Interface/implementation Separation Mechanism for Integrating Object-oriented Management Systems and General-purpose Programming Languages

EUN-SUN CHO[1] AND HYOUNG-JOO KIM[2]

[1]*Graduate School of Information and Communication, Ajou University, Suwon, 442-749, Korea*
[2]*Department of Computer Science, Seoul National University, Seoul 151-742, Korea*
Email: eschough@madang.ajou.ac.kr

When a database is shared in different database applications, it is helpful for data-independence and program readability to screen off the implementation details of a schema class from the programs other than the method implementation code. This makes some systems allow their users to define the schema class in two parts—the interface and the implementation. In this paper, we propose a preprocessing-based approach with the object model for OODBPLs (object oriented database programming languages) with interface/implementation separation.

1. INTRODUCTION

'Class' in object-oriented terms has two aspects—one is *interface*, and the other is *implementation*. Interface represents the semantics of the class, and the information only with which its users can access the objects of the class. On the other hand, implementation includes the internal implementation of the class.

These days, it is not rare to define a class separately into an interface unit and an implementation unit [1, 2, 3, 4]. For example, with the separate interface only, the clients in distributed environments are allowed to be served from the remote servers without knowing the implementation details.

As another example, in the database area, users can access data as long as they know the schema interface, while the implementation of a schema class is needed for implementing the method body of the class and for creating the objects. In addition, interface/implementation separation prevents implementation changes from influencing its interface specification and related programs, which reduces the schema evolution cost [5].

In this paper, we focus on the interface/implementation separation in the database area, and propose a preprocessing-based approach for OODBPLs (object-oriented database programming languages) with the object model with interface/implementation separation. The sequence of this paper is as follows. The way of defining an interface unit and an implementation unit is introduced in Section 2, while Section 3 presents the semantics of the hierarchical relationships between interfaces and implementations. In Section 4, the semantics of relationships between interfaces and implementations is elaborated. Each of these sections covers formal definitions, related works and discussions. Section 5 concludes the paper. An example of a relatively complete code segment is in the Appendix.

2. DEFINING INTERFACE AND IMPLEMENTATION

In this section and the three following sections, we introduce our preprocessing approach for the interface/implementation separation for OODBPLs. Before proceeding further we define terms used throughout this paper. A class *definition* is usually specified by data structures and method bodies of a class, but in this paper it is also used to mean the class *declaration*, whenever no confusion arises. The *base language* is an OOPL (object-oriented programming language) that is extended to database access. The base language is assumed to have a static type system [6], and must have an object constructor called *class* and user-defined hierarchical relationships among the classes. Subtyping is identified by the hierarchical relationship.

In the proposed mechanism, two more object constructors other than class are introduced—interface and implementation. An interface is an object constructor consisting of public data and public method signatures. An implementation is the implementation part of a schema class, including internal data structures and method implementations. When a schema class is divided into an interface and implementation, it is said that the implementation implements the interface. If the implementation `M` implements the interface `I`, the object created by `M` is also considered as an

object of `I`, and `I` and `M` are in an *implementing relationship*. When a user explicitly defines the implementing relationship between an interface and an implementation, the interface and the implementation are said to be *bound* to each other, or in a *binding relationship*.

A C++-like syntax, which is one of the most popular APIs (application programming interfaces) for OODBMSs (object-oriented database management systems) [7, 8, 9, 10, 11], is used throughout this paper. However, for simplicity, some restrictions are introduced to enable concentration on database semantics. For type constructors, only the function type is considered; the pointer type and the array type are ignored. Template classes, method overloading and protected attributes/methods [12] are not included either.

2.1. Definitions of interfaces and implementations

In the conventional C++ interface for OODBMSs, the keyword `persistent class` [7, 8] or `dbclass` [13] is used to specify which class is involved in a schema construction. For example, usually the persistent class `Deposit` could be declared in the conventional C++ interface as follows.

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

Similarly, the `persistent` keyword is also used in our approach. However, a persistent class here, unlike the ones in the previous approaches, consists of its `public` part, that is, of its interface.

```
// our mechanism
persistent class Deposit {
public :
    number account_number;
    money show_amount();
    time show_date();
    void put_money();
    ....
};
```

Interfaces do not have the member property keywords in them; `public` is assumed.

To allow different application programs to get information by sharing only interfaces, each interface declaration should not contain implementation names in their specification of the attributes/methods. That is, in order to make users understand the schema only with the set of interfaces, each interface declaration should be constructed with only the names common to those applications, such as primary types or the interfaces themselves. This property, named the *self-containment* of the set of interfaces, enables users to access data without knowing concrete implementations.

A definition of an implementation is similar to a non-database class except for the additional keyword `implements` which binds to an interface. This allows one-to-many mappings from interfaces to implementations. For example, implementations of `Deposit` may be:

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

or

```
class Money_Deposit {   // implementation
    implements Deposit;
    ...
};
```

The attributes/methods declared in the corresponding interface can be omitted or rephrased. If omitted, the declarations in its interface are automatically copied by the system.[1]

In our approach, both interfaces and implementations are user-defined types. Using interfaces as types is similar to using classes or tables as types in existing OODBMSs and RDBMSs, which ensures the right use of the database classes/tables. Using implementations as types is also helpful for detecting errors in the use of implementations, ahead of the run time.

Here, we give a formal description of interfaces and implementations of our approach. Some definitions of domains are given before proceeding further. A set of system-defined types such as `int`, `float` and `char` is *BT*. The domain of the names of interfaces is *DNI*, that of implementations is *DNM*, and that of the attribute names and method names is *DNSM*. It is assumed that each interface or implementation has a unique name.

In what follows the user-defined interfaces and implementations of schema classes are defined.

DEFINITION 1. (Set of interfaces and set of implementations) *A user-defined set of interfaces I is a pair* $I = \langle IN, \nu \rangle$ *s.t.*

$$IN \subseteq DNI$$
$$\nu : IN \rightarrow 2^{DNSM}$$

where ν *is a function which maps each interface name to a set of attributes/methods.*

A user-defined set of implementations M is a pair $M = \langle IM\delta, \theta \rangle$ *s.t.*

$$IM \subseteq DNM$$
$$\delta : IM \rightarrow 2^{DNSM}$$
$$\theta : IM \rightarrow 2^{DNSM}$$

[1] Currently, unless they are inherited from the superclasses, we assume that all the methods of the implementation are always accompanied by their code bodies.

where δ is a function which maps each implementation name to a set of private attributes/methods, and θ is a function mapping each implementation name to a set of public attributes/methods.

IN is the set of user-defined interface names. *I* defines the set of user-defined interfaces, by the names accompanied by their attribute/method names. The attributes and methods of an interface named i can be obtained by $\nu(i)$. *IM* is the set of user-defined implementation names. All the data and methods of an implementation m can be obtained from $\delta(m) \cup \theta(m)$. *M* defines the set of user-defined implementations, by the names accompanied by the public/private attribute/method names of them. Usual OOPL classes belong to *IM*. We call an element of $IN \cup IM$ a *class*.

In the following definitions, Ξ_I and Ξ_M are respectively the sets of type specifications used in the declarations of attributes/methods of interfaces and of implementations.

DEFINITION 2. (Type specifications for interfaces) *A set Ξ_I is defined inductively as follows:*

(i) *if $a \in BT$, then $a \in \Xi_I$,*
(ii) *if $a \in IN$, then $a \in \Xi_I$,*
(iii) *if $a_0, \ldots, a_n \in \Xi_I$, then function $(n, a_0, \ldots a_n) \in \Xi_I$,*

where an application of the function `FUNCTION` *to $\langle n, a_0, \ldots, a_n \rangle$ yields $\langle$*`FUNCTION`*$, n, a_0, \ldots, a_n \rangle$ for any $a_i \in \Xi$ $(0 \leq i \leq n)$, such that a_0 means the result type-specification, and $a_1, \ldots a_n$ mean argument type specifications for the function.* `FUNCTION` *is a system-defined symbol used in the type specifications.*

DEFINITION 3. (Type specifications for implementations) *A set Ξ_M is defined inductively as follows:*

(i) *if $a \in BT$, then $a \in \Xi_M$,*
(ii) *if $a \in (IN \cup IM)$, then $a \in \Xi_M$,*
(iii) *if $a_0, \ldots, a_n \in \Xi_M$, then function $(n, a_0, \ldots a_n) \in \Xi_M$,*

under the same conditions described in Definition 2.

Note that we define Ξ_I and Ξ_M separately. According to the self-containment property mentioned above, only the interfaces and primary types are allowed to be the type specifications of interfaces. The elements of *IN* or *IM* shown in a type specification are called *components* of the type specification.

2.2. Acceptable binding

At the `implements` phrase in each implementation declaration, the definition of the implementation and its bounded interface are checked if the user-defined binding destroys the type safety. All destructive bindings are rejected, otherwise we call them *acceptable* bindings. By a check on the involved definitions of interface and the implementation, the acceptability of a binding can be syntactically confirmed. For example, we could use the sufficient condition for the acceptable binding that all attributes/methods in the involved interface are also defined in the corresponding implementation in the same way, that is, with the same type specifications.

However, this condition is so restrictive for database applications that the implementation could not have attributes/methods of implementation types freely: when the attributes/methods belong to the interface, an implementation bound to it can have them only with interface types or primitive types, with self-containment property of interfaces. The implementation therefore will miss the chance of more concrete description of the attributes/methods, by forcing the interface and the implementation to have the same type specification for their common attributes/methods.

Consequently, to make it possible to use implementations and interfaces as type specifications in the implementation, we make the condition for acceptable bindings less strict as follows: (1) all attributes/methods in the interface declaration exist also in the implementation declaration; and (2) the type specifications of the attributes/methods in the implementation are identical to their corresponding type specifications in the interface, or in other binding relationships with their corresponding type specifications in the interface, if they are not method argument type specifications. Note that only the type specifications of data, and the result of methods in an interface, are allowed to have variants in the corresponding implementations. Otherwise, the type correctness could be violated by the covariance in the argument type specifications [14].

For example, the implementation `256Color` and the implementation `256ColorImage` are bound to the interfaces `Color` and `ColorImage` respectively, in the following definition.

```
persistent class  Color{ ... };
class 256Color{ ... implements Color; ... };
persistent class ColorImage {
     ...
     Color color();         // return type is Color
};
class 256ColorImage {
     ...
   implements ColorImage;
     256Color color();      // 256Color is an
                               implementation of Color
     ...                    // others are same as in the
                               ColorImage
};
```

The return type of the method `color()` is `Color` in `ColorImage`, but is `256Color` in `256ColorImage`. However, since `256Color` is already bound to `Color`, the binding of `256ColorImage` and `ColorImage` will not be rejected, if `256Color` and `Color` are already accepted (see Figure 1).

To describe user-defined bindings in a formal way, we define a function $\mathcal{Z}$ which inductively maps each element of $\Xi_I \cup \Xi_M$ to its components according to the definition of $\Xi_I \cup \Xi_M$.

DEFINITION 4. (Names in a type specification) *For every $ty \in \Xi_I \cup \Xi_M$, a function $\mathcal{Z}$: $\Xi_I \cup \Xi_M \mapsto 2^{IN \cup IM}$ is defined by recursion on ty:*

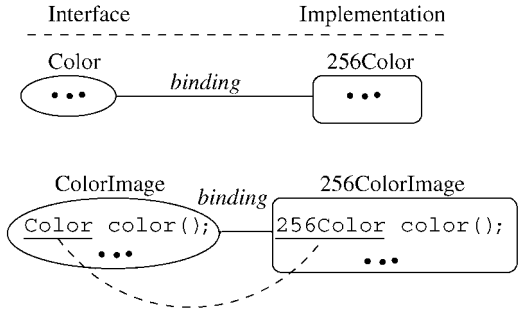

FIGURE 1. Example acceptable classes.

(i) *if* $ty \in BT$, *then* $\mathcal{Z}(ty)$ *is* ϕ,
(ii) *if* $ty \in (IN \cup IM)$, *then* $\mathcal{Z}(ty)$ *is* $\{ty\}$,
(iii) *if there is* $a_0, \ldots, a_n \in \Xi_I \cup \Xi_M$ *s.t.,* $ty = function(n, a_0, \ldots, a_n)$, *then* $\mathcal{Z}(ty)$ *is* $\mathcal{Z}(a_0)$.

The interfaces or implementations used in the specification of the data sm of a class m are represented by $\{x \mid x \in \mathcal{Z}(\mathcal{P}(m, sm))\}$. As for functions, $\mathcal{Z}$ considers only their result type specifications.

Then, we introduce a function $\mathcal{P}: (IN \cup IM) \times DNSM \mapsto \Xi_I \cup \Xi_M$ which maps the name of an attribute/method of a class to the type specification of the attribute/method. That is, $\mathcal{P}(m, f)$ symbolizes the type specification of an attribute/method f of a class m. $\mathcal{P}$ is defined by users during the class declaration time. Now, the definition of the functions π_I and π_M, which describes the set of declarations of the attributes/methods in a given interface and an implementation respectively are in order.

DEFINITION 5. (Set of data and methods of an interface and an implementation) *A function* $\pi_I: IN \mapsto DNSM \times \Xi_I$ *and a function* $\pi_M: IM \mapsto DNSM \times \Xi_M \times \{PUBLIC, PRIVATE\}$ *are defined by the following equations:*

(i) *for all* $i \in IN$, $\pi_I(i) = \{\langle sm, \mathcal{P}(i, sm)\rangle | sm \in \nu(i)\}$,
(ii) *for all* $m \in IM$, $\pi_M(m) = \{\langle sm, \mathcal{P}(m, sm), PUBLIC\rangle | sm \in \theta(m)\} \cup \{\langle sm, \mathcal{P}(m, sm), PRIVATE\rangle | sm \in \delta(m)\}$.

This definition is useful for defining the acceptability of bindings.

DEFINITION 6. (Set of acceptable bindings) $\trianglerighteq \subseteq IN \times IM$ *is a subset of user-defined bindings with the following properties: if* $\langle i, m\rangle \in \trianglerighteq$, *for each* $d_1 = \langle sm_1, t_1\rangle \in \pi_I(i)$, *there is a corresponding* $d_2 = \langle sm_2, t_2, pr\rangle \in \pi_M(m)$ *s.t.* $sm_1 = sm_2$, $pr = PUBLIC$, $\mathcal{Z}(t_1) = \{x_1, \ldots, x_n\}$, $\mathcal{Z}(t_2) = \{y_1, \ldots, y_n\}$ *and for all* x_i, *there is corresponding* y_i *s.t.*

(i) $x_i = y_i$ *or* $\langle x_i, y_i\rangle \in \trianglerighteq$ *and*
(ii) $t_1[y_1/x_1][y_2/x_2]\ldots[y_n/x_n] = t_2$.

2.3. Related works and discussions

Galileo [15, 16], one of the early active OODBMSs, provides three object constructors. Among them, *abstract types* and *concrete types* correspond to our interfaces and implementations, respectively. However, Galileo does not preserve the self-containment property of the abstract types.

The ODMG-93 object model [17], a *de facto* standard object model suggested by OMG (object management group), is also based on the interface/implementation separation idea. An ODL (object definition language) [17] is a specification language which is not dependent on any specific application programming languages. It requires that another application programming language such as C, C++, or Smalltalk implements the classes defined in the ODL [17]. An ODL class may have more than one implementation in different programming languages. Thus, an ODL class represents an interface of a schema class, while the corresponding classes in programming languages are similar to implementations in our approach. However, bindings of implementations and ODL classes are preset in ODMG-93, and the programmers consider the implementation and the corresponding ODL class identical. This loses the flexibility of one-to-many mappings, which is discussed further in the next section.

The interfaces and implementations in our approach look like those in Java and CORBA. However, we propose a distinctive way of integrating database classes and general purpose OOPLs by using interfaces as persistent classes. This allows database programming under a single object model, even in general-purpose OOPLs. This approach contrasts with the current OOPL extensions providing both CORBA and OODB interface [18, 19, 20], where the users are expected to be used to the semantics of database classes and that of separate classes (interfaces and implementations), simultaneously.

3. HIERARCHIES

3.1. Distinct hierarchies in interfaces and in implementations

In our approach, each database application program maintains two hierarchies—one for the schema interfaces and the other for the implementations and non-database OOPL classes. The hierarchies for interfaces are based on is-A relationships, that is, subset relationships of their instance sets, while those for implementations are for the reuse of the code bodies.

For example, we can define the interface `Deposit` as a subclass of another interface as follows.

```
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();
                                 ...};
```

The implementation `Money_Deposit`, which implements the interface `Deposit`, can be placed in a hierarchy,

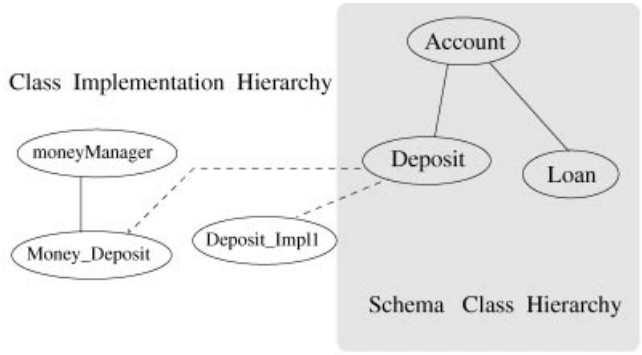

FIGURE 2. Separated class hierarchies for interfaces and implementations in the example.

as follows. This is independent of the schema interface hierarchy including `Deposit`, `Account` and `Loan`. Assume that the class `moneyManager` is an existing class for managing the data of the `money` type.

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

These example hierarchies are depicted in Figure 2. Each of the two hierarchies bears their own inheritance relationships. Both hierarchies for interfaces and those for implementations identify subtyping.

Next, we provide the definition of $\prec_I$ and $\prec_M$, which specifies type-correct and user-defined inheritance relationships for interfaces and implementations, respectively.

DEFINITION 7. (Inheritance relationships of interfaces and implementations) *$\prec_I \subseteq IN \times IN$ is a subset of the user-defined inheritance relationship set with the property that if $\langle i_1, i_2\rangle \in \prec_I$ and $\pi_I(i_2) \subset \pi_I(i_1)$. $\prec_M \subseteq IM \times IM$ is a subset of the user-defined inheritance relationship set with the property that if $\langle m_1, m_2\rangle \in \prec_M$, $\pi_M(m_2) \subset \pi_M(m_1)$.*

The above definition shows that the re-definition with inheritance follows the no-variance rule [21]. Access specifiers like `PRIVATE` and `PUBLIC` cannot be re-defined. We let $\prec_I^*$ and $\prec_M^*$ denote the transitive and reflexive closure of $\prec_I$ and $\prec_M$, respectively.

3.2. Related works and discussions

Our approach has an explicit subclassing mechanism, which means that the subtyping is identified only by the user-defined inheritance specification and the transitivity rule. In some of the existing programming languages [2, 3, 22, 23, 24], subtyping is identified by the structural conformity, based on signatures of the classes. However, it is more attractive to us to distinguish subtyping by explicit user specification of inheritance, since implicit subtyping may introduce inadvertent relationships between schema classes, contrary to users' intentions [25]. Especially for database applications, subtyping based on explicit user-specification is preferred, so that the hierarchies defined by schema designers can be made meaningful in the type system. Moreover, explicit subtyping of implementations is more useful when the features of the database are integrated with a base language like C++ which supports explicit subtyping.

Due to the decoupling of interface hierarchies and implementation hierarchies, schema evolution in our approach is made simpler. For example, let us assume that there is an interface of a persistent class `Deposit` which has multiple implementations named `Deposit_Impl1`, `Deposit_Impl2` and `Deposit_Impl3`. In existing OODBMSs without interface/implementation separation, such implementations have to be made subclasses of `Deposit` in a class hierarchy. This can be represented in a C++ like syntax as follows.

```
// a schema class
  persistent class Deposit { ... };
// various ways of implementing Deposit
  persistent class Deposit_Impl1: virtual Deposit
                                          { ... };
  persistent class Deposit_Impl2: virtual Deposit
                                          { ... };
  persistent class Deposit_Impl3: virtual Deposit
                                          { ... };
```

At this time, if a new interface `SpecialDeposit` is created as a subclass of `Deposit` in the schema, it should be inherited from the three classes.

```
  // a subclass of Deposit
persistent  class SpecialDeposit:
  Money_Deposit, Deposit_Impl1, Deposit_Impl2 { ... };
```

In such a case, ambiguities [26] may arise if any pair of those three subclasses happen to share names of attributes/methods, which is not rare, and users have to override them in the class `SpecialDeposit`.

In our approach, such a schema class can be added in a simpler and more elegant way. Since a hierarchy for implementations can be built regardless of the interface hierarchy, `DepositImpl1`, `DepositImpl2` and `DepositImpl3` in the above example do not have to be subclasses of `Deposit` any more. Instead, they are bound to `Deposit`.

```
class DepositImpl1{implements Deposit;...};
class DepositImpl2{implements Deposit;...};
class DepositImpl3{implements Deposit;...};
```

Since the changes in the implementation hierarchy do not affect the interface hierarchy, and *vice versa*, the new interface `SpecialDeposit` can inherit from `Deposit` directly, without consideration of the implementations of `Deposit`.

```
persistent class SpecialDeposit: Deposit{...};
```

Also, when users want to add/delete/modify the private attributes/methods in the implementation of the class `Deposit`, they do not have to change the whole class declaration or application programs. Instead, they are supposed to change only the specific implementation, or add a new implementation of the interface and use it from then on. Although this point is also applicable to general non-database programming, it is especially useful for database programming where the degeneration of schema evolution cost is important.

4. IMPLEMENTING RELATIONSHIPS AND SUBTYPING

4.1. Side-effect of the implementing relationships

Similar to other database programming languages [8, 10, 27], a database object in our approach is created through an implementation and handled by an *object handler* in application programs. The type of such an object handler is the pointer to an interface type, and the actual implementation of the object does not have to be known to the users of the object handlers. For example, if the interface `Deposit` is implemented by both `Deposit_Impl1` and `Deposit_Impl2`, instances can be created and used as follows ('`obase`' means the name of an objectbase).

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

By the object handler `x`, all attributes/methods described in `Deposit` can be accessed, whether the actual implementation is `Deposit_Impl1` or `Deposit_Impl2`. Such assignment statements should be allowed only if `Deposit_Impl1` and `Deposit_Impl2` are in implementation relationships with the interface `Deposit`.

However, if the implementation `DepositImpl1` implements the interface `Deposit`, this does not simply mean that `DepositImpl1` is bound to the interface `Deposit`. Let us consider more cases of implementation relationships.

First, by the subset property of the schema hierarchy in OODBMSs, an object of an interface is also considered an instance of the superclasses of the interface. Thus, `DepositImpl1` also implements the interface `Account`, if the interface bound to the implementation `DepositImpl1` is a subclass of the interface `Account`.

Second, although it appears that `DepositImpl1` implements `X` if one of the superclasses of `DepositImpl1` is bound to the interface `X`, it is not always true. It is because both the subset relationships between interfaces and the reuse relationships between implementations are involved in the type system as subtyping, and because such mixing might cause unexpected side-effects by transitivity of the subtyping. For example, consider the schema interface `Zoo_Animal` which represents animals in a zoo, bound to the implementation `biologicalinfo_record` in the following example. Of course, the object created by `biologicalinfo_record` also belongs to `Zoo_Animal`.

```
persistent class Zoo_Animal {...};
class biologicalinfo_record { implements Zoo_Animal;
                              ... };
```

Now, in addition to `biologicalinfo_record`, let us consider some other implementations, such as `biologicalinfo_and_price_record` and `biologicalinfo_and_feeding_record` for the interface `Zoo_Animal`. Such implementations are often defined with inheritance from the existing implementation `biologicalinfo_record` for code reuse.

```
class biologicalinfo_and_feeding_record:
    public biological_record{...};
class biologicalinfo_and_price_record:
    public biological_record{...};
```

Now, let us assume that there is a new schema interface `Person` and an implementation `biologicalinfo_and_intelligence_record` is bound to `Person`. Also, it is possible that `biologicalinfo_and_intelligence_record` is defined with inheritance from the existing implementation `biologicalinfo_record` (Figure 3);

```
persistent class Person {...};
class biologicalinfo_and_intelligence_record :
    public biological_record
    { implements Person; ...};
```

However, the object might be used as an instance of a wrong interface, if we make intuitive use of both bindings and subclass relationships for subtyping. In the following example, an instance of `Person` will be handled by the handler for the `Zoo_Animal`.

```
// create a 'Person' object
biologicalinfo_and_intelligence_record * p2 =
    new(obase) biologicalinfo_and_intelligence_record; //
    a Person
biologicalinfo_record * p3 = p2;
Zoo_Animal * q0 = p3;       // a wrong assignment
```

This code segment shows that it is expected the `Person` will be handled by the handler for the `Zoo_Animal` object. However, a `Person` object should not be considered as a `Zoo_Animal` object, since there exists no is-A relationship between `Person` and `Zoo_Animal` in the schema definition. Actually, such a wrong assignment is due to the implementations of the two classes, `biologicalinfo_and_intelligence_record` and `biologicalinfo_record`, which are related to each other by subclassing for code reuse.

For database applications, such a result causes more serious side-effects. Let us consider an additional example:

```
Set<Zoo_Animal> * S;
Zoo_Animal * x;
forall(x << S) {
    x->...
    ...
};
```

In the case of the automatic management of the extents of schema classes, where the extent of a class is obtained from the database type system, the objects of `biologicalinfo_and_intelligence_record` are in the extent of `Zoo_Animal`; they can be handled by the `Zoo_Animal` pointers. Moreover, the extent of `Person` becomes a subset of `Zoo_Animal`, which means a `Person` *is* a `Zoo_Animal`. Thus, to ensure the correct results of queries, we have to investigate how such wrong assignments may be avoided.

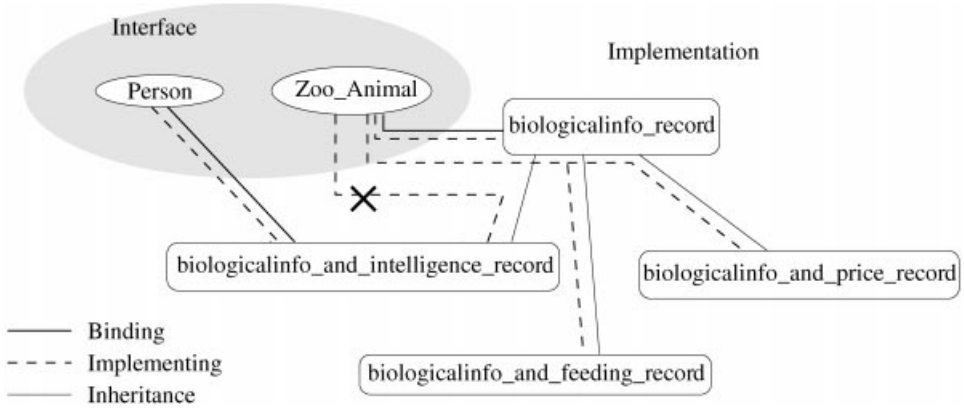

FIGURE 3. Example of inheriting from implementations.

TABLE 1. General subtyping rules.

[Refl]	$\dfrac{}{\vdash \tau \leq \tau}$
[Trans]	$\dfrac{\Sigma; \vdash \tau \leq \tau' \text{ and } \tau' \leq \tau''}{\vdash \tau \leq \tau''}$

TABLE 2. Subtyping rules.

[I-Sub]	$\dfrac{\Sigma; \vdash i_1 \prec_I^* i_2}{\vdash i_1 \leq i_2}$
[M-Sub]	$\dfrac{\Sigma; \vdash m_1 \prec_{M+}^* m_2}{\vdash m_1 \leq m_2}$
[I-M-implementing]	$\dfrac{\Sigma; \vdash i \trianglerighteq^* m}{\vdash m \leq i}$

We propose that if an implementation has the `implements` phrase, it is semantically related to the bound interface; but the inheritance should be considered as the reuse of the code of the superclasses. For example, `biologicalinfo_and_intelligence_record`, with `implements` phrase, does not implement `Zoo_Animal`, which is an interface of its superclass `biologicalinfo_record`. In other words, only the implementations without the `implements` phrase are considered to be the implementation of the interfaces of