C++-based OODBPL에서 클래스의 분리를 지원하기 위한 구현 기법
(Techiniques to Implement Class-Separation on C++-based OODBPL)

조 은 선 † 김 형 주 ††
(Eun-Sun Cho) (Hyoung-Joo Kim)

요 약 하나의 클래스를 인터페이스 (interface)와 구현부 (implementation)로 나누어 다루는 '클래스-분리'에 대한 연구는 최근 프로그래밍 언어 분야 또는 분산 처리 분야를 비롯하여 데이타베이스 분야에서도 그 유용성이 입증되고 있다. 그러나, 데이타베이스를 위한 클래스-분리에 관한 문헌들은 아직까지는 추상적이고 간단한 설명만을 제시하고 있기 때문에, 실제 DBMS (Database Management System)를 구현할 때 참조할 만한 자료가 없는 실정이다. 본 논문은 ODMG-93 기반 C++ 데이타베이스 프로그래밍 언어를 위해 클래스-분리를 지원 하는 구체적인 방안을 제시하고, 기존의 타분야에서 사용된 기타 방법들과 비교한다.

Abstract The importance of class-separation (which separates interface and implementation of a class) is well recognized in many areas such as programming languages, distributed computing, and database systems. However, when it comes to the implementation of class-separation for object-oriented persistent languages in DBMSs, there is not much reference material. This paper proposes an implementation framework for class-separation for an ODMG 93-based C++ database language, and compares existing implementation works in other areas than databases.

1. 개 요

클래스를 인터페이스 (interface)와 구현부 (implementation)로 나누어 다루는 것 (이하 '클래스-분리')에 대한 연구는 프로그래밍 언어 분야 또는 분산 처리 분야에서 그간 적지 않게 이루어져 왔다[1,2,3].

데이타베이스 분야에서도 클래스의 분리가 유용한 개념으로 등장하기 시작하였는데[4,5,6], 이것은 클래스의 구현부를 그 클래스를 실제 구현하는 프로그래머들에게만 공개하고, 질의어 사용자 및 스키마 설계자 (schema designer)를 비롯한 나머지 사용자들에게는 인터페이스만을 볼 수 있도록 하는 것이다. 궁극적으로는 데이타베이스 클래스의 각 부분 (인터페이스와 구현부)의 역할에 따른 분리를 통해, 클래스의 구현이 변경되어도 데이타베이스 응용 프로그램들이나 기타 사용자들에게 가능한 한 영향을 미치지 않도록 하는 데에 그 목적이 있다. 이로써 뷰단계(view level)-개념적 단계(conceptual level)-물리적 단계(physical level)로 나뉘어 온 데이타베이스 추상화 단계들이 클래스-분리로 인해 뷰단계(view level)-개념적 단계(conceptual level)-클래스-구현부 단계(class-implementation level)-물리적 단계(physical level)로 세분화되고, 개념적 단계에서는 인터페이스만을 접할 수 있도록 하게 된다[7].

현재 객체지향 데이타베이스의 표준안으로 제시되어 주목을 받고 있는 ODMG-93 모델에서도 이러한 필요성을 인식하여 객체 모델 단계에서 클래스-분리를 고려하고 있다[4]. 그러나, ODMG-93에서는 클래스-분리에

· 본 논문은 1998년도 과학기술처 특정연구(핵심S/W)기술개발 사업에서 지원한 '객체지향 기술을 이용한 인터넷 상의 트랜잭션 처리 기술 개발' 중 서울대학교 공학연구소에서 수행한 '웹트랜잭션 서버를 위한 객체지향 컴포넌트 기술개발'의 일부로 연구되었음
† 학생회원 : 한국과학기술원 프로그램분석시스템연구단 연구원
cschough@compiler.kaist.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학과 교수
hjk@oopsla.snu.ac.kr
논문접수 : 1997년 10월 31일
심사완료 : 1999년 3월 5일

Copyright (C) 2005 NuriMedia Co., Ltd.

대한 추상적이고 간단한 설명만을 제시하고 있기 때문에, 이를 참조하여 실제 데이타베이스 관리 시스템을 구현하기는 아직은 사실상 불가능하다. 게다가, ODMG C++ 바인딩 (binding)이나 ODMG Smalltalk 바인딩 등, 객체 모델 이외의 ODMG 기술에서는 클래스-분리에 대한 사항을 완전히 배제하고 있음을 알 수 있다. 이는 ODMG가 기존의 여러 시스템들의 장점을 단순한 '모으기' 식으로 결합한 데에 그 원인이 있는데, ODMG 객체 모델과는 달리 ODMG C++ 바인딩 등은 기존의 ObjectStore[8], Objectivity[9] 등의 장점을 취합해 구성한 것이기 때문이다. 물론 이러한 기존의 C++ 바인딩들은 클래스-분리에 대한 필요성을 아직 인식하지 못한 단계이다.

SQL3에서도 클래스-분리 개념과 비슷한 개념이 ADT (abstract data type)와 테이블 (table)의 관계로 나타난다[6]. ADT는 사용자에게 명시되는 각 데이타의 사용 방법이라 할 수 있고, 테이블은 구체적인 구현을 의미하므로, 각각 인터페이스와 구현부에 해당되게 된다. 이로써 사용자에게 하나의 단계의 추상적인 관점을 새로 제공하여, 클래스 구현의 변동이 여타 데이타베이스 응용 프로그램에 가능한 영향을 미치지 않도록 한다. 그러나, SQL3의 경우는 자체 언어 SQL 외의 C++ 등의 프로그래밍 언어들을 통한 데이타베이스 프로그래밍이 전혀 고려되지 않았기 때문에, 기존의 언어들에 익숙한 프로그래머들을 위주로 하는 경우를 위해서는 그 적용 방식에 대한 연구가 먼저 선행되어야 하는 상황이다.

본 논문은 ODMG-93 C++바인딩을 근간으로 하여 클래스-분리를 지원하는 데이타베이스 언어인 'LOD*[1]'를 소개하고, 클래스-분리를 추가 확장하는 구체적인 방식을 제안, 그 구현에 대한 경험을 공유하고자 하는 데에 그 목적이 있다.

