객체지향 데이타베이스에서의 새로운 데이타 추상화 단계
(A New Data Abstraction Level for Object-Oriented DBMSs)

조 은 선 † 한 상 영 †† 김 형 주 †††
(Eun-Sun Cho) (Sang-Yong Han) (Hyoung-Joo·Kim)

요 약 기존의 CODASYL과 관계형 데이타베이스 시스템등에서는 물리적 단계(physical level) - 개념적 단계(conceptual level) - 뷰 단계(view level)의 세가지 데이타 추상화 단계를 지원하고 있다. 이로써, 데이타베이스 사용자들은 데이타베이스의 복잡한 구현 부분을 추상화 시켜 데이타를 접근하게 되며, 하위 단계에서 다루어지는 보다 물리적인 변화에 의해 상위 단계에서 작성된 응용 프로그램을 다시 작성하는 것을 방지한다.

그러나, 객체지향 데이타베이스 모델은, 메쏘드, 클래스 계층 구조 등 기존의 데이타 모델과는 다른 새로운 개념들을 제공하고 있기 때문에, 데이타 추상화도 기존의 방식만으로는 한계가 있다. 본 논문에서는 이에 따르는 문제점들을 검토하고, 객체지향 데이타베이스의 특성에 의해 요구되는 새로운 데이타 추상화 단계의 도입을 제안한다. 그리고, 제안된 새로운 데이타 추상화에 따르는 잇점과 그 지원 방식에 대해 고찰한다.

Abstract CODASYL and relational DBMSs take traditional three level data abstraction approaches - physical level - conceptual level - view level. This means that complicated data implementation details and complicated data structures are hidden from users As such, data abstraction prevents modification in a lower abstraction level from affecting the applications in a higher level.

However, object-oriented models include such new features like methods and class inheritance that the previous abstraction approach shows some limitations in object-oriented database systems. In this paper, we introduce a new level of data abstraction, called the 'class implementation level', which may overcome the limitations in previous abstraction approach. We describe how to support the new abstraction level in current object-oriented database systems.

1. 개 요

데이타베이스 사용자들은 데이타베이스의 복잡한 구현 부분과 무관하게 데이타를 접근하게되는 경우가 많다. 이를 위해 데이타베이스 관리 시스템에서는 몇가지 데이타 추상화 단계를 두고, 사용자의 성격에 따라 각 단계에 의한 추상화된 관점으로 접근할 수 있도록 해준다. 그리하여, 하위 단계에서 다루어지는 보다 물리적인 데이타 부분이 변화되어도 상위 단계 응용 프로그램을 다시 작성할 필요가 없도록 '독립성(independence)'을 지원한다.

CODASYL과 관계형 데이타베이스 시스템등에서는 데이타의 추상화 단계로서 그림 1과 같이 물리적 단계(physical level) - 개념적 단계(conceptual level) - 뷰 단계(view level)의 세가지 단계를 지원하고 있다[1]. 물리적 단계에서 데이타베이스를 변화시켜도 응용프로그램이 변하지 않는 성질을 '물리적 독립성', 개념적 단계에서 데이타베이스를 변화시켜도 응용프로그램이 변하지 않는 성질을 '개념적 독립성'이라한다[1]

· 본 연구는 한국 과학재단에서 부분적으로 지원을 받아 수행되었음(과제번호 : 94-2180, 과제명 : 'ODMG 표준 객체모델을 근간으로 C++을 확장한 객체지향 데이타베이스 프로그래밍 언어에 관한 연구'
· 본 연구는 통상 산업부에서 지원하는 객체지향 데이타베이스 'SOP(SNU OODBMS Platform)' 개발 과제의 일부인 객체지향 데이타베이스 프로그래밍 언어 개발 과정 중 수행되었음
† 학생회원 : 서울대학교 컴퓨터공학과
†† 종신회원 : 서울대학교 전산과학과 교수
††† 종신회원 : 서울대학교 컴퓨터공학과 교수
논문접수 : 1997년 1월 13일
심사완료 : 1997년 7월 31일

Copyright (C) 2005 NuriMedia Co., Ltd.

객체지향 데이타베이스 시스템에서도 관계형 데이타베이스의 '테이블'을 '클래스'로, '레코드'를 '객체'로 바꾸어 단순하게 추상화 단계를 설정하여 생각할 수 있다 그러나, 객체지향 데이타베이스 모델은 기존의 데이타 모델들과는 다른 새로운 개념들이 많이 추가되었기 때문에, 이와 같은 지원은 한계를 가지게 된다. 본 논문에서는 이러한 문제점들을 검토하고, 객체지향 데이타베이스에 새로운 추상화 단계로서, '클래스-구현부 단계(class-implementation level)'의 도입을 제안한다. 이것은 일반 개념적 단계의 사용자와, 해당 스키마 클래스의 메쏘드를 구현하는 클래스-구현부 단계의 사용자를 분리함으로써, 스키마의 정의와 사용의 독립성을 보장하는 것이다. 이로써 데이타 공유가 수월해지고, 스키마 변환 비용이 낮아지는 등의 여러 잇점이 생기게 됨을 보인다.

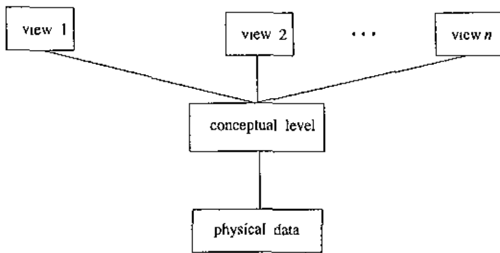

그림 1 기존의 추상화 단계[1]

논문의 구성은 다음과 같다. 먼저, 다음 절에서는 객체지향 데이타 모델을 위해 고려해야할 사항들을 나열한다. 3절에서는 기존의 세 단계 추상화 방식을 따르는 객체지향 데이타베이스 시스템들이 가지는 한계점들에 관해 검토하여, 새로운 단계의 추상화를 도입하는 동기를 밝힌다. 4절에서는 클래스-구현부 추상화의 도입에 대한 설명과 프로그래밍 방식의 변화에 대해 기술하고, 이로써 지원되는 데이타 독립성이 주는 잇점에 대해 고찰한다. 5절에서는 기존의 객체지향 데이타베이스에서 클래스-구현부 추상화를 지원하기 위한 방법과 추가되어야하는 기능들에 관해 살펴보고, 6절에서는 관련 연구들과의 관계를 보인다.

2. 객체지향 모델을 위한 고려 사항

객체지향 데이타베이스를 위한 추상화 단계를 결정하기 위해서는 객체지향 데이타베이스만의 여러 특성에 대하여 고려해야한다. 먼저, 객체지향 데이타베이스는 모델 자체가 관계형 데이타 모델과는 달리 여러 가지 새로운 개념들을 가지고 있는데, 대표적으로 클래스 내부에 정의되는 함수인 '메쏘드'와 클래스 간의 상하 관계인 '클래스 계층 구조'를 들 수 있다. 각 클래스는 메쏘드의 정의시에만 접근할 수 있는 데이타나 메쏘드인 '전용 부분(private part)'을 클래스 내부에 가질 수 있는데, 이것을 캡슐화(encapsulation)[2]라고 한다. 클래스 계층 구조는 클래스 간의 상하 관계에 기인한 것으로, 스키마의 모델링 능력을 향상시키고, 메쏘드 코드(전용 부분도 포함)에 대한 재사용성을 준다.

