하는 이들간의 매핑(Mapping)은 새롭게 심각한 부담으로 나타나고 있다. 이런 문제는 아직 완벽한 기준도 없이 실전에서 함부로 적용되고 있다. 객체지향의 입장에서 클래스를 설계한 후 다시 최상의 데이터 모델을 만들고 이를 서로 매핑하여야 함에도 불구하고 이를 1:1 로 테이블을 설계하는 곳이 많다는 것이다. 이것은 앞서 예로 들었던 영화의 모든 배역에 그 비중이 주연이든, 조연이든, 엑스트라든 모두 하나씩의 배우를 캐스팅한 것이나 다를 것이 없다. 물론 등장인물이 매우 적은 소형 영화이거나 배우의 수가 비용과 직결되지 않는다면 문제될 것이 없다. 다시 말해서 클래스 수가 많지 않거나 데이터가 소량인 시스템이라면 이것도 충분히 고려해 볼만한 방법이겠지만 기간 시스템과 같은 곳에 이러한 방법을 적용한다는 것은 어불성설이라 하지 않을 수 없다.

처음에는 객체지향을 하는 사람들이 자신이 하는 일에 대한 자부심에 너무 고무되어 있는 나머지 객체지향이 제일이고 이것만이면 모든 것이 해결된다고 맹신하였다. 그들은 데이터 모델링은 전혀 필요 없다고 고집을 부렸고, 그렇게 해서 실제로 성공(소형 시스템)을 하면서 자가당착에 빠졌다. 이들이 선택한 방법은 당연히 위에서 말한 1:1 매핑을 하는 방법이다. 작은 성공으로 인해 가속이 붙은 자신감은 급기야 대형 시스템에 이와 같은 방법을 시도하게 했고 필연적으로 엄청난 문제를 만들게 되었다.

객체지향 기술이 대형 기간 시스템에 적용되기 위해서는 데이터 모델에 대한 현실적인 문제를 반드시 해결해야 한다. 그렇지 않고서는 절대로 파고들 수가 없다는 것을 분명하게 인식하지 않으면, 지금까지 그래왔듯이 비록 우수한 개념임에도 불구하고 계속해서 부수적인 소형 시스템에만 적용되는 한계를 결코 벗어날 수 없을 것이다. 객체지향 방법론이 아무리 순환적으로 진행하면서 점차적으로 접근해 가는 방식이라 하지만 시작이 너무 부실하면 시행착오가 늘어나는 법이기 때문에 초기에 중요한 것들을 견고하게 하는 일은 논리적으로 볼 때도 중요한 의미를 가진다는 것은 확실하다.

데이터 모델링에서 강조하는 '시스템의 골격부터 명확히 하자'는 원칙이 그대로 적용된다는 것을 뜻한다. 다시 말해서 객체지향 방법론에서도 보다 근본적인 객체는 '데이터'일 수 밖에 없기 때문에 데이터 형태의 객체들을 먼저 정의해 두는 것도 그리 나쁘지는 않을 것이란 뜻이기도 하다. 물론 순환적, 점차적으로 접근해 가기 때문에 크게 보면 데이터형 클래스와 프로세스형 클래스, 이들이 서로 결합된 클래스, 이를 더 크게 추상화한 클래스는 같이 정의해 가는 것처럼 보이기는 한다.

기존의 방법론에서 프로세스 모델링을 마친 후 이를 이용해 데이터 모델링을 하듯이 어느 정도 클래스 정의를 끝낸 후 이를 토대로 정확한 데이터 모델을 수립하는 방법도 생각해 볼 수 있다. 만약 여러분이 업무에 전혀 무지한 상태에서는 도저히 객관적이고 구체적인 데이터 모델링을 할 수 없다면 오히려 이 방법이 바람직할 수도 있다. 또한 이 방법을

적용해도 과거에 비해서는 문제가 크지 않을 수가 있다. 그것은 객체지향 방법론이 순환적으로 접근하기를 강요하고 있기 때문에 정확한 데이터 모델이 뒤 늦게라도 나타나기만 했다면 이것이 다시 전체에 제대로 반영 되어질 것이기 때문이다.

필자는 여러분이 가장 근본이 되는 데이터 집합부터 명확히 하고 이를 토대로 객체와 클래스를 설계해 가는 것이 가장 바람직하다고 믿으며, 그렇게 접근하기를 추천한다.

RDBMS 의 출현과 거의 비슷한 시점에 이미 OODBMS 가 출현했지만 기세 싸움에서 패퇴하였음을 간과해서는 안 된다. 우리가 시스템화를 하고자 하는 영역 중에는 객체지향 데이터베이스 개념을 적용해야만 하는 일도 분명히 있다. 그러나 인간 세상의 업무라는 것에는 특이한 것 보다 평범한 것이 더 많이 있다는 사실을 잊어서는 안 된다. 평범한 업무의 적용에는 아직은 관계형이 더 적합하고, 객체 기술이 이를 밀어내고 그 자리를 차지하기에는 아직까지는 힘이 부족해 보인다. 이러한 지지세력을 바탕으로 RDBMS 벤더가 세상의 흐름을 지배하고 있는 이상, 당분간 이러한 체제는 쉽게 무너지지 않을 것이 분명하다.

결론적으로 여러분들은 기존의 관계형 개념에 객체 개념을 접목한 객체 관계형 데이터베이스를 남들 보다 앞서 활용하기 위해 각고의 노력을 경주하는 것이 훨씬 현실적인 판단이라는 것을 인정해야 할 것이다.

