A New Data Abstraction Layer Required For OODBMS

Eun-Sun Cho
Department of Computer Science
Seoul National University
Seoul, Korea 151-742

Sang-Yong Han
Department of Computer Science
Seoul National University
Seoul, Korea 151-742

Hyoung-Joo Kim
Department of Computer Engineering
Seoul National University
Seoul, Korea 151-742

e-mail:{eschough@candy, syhan@pandora, hjk@papaya}.snu.ac.kr

Abstract

A 'class' in object-oriented paradigm represents both interface and implementation of the class. However, interface and implementation of a class are needed on different purposes, since class interface is shared among most users, while class implementation is used only to the implementors of the class.

In this paper[1], we introduce a new level of data abstraction, called the 'class-implementation level', which is based on the separated management of interface and implementation of a class. And we describe a new model for OODBMS which provides users with the abstract view of the class implementation.

1 Introduction

Most OODBMSs are based on 'classes[6]'. The definition of a 'class' can be divided into two part – class *interface* and class *implementation*. Class *interface* represents data semantics which is shared among users, while class *implementation* implements a class including data structures and method definitions. Many extended relational database management systems(ERDBMS) and OODBMSs allow users to identify class interface and class implementation by the keywords 'public/private'. For example, the interface of a class is a collection of data/method declarations which are specified with 'public'.

However, according to the semantics of database schema, class interface is shared by more than one applications, while class implementation is used only for the implementors of the class. Since interface and implementation of a single class are on such different purposes that there are many limitations in OODBMSs as well as in conventional application programs. In this paper, we introduce a new data-abstraction level called 'the *class-implementation level*' for OODBMSs, This new layer solves many problems rising in the previous abstraction model, and guarantees '*class-implementation independence*' which means the ability to modify the class implementation without causing application programs to be rewritten.

The sequence of the paper is as follows. The next section shows the motivation of a new integration mechanism. Section 3 presents an overview of the concept of the class-implementation abstraction. And section 4 describes how the new abstraction concept effects the data model and APIs(application programming interface) for OODBMSs. Section 5 investigates more on the semantics of the OODBMSs with the new abstraction layer. Section 6 introduces an example of the proposed OODBMS architecture which is currently being implemented. Section 7 covers some related works. Finally, section 8 concludes this paper.

2 Motivations

2.1 Implementation hiding

These days, object-oriented concept has proved to be the solution for distributed computing with its message sending mechanism[5, 10]. And, in many distributed OOPLs[5, 10], a class is required to be shown as two separate modules – one is the '*interface*' of a class which can be accessed by all users regardless of their sites, and the other is the '*implementation*' of a class, used only for implementation of the class. So

[1]This work is supported in part by KOSEF under grant KOSEF94-2180, "Research on Object-Oriented Database Programming Language Based on C++ and ODMG Standard Object Model".

0-8186-8114-4/97 $10.00 © 1997 IEEE

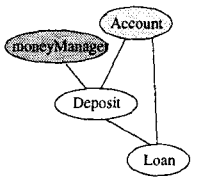

Figure 1: A mixed class hierarchy

does some distributed DBMSs to allow users to access distributed data in more elegant ways[14, 18].

ODMG-93[6], a de facto standard of OODBMS model, also suggests the separation of interface and implementation of classes. The main purpose of such separation in ODMG model is to provide multi-language environments and allow sharing the database among applications in various languages, such as OQLs(object query language)[6], C++, Smalltalk, and so on. Thus, ODMG proposes an application language-independent ODL(object definition language) to define database schema[6]. It is not surprising that most ODL-compliant OODBMSs are based on CORBA[10]. But, in such case, no changes are made in DBMS architectures or semantics, but they provide only a kind of CORBA gateways. Even in the OODBMSs tightly bound to CORBA[12], the library interface for CORBA has limitations[3].

In this paper, we suggest the new abstraction layer, called the 'class-implementation layer', which effects all database APIs, and propose that OODBMSs should be designed with consideration for the new layer so that users could take advantage of the class-separation easily. The new layer with class-separation is useful not only for the distributed or multi-language applications, but also for the normal OODBMSs.