관계형 데이타베이스에서 SQL, C 등 여러 인터페이스를 가지는 것과 같이, 객체지향 데이타베이스에서도 C++, Smalltalk, SQL3, 객체 질의어 등 여러 인터페이스를 제공하는 것이 바람직하다 그러나, 객체지향 모델은 관계형 모델과는 달리 그 종류가 다양하기 때문에 데이타베이스 자체의 객체모델과 C++, Smalltalk 등의 인터페이스 언어 모델이 다르다. 따라서 이들을 자연스럽게 통합하는 일이 필요하다

또, OMG(object management group)에서 여러 객체지향 데이타베이스들의 모델의 장점을 결합시켜 제안한 ODMG-93 객체지향 데이타베이스 모델[3]에 따르면, SQL과 비슷한 객체 질의 언어 'OQL(object query language)', 응용 프로그램에서 사용되는 객체 관리 언어 'OML(object management language)', 특정 언어에 무관한 데이타(스키마) 정의 언어 'ODL(object definition language)'의 세가지 종류의 데이타베이스 인터페이스를 지원하는 것을 고려할 필요가 있다[1].

3. 기존의 추상화 방식의 적용과 그 한계

이 절에서는 객체지향 데이타베이스 시스템에 기존의 세가지 단계의 추상화가 적용되는 형태에 관해 검토하고, 여러 가지 한계들에 관해 고찰한다.

본문에서 가정되는 객체지향 데이타베이스 관리 시스템은, 스키마를 정의하는 언어인 데이타 정의어를 제공하고 SQL문과 비슷한 데이타 질의어를 지원한다. 또, 스키마의 구현및 일반 응용프로그램의 작성을 위한 데이타베이스 프로그래밍 언어를 제공하는데, 대부분의 상용 객체지향 데이타베이스 시스템에서와 같이 C++을 기반으로 하는 것으로 가정한다 이것은 단지 구문상의 편의를 위한 것이므로, C++외의 다른 객체지향 언어에도 쉽게 적용될 수 있으리라 본다 또 본 논문에서는 클래스의 메쏘드 구현을 제외한 나머지 기술을 '선언

1) 객체지향 데이타모델에서 특기할 사항중 객체 식별자(OID)에 관한 것은 데이타 추상화에 별 영향을 미치지 않는다고 판단되어, 고려 대상에서 제외하였다

Copyright (C) 2005 NuriMedia Co., Ltd.

(declaration)'이라고 하고, 메쏘드 구현을 포함한 기술을 '정의(definition)'라고 한다.

3.1 개념적 단계에 내재된 서로 다른 두가지 추상화 관점

뷰-단계는 가장 상위 단계의 사용자들이 질의등으로 데이타베이스를 접근하는 관점을 추상화한 것이다. 객체지향 데이타베이스의 뷰가 관계형 데이타베이스와 다른 점은, 기존의 스키마로 부터 도출되는 것이 새로운 테이블이 아니라, 새로운 클래스와 클래스 계층 구조라는 것이다. 현재 O_2[4]의 상용 객체지향 데이타베이스 시스템들에서 뷰를 지원하며, ODMG-93 모델에서는 향후 연구과제로 남아 있다.

물리적 단계에서는 객체지향 모델에 맞도록 데이타를 객체 식별자를 포함한 데이타 개체에 의해 표현하고, 각 개체간의 관계를 이 객체 식별자를 써서 표현한다.

개념적 단계에서는, 관계형 데이타베이스와 마찬가지로, 스키마가 설계된 그대로 사용자에게 제공된다. 그런데, 객체지향 데이타베이스 시스템은 클래스에 캡슐화가 적용되기 때문에, 관계형 데이타베이스와 달리 개념적 단계 내에 두 가지 서로 다른 관점이 존재하게 된다. 즉, OOER 다이어그램이나 데이타 정의언어 등으로 스키마를 설계하거나 데이타 질의어등으로 접근하는 '사용자'의 관점과, 스키마 클래스의 메쏘드를 정의하는 '구현자' 관점의 두 가지가 있기 때문이다. 이 두 관점의 차이는 클래스의 캡슐화가 적용되는지의 여부, 즉, 메쏘드의 내부와 전용 부분을 접근할 수 있는지의 차이이다. 예를 들어 데이타 정의 언어나, 데이타베이스 프로그래밍 언어를 사용하여 클래스 'Deposit'을 선언한다면 다음과 같다.

```
class Deposit {
  private:
        money amount;
        time  issue_date;
  public:
        number account_number;
        money show_amount();
        time show_date();
        void put_money(money);
     ....
};
```

메쏘드를 정의하려는 프로그래머들은 위의 선언에서 'account_number', 'show_amount' 등 뿐 아니라, 'amount' 이나 'issue_date'와 같이 'private'으로 명세된 전용 부분들을 접근할 수 있지만, 단지 메쏘드를 선언하거나, 사용하는 사용자들은 이러한 전용 부분을 접근할 수 없다.

따라서 다음과 같은 현상들이 발생한다.

첫째, 메쏘드의 구현과 무관한 사용자들은, 클래스의 전용 부분을 볼 수는 있지만 접근할 수는 없게 된다. 즉, 안전성, 가독성을 근원적으로 저하시키기 때문에 바람직하지 못하다.

둘째, 위의 선언은 구현을 위한 전용 부분을 포함하고 있지만, 대부분의 상용 시스템에서는 메쏘드 구현과 무관한 객체 정의어 사용자에게도, 위와 같이 클래스를 선언해줄 것을 요구하고 있다[5,6]. 이로써, 각 클래스당 시스템 전체적으로 하나의 선언만 관리되면 된다는 잇점이 있게 되지만, 객체 정의어로 된 클래스의 정의 안에 특정 언어와 메쏘드 구현에 관련된 부분이 포함되게 된다. 따라서, 객체 정의어의 장점인 서로 다른 언어로 작성된 여러 응용 프로그램에서 스키마 정보의 공유를 쉽게 해주는 성질을 잃게 된다.

기존의, 서로 다른 여러 언어를 지원하는 객체지향 데이타베이스 시스템 중에서는, 공용 메쏘드의 선언으로만 정의된 클래스를 객체 정의어로 정의하여, 객체 요구 중개 인터페이스인 ORB(object request broker)[7]를 통해 데이타를 접근하는 방식을 부분적으로 지원한 것들이 있었다[8,9,10,11]. ODMG-93[3]에서도 이를 반영하여, 구현이 없는 클래스 선언인 타입(type)과 구현이 들어있는 클래스(class)의 이중 구조를 그 데이타 모델에 포함시켰다. 그러나, ODMG-93에 있는 기술은 정의가 모호해서, 이를 따르는 객체지향 데이타베이스 시스템들 간에도 각기 다르게 해석되고 있다. 심지어는 ODMG-93내의 C++-바인딩이나 Smalltalk-바인딩의 정의에서도 이러한 이중 구조에 대해 전혀 고려하지 못하고 있다. 따라서 ORB를 지원하는 객체지향 시스템들도, 시스템 전체를 이중 구조로 재구성하기 보다는 자신의 언어 인터페이스의 일부에 ORB 사용을 허용하고 있는 정도여서[8,9,10,11], 일관된 객체 모델을 가질 수가 없다.

본 논문에서는 이것이 시스템 전체에 적용될 구성 원칙이 없는데서 기인하였다고 보고, 새로운 데이타 추상화 단계를 통해 전체 데이타 모델을 정리함으로써, 이러한 원칙을 제안한다.

3.2 뷰를 이용한 잠정적인 극복 방식과 한계

앞 절에서 밝혀진, 개념적 레벨내에 두 개의 서로 다른 관점이 존재함으로써 가지는 단점들은, 기존의 세가지 단계의 추상화 중 뷰를 적절히 활용하면 줄어들게 된다.

Copyright (C) 2005 NuriMedia Co., Ltd.