먼저 2장에서는 LOD*에 대한 전반적인 소개를 한다. 3장에서는 LOD*에서 클래스-분리의 지원을 위한 구현 방식에 대해 구체적으로 살펴본다. 4장에서는 이러한 구현 방식에 대해 토의하고, 기존의 프로그래밍 언어나 분산 환경 등의 타 분야에서 도입된 클래스-분리 방식들과 비교 후 차이점을 분석한다. 5장에서는 결론을 맺는다

2. LOD*의 소개

2.1 기본적인 성격과 기능

여기서는 LOD*가 데이타베이스 프로그래밍 언어로써 가지는, 클래스-분리를 제외한 성격이나 기능들에 관해 간략하게 살펴본다. LOD*는 기존의 사용자들에게 친숙하도록 C++ 자체나, 기존의 C++기반 데이타베이스 언어들에서 그 의미나 구문이 크게 벗어나지 않도록 정의되었다.

LOD*에서 객체의 지속성 (persistence) 여부는 객체의 생성 방법에 의해 결정된다. 예를 들어 클래스 A의 지속적 포인터와 그 객체를 생성하는 명령은 다음과 같다.

```
persistent A * p = new (obase) A;
```

여기서, p와 같은 포인터는 지속적 객체를 다루는 핸들러이며[10,11], 'obase'는 객체베이스 (데이타베이스)를 나타내는 객체이다. 이와 같은 방식으로 지속적 객체를 생성하는 인터페이스를 제공하는 객체지향 데이타베이스는 이외에도 ODE[10], ObjectStore[8,11], Objectivity[9] 등이 있다.

또, 데이타베이스 프로그램시에는 질의어와 비슷하게 데이타 추출을 하기 위해, 특정 집합의 각 원소들에 대한 조건 검사를 자주 수행하게 되는데, LOD*에서는 이러한 연산용으로 특수한 반복 연산자 (iterator)[2]인 'forall'구문을 지원하여 간단하게 프로그래밍 할 수 있도록 돕는다.

기타, 클래스 Set<T>, Bag<T>, List<T>, Varray<T> 등 ODMG C++ 바인딩에서 제공되는 여러 클래스들이 그대로 사용될 수 있다.

2.2 클래스 정의와 클래스-분리

C++에 기반한 데이타베이스 프로그래밍 언어에 클래스-분리를 도입하기 위해서는, 객체지향 데이타베이스 모델과 C++, 그리고 클래스-분리 개념과의 결합이, 가능한 한 일관되고 매끄럽게 연결되도록 하여, 사용자에게 불편을 주지 않도록 하는 것이 좋다. 이를 위해서는 ODMG C++ 바인딩과 같은 클래스 라이브러리 바인딩 (class library binding)보다는 전위 처리기 (preprocessor)를 통한 C++ 포인터 인터페이스[8,9,10,11]가 더 적합함을 알 수 있다[11]. 따라서, LOD*도 전위 처리를 통해 클래스-분리와 데이타베이스 프로그래밍을 지원하도록 하였다.

일반적으로 전위 처리를 하는 C++ 데이타베이스 프로그래밍 언어에서 스키마 클래스를 정의할 때는, 키워드 'class' 대신 O++[10]에서와 같이 'persistent class'

1) [lo:dsta:r] : 서울대학교에서 개발된 객체지향 데이타베이스 관리 시스템인 'SOP (SNU OODBMS Platform)'에서 지원되는 C++기반의 데이타베이스 언어

2) 특정 문장들을 반복 수행할 것을 의미하는, C나 C++의 for나 while 등과 같은 구문이다.

Copyright (C) 2005 NuriMedia Co., Ltd.

라고 하거나, E[12]에서 처럼 'dbclass'등을 사용하여, 일반 클래스들과 구분하게 된다[3]. 다음은 스키마 클래스 'Deposit'가 'persistent' 키워드를 가지고 정의된 예이다[4].

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

그런데, 클래스-분리를 위해서는 '클래스 (class)' 외에도 '인터페이스 (interface)'와 '구현부 (implementation)'라 불리는 두 개의 객체 생성자 (object constructor)에 대한 개념이 더 필요하다. '인터페이스'는 클래스 선언에서 공용 (public) 멤버, 공용 멤버 함수 선언만을 뽑아 이루어지는 객체 생성자이고, '구현부'는 이와 달리 전용(private) 멤버나 멤버 함수 구현을 모두 포괄하는 객체 생성자이다. 스키마 클래스가 인터페이스 I와 구현부 M으로 나뉘어진 경우, M이 I를 '구현'한다고 하고, M에 의해 생성된 객체는 I에도 속하는 것으로 간주된다.

LOD*에서의 스키마 클래스는 앞서 설명된 C++ 기반 데이타베이스 프로그램에서와 비슷하게 키워드 'persistent'로 구별된다. 하지만, 이러한 클래스들이 인터페이스만을 나타낸다는 점이 기존의 경우와 다르다. 예를 들어, LOD*에서 스키마 클래스 'Deposit'은 공용 부분만으로 다음과 같이 정의되게 된다.

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

'Deposit'은 다른 스키마 클래스의 하위 타입 (subtype)으로 정의할 수도 있다.

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

이러한 스키마 인터페이스는 C++ 바인딩등 일반 API (Application Programming Interface)외에도 ODL (Object Definition Language)[4]나 다른 스키마 입력 도구로도 정의될 수 있다.

인터페이스와 달리, 구현부는 API로만 정의된다. 구현부는 공용, 전용 부분이 모두 정의된다는 점에서 일반 클래스와 똑같지만, 'implements'라는 키워드의 추가로 자신의 인터페이스와 연결된다. 다음의 구현부 'Deposit_Impl1'은 인터페이스 Deposit를 구현하는 구현부이다.

```
class Deposit_Impl1 { // implementation
 implements Deposit;
 .... // same as declaration in Deposit
      (can be omitted)
 money amount; time  issue_date;
 .... // method implementation ... };
```

인터페이스에 정의된 데이타/메쏘드는 구현부에서 다시 정의될 수도 있고, 생략될 수도 있는데, 다시 정의되는 경우 재정의된 것이 이전의 정의와 타입 등이 잘 맞는지에 대해 '적합성 (acceptability)'이 검사되게 된다. 만일 잘 맞는다면, 이것은 '적합한 바인딩 (acceptable binding)'이라 불린다[13].

인터페이스와 구현부는 각기 독립적인 계층 구조를 가진다. 따라서, 인터페이스의 계층 구조는 스키마 모델링과 관련된 것인 반면, 구현부가 이루는 계층 구조는 이러한 스키마와 상관이 없이 설정 된다. 예를 들어, 구현부 'Money_Deposit'이 인터페이스 'Deposit'을 구현하고 있는 경우에도 'Deposit'이나 'Account'로 이루어지는 계층 구조와는 무관한 계층 구조에 위치하게 된다. 다음 예에서 클래스 'moneyManager'는 'money'타입 자료를 다루기 위해 기존에 만들어 놓았던 클래스라 가정한다.

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

따라서, 이론적으로 한 프로그램 내에는 클래스 계층 구조가 인터페이스들을 위한 것과 구현부들을 위한 것, 그리고 클래스-분리 없이 정의 된 일반 C++ 클래스들을 위한 것의 세 가지가 공존하게 된다. 그런데, 구현부

3) 단, Objectivity[9]의 '.ddl'확장자 화일이나, ObjectStore[8]의 '.H' 확장자 파일 등과 같이 별도의 특수한 헤더 (header) 화일을 써서 데이타베이스 클래스를 나타내게 될 경우에는 class 키워드를 그대로 사용해도 혼돈이 없다. LOD*에서도 별도의 특수한 헤더 화일인 '.sch' 확장자 화일에서 스키마 클래스를 정의하는 것을 지원한다.

4) number, money, time 등은 미리 정의된 타입으로 가정한다.

Copyright (C) 2005 NuriMedia Co., Ltd.

들과 일반 C++ 클래스들은, 데이타베이스 스키마와 비교적 무관하고 구현에 관계되어 있는 등 그 성격이 비슷하므로, LOD*에서는 이들을 합하여 한 개의 계층 구조를 형성하도록 하였다. 따라서, 사용자는 한 가지가 줄어든 두 가지의 독립된 계층구조를 인지하게 되며, 이는 Java[1]나 Objective-C[14], CORBA[3] 등의 기존의 클래스-분리 언어 사용자에게 익숙한 형태가 된다는 장점이 있다.

한 인터페이스는 동시에 두 개 이상의 구현부를 가질 수도 있다. 예를 들어 'account_data'가 계정을 위한 시스템 제공 타입이라고 가정하면, 인터페이스 'Deposit'은 동시에 다음과 같이 여러 구현부를 가질 수 있다.

```
class Money_Deposit :
 moneyManager{ // same as above definition }
class Deposit_Impl1{ // same as above
definition };
class Deposit_Impl2 :
 account_data { implements Deposit; .. };
```

클래스-분리가 도입되었을 때에의 지속적 객체의 사용은 앞 절에 기술한 지속적 객체 핸들러 방식을 기초로 한다. 단, 데이타베이스 객체는 구현부를 통해 생성되고, 인터페이스 포인터 타입인 객체 핸들러에 의해 운용된다.

예를 들어 인터페이스 'Deposit'이 구현부 'Deposit_Impl1'과 'Deposit_Impl2'에 의해 구현되었다면, 인스턴스들은 다음과 같이 사용된다.

```
persistent Deposit * x =
 new(obase)Deposit_Impl1;
...
x = new(obase) 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 있는, 새로운 차원의 다형성 (polymorphism)을 제공하게 된다는 장점을 가진다. 이와 같은 지정문은, 구현부 'Deposit_Impl1'나 'Deposit_Impl2' 타입의 객체들이, 인터페이스 'Deposit' 타입으로 간주되어 접근되어도 무리가 없는 경우에만 허용된다. 이렇게 한 프로그램 내에서 클래스-분리의 인지와 구현부 선택을 허용하는 것은, 현재 ODMG 모델에서는 배제되고 있으나, CORBA나 Java 등에서는 이미 제공되고 있는 상태이다[1,3,4].

3. 구 현

앞서 설명되었듯이 본 절에서는 LOD*에서 클래스-분리를 지원한 방식을 구체적으로 소개한다. LOD*의 주요 특징 중 하나는, 전위처리(preprocessing)된 결과가 ODMG C++ 바인딩을 따르는 코드가 된다는 점이다. 이로써, LOD* 전위처리기만 있다면 하나의 LOD* 코드가 ODMG C++ 바인딩을 제공하는 여러 객체지향 데이타베이스에서 그대로 사용될 수 있기 때문에, ODMG가 표준으로서 가지는 이식성 (portability) 등의 잇점을 그대로 살릴 수 있다.

3.1 인터페이스, 구현부 및 객체의 구성

LOD*의 인터페이스나 구현부들은 전위처리 후에 각각 하나의 ODMG C++ 바인딩 상의 특수한 클래스들로 변형된다. 먼저, 구현부들은 클래스 'ImplObject'의 하위 클래스 (subclass) 정의로 변환되는데, 이것은 ODMG C++ 바인딩의 일반적인 데이타베이스 클래스가 루트 클래스 PObject의 하위 클래스로 정의되는 것과 비슷하다. ImplObject는 인터페이스와 구현부로 이루어지는 다형성을 동적으로 지원하기 위해, 멤버 함수 포인터 테이블로의 포인터 my_class_tbl를 가지고 있다.

```
struct ImplObject :public PObject{
 FptrtblClass * my_class_tbl;
 ImplObject(char * class_name){
  my_class_tbl = &ftbl[ class_name ]; ... };
```

처음 ImplObject의 생성시에 자신의 구현부 이름을 가지고 적절한 my_class_tbl을 찾아낸다. 'ftbl'은 전역변수로서 모든 구현부의 멤버 함수 테이블들을 원소로 하는 테이블이며, 구현부 이름을 가지고 멤버 함수 테이블을 탐색하기 위해 존재한다.

인터페이스들은 다음과 같은 클래스 InterObject의 하위 클래스로 정의된다. InterObject는 실제 구현부 객체 핸들러인 Ref<ImplObject>의 하위 클래스로 정의된다.

```
struct InterObject : public Ref<ImplObject> {
 InterObject(){..}
 InterObject(Ref<ImplObject> t) :
  Ref<ImplObject>(t){ ... };
```

인터페이스의 각 멤버를 위해 'get_'으로 시작하는 이름의 함수와 'set_'으로 시작하는 이름의 함수가 정의된다. 이를 위해 전위 처리기는 get_/set_ 함수 이름 앞에 해당 구현부 클래스 이름을 붙인 전역 함수들을 준비하게 되며, get_/set_ 함수 각각은 내부에서 이 전역 함수들을 호출하게 된다. 하지만, 사용자는 get_/set_ 함수를 직접 호출하는 것이 아니라, 일반 멤버를 접근하는

Copyright (C) 2005 NuriMedia Co., Ltd.

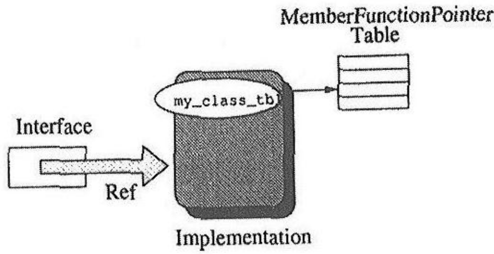

그림 1 클래스-분리를 따르는 객체 구성

것과 다름없이 사용할 수 있도록 전위 처리된다. 사용자가 인터페이스에서 선언한 모든 멤버 함수들은 전위 처리기에 의해 구현부에 있는 멤버 함수들을 호출하는 것으로 변환된다.

결과적으로, 인터페이스 객체와 구현부 객체는 그림 1과 같이 구성된다. 데이타베이스에 저장되는 객체의 실질적인 부분은 모두 구현부가 관할하며, 인터페이스는 이에 대한 아무 영향력이 없고 다만 객체 핸들러 역할을 한다. 따라서, 전위처리 후 변환된 결과 코드는, 해당 구현부 객체가 Ref 핸들러에 의해 저장되는 것과 비슷한 효과를 얻는다.

3.2 변환의 예

3.2.1 인터페이스와 구현부의 변환

먼저, 다음과 같은 간단한 스키마 인터페이스인 SCollection을 살펴보자.

```
persistent class SCollection {
 public:
  void insert();
  char name[10]; };
```

이것이 전위 처리되면, InterObject클래스의 하위 클래스로 바뀌어 다음과 같이 여러 가지 메쏘드와 명세가 첨가된다.

```
class SCollection: public InterObject
{ public:
  void  insert(){
   void(*f)(Ref<ImplObject>);
   f=(void (*)(Ref<ImplObject>))
    (*(((ImplObject*)operator->())->
     my_class_tbl))["insert"]->ptr;
   f(*this);  }
  void set_name(char n[]){
   void (*f)(Ref<ImplObject>, char []);
   f = ( void (*)(Ref<ImplObject>, char[]))
    (*(((ImplObject*)operator->
     ())->my_class_tbl))["set_name"]->ptr;
   f(*this,n);  }
  char * get_name() {
   char * (*f)(Ref<ImplObject>);
   f = ( char * (*)(Ref<ImplObject>))
    (*(((ImplObject *)operator->())->
     my_class_tbl))["get_name"]->ptr;
   return f(*this);  }
   SCollection(){}
public:
  SCollection( Ref<ImplObject> t) :
   InterObject(t) {}
  SCollection& operator= (Ref<ImplObject> t)
{ Ref<ImplObject> * nthis = this;
   (*nthis) = t;  }
  ... };
```

SCollection과 같이 상위 클래스가 없는 인터페이스는 InterObject를 상위 클래스로 하여 변환된다. 여기서 SCollection은 InterObject에서 물려받은 멤버 이외의 멤버는 가지지 않음을 알 수 있다. 위의 insert와 같이 인터페이스의 멤버 함수의 내용은 자동으로 채워지게 되는데, Ref<ImplObject>의 my_class_tbl에 저장되어 있는 그 함수의 포인터를 적절한 타입으로 강제 타입 변환해서, 호출하는 것이 된다. 만일 멤버가 있다면, get_/set_ 함수가 추가되며, 이들은 전위처리기가 마련한 해당 전역 함수를 호출하는 내용을 가지게 된다.

구현부의 변환도 인터페이스의 변환과 비슷하나, InterObject 대신 ImplObject에서 계승받게 만든다는 차이가 있다. 또, implements 키워드를 만나면, 해당 인터페이스로 가서 그의 멤버와 멤버 함수를 가져와 재정의하는 일도 전위처리시 해주게 된다. 예를 들어 다음과 같은 구현부 정의는

```
class BSet{
 implements SCollection;
  void insert(){ ...}
 private:
  SCollection * other;
  int number_of(){ ...} };
```

다음과 같이 변환된다.

```
class BSet : public ImplObject{
  SCollection  other;
  int number_of(  ){ ...}
 public :
  char name[ 10 ];
  void insert( ){...}
   BSet( ) : ImplObject("BSet"){ ... }
  ... };
```

Copyright (C) 2005 NuriMedia Co., Ltd.

3.2.2 전역 함수의 정의와 등록

인터페이스와 구현부를 연결해주는 역할을 하는 전역 함수들이 전위처리기에 의해 정의된다. 각 인터페이스의 멤버 함수들에 대해서는 해당 구현부 클래스 이름과 멤버 함수 이름이 결합되어 전역 함수 이름이 생기게 되는데, 이 전역 함수들의 내용은 해당 구현부 클래스의 멤버 함수의 호출이다. 인터페이스의 멤버에 대한 get_/set_ 전역 함수도 이 때 정의되는데, 구현부의 멤버를 리턴해주거나 지정 (assignment)하는 내용으로 정의된다. 이러한 전역함수들은 궁극적으로는 구현부 이름, 함수 이름, 전역 함수 이름을 인자로 하여 전역 변수 ftbl에 등록되게 된다.

3.2.3 표현식의 변환

앞서 2절에서 밝혔 듯이 데이타베이스 객체의 생성은 C++의 new 연산자를 객체베이스 인자를 가지고 호출함으로써 이루어지며, 생성된 객체의 접근은 포인터 타입의 객체 핸들러를 통하여 수행 된다.

다음과 같은 LOD* 프로그램이 있다고 가정하자.

```
persistent SCollection * sp3 =
 new(obase) BSet;
 .
cout << sp3->name;
sp3->insert();
```

이와 같은 프로그램은 전위 처리를 거친 후 다음과 같이 변경된다.

```
SCollection  sp3=
 Ref<BSet>(new (obase) (BSet )) ;
...
cout << sp3.get_name();
sp3.insert ( ) ;
```

인터페이스 자체가 구현부 객체를 가리키고 있는 스마트 포인터의 일종이므로, 인터페이스의 포인터 타입으로 정의된 것은 인터페이스 타입 객체 자체로 바꾸어준다. 결국 인터페이스 객체는 Ref와 같은 역할을 하게 된다.

3.3 타입 검사

전위 처리기에서의 타입 검사는 크게, 구현부의 정의시에 implements 키워드로 명시된 인터페이스와의 바인딩 관계가 받아들여질 수 있는 것인지를 검사할 때와, 표현식의 타입을 검사할 때로 이루어진다. 표현식의 타입 검사는 다시 인터페이스와 구현부간의 구현 관계 검사, 두 인터페이스나 두 구현부 간의 관계 검사 등의 여러 가지 타입 검사가 적용될 수 있다. 여기서 인터페이스나 구현부들은 각각 클래스로 변환되므로 두 인터페이스나 두 구현부 간의 관계 검사는 C++의 클래스 계층 구조에 맡길 수 있다. 그러나, 인터페이스-구현부 간의 구현 관계 등은 반드시 전위 처리기에 의해 검사되게 된다.

그런데, ODMG C++ 바인딩[4]에 의하면, 객체 핸들러 'Ref'가 가지는 다양한 생성자와 변환 연산자들 때문에 그 템플릿 타입 인자가 무엇이든간에 서로 지정될 수 있게 된다[15]. 즉, ODMG C++ 바인딩에서는 'Person'의 객체가 'Ref<Person>' 타입의 핸들러들뿐아니라 'Ref<Bycycle>', 'Ref<School>' 등과 같이 전혀 관련 없는 타입의 핸들러들에도 컴파일 오류없이 지정이 가능하며, 이것은 수행 시간까지 넘겨져서 트랜잭션 도중 동적으로 예외 처리가 되게 된다. 따라서, 이러한 핸들러들간의 정적인 타입 검사의 결과가 C++의 포인터 타입간의 타입 검사와 거의 동일하게 이루어지도록 하기 위해서는, 어디서든 이러한 타입 검사를 맡아 수행해야 한다. LOD*는 전위 처리기를 통해 정적인 타입 검사를 하고 있기 때문에 객체 핸들러의 타입이 C++의 포인터의 타입과 비슷하게 부여되고, 결과적으로는 데이타베이스 프로그래밍이 보다 편리해지며, 실행 중 예외 처리 결과 중단 (abort)되는 트랜잭션들을 줄일 수 있다는 장점을 얻는다.

타입 검사 동안 스키마 정보가 계속 참조되므로 프로그램을 전위 처리하는 동안에는 데이타베이스뿐 아니라 주기억 장치에도 스키마를 위한 테이블을 두어 클래스들에 대한 정보를 저장하게 된다. 이를 '클래스 정보 테이블'이라고 한다 이 클래스 정보 테이블은 현재 전위 처리되고 있는 클래스들에 대한 정보와 스키마 관리자로 부터 얻은 정보를 함께 보관하게 된다. 전위처리기가 따르는 타입 검사 원칙에 대한 자세한 내용은 [13]에 있다.

3.4 전위 처리기 외적인 사항들

기존의 스키마 관리자는 클래스만을 다루었지만, 클래스-분리를 도입하면 인터페이스와 구현부를 모두 다루어야한다. 현재 인터페이스와 구현부 각각은 하나의 클래스의 형태로 스키마 저장소에 저장되는데, 스키마 관리자는 기존의 스키마 정보외에도 인터페이스인지 구현부인지를 나타내는 플래그 (flag)를 가지며, 인터페이스와 구현부간의 구현 관계를 보관/처리할 수 있도록 확장되게 된다. 이로써 기존의 스키마 관리자 및 스키마 저장소를 가능한 그대로 사용할 수 있도록 하였고, 비슷한 방식으로 여타 데이타베이스 관리 시스템들도 비교적 간단하게 클래스-분리를 지원할 수 있도록 확장 가능하리라 본다.

Copyright (C) 2005 NuriMedia Co., Ltd.

ODMG 객체 모델[4]에 의하면, 특정 프로그래밍 언어에 의존하지 않고 ODL (Object definition language)이나 그밖의 온라인 스키마 저작 도구 (on-line import tool) 등으로 정의된 클래스들은 클래스의 구현과 무관하다. 따라서, 클래스-분리를 도입하면, 이러한 클래스들은 인터페이스로 간주되어야 한다.

질의어를 사용하는 사람이나 대다수의 응용 프로그램들에서는 구현부에 관해서는 모르는 상태로 오로지 인터페이스만을 가지고 데이타를 접근할 수 있다. 이로써 클래스의 구현을 변경하더라도 질의어를 사용하는 사람이나 응용 프로그램에 가능한 영향이 가지 않도록 하여, 또다른 차원의 데이타 독립성을 보장하게 된다.

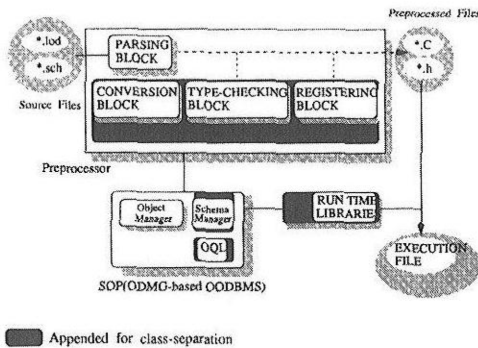

그림 2 전위 처리기 중심의 지원 방식을 위한 구조도

이러한 전위 처리 절차는 그림 2와 같이 요약될 수 있다. LOD*의 전위 처리기는 .lod와 .sch의 확장자를 가진 LOD* 화일들을 파싱하여 유용한 정보를 저장/추출하기 위한 구문 트리를 생성하게 된다. 다음엔 트리의 각 노드를 하나씩 모두 방문하면서 여러 가지 정보들을 추출하고, 이러한 정보들에 따라 지속적 자료인가 아닌가와 그 타입을 판별하여 적당한 변환을 하게 된다. 이 때, 스키마 관리자와 정보 교환이 이루어지며, 스키마 클래스의 정의라면 데이타베이스에 등록을 하게 된다. 결과로 나온 C++ 코드는 LOD* 라이브러리 및 SOP에서 제공하는 각종 라이브러리들을 써서 링크되어 수행 화일로 컴파일 된다. 전위 처리기의 파싱 부분은 Brown 대학에서 만들어 공용으로 배포한 C++ 파서(parser)를 수정, 확장하여 사용하였다.

4. 비교와 토의

C++과 같이 프로그램의 수행 속도를 중시하는 언어들은 그 의미론(semantics)도 언어의 구현 방식에 따라 좌우되는 부분이 많다. 이 때문에 의미론의 확장이 구현에 의해 제약을 받게 되므로, C++과 같은 언어들을 확장하는 것은 그리 간단한 일이 아니다. 클래스-분리를 추가할 때도 이러한 어려움이 예외는 아니기 때문에, LOD*에서는 몇 가지 제한을 두고 지원하기 시작하였다. 먼저, 데이타베이스를 위한 클래스의 인터페이스나 구현부는 반드시 main 함수 바깥에 전역 (global)으로 선언된 것만을 취급하고, 인터페이스나 구현부 내에 중첩된 다른 클래스의 선언이나 정의가 없다고 가정하였다. 이로써, 인터페이스나 구현부의 이름 인식 공간에 대한 구현과 같은 복잡한 부분들을 가능한 단순화시켜, 클래스-분리의 구현 쪽에 더 비중을 둘 수 있었다. 그리고, InterObject나 ImplObject가 다중 계승시 중복 계승될 수가 있으므로, 단일 계승만을 고려하였다. 이것은 클래스-분리뿐만 아니라, PObject를 계승받는 ODMG C++과 Smalltalk 바인딩 일반에 관한 문제이기 때문에 보류하였다. 또, 인터페이스의 정의시 사용자 정의 타입의 배열을 멤버나 멤버 함수로 가지는 것은 현재 배제하였다. 이유는 그 인터페이스에서 인터페이스 타입의 배열로 선언된 멤버/멤버 함수가 해당 구현부들에서는 구체적으로 구현부의 배열을 써서 재정의 될 수 있으므로, 이 경우 인터페이스 포인터를 통한 배열 멤버 접근시, 그 배열 원소의 위치 계산이 각 구현부가 정의되기 전 인터페이스 정의만 가지고는 수행되기 어렵기 때문이다. 또, 템플릿으로 구현된 인터페이스나 구현부는 별도로 연구하여 의미론을 결정해야 하는 부분이므로 본 논문의 범위에 넣지 않고 있다. 현재, 클래스-분리를 도입한 Java[1]등에서도 템플릿을 고려하지 않고 있지만, 이 부분에 관한 연구들은 여러 곳에서 계속 수행되어 오고 있다[16,17,18].

LOD*의 구현을 위해 기존의 타분야에서 이루어져 왔던 여러 클래스-분리 방식들이 참조되었다. 하지만, 이들 기존의 방식들을 그대로 사용하지는 못하였는데, 여기서는 각각에 대해 살펴보고 LOD*와 비교한다.

4.1 기존 시스템들의 고찰

4.1.1 CORBA C++ 기반 데이타베이스 시스템

CORBA[3]도 클래스-분리를 지원하고 있다. CORBA에서 객체는 'TIE 방식'과 'BOA 방식' 중 한 가지에 의해 구현된다[19]. TIE 방식은 인터페이스와 구현부 사이에 연결하는 객체를 하나 더 두어 타입 검사 등을 도맡는 방식이며, BOA 방식은 구현부가 인터페이스의 하위 클래스로 들어감으로써 클래스-분리를 하는 방식이다. 이 때 TIE 방식은 인터페이스 포인터나 구현부 외에 새로운 중간 객체를 생성하는 오버헤드가 있으며

Copyright (C) 2005 NuriMedia Co., Ltd.

[19], BOA 방식은 인터페이스와 구현부 계층 구조가 섞임으로 인한 스키마 복잡성과 불안정성을 가지게 된다[20]. CORBA에서 지원되는 지속적 객체들도 BOA 방식이나 TIE 방식에 근거하여 CORBA 객체에 지속성을 부여하고 있기 때문에, 이 한계 내에서 구현되고 있다[21,22]. 그 외에 별도의 지속적 서버를 도입하는 경우나[23], 또는 관리자 객체를 별도로 두고 이를 통해 데이타베이스를 접근하는 방식[24]은, ODMG 모델을 따르는 데이타베이스 스키마의 클래스-분리와는 그다지 관련이 없다고 할 수 있다.

4.1.2 GNU C++의 전위 처리기

GNU C++ 컴파일러의 전위 처리기 형태로 구현된 [16]의 방식에서는, 인터페이스 객체가 멤버 함수 포인터 테이블로의 포인터와 구체적인 구현부 객체로의 포인터 모두를 가지는 구조이다. 이는 LOD*에서 하나의 인터페이스 객체가 ODMG C++ 바인딩의 Ref처럼 구현부 객체 하나만을 가리키는 것과 대별된다.

멤버 함수 포인터들의 테이블은 각 구현부의 객체가 인터페이스에 바인딩 될 때마다 동적인 객체로서 하나씩 생성되며, 생성된 테이블 객체의 포인터는 해당 인터페이스 객체 내부에 저장된다. 테이블 객체에는 구현부의 멤버 함수들의 주소들이 인터페이스의 멤버 함수 타입으로 변환되어 저장되며, 인터페이스 객체에 대해 멤버 함수가 호출될 때 직접 호출되어 수행된다. 반면, 앞서 언급했듯이 LOD*는 이와 달리 처음 프로그램을 시작할 때 정적으로 멤버 함수 테이블이 구성된 후, 인터페이스 객체가 아닌 구현부 객체에 테이블 객체 포인터가 저장되기 때문에, 인터페이스 객체에 대해 멤버 함수가 호출될 때는 해당 구현부 객체를 거쳐서 수행되게 된다.

결과적으로 이 방식은 LOD*에 비해 인터페이스에 대한 멤버 함수 호출시 구현부 객체를 한 단계 거치지 않아도 된다는 장점이 있는 반면, ODMG C++ 바인딩의 Ref를 중심으로 하는 객체 구조를 반영하는 데는 LOD*에 비해 적합하지 못함을 알 수 있다. 그리고, 이 방식은 멤버 함수 포인터 테이블 구성시 구현부의 멤버 함수 포인터 타입을 인터페이스의 멤버 함수 포인터 타입으로 변환하는 과정에서, C++에서 허용되지 않는 타입 변환이 이루어지기 때문에 GNU 컴파일러 자체의 수정이 불가피하여[16], 다른 C++ 컴파일러를 사용할 수 있는 이식성이 떨어지게 된다는 단점이 있다.

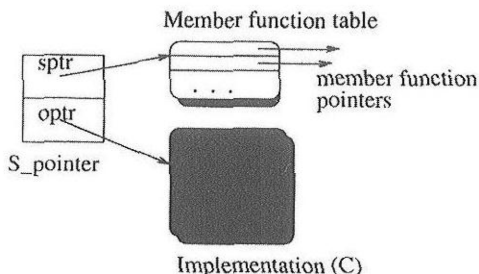

그림 3 GNU C++에서 사용된 클래스-분리 기법에서의 객체 구성

4.1.3 HP 방식

C++에 클래스-분리를 지원하는 방식 중 가장 직관적이고 구현이 간단한 것은, HP가[25]에서 제안한 것이다. 여기서는 인터페이스 각각을 클래스로 두고, 인터페이스의 구현부를 그 인터페이스의 하위 클래스로 만들어 구현하였다. 따라서, 사용자가 인터페이스와 구현부를 위해 독립적인 계층 구조를 정의하고 나면, 구현부는 자신의 상위 클래스와 인터페이스를 다중 계승을 하게 된다. 결과적으로는 두 개의 서로 독립적인 계층 구조가 하나로 섞이게 되므로, 클래스 계층 구조가 인터페이스 계층 구조와 구현부 계층 구조가 얽혀 있는 모양을 하며 복잡해지게 된다. 따라서, CORBA의 BOA 방식과 마찬가지로 복잡성과 불안정성이 있고[19], 스키마 변환 오버헤드가 커지게 된다[20].

4.2 전체 비교

인터페이스의 포인터는 OID 혹은 스마트 포인터로 변형될 수 있다. 하지만, OID로 변형되면 인터페이스 포인터를 위한 스위즐링 (pointer swizzling)[26] 등의 처리를 필요로 하게 되는데, 이는 이미 ODMG C++ 바인딩의 스마트 포인터인 Ref 클래스에서 제공되고 있다[22]. 따라서, 인터페이스 포인터를 Ref 타입으로 변환하는 것도 하나의 방법이 될 수 있다. 이 때 구현부를 통해 생성된 객체를 인터페이스 포인터에 지정할 수 있도록 하기 위해서는, 템플릿 Ref에 해당 구현부에서 변환된 클래스를 타입 인자로 하는 타입을 인터페이스 포인터의 타입으로 하게 된다. 하지만, 이 경우에는 인터페이스 포인터의 타입이 지정문 우측에 나타나는 객체의 구현부에 따라 달라지게 된다는 뜻이므로, 실제 수행 전에는 정확한 타입을 알 수 없다는 단점이 있다. 결과적으로 이 방식으로는 전위 처리시 인터페이스 포인터를 선언하는 코드의 생성이 불가능하다.

대신 타입에 자유로운 스마트 포인터인 RefAny로 변환될 수도 있다. 하지만, RefAny는 ->와 같은 연산자가 제공하지 않으므로, 이 경우도 아래와 같은 문에서 i->f()는 컴파일 오류를 동반한다.

Copyright (C) 2005 NuriMedia Co., Ltd.

```
Interface1 * i = new(obase) Implementation1;
// when converted to
// RefAny ı = new(obase) Implementation1;
...
ı->f(); // there occurs a compile error in C++
```

따라서, 이를 극복하는 다른 몇 가지 방법을 소개하면 다음과 같다. 먼저, Interface1이 Implementation1의 상위 타입이면, 인터페이스 포인터를 따로 변환하지 않아도 지정문에 타입 오류가 없게 된다. 이는 HP에서 제안된 클래스-분리 방식[25]과 CORBA의 BOA[19]에서 쓰고 있다. 하지만, 언급되었듯이 클래스 계층 구조가 복잡하게 되어 객체의 구조가 이에 영향을 받아 안정성에 문제가 생긴다는 점이 있다. 둘째, 인터페이스와 구현부 중간에 타입과 관련된 사항을 관장하는 중간 객체 (TIE)를 두고, 이 중간 객체의 클래스는 인터페이스의 하위 클래스이며, 구현부로 부터의 변환 연산자를 보유하게 한다. 이 방식은 CORBA의 TIE 방식[19]에서 사용한 것으로 중간 객체 생성의 오버헤드가 단점이다. 셋째, f() 호출 전에 i를 강제적으로 Implementation1으로 타입 변환을 할 수 있는데, 즉, ((Ref<Implementation1>)i)->f() 등으로 변형한다. 그러나, C++과 같이 동적 타입 확인과 타입의 1차 객체화가 지원되지 않는 언어에서, Implementation1을 전위처리중에 인식하여 처리해야 하므로, 이 방법은 여전히 적용이 불가능하다[5]. 넷째, 본 논문에서와 같이 새 스마트 포인터('Interface'와 하위 클래스들)를 도입하여 이 스마트 포인터 객체에 직접 f()를 호출하게 되면, 구현부 객체를 판단하여 메쏘드를 호출하는 일이 자동적으로 관장되게 된다. 다섯째, 앞서 언급했듯이 GNU C++ 처럼 직접 컴파일러를 수정하여, 멤버 함수 바인딩과 타입 변환을 강제하는 방법이 있으나, 수정된 컴파일러외의 다른 C++ 컴파일러를 사용할 수 없기 때문에 이식성에 문제가 있다[16].

표 1 LOD* 및 기타 클래스-분리 지원 방식간의 비교

		타입검사	바인딩	동적 구현부 판단	스키마 오버헤드	함수 포인터 테이블	멤버 접근 오버헤드	함수 호출 오버헤드
HP		컴파일러	명시적	계승	유	무	무	무
GNU C++		전위처리기	암시적	강제	무	유(동적)	유	유
CORBA	TIE	컴파일러	명시적	중간 객체	무	무	유	무
	BOA	컴파일러	명시적	계승	유	무	유	무
LOD*		전위처리기	명시적	스마트 포인터	무	유(정적)	유	유

5) 현재 새 C++ 릴리즈에 동적 타입 변환은 추가가 되었다[15].

표 1은 앞서 언급되었던, 클래스-분리의 각 지원 방식에 따르는 비교를 간단히 나타낸 것이다. 타입 검사는 크게 전위처리기가 하는 것과 C++ 컴파일러에게 맡기는 두 가지 방식이 있다. LOD*과 같이 전위처리기가 맡는 경우, 컴파일러나 특정 객체 구조에 의존하지 않는 장점이 있다. 인터페이스와 구현부간의 바인딩은 암시적인 것과 명시적인 것이 있으나, 데이타베이스 응용 프로그램에서는 스키마를 다루므로 LOD*와 같이 명시적인 것이 더 적합하다[7].

스키마 변환 오버헤드는 클래스-분리에 따라 스키마가 복잡해지는 HP나 CORBA BOA 방식이 가장 크고, 상대적으로 LOD*등은 적다. GNU C++은 함수 호출 테이블이 거의 매 객체 생성시마다 또 하나의 동적인 객체로 생성된다는 점에서 오버헤드가 있다. LOD*에서는 선언된 모든 인터페이스, 구현부에 대해 함수 호출 테이블을 정적으로 매 프로그램 시작마다 구축하게 된다.

구현부의 형태를 변형하여 멤버 접근을 인터페이스가 직접 하는 경우, 멤버 접근의 오버헤드가 적다는 장점이 있다. 하지만 융통성과 다중언어 지원이 쉬워지는 잇점 때문에, LOD*와 같이 멤버 접근을 함수 호출로 변형하고, 멤버 접근 오버헤드를 감수하는 것이 일반적이다[5]. 멤버 접근을 함수 호출로 바꾸었을 때 드는 오버헤드에 대해서는, [5,26]등에서 볼 때 활발히 연구되고 있음을 알 수 있다. 더우기 전위처리 방식을 택한 LOD*와 같은 경우에는 보다 많은 최적화 기회가 있을 것으로 생각된다.

GNU C++ 방식이나 LOD*과 같은 함수 포인터 테이블을 통하여 접근하는 방식에는 함수 호출 오버헤드, 특히 동적 바인딩에 드는 오버헤드가 있게 되지만, CORBA에서와 같은 간접 객체 등은 쓰지 않는다는 잇점을 가질 수 있다. 이것과 관련해서는 메쏘드 추출(method dispatch) 기법[27] 등의 연구가 현재도 활발히 진행되고 있다.

5. 결 론

본 논문에서는 ODMG-93 C++바인딩을 기반으로 확장한 언어인 LOD*에 클래스-분리를 추가 확장하는 구체적인 방식을 소개하였다. 먼저 클래스-분리의 지원 방식에 관한 방향을 정의하고, 이를 따르는 구현을 하였다. 전위 처리 방식을 따랐기 때문에, 목적에 맞는 독자적인 모델의 개발과 타입시스템 및 구문 선정이 가능하였으며, ODMG C++ 바인딩 코드로의 변환을 통해 기존에 제공되고 있는 스마트 포인터의 기능을 최대한 이용할 수 있게 되고 프로그램의 이식성도 높였다[13]

Copyright (C) 2005 NuriMedia Co., Ltd.

본 논문에서 소개된 클래스-분리 구현 방식은 데이타베이스 관리자 전반에 관해 구체적으로 고려되었으며, ODMG C++ 바인딩을 지원하는 시스템 위에서 일반적으로 수행 가능하여, 데이타베이스 응용 프로그램에 비교적 적합한 형태로서 이끌어내어졌다고 판단된다. 현재는 앞서 언급된 비교 대상들과 함께 성능을 비교 분석 중이다.

참 고 문 헌

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

[2] A. Black, N. Hutchinson, E Jul, H. Levy, and L. Carter. 'Distribution and Abstract Types in Emerald'. *ACM Computing Surveys*, 19(2): 105190, June 1987.

[3] DEC, HP, HyperDesk, NCR, Object Design, and SunSoft. *The Common Object Request Broker : Architecture and Specification*. OMG group, February 1997.

[4] R. G. G. Cattell. *Object Database Standard : ODMG-93 release 2.0,*. OMG group, 1997.

[5] B. Liskov, A Adya, M. Castro, and Q. Zondervan.'Type-Safe Heterogeneous Sharing can be Fast'. Technical report, MIT, 1996.

[6] Nelson M. Mattos. 'An Overview of the SQL3 Standard'. Technical report, Database Technology Institute IBM - Santa Teresa Lab. San Jose - California, March 1995.

[7] E.S. Cho, S. Y Han, and H. J. Kim.'A New Data Abstraction Layer Required For OODBMS'. In *Proc. of International Database Engineering and Applications Symposium*, August 1997.

[8] Object Design, Inc. *ObjectStore Release 4.0 Online Documentation*, March 1995.

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

[10] 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.

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

[12] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. 'The Design of the E Programming Language'. Technical Report #824, Computer Science Department, University of Wisconsin-Madison, February 1989.

[13] Eun-Sun Cho. *'LOD* : An Object-Oriented Database Programming Language with High-Degree of Class-Separation Support'*. Ph.D thesis, Dept. of Comp. Sci. and Stat., Seoul National University, August 1998.

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

[15] Bjarne Stroustrup, ed *The C++ programming language third edition*. Addison-Wesley Publishing Company, Inc., 1997.

[16] 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.

[17] A. C. Myers, J. A. Bank, and B. Liscov. 'Parameterized Types for Java'. In *Proc. of SIGPLAN Conf. on Principle of Programming Languages*, 1996.

[18] P. O'Brien, B. Bullis, and C. Schffert. 'Persistent and Shared Objects in Trellis/Owl'. In *IEEE International Workshop on OODBS*, pp. 113-123, 1986.

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

[20] E S. Cho, S. Y. Han, and H. J. Kim.'A Semantics of the Separation of Interface and Implementation in C++' In *Proc. of Int'l conference on COMPSAC*, 1996.

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

[22] F. Reverbel. *'Persistent in Distributed Object Systems: ORB/ODBMS Integration'*. Ph.D thesis, University of New Mexico, 1996.

[23] Jan Kleindienst, Frantisek Plasil, and Petr Tuma. 'Lessons Learned from Implementing the CORBA Persistent Object Service'. In *Proc. of the ACM OOPSLA Conf.*, 1996.

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

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

[26] A. L Hosking and J. E. B. Moss. 'Compiler Support for Persistent Programming'. Technical report, University of Messacgusetts, March 1991.

[27] 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.

Copyright (C) 2005 NuriMedia Co., Ltd.

조 은 선

1991년 2월 서울대학교 계산통계학과 졸업. 1993년 2월 서울대학교 계산통계학과 전산학 전공 이학 석사. 1998년 8월 서울대학교 계산통계학과 전산과학전공 이학 박사. 1999년 4월 ~ 현재 한국과학기술원 프로그램 분석 시스템 연구단 전임연구원. 관심분야는 객체지향 프로그래밍 언어, 객체지향 데이타베이스, 프로그램 분석

김 형 주

제 26 권 제 1 호(B) 참조

Copyright (C) 2005 NuriMedia Co., Ltd.