앞으로의 객체관계형 데이터베이스가 갖춰야 할 사항을 나름대로 정리해 보았다. 객체관계형 데이터베이스는 관계형의 데이터 저장 방식과 객체 모델링이 가장 효과적으로 결합될 수 있어야 한다. 사용자가 정의한 데이터 타입을 유연하게 사용할 수 있어야 하고, 관계형 데이터베이스에 비해 멀티미디어나 대형 오브젝트(Large Data Object)를 더욱 잘 지원할 수 있어야 하며, SQL 과 같은 DBMS 표준은 물론 CORBA 나 DCOM 과 같은 객체 표준도 준수해야 한다. 게다가 기업 내의 주요 업무를 지원하는 애플리케이션을 위해 질의 성능이나 확장성, 보안 등의 측면에서 현재 RDBMS 의 수준을 능가할 수 있어야 한다.

관계형 데이터베이스에서 사용할 수 있는 데이터 타입은 Character, Number 그리고 Date 와 같이 제품에 따라 약간의 차이가 있기는 하지만 한정이 되어 있다. 이들 데이터 타입에 주어지는 오퍼레이션 역시 데이터베이스 엔진 내에 이미 고칠 수 없도록 코딩되어 있다. 객체관계형 데이터베이스는 사용자들이 정의하는 데이터 타입과 이들 데이터 타입 상에서 실행되는 오퍼레이션을 지원할 수 있어야 한다. 사용자가 정의한 데이터 타입은 시스템 내에서 다른 데이터들과 아무런 문제없이 함께 사용되어야 한다.

예들 들어 사용자가 주문이라고 하는 데이터 타입을 Order 라고 정의하는 경우, 이 타입의 속성(Attribute)은 관계형 테이블의 컬럼과 유사할 것이다. 메소드(Method)는 관계형 데이터베이스에서 저장형 프로시져(Stored Procedure)를 정의할 때와 마찬가지 방법으로 정의되면 된다. 가령, Order 라는 데이터 타입의 속성에는 주문번호, 주문 고객명, 주문품목, 수량 등이

정의될 수 있고, 메소드로는 Shipping, Holding, Cancel 등이 정의 될 수 있을 것이다.

‘POINT’와 같은 데이터 타입은 더욱 복잡할 수 있는데, 이는 X 축과 Y 축의 좌표 개념으로 정의될 수 있다. 이 같은 데이터 타입에 작동하는 함수로는 거리를 산출하는 것이라든지, 주어진 반경내의 영역을 찾아내는 것 등이 가능할 것이다. 이럴 경우 위상연산을 통해 "새로 개설한 매장으로부터 반경 2Km 안의 경쟁사 매장의 위치 및 개수는?" 과 같은 질의를 만족시킬 수 있다.

객체관계형 데이터베이스는 또한 오브젝트(객체)를 상세한 정도에 따라 모델링할 수 있어야 하고, 필요시 더 복잡한 오브젝트를 만들어내기 위해 다른 오브젝트를 참조할 수 있어야 한다. 예를 들면, 회사 내에서 하나의 부서는 직원, 자산, 관리자, 위치 등 몇 개의 오브젝트가 모여 구성된다. 따라서 ORDBMS 내에서 ‘부서’를 모델링하기 위해 부서를 구성하는 소위 '컴포넌트 오브젝트(Component Object)'들의 특성을 잘 정의해야 한다.

시스템이 진정한 객체관계형이 되기 위해서는 데이터 타입의 확장들이 메소드에 동적으로 링크되어 서버 코드를 다시 코딩하는 일이 없도록 해주어야 하고, 클라이언트나 서버에서 선택적으로 활성화되고 실행될 수 있어야 하며, 적절한 보안/통제 기능이 제공 되어져 서버 자체의 보안이 영향을 받아서는 안된다.

멀티미디어와 같은 비정형 데이터는 기존 관계형 데이터(행과 열로 표현되기 때문에 ‘스칼라 데이터’라고도 함) 기반 애플리케이션의 유용성을 높일 수 있지만, 오브젝트 저장을 위해 스페이스 요구량이 급격히 증가한다. ORDBMS 는 대형 오브젝트(LOB: Large OBjects)를 수준급의 성능으로 처리할 수 있어야 한다. 또한 데이터베이스는 클라이언트 애플리케이션에서 커다란 객체 가운데 원하는 부분만을 끌어내는 'Piecewise' 액세스를 지원할 수 있어야 한다.

관계형 데이터베이스를 발전시켜 나감으로써 객체지향에 접근해 가는 방식은 몇 가지 이점을 지닌다. 첫째, 십 수 년간 입증된 또 확산된 - 그래서 어찌 보면 익숙한 - 관계형 데이터베이스 기술을 지렛대를 활용하듯이 이용할 수 있다는 것이다. 둘째, 현재 전 세계적으로 가장 많이 사용되는 관계형 데이터베이스 기반의 애플리케이션에 객체 기술로의 전이를 위한 ‘통로(Paths)’를 제공하게 된다는 것이다.

다시 말해 객체지향 애플리케이션과 기존에 사용해오던 애플리케이션이 공존할 수 있다는 것이다. 뿐만 아니라, 관계형 데이터베이스가 제공하는 신뢰성, 확장성, 보안, 질의 최적화 등이 객체 기술이 도입되더라도 그대로 보장될 가능성이 가장 크다.