2.2 Two different kinds of class hierarchies

In OOPLs, subclass relationships are identified with inheritance for code-reuse. For example, the definition of class 'Deposit' reuses the implementation of the system defined class 'money'.

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

However, in a database schema, subclass relationships are defined based on subset relationships, instead of reuse relationships. So, the subclass relationship sets in an OOPL and the database schema do not go with each other. In above example, class 'Account' can be a super class of 'Deposit' in the database schema.

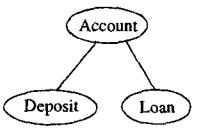

Figure 2: A class hierarchy for database schema

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

Although both class 'Deposit' and class 'Loan' are the subclasses of class 'Account', they might be implemented differently that it can be, for example, a subclass of 'Deposit' itself for code-reuse. In current OODBMSs, this situation can be realized by multiple inheritance, as shown in figure 1. However, such multiple inheritance from the mixture of the two independent hierarchies is known to cause serious complexity[4, 8]. And it may be more serious in database applications, because the schema evolution cost, in proportion to the complexity of the schema, is much higher than in conventional object-oriented programs.

Note that the hierarchy for persistent classes becomes more complicated than originally intended for a schema classes shown in figure 2, which makes the database users confused, and schema evolution costs increased. For example let us assume that a class 'Deposit' has multiple implementations named 'Money_Deposit', 'Deposit_Impl1', and 'Deposit_Impl2'. They have to be made subclasses of 'Deposit' in the persistent class hierarchy. These can be represented in a C++ like syntax as follows.

```
class Deposit{ ... }; // a schema class
// various ways of implementing Deposit
class Money_Deposit : virtual Deposit{ ... };
class Deposit_Impl1 : virtual Deposit{ ... };
class Deposit_Impl2 : virtual Deposit{ ... };
```

At this time, if a new class 'SpecialDeposit' is created as a subclass of 'Deposit' in the schema, it should be inherited from all the three classes. However, an ambiguity[11] can arise if any pair of those three classes happen to have common data/methods. And users have to overriding the data/methods from three super classes in order to implement the class 'SpecialDeposit' of its own.

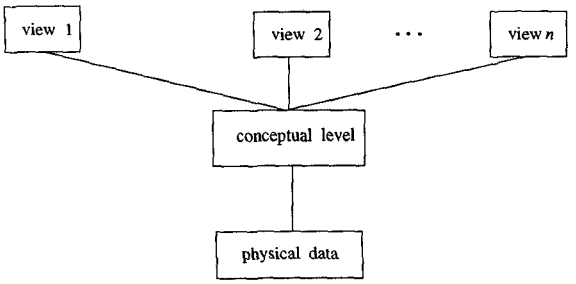

Figure 3: The existing database abstraction layers[15]

3 Overview of the Class Implementation Abstraction

Conventionally, there are three levels of data abstraction – the physical level, the conceptual level and the view level, as shown in the figure 3[15]. In this paper, we introduce a new level of abstraction, the '*class-implementation level*', as illustrated in figure 4. This new level includes method definition and data structure definition which are related to implementing the class.

To achieve the class-implementation independence, a schema class is defined as two separated classes – an '*interface*' and an '*implementation*'. For example, a schema class 'Deposit' mentioned above, is defined as follows.

```
// interface
persistent class Deposit {
    number account_number;
    void show_amount();
    void show_date(); .... };
// implementation
class DepositImpl1 {
    account_data p;
    implements Deposit; .... };
```

The implementation 'DepositImpl1' is bound to the interface 'Deposit' with the keyword 'implements', which means that the implementation 'DepositImpl1' *implements* the interface 'Deposit'. The definitions of implementations are shown only at the class-implementation level, that is, only to the application programmers implementing the very classes. Note that without the class-implementation level abstraction, all users have to see the whole classes including the implementation-related portions. The separated definition of an interface also allows ODL users to ignore the private data/methods of the class.

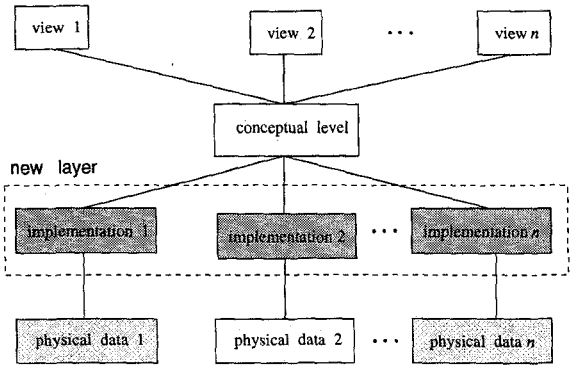

Figure 4: The proposed database abstraction layers

4 OODB Language Interfaces and The Class-Implementation Layer

In this section, we describe what the language interfaces for OODBMSs will be like when they provide users with the views of the class-implementation abstraction level.

4.1 ODLs(Object definition languages

ODLs are almost same as ODMG-ODL[6]. However, as mentioned earlier, ODL users do not concern about `private` part of classes. Thus, class 'Deposit', class 'Loan' and class 'Account' are defined as follows in an ODL.

```
// all data/methods are public
class Account {
    number account_number;
    void show_amount();
    money show_date(); .... };
class Deposit : Account {...};
class Loan :    Account {...};
```

4.2 Application programming languages

In this section, we introduces a language extension to C++, to show how the models of such APIs are changed. Most of all, in order to support the class-implementation independence, the database programming language should provide the facilities for the separated definition of schema interface and implementation. And in an application program two separated hierarchies are maintained – one for persistent class interfaces, and the other for implementations and usual non-database classes.

A hierarchy for class interface is intended only for modeling real worlds. The declaration of interfaces in an application program are same as those in an ODL except for they are specified with the keyword 'persistent' in an application program, which is a

conventional way to specify schema classes in database programming languages[6].

An implementation is similar to a usual non-database class except for the additional keyword 'implements' which binds it to an persistent class interface. A hierarchy for class implementation can be built at the class-implementation level, regardless of its class interface hierarchy. Thus, any changes in a class implementation hierarchy can be made without any affection on its schema interface class hierarchy, and vice versa. For example, both 'Money_Deposit' and 'DepositImpl1' may implement class 'Deposit'.

```
class Money_Deposit:money{implements Deposit;..};
class DepositImpl1 {
   implements Deposit;
   void compute_date(); .... };
```

In an application program, an object is created in a database through an implementation, and handled by an 'object handler' of an interface pointer type. For example, when an interface 'Image' is implemented by the implementations 'BitImage1' and 'BitImage2', their objects are used as follows.

```
persistent Deposit * x =  new DepositImpl1; ...
x =  new DepositImpl2;
```

4.3 OQLs(Object query language)

OQL queries are almost same as in conventional object-oriented queries. However, for implementation independence, an extent for a schema class is collected by traversing all the implementations which implement the interface, as well as its sub-interfaces in the interface hierarchy tree.

5 Discussions

5.1 More on class-separation

Separation semantics of interface and implementations is not new in programming languages[2, 4, 5, 9]. However, not all of them are satisfactory for the class-implementation layer. In this section, we investigate on the separation model appropriate for OODBMSs.

First, since database schema designers use sub-classing for real-world modeling, explicit sub-classing specified by users is preferable to implicit sub-classing based on signatures[4, 7].

Second, there should be no obligation to bind interfaces and implementations by one-to-one mapping. By means of the multiple implementations for an interface, some kinds of schema evolution can be also simplified, as shown in the later sections.

Third, since different users have to get information only from interfaces, an interface declaration itself is also required to be constructed with the known types to those users, such as primary types or common interfaces. That is, the whole set of interfaces must be described only with itself and primary types, named the 'self-containment' property of the interface set.

5.2 Schema evolution costs

In general, the cases of schema evolution in OODBMSs fall into two categories: one is the changes of the entities and relationships in real worlds, the other is modifying the class implementation usually for the sake of performance. Although the former is unavoidable, the latter kind of evolution is possible to reduce its cost by class-implementation abstraction.