먼저, 클래스의 공용 부분만을 접근하는 일반 사용자들, 즉, 객체 정의어, 객체 질의어 등의 일반 사용자들을 위한 뷰를 마련한다. 대신 개념적 단계에서는 데이타베이스 프로그래밍 언어 사용자들이, 객체 정의어로 정의된 클래스에 전용 부분을 덧붙이거나, 자체적인 정의에 의해 구체적인 클래스 구현을 만들어낸다. 정리하자면, 객체 정의어, 객체 질의어 사용자들은 뷰 단계에서, 클래스의 구현을 담당하는 사용자들은 개념적 단계에서 각각 데이타베이스를 접근하게 된다. 이로써, 객체 정의어 사용자는 클래스를 정의할 때 전용 부분을 명시하지 않으며, 시스템에서 객체 질의어 등의 사용자들에게는 클래스의 전용 부분을 보여주지 않게 되어 앞절에서 지적한 단점들이 어느 정도 7없어지게 된다.

그러나, 이 방법도 역시 몇가지 개선이 필요하다. 첫째, 이와 같은 객체 정의어, 객체 질의어 등의 일반 사용자를 위한 뷰가 매 데이타베이스 시스템마다 반드시 필요하게 된다. 따라서, 이러한 뷰는 아예 시스템에서 제공되는 것이 더 바람직하다.

둘째, 단순히 전용 부분을 감추는 것만으로는, 객체 정의어 사용자가 클래스 구현으로부터 완전히 자유로와 지는 것은 아니다. 왜냐하면, 응용 프로그래밍 언어가 아닌 객체 정의어로 스키마를 설계하는 것일지라도, 클래스 계층 구조를 정의할 때에는 상위 클래스로부터 전용 부분이 계승되는 것을 고려할 수 밖에 없기 때문이다.

셋째, 위의 경우 객체 정의어로 된 한 클래스마다 이를 구현하는 데이타베이스 프로그래밍 언어로 된 클래스가 하나씩 존재하게 되는데, 이것 역시 한계를 가지고 있다. 왜냐하면, 데이타베이스 스키마 클래스 계층 구조는 부분 집합 구조에 기반을 두고 있는데 반해, 객체지향 프로그래밍 언어에서는 클래스 계층 구조가 일반적으로 클래스 정의의 재사용에 기반을 두어 형성되기 때문에, 프로그래밍 언어에서 정한 클래스 계승 관계가 데이타베이스 스키마에서 정한 관계와 달라지는 경우가 발생될 수가 있기 때문이다.

예를 들어, 앞서 기술된 클래스 'Account', 'Deposit', 'Loan'이 데이타베이스 클래스들이고, 이 중 클래스 'Account'가 나머지 두 클래스의 상위 클래스라 하면, 다음과 같이 선언될 수 있다.

```
class Account {
  public:
        number account_number;
        money show_amount();
        time show_date();
        ....
};
class Deposit : public Account{ ... void put_money(); ... };
class Loan : public Account{ ... void borrow_money(); ... };
```

그런데, 클래스 'Deposit'이 데이타베이스 프로그래밍 언어에서 구현되기 위하여, 사칙연산과 환전등의 메쏘드를 가진 기존의 클래스 'moneyManager'의 정의를 재사용하는 경우에는, 앞서 정의된 데이타베이스 스키마의 계층 구조와는 다른 모습으로 다음과 같이 정의될 수 있다.

class Deposit : moneyManager { ... }

이러한 경우 대부분의 객체지향 데이타베이스 시스템에서는, 클래스 'Deposit'은 다중 계승을 통하여 클래스 'Account'와 클래스 'moneyManager'에서 동시에 계승받는 것으로 해결한다.

```
class Deposit : moneyManager, public Account { ... }
```

그런데, 이러한 다중 계승은 두 개의 서로 다른 성질의 계승 관계를 섞기 위한 것으로, 클래스 계층 구조를 기하 급수적으로 복잡하게 한다는 지적이 있어왔다[12,13,14,15,16]. 예를 들어, 클래스 'Deposit'과 'Loan'이 공통적인 상위 클래스 'Account'를 두고 있다 하더라도, 클래스 'Deposit'를 구현하는 방식과 'Loan'를 구현하는 방식은 같지 않을 수도 있다. 즉, 클래스 'Deposit'는 클래스 'moneyManager'에서 계승받아 구현되었지만, 클래스 'Loan'은 독자적으로 또는 다른 클래스에서 계승받아 구현되는 것이 나을 수도 있다. 경우에 따라서는 그림 2와 같이, 'Loan'이 아예 클래스 'Deposit'의 정의를 계승받는 것이 유용한 경우도 있다. 또, 클래스 계층 구조가 복잡하면, 가독성이 떨어지고, 클래스 변경이 어려워진다. 따라서 이것은 객체지향 데이타베이스과 같이 클래스(스키마)에 관련된 연산들의 비용이 큰 경우에는, 심각한 문제가 된다고 볼 수 있다.

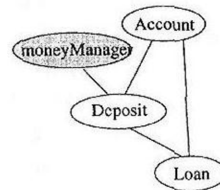

그림 2 스키마와 재사용을 위한 두 개의 독립적인 클래스 계층 구조가 섞인 계층 구조

Copyright (C) 2005 NuriMedia Co., Ltd.

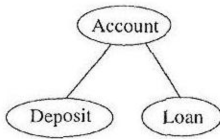

그림 3 데이타베이스 스키마만을 위한 클래스 계층 구조

이상에서 살펴본 바와 같이, 그 클래스를 구현하는 사람들에게만 사용되는 전용 데이타/메쏘드들이 모든 사용자에게 공개됨으로써, 여러 가지 문제점들이 발생한다는 것을 알 수 있다. 이러한 문제점들은 기존의 세단계 데이타 추상화와는 다른, 좀더 체계적인 데이타 추상화를 객체지향 데이타베이스에 도입함으로써 해소될 수 있다. 새로 도입될 추상화 단계는 앞에서 언급된 문제들의 해결 외에도 몇가지 장점을 더 가지고 있는데, 다음 절에서는 이에 대하여 구체적으로 설명된다.

4. 새로운 추상화 단계:클래스-구현부 추상화

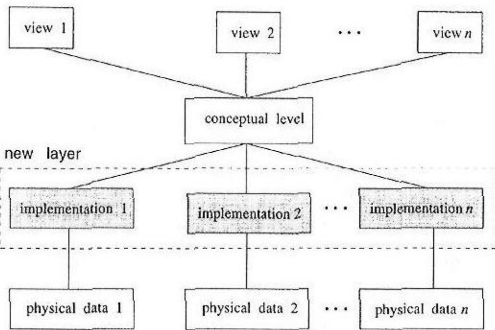

그림 4 제안된 새로운 데이타 추상화 단계

본 논문에서는, 기존의 세단계 데이타 추상화 외에도 객체지향 데이타베이스를 위한 '클래스-구현부 추상화(implementation abstraction) 단계'를 새로 도입한다. 이러한 클래스-구현부 추상화는 클래스 전용 부분의 변화에도 그 클래스를 구현하는 사람들이 아닌 다른 사용자들에게 아무런 영향을 미치지 않는 독립성을 보장해 준다. 그림 4에서 볼 수 있듯이 클래스-구현부 단계는 개념적 단계보다는 시스템에 가까우며, 물리적 단계보다는 사용자에 가깝다.

4.1 클래스의 분리

클래스-구현부 단계가 도입되면, 객체 정의어, 객체 질의어 등을 사용하는 일반 사용자와, 데이타베이스 프로그래밍 언어를 사용하여 해당 스키마 클래스의 메쏘드를 구현하는 사람들은 각각 개념적 단계와 클래스-구현부 단계로 관점이 분리된다.

개념적 단계의 사용자들은 공용(public)으로 선언된 데이타와 메쏘드들을 각자의 접근 권한에 따라 접근하게 된다. 이러한 클래스의 공용 부분을 클래스의 '인터페이스(interface)'라 한다. 즉, 인터페이스는 개념적 단계의 사용자들에게 보여지는 클래스를 나타내므로, 개념적 단계의 사용자가 인터페이스를 접근하는 방식은 기존의 시스템에서 데이타베이스 사용자들이 스키마 클래스를 접근하는 것과 동일하다.

클래스-구현부 단계의 사용자들은, 자신이 구현하는 몇가지 클래스들에 대해서는, 인터페이스 부분 외에 그 클래스를 구현하는 데에만 사용되는 전용(private) 부분에도 접근이 가능하다. 이러한 클래스의 전용 부분이 포함된 클래스 전체는 '구현부(implementation)'라고 불린다. 즉, 구현부란 그 클래스를 구현하는 클래스-구현부 단계의 사용자들에게 보여지는 클래스를 일컬으며, 개념적 단계에서는 접근 될 수 없다.

예를 들면, 앞서 선언된 클래스 'Deposit'은 인터페이스와 구현부를 별도로 하여 다음과 같이 표현될 수 있다. 먼저 인터페이스는 다음과 같다.

```
class Deposit { // 인터페이스
    number account_number;
    money show_amount();
    time show_date();
    void put_money();
  ....
};
```

그리고 이에 따른 구현부는 다음과 같이 나타내어진다.

```
class Deposit_Impl1 { // 구현부
    implements Deposit;
    .... // 인터페이스 Deposit내의 선언과 동일
    money amount;
    time  issue_date;
    ....
};
```

'number', 'money', 'time' 등은 각각 숫자, 금액, 날짜 등을 위해 시스템에서 제공되는 데이타 타입이라고 가정한다. 구현부 'Deposit_Impl1'에 있는 키워드

Copyright (C) 2005 NuriMedia Co., Ltd.

'implements'는 구현부 'Deposit_Impl1'와 인터페이스 'Deposit'을 연결하는 매개이다. 이것은 구현부 'Deposit_Impl1' 가 인터페이스 'Deposit'을 구현한다는 의미이다[2]. 위와 같이 인터페이스에서 선언된 것들이 구현부의 public 부분에 반드시 나타날 필요는 없다. 인터페이스에서 선언되었지만, 구현부의 public 부분에 나타나지 않은 것은 시스템에서 자동적으로 삽입 처리해 준다.

클래스-구현부 추상화에 따라 구현부 'Deposit_Impl1'은 개념적 단계에서는 보이지 않고, 대신 'Deposit'의 선언만이 보이게 된다. 이 때, 'Deposit_Impl1'의 전용 부분을 변경하게 되더라도, 인터페이스 'Deposit'을 접근하는 개념적 단계의 일반 사용자들은 변화 없이 클래스를 접근하게 된다. '클래스-구현부 독립성(*class-implementation independence*)'이란, 클래스-구현부를 변경해도 개념적 단계의 응용 프로그램을 수정할 필요가 없도록 보장함을 뜻한다.

4.2 개념적 단계

앞서 언급한 바와 같이, 데이타 정의 언어를 써서 스키마를 설계하는 것은 개념적 단계의 추상화를 통하여 이루어진다 그리고, 클래스 대신 인터페이스를 접근하는 것 외에는, 기존의 스키마 설계 방식과 달라지지 않는다. 다만, 인터페이스만을 선언하기 때문에 클래스의 전용 데이타/메쏘드들에 대해서는 고려하지 않게 된다는 잇점을 갖는다. 이러한 클래스들을 구현하게 될 구현부는, 개념적 단계가 아닌 클래스-구현부 단계에서 별도의 데이타베이스 프로그래밍 언어로 분리되어 선언되게 되는데, 전용 데이타/메쏘드들도 이 때에 비로소 정의된다

또, 개념적 단계에서의 스키마 계층 구조는 인터페이스만으로 구성되므로, 스키마 설계자들은 클래스의 세세한 구현에 얽매이지 않고 자유롭게 실세계를 모델링할 수가 있다. 예를 들어, 우선 인터페이스 'Deposit', 'Loan', 'Account'의 계층 구조는 구현에 무관하게 다음과 같이 선언된다.

```
class Account {
    number account_number;
    money show_amount();
    time show_date(),
    ...
};
class Deposit : Account {... void put_money(); . };
class Loan : Account {... void borrow_money(), ...};
```

2) 이러한 인터페이스와 구현부의 연결을 정하는 것은 구현부 선언 내에 키워드 '*implements*'를 쓰는 방식 외에도, 별도의 매크로 정의등 여러 가지 구문(syntax)으로 나타내어질 수 있다.

이로써, 데이타 정의 언어 사용자가 미리 구현을 염려하여 클래스의 전용 부분을 고려하지 않아도 되며, 인터페이스 계층 구조가 클래스 구현과 완전히 분리되어 있으므로, 클래스의 전용 부분을 계승 받기 위한 구현부의 재사용 계층 구조는 고려하지 않아도 된다. 그리고, 인터페이스만으로 이루어진 클래스들은, 특정 언어에 의존하지 않는 데이타 정의 언어로 쉽게 표현될 수 있다는 장점이 있다.

서로 다른 언어의 여러 사용자들이 인터페이스들만을 공유함으로써 클래스에 관한 정보를 얻어낼 수 있도록 하기 위해서는, 인터페이스의 선언 자체도, 기본 타입이나 다른 인터페이스들과 같이 사용자들이 공통적으로 알고 있는 것으로만 이루어져야 한다. 즉, 각 사용자가 이들 인터페이스들의 구현부에 대해서는 전혀 알지 못해도 인스턴스들을 접근할 수 있어야 하는데, 이것은 스키마 인터페이스 집합의 '자기-포함성(self-contained-ness property)[17]'을 필요로 한다. '자기 포함성'이란, 인터페이스의 데이타나 메쏘드에 나타나는 타입 명세들(위의 예에서는 'number', 'money', 'date')에, 구현부 이름 같이 여러 응용 프로그램 간에 공유되고 있지 않은 것들이 나타나서는 안된다는 것을 의미한다. 따라서, 인터페이스 선언은 반드시 인터페이스 이름들이거나 기본 타입들로 이루어져야 한다.

객체 질의어 등을 비롯한 일반적인 데이타베이스 사용자들은, 개념적 단계에서 데이타베이스를 접근하므로, 인터페이스만을 스키마로 보게되고, 클래스 계층 구조도 인터페이스의 계층 구조만을 다루게된다. 따라서, 이들 사용자들이 객체를 접근하는 것은 기존의 방식과 달라지지 않지만, 결과적으로는, 클래스 선언을 참조할 때 자신이 접근 불가능한 전용 데이타/메쏘드를 보지 않게 되므로, 가독성, 안전성 면에서 기존에 방식에 비해 유리하다.

4.3 클래스-구현부 단계

구현부를 선언하는 데이타베이스 프로그래밍 언어 사용자들은 클래스-구현부 단계에서 데이타베이스를 접근한다. 따라서, 이들 구현자들은 앞서 소개되었던 인터페이스와 분리된 구현부 정의를 모두 다루게된다. 결과적으로 데이타베이스 프로그래밍 언어에는 몇가지 기능이 추가로 지원되어야하고, 이에 따라 새로운 프로그래밍 방식이 제시된다.

<인터페이스와 구현부의 선언>

Copyright (C) 2005 NuriMedia Co., Ltd.

데이타베이스 프로그래밍 언어를 사용해서도 앞 절의 데이타 정의 언어와 같은 방식으로 인터페이스를 선언할 수 있다. 단, 인터페이스의 선언 앞에 'persistent' 키워드를 붙임으로써 다른 클래스들과 구분한다. 이것은 새로운 구문이 아니라, 기존의 객체지향 시스템의 데이타베이스 프로그래밍 언어들에서, 'persistent' 키워드를 붙여 스키마 클래스를 나타냄으로써 타입 직교성[19]을 보장하던 것을 그대로 이용한 것이다[5,6,18]. 예를 들어 앞서 선언된 클래스 'Account'를 선언한다면 다음과 같다.

```
persistent class Account { ... // 앞의 4.2절에서와
  동일 }
```

구현부는 데이타베이스 응용 프로그램에서만 선언되고 정의된다. 구현부의 선언은 비-데이타베이스의 클래스 선언과 거의 비슷하나, 4.1절에서 기술된 'Deposit_Impl1'의 선언에서 본 바와 같이 'implements' 키워드를 가지고 특정 인터페이스와 '바인딩(binding)'된다는 점이 다르다.

데이타베이스 프로그램에서는 인터페이스를 위한 계층 구조와, 구현부 및 비-데이타베이스 클래스들을 위한 계층 구조를 서로 독립적으로 유지하게 된다. 예를 들어, 인터페이스 'Deposit'를 구현하는 구현부 'Money_Deposit'는 인터페이스 계층 구조의 영향을 받지 않고, 다음과 같이 별도의 계층 구조를 가지고 선언된다.

```
class Money_Deposit : moneyManager
      { implements Deposit; ... };
```

또, 이와 같이, 인터페이스 계층 구조와 클래스 구현 계층 구조가 완전히 분리되어 있기 때문에, 전체 스키마 계층 구조가 간단해진다. 그리고, 한쪽의 변경이 다른 쪽의 변경에 영향을 덜 미치게되므로, 스키마 변경의 복잡성이 줄어들게 된다.

<바인딩>

클래스의 인터페이스와 구현부는 반드시 일대일 사상으로 연결될 필요는 없다. 즉, 한 인터페이스는 여러 가지 형태의 구현부들을 동시에 가질 수 있으므로, 한 스키마 클래스는 이에 따라 여러 형태로 구현된 인스턴스들을 가질 수가 있다. 예를 들어 인터페이스 'Deposit'는 다음과 같이 다양한 구현부들을 가질 수도 있다. 'account_data'는 시스템에서 제공하는, 계정을 위한 레코드 타입이라고 가정한다.

```
// implementations for 'Deposit'
 class Moeny_Deposit : moneyManager{ // 위의 정의와
   동일 };
 class Deposit_Impl1{ // 4.1의 정의와 동일 };
 class Deposit_Impl2 : account_data { implements
   Deposit; .. };
```

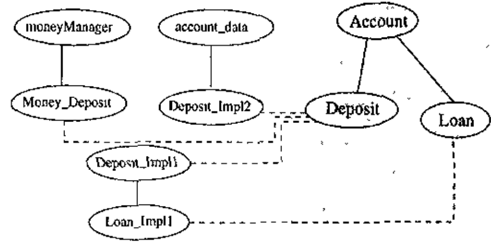

그림 5 인터페이스와 구현부를 분리한 클래스 계층 구조

그런데, 하나의 인터페이스에 여러 구현부가 선언 될 수 있는 성질에 의해, 몇가지 스키마 진화가 간단해질 수 있는 잇점이 있다. 먼저, 만약 'Deposit'의 스키마상 하위 클래스인 'SpecialDeposit'이 존재한다고 한다면, 기존의 계층 구조에서 이러한 하위 클래스의 구현은 세 개의 상위 클래스 즉, 'Money_Deposit', 'Deposit_Impl1', 'Deposit_Impl2'에서 다중 계승 받아야 한다. 이러한 다중 계승은 각 구현부들간의 이름 충돌등을 유발하게 되며[13], 결과적으로 클래스 'SpecialDeposit'의 본래 의도와는 동떨어진 구현을 가지게 될 가능성이 높다. 그러나, 그림 5와 같은 분리된 계층 구조에 'SpecialDeposit'이 추가될 때는, 여러 구현부를 가지는 계층 구조가 스키마와 별도로 존재하므로 'SpecialDeposit'은 'Deposit'에서만 계승받으면 된다. 'SpecialDeposit'의 구현은 'Deposit'의 구현부들을 그대로 이용하거나, 이들과 무관하게 새로 정의할 수도 있다.

또, 경우에 따라서는 클래스 변경을 새로운 구현부를 추가하는 작업으로 대체할 수 있다. 예를 들어, 3절에서 소개된 클래스 'Deposit'의 전용 데이타 'money amount;'를, 성능 향상을 위해 'int amount;'로 바꾸려고 할 때, 기존의 방법으로는 클래스의 선언 자체를 변경함으로써 가능했다. 이렇게 클래스 자체의 내부 구조

Copyright (C) 2005 NuriMedia Co., Ltd.

를 변경하는 것은, 응용 프로그램 뿐아니라 그 클래스의 인스턴스들을 모두 따라 변경시켜야하는 것으로, 스키마 진화 중에서도 비용이 많이 드는 것으로 알려져있다[2]. 클래스 구현부 추상화가 도입된다 하더라도, 실세계에 대한 모델링 자체가 바뀌는 경우에는 스키마 변환이 불가피 하지만, 성능 향상 등을 위해 클래스의 구현 부분이 수정될 때에는, 클래스가 인터페이스와 구현부로 분리되어 있으므로, 구현부를 하나 추가 시키는 것으로 대신할 수 있다. 앞의 'Deposit'의 예에서는 다음과 같은 구현부를 추가한다.

```
// 'Deposit'을 위한 새로운 구현부
class Deposit_Impl3{
    implements Deposit;
    int amount;
    ... // Deposit_Impl1와 동일
};
```

개념적 단계 사용자는 클래스-구현부 추상화에 의하여 이와 같은 구현부의 추가에 전혀 영향을 받지 않게 된다. 또, 새로운 구현부는 특별한 경우가 아니면 앞으로 정의될 객체에 대해서만 적용되므로, 이 작업들은 기존의 인스턴스들에게 영향을 주지 않고 수행될 수가 있다. 결과적으로는 스키마 진화의 비용[2]이 줄어든다.

컴파일러는, 각 구현부의 키워드 'implements' 구를 만나면, 그 구현부의 선언과 바인딩된 인터페이스의 선언간에 타입 안전성(type safety)[17]이 지켜지는지 여부를 검사한다. 검사 결과 타입 공간을 파괴하는 것으로 판명된 바인딩은 컴파일이 거부된다. 따라서, 타입 안전성을 보장하기 위해서는, 인터페이스의 모든 데이타와 메소드가 구현부에도 존재하면서, 이들의 타입 명세가, 대응되는 구현부의 데이타/메소드들의 타입 명세와 같은 경우에만 바인딩을 받아들이는 것도 한가지 방법이다. 그러나, 인터페이스의 자기-포함성때문에 인터페이스의 각 데이타와 메소드의 타입 명세에 인터페이스이거나 기본 타입만을 허용되므로, 이 경우 구현부 선언 내의 타입 명세에도 구현부가 허용되지 않게 되어버린다.

따라서, 타입 안전성이 보장되는, 받아들여지는 (acceptable) 바인딩은 (1) 인터페이스의 모든 데이타나 메소드가 구현부에도 존재하면서, (2) 인터페이스의 각 데이타와 메소드의 타입 명세가, 구현부의 대응되는 타입 명세와 같거나 또 다른 바인딩에 의해 연결되어 있는 경우로 정의할 수 있다. 예를 들어 다음 클래스 선언들에는 몇가지 바인딩들이 나타나 있다.

```
persistent class  Color{ ... };
class 256Color{ ... implements Color; ... };
persistent class ColorImage {
    ...
    Color color();        // 리턴 타입은 Color
};
class 256ColorImage {
    ...
implements ColorImage;
    256Color color(); // 256Color는 Color의 구현부
        };
```

구현부 '256Color'와 '256ColorImage'는 인터페이스 'Color'와 'ColorImage'에 각각 바인딩되어있다. 이 때, 메소드 'color()'의 결과 타입은, 'ColorImage'의 선언에서 'Color'이지만, 구현부 '256ColorImage'에서는 '256Color'이다. 그런데, '256Color'는 'Color'에 이미 바인딩 되어 있으므로, 이 바인딩이 이미 컴파일러에 의해 받아들여졌다면, 구현부 '256ColorImage'와 인터페이스 'ColorImage'의 바인딩도 거부되지 않게 된다.

<재사용성>

인터페이스 'Deposit'의 구현부는 다른 클래스 인터페이스를 위해 재사용될 수도 있다. 다음 예는 구현부 'DepositImpl'가 인터페이스 'Loan'를 구현하는데 사용되는 것을 보여준다. 구현부 'Loan_Impl1'은 구현부 'Deposit_Impl1'에서 계승받아 인터페이스 'Loan'을 구현한다. 결과 클래스 계층 구조는 그림 5에 나와 있다.

```
class Loan_Impl1 : Deposit_Impl1{ implements
    Loan; ... };
```

<프로그래밍 언어와 데이타베이스의 결합>

앞서 언급한 대로 인터페이스 계층 구조와 별도로, 구현부들과 비-데이타베이스들이 어우러져 하나의 계층 구조를 형성하도록 하고 있기 때문에, 보다 정리된 방식으로 데이타베이스 응용 프로그램과 스키마가 결합된다. 궁극적으로는 인터페이스의 계층 구조가 데이타베이스 모델을 따르고, 구현부들과 비-데이타베이스 클래스들은 프로그래밍 언어의 의미론을 따르는 것이 가능하다. 따라서, CLOS[20]로 ODMG-93 모델을 지원할 수도 있고, C++로 O_2[21]와 같은 다중 메소드 데이타 모델[22]의 OODBMS를 지원할 수 있다.

<객체 생성과 사용>

응용 프로그램에서, 데이타베이스 객체는 구현부를 통해 생성되고, 인터페이스 포인터 타입인 객체 핸들러[

Copyright (C) 2005 NuriMedia Co., Ltd.

5,18,23]에 의해 운용된다. 예를 들어 인터페이스 'Deposit'이 구현부 'Deposit_Impl1'과 'Deposit_Impl2' 에의해 구현되었다면, 인스턴스들은 다음과 같이 사용된다.

```
persistent Deposit * x =  new Deposit_Impl1;
  ...
  x =  new Deposit_Impl2;
  x->put_money(1000);
```

따라서, 실제 구현부가 'Deposit_Impl1'인지 'Deposit_Impl2'인지 상관없이, 객체 핸들러 'x'를 통해 인터페이스에서 기술된 모든 데이타를 접근할 수 있고, 메쏘드를 호출할 수 있다. 이와 같은 지정문은, 구현부 'Deposit_Impl1'나 'Deposit_Impl2' 타입의 객체들이, 인터페이스 'Deposit' 타입으로 간주되어 접근되어도 무리가 없는 경우에만 허용되어야한다. 이러한 경우 'Deposit_Impl1'와 'Deposit_Impl2'가 인터페이스 'Deposit'를 '구현(implementing)'하는 관계에 있다고 한다.

개념적 단계에서 자신이 생성하지 않은 객체를 접근할 경우에는 일반 함수 호출 결과나, 질의문, 또는 이름을 통한 찾기에 의해 객체를 데이타베이스에서 가져와야 한다. 예를 들어, 이름 "CHAIRMANSDEPOSIT"을 통하여 이미 생성된 객체를 접근하는 예는 다음과 같다.

```
...
y = obase.lookup("CHAIRMANSDEPOSIT"); // obase는
    데이타베이스를 나타내는 객체
y->put_money(1000);
```

이것은 ODMG-93에서 정의된, 이름으로 객체를 찾는 표준 함수를 사용한 예이다. 이를 위해 클래스-구현부 단계에서 처음 객체를 정의할 때 다음과 같이 해당 객체에 이름이 미리 부여되어 있어야한다.

```
persistent Deposit * x =  new Deposit_Impl1;
obase::mapname(x, "CHAIRMANSDEPOSIT");
 ...
```

결과적으로 클래스-구현부 추상화를 도입함으로써, 인터페이스만을 일반 사용자에게 공개하여 클래스의 전용 부분을 감출 수 있고, 이러한 인터페이스를 특정 언어에 의존하지 않는 데이타 정의 언어로 쉽게 바꿀 수가 있다. 따라서, 여러 다른 언어들로 작성된 응용 프로그램간의 공유성(sharability)이 좋아지게 된다.

또, 별도로 존재하는 두 계층 구조는, 인터페이스와 구현부가 분리되지 않은 경우 다중 계승을 통해 만들어 질 수 있는 계층 구조에 비해 간단하고 의미적으로도 명확하다. 또, 프로그래밍 언어와 데이타베이스의 모델이 다른 경우 나타나는 불일치를 체계적으로 해소할 수 있도록 해준다. 그리고, 한 인터페이스에 대해 여러 개의 구현부가 존재 가능한 성질에 의해서, 클래스의 구현부를 수정함으로써 발생되는 스키마 변환을, 새 구현부를 추가하는 것으로 대치하여 비용을 절감할 수 있다.

5. 클래스-구현부 추상화의 실제 지원

기존의 객체지향 데이타베이스 시스템이 클래스-구현부 추상화를 지원하기 위해 추가되어야할 기능은 몇가지로 요약해볼 수 있다. 첫째, 구현부와 인터페이스가 분리된 객체의 새로운 데이타 구조를 지원하는 것이다. 둘째, 스키마 관리자가 다루는 클래스 정보에 기존의 클래스 자체에 관한 사항들외에도 인터페이스인지 구현부인지 여부를 추가하는 것이다. 데이타 정의 언어로 작성되었거나, 'persistent'가 명기된 클래스들을 인터페이스로, '*implements*'가 명기된 클래스들을 구현부로 스키마 관리자에게 등록한다. 셋째, 구현부의 선언시, 명시된 인터페이스와의 바인딩 관계가 받아들여질 수 있는 것인지를 검사한다. 넷째, 표현식의 타입 검사에서는 인터페이스와 구현부간의 구현 관계를 검사한다. 다섯째, 익스텐트를 구하려면, 그 스키마 인터페이스의 하위 클래스들 외에도 그 인터페이스를 구현하는 모든 구현부들까지 고려하여 인스턴스들의 집합을 구한다.

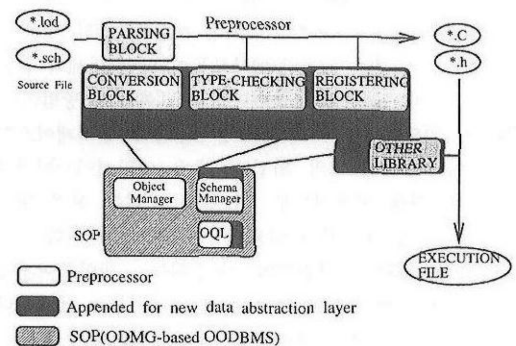

그림 6 전위 처리기 중심의 지원 방식을 위한 구조도

이러한 기능을 가장 간단히 지원하기 위한 방법으로는 그림 6과 같이 데이타베이스 전위처리기를 확장하는 방법이 있다. 즉, 클래스-구현부 단계에서 인터페이스와

Copyright (C) 2005 NuriMedia Co., Ltd.

구현부로 분리된 클래스를 지원하기 위해, 'persistent'와 'implements' 키워드를 전위 처리한다. 그리고, 인터페이스의 포인터들을 ODMG-93 C++ 바인딩의 객체 핸들러인 'Ref' 대신 그의 하위 클래스인 'Interface'의 객체로 변환한다. 예를 들어 앞 절의 객체의 생성과 사용의 예에서 선언되고 초기화된 'persistent Deposit *x = new Deposit_Impl1;'는 다음과 같이 변환된다.

```
Interface<Deposit> x = new Deposit_Impl1;
...
```

객체의 데이타 구조는 기존의 데이타 구조을 그대로 사용하고, 대신, 클래스 'Interface'에서 데이타 모델에 관련된 작업을 전담한다. 예를 들어 인터페이스 'Deposit'의 포인터 'x'는 'Interface<Deposit>'와 같이 변환되는데, 'Ref' 와는 달리 다른 타입 인자의 'Interface' 클래스인 'Interface<Account>'와 서브 타입 관계가 성립한다. 이러한 서브 타입 관계는 전위 처리기에서 인위적으로 타입 검사를 하여 정의하는 것이다. 모든 구현부 클래스들은 'PObject'의 하위 클래스인 'ImplObject'에서 계승받는다. 구현부 클래스의 정의시 인터페이스와의 바인딩에 관한 타입 정확성 검사나, 일반 표현식의 타입 검사도 전위 처리기에서 담당한다[3].

클래스-구현부 단계의 사용자는 데이타베이스 언어로 작성한 프로그램을 전위처리하거나 별도의 도구를 이용함으로써 클래스를 데이타베이스에 등록하게 된다. 개념적 단계의 사용자는 데이타 정의 언어로 인터페이스를 선언한 후 온라인 도구나 각 인터페이스에 대응되는 헤더화일을 통해 데이타베이스에 등록하거나, 인터페이스 포인터를 통해 객체를 접근하게 된다. 스키마 관리자에서는 등록되는 클래스가 인터페이스인지 구현부인지를 판별하는 정보를 추가로 관리하게 되며, 데이타 정의 언어로 작성되는 모든 클래스는 인터페이스로 간주하여 등록된다. 각 클래스마다 추가된 정보는 타입 판별외에 익스텐트 계산등에 유용하게 쓰인다.

일례로 클래스-구현부 추상화 단계는 이러한 방식을 따라 ODMG-93을 따르는 OODBMS인 'SOP(SNU OO-DBMS Platform)[25]'[4] 위에 적용중이며, 현재는 데이타베이스 프로그래밍 언어 전위처리기 부분이 거의 완성된 상태이다. 앞서 소개된 예제들은 이에 따른 구문을 사용하였다.

6. 관련 연구 및 토의

클래스-구현부 추상화를 도입하면, 기존의 추상화 단계만으로 데이타베이스를 접근할 수 있는 것에 비해 여러 가지 장점을 가지는 것은 앞 절들에서 언급되었다.

클래스-구현부 추상화를 위해 기본적으로 지원되어야 하는 클래스의 인터페이스와 구현부를 분리하는 것은 프로그래밍 언어 분야에서는 새로운 것이 아니다. 특히 비교적 널리 사용되는 Java[26]나 Objective-C[27]에도 인터페이스와 구현부가 분리되는 것이 지원되고 있다. 그밖에도 Cecil, Modula-3, Emerald, Theta[28,29,30,32,15]와 같은 언어에서 인터페이스와 구현부의 분리를 지원해왔다. 따라서, 이들 언어의 프로그래밍에 익숙한 사용자들에게는 클래스-구현부 단계의 데이타베이스 프로그래밍이 더욱 친숙할 것이다.

그런데, ODMG2.0의 Java 바인딩 초안에서는 Java의 'Interface' 부분이 고려되지 않았다[33]. 또, Java에서 관계형 데이타베이스를 호출하는 인터페이스인 JDBC[34]도 인터페이스와 구현부의 분리와는 전혀 관계가 없다. Theta[30,32]를 데이타베이스 언어로 사용하는 분산 객체지향 데이타베이스 시스템 Thor[31]는 인터페이스와 구현부 분리를 이용하기는 하지만, 데이타 모델이 Theta 자체에 완전히 의존하고, 분산 환경에서 객체의 공유나 캐쉬등에 주안점을 두고 있기 때문에, 클래스-구현부 추상화에 관해서는 별 다른 언급이 없다.

또, 기존의 프로그래밍 언어 인터페이스와 구현부의 분리 기법을 그대로 데이타베이스 프로그램에 적용할 수 없다. Cecil[29]은 다중 메쏘드(multi method)에 기반을 두고 있기 때문에, Cecil의 구현 분리 개념으로는 ODMG-93을 포함한 대부분의 보편적인 객체지향 데이타베이스 모델에서는 타입 안전성을 보장할 수 없다[21,31]. Modula-3[28]이나 Emerald[15]는 한 인터페이스당 한 구현부만을 허용하며, 구현부는 독립적인 타입 검사의 단위가 되는 것이 아니므로, 구현부만의 계층 구조도 존재하지 않게 된다. 대부분의 경우, 인터페이스가 스키마와 같은 모델링 개념보다는 클래스에 내재되어 있는 타입을 의미하기 때문에, 데이타베이스에 응용하기에는 적절하지 않다. 따라서, 클래스-구현부 추상화에서 요구되는 인터페이스-구현부의 분리의 의미가 이러한 분리를 이미 제공하는 기존의 몇몇 프로그래밍 언어들의 그것과 충돌없이 결합될 수 있도록 하는 것에 대한 보다 많은 연구가 필요하다.

새로운 추상화 단계에서의 스키마 변환에 따른 장점

3) 이 방식은 두 클래스 계층 구조를 단순 병합[14]하지 않기때문에 불합리성(anomaly)을 갖지 않게 된다[24].

4) 1992년에서 1996년까지 서울대학교에서 개발된 ODMG-93 기반 OODBMS 이다.

Copyright (C) 2005 NuriMedia Co., Ltd.

은, 마치 역사(history)를 사용한 스키마 변환에서 얻는 이득과 비슷하다[35]. 하지만 이러한 스키마 변환은 변경 직전의 클래스와의 연결 리스트로 스키마 변환의 역사(history)가 저장되는 형식이기 때문에, 재사용 관계가 선형적이다[35]. 본 논문에서는 구현부간의 계층 구조가 허용되므로 선형적인 구조보다 더 발전된 형태라고 볼 수 있다.

클래스-구현부 단계를 지원하는데 드는 오버헤드는 전위 처리시의 추가적인 타입 검사, 스키마 관리자의 추가 정보 관리, 익스텐트의 확장, 응용 프로그램들의 복잡성, 데이타베이스 모델에 부합하기 위한 인터페이스들의 동작 등을 들 수 있다. 첫째, 전위 처리시 타입 검사는 전위처리기에서 특정 객체지향 데이타베이스 모델을 지원하기 위해서 기존의 'Ref'위에서 이루어졌던 것을 'Interface'에 적용한 것이다. 이것은 표현식 타입 검사 등을 포함하므로 컴파일 동안의 오버헤드가 가중되는 것은 사실이나, 수행 시간 오버헤드는 없다. 둘째, 스키마 관리자 정보에 인터페이스인지 여부를 추가하는 것과 익스텐트 확장에 따른 작업에 따른 오버헤드의 증가는 클래스 계층 구조의 성격에 따라 좌우된다. 즉, 인터페이스와 구현부 분리의 잇점이 가장 적은 경우는, 구현부의 계층 구조가 분리되기전 계층 구조와 동일한 경우이므로, 익스텐트 계산의 추가 오버헤드는 이런 최악의 경우 인터페이스들이 이루는 계층 구조의 복잡도에 비례하게 된다. 이러한 오버헤드는 클래스 계층 구조 인덱스[2] 등 클래스 계층 구조를 효과적으로 탐색하기 위한 방법들을 그대로 적용하여 줄일 수 있다. 셋째, 응용 프로그램의 복잡성은, 인터페이스와 구현부 분리 이전에 이미, 한 개의 계층 구조에 여러 계승 관계가 뒤섞인 형태로 존재해 왔기 때문에 새로운 것은 아니다. 다만, 인터페이스와 구현부 간의 연결 고리에 대해 새롭게 정의되었기 때문에, 메쏘드 구현자들에게 부담이 될 수가 있다. 그러나, 앞서 밝혔던 Java[26]를 비롯한, 인터페이스와 구현부 분리를 지원해온 여러 프로그래밍 언어가 이미 등장해왔으므로, 이에 대한 친숙도가 큰 문제가 되리라고는 보지 않는다. 넷째, 데이타 모델을 반영하는 인터페이스 자체의 동작, 즉 클래스 'Interface'의 동작은 수행 속도에 대해서는 면밀히 고려되어야 할 것이다. 이것은, 일반적인 메쏘드 추출(method dispatch) 기법[36]의 효율성에 관한 문제와 거의 같은 것으로 다루어질 수 있으며, 연구 과제로 남는다.

7. 결 론

본 논문에서는 객체지향 데이타베이스의 클래스가 가지는 성질을 감안한 새로운 데이타 추상화 단계인 '클래스-구현부 단계(class-implementation level)'를 도입할 것을 제안하였다. 기존에 개념적 단계에서 접근 가능했던 하나의 스키마 클래스는, 개념적 단계에서 접근 가능한 부분과 클래스-구현부 단계에서만 접근 가능한 부분으로 분리되어 추상화된다.

이러한 새로운 단계의 추상화의 도입이 기존의 세 단계 추상화 방식에 비해 어떠한 잇점이 있는지 살펴보았고, 클래스-구현부 추상화가 도입됨으로써 일어나는 데이타 모델과 데이타베이스 프로그래밍 방식의 변화에 대해 기술하였다. 또, 데이타베이스 관리 시스템들의 스키마 관리자나 데이타베이스 프로그래밍 언어 전위처리기등에서 추가로 지원되어야하는 기능들에 관해 논하고, 시스템 구현의 예를 제시하였다.

현재 기존의 프로그래밍 언어들에서 제공되는 인터페이스-구현부의 분리 의미를 클래스-구현부 추상화에서 요구되는 형태로 사상시키는 것과, 클래스-구현부 추상화의 보다 효율적인 지원 방법에 대해 연구 중에 있다.

참 고 문 헌

[1] Henry F. Korth and Abraham Silberschatz Database System Concepts. McGraw-Hill, Inc, second edition, 1991.

[2] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[3] R. G. G. Cattell. *Object Database Standard : ODMG-93 release 1.2*. OMG group, 1996.

[4] O_2 Technology. *C++ Interface to O_2*, March 1994.

[5] Thomas Atwood. "Two Approaches to Adding Persistence to C++". In *The Fourth International Workshop on Persistent Object Systems*, pages 369--383, 1991.

[6] Objectivity Inc. *Objectivity/DB : Getting Started with C++*, March 1994.

[7] IONA Technologies Ltd. *Orbix :distributed object technology -Programmer's Guide-(release 1.3)*, July 1995

[8] IONA Technologies Ltd. *Orbix+ObjectStore Adapter*, April 1996.

[9] F. Reverbel. "*Persistent in Distributed Object Systems: ORB/ODBMS Integration*" PhD thesis, University of New Mexico, 1996.

[10] E. Kilie and et al. "Experiences in Using CORBA for a Multidatabase Implementation". In *Proc. of Database and Expert System Applications(DEXA)*, London, 1995.

[11] Ron Ben-Natan. *CORBA : A Guide to the Common Object Request Broker architecture*. McGraw-Hill,

Copyright (C) 2005 NuriMedia Co., Ltd.

1995.

[12] G. Baumgartner and V. F. Russo. "Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism". Technical Report CSD-TR-93-059, Purdue University, September 1993.

[13] William R. Cook. "Inheritance Is Not Subtyping". In *Proc. of SIGPLAN Conf. on Principle of Programming Languages*, pages 125--135, 1990.

[14] Bruce Martin. "The Separation of Interface and Implementation in C++". In *Proceeding of Usenix C++ conference*, pages 51--63, 1991.

[15] R. K. Raj and et al. "The Emerald Approach to programming". Technical Report 88-11-01, University of Washington, November 1989.

[16] Craig Schaffert and et al. "An Introduction to Trellis/Owl". In *Proc. of the ACM OOPSLA Conf.*, pages 9--16, September 1986.

[17] John B. Fraleigh. *A First Course in Abstract Algebra - Third edition.* Addison-Wesley Publishing Company, Inc., 1982.

[18] R. Agrawal and N. H. Gehani. "Rationale for the Design of Persistency and Query Processing Facilities in the Database Programming Language O++". In *2nd Int'l Workshop on Database Programming Languages*, Portland OR, June 1989.

[19] Malcolm P. Atkinson and O. Petter Buneman. "Types and Persistence in Database Programming Languages". *ACM Computing Surveys*, 19(2):105--190, June 1987.

[20] Sobya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, Inc. , 1989.

[21] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. "*Object-Oriented Database System - The Story of O_2*". Morgan Kanfmann Publishers, Inc., 1991.

[22] Giuseppe Castagna. "Covariance and Contravariance : Conflict without a Cause". *ACM Transactions on Programming Languages and Systems}*, 17(3): 431--447, 1995.

[23] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore Database System". *Communications of the ACM*, 34(10), October 1991.

[24] E. S Cho. "A Semantics of the Separation of Interface and Implementation in C++". In *Proc. of Int'l conference on COMPSAC*, 1996.

[25] J. H Ahn, K. W. Lee, H. J. Song, and H. J. Kim. "SOPRANO : Design and Implementation of Object Storage System". *Journal of The Korea Information Science Society*, 23(9):243--255, September 1996.

[26] Ken Arnold and James Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[27] Brad J. Cox and Andrew J. Novobilski, editors. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley Publishing Company, Inc., second edition, 1991.

[28] Luca Cardelli, James Donahue, and Lucille Glassman. "Modula-3 Language Definition". *ACM SIGPLAN Notice*, 8(29):15--42, August 1992.

[29] Craig Chambers. "The Cecil Language Specification and Rationale(Version2.0)". Technical report, University of Washington, December 1995.

[30] Babara Liskov. "Specifications and Their Use in Defining Subtypes". In *Proc. of the ACM OOPSLA Conf.*, October 1993.

[31] B. Liskov and et al. "Safe and Efficient Sharing of Persistent Objects in Thor". In *Proc of the ACM SIGMOD Conf. on Management of Data*, June 1996.

[32] A. C. Myers. "Bidirectional Object Layout for Separate Compilation". In *Proc. of the ACM OOPSLA Conf.*, 1995.

[33] OMG Group. *ODMG 2.0 draft*, December 1996.

[34] Brian Jepson. *Java^{TM} Database Programming.* John Wiley & Sons, Inc., 1996.

[35] F. Ferrandina and G. Ferran. "Schema and Database Evolution in the O_2 Object Database System". In *Proc. of the Conf. on VLDB*, 1995.

[36] Pascal Andre and Jean-Claude Royer. "Optimizing Method Search with Lookup Caches and Incremental Coloring". In *Proc. of the ACM OOPSLA Conf.*, pages 110--126, 1992.

조 은 선
제 24 권 제 2 호(B) 참조

한 상 영
제 24 권 제 2 호(B) 참조.

김 형 주
제 24 권 제 1 호(B) 참조

Copyright (C) 2005 NuriMedia Co., Ltd.