For example, let us consider the previous example of class 'Deposit' with three subclasses for implementations. In our mechanism, such classes do not have to be subclasses of the class 'Deposit' any more, but instead, they are bound to the interface 'Deposit'. Thus, the new class 'SpecialDeposit' simply inherits from the class 'Deposit' directly, without consideration of these implementations. And, when a user wants to change, for efficiency, the private data declaration 'money amount;' to 'int amount;' in class 'Deposit', it is unnecessary to change the whole class declaration, but the user just creates a new implementation and binds it to the interface.

6 Implementing The Proposed Architecture

Implementation abstraction is being realized on an OODBMS named 'SOP(*S*NU *O*O-DBMS *P*latform)[1]'[2] as shown in figure 5, which provides many facilities including those mentioned in this paper. This system supports explicit sub-classing specified by users, and one-to-many mapping from interfaces to implementations. Type-safety in bindings of interface and implementation with 'implements' phrase and self-containment of interfaces are checked by the preprocessor of C++ API.

7 Related Works

Related works on DBMSs are listed in section 2. Here, we examine other works related to the

[2]An ODMG-based OODBMS developed from 1992 to 1995 at Seoul National University

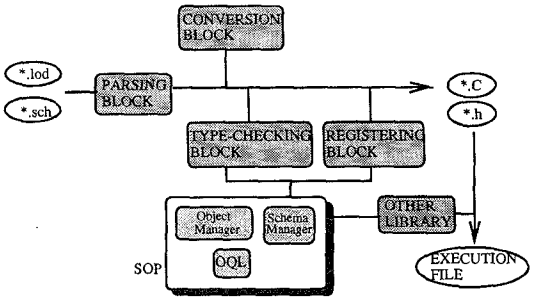

Figure 5: The architecture of a C++ preprocessor for the class-implementation level

class-separation concept. As mentioned earlier, the concept of separation of interface and implementations of a class is in programming languages such as Java, Objective-C and Emerald[2, 4, 5, 9]. However, those focus on distributed programming or separation of inheritance hierarchies, without explicitly-defined interface hierarchies, interface-implementation bindings, implementation type-abilities and other issues needed for database applications[2, 4, 5, 9]. Moreover, their separation semantics are not properly used for OODBMSs. For example, either ODMG2.0-Java binding draft[17] or JDBC[13][3] does not concern about the '`interface`' feature in Java[2].

8 Conclusions

This paper introduces new abstraction level called '*class-implementation layer*', which is based on the separated management of interface and implementation of a class. The class-implementation abstraction allows a schema class to have multiple implementations, which also reduces the schema evolution. And it is also useful for distributed or multi-language applications.

Currently, we are investigating on the semantics of class-separation for OODBMSs, and plan to complete the realization of the new layer.

References

[1] J.H Ahn and H. J. Kim. Seof: An adaptable object prefetch policy for object-oriented database systems. In *Proc. of the Conf. on Data Engineering*, 1997.

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

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

[4] 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.

[5] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald". *ACM Computing Surveys*, 19(2):105–190, June 1987.

[6] R. G. G. Cattell. *Object Database Standard : ODMG-93*. OMG group, 1993.

[7] R.C.H Connor, A.L. Brown, Q.I Cutts, and A. Dearle. "Type Equivalence Checking in Persistent Object Systems". In *The Fourth International Workshop on Persistent Object Systems*, pages 154–167, 1990.

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

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

[10] DEC, HP, HyperDesk, NCR, Object Design, and SunSoft. *The Common Object Request Broker : Architecture and Specification* . OMG group, 1991.

[11] Margaret A. Ellis and Bjarne Stroustrup, editors. *The Annotated C++* . Addison-Wesley Publishing Company, Inc., 1990.

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

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

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

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

[16] Microsoft. *Microsoft ODBC 2.0 : Programmer's Reference and SDK Guide (version 2.0)*, 1995.

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

[18] M. Shan. "Pegasus Architecture and Design Principles". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1993.

[3] A calling interface for relational database like ODBC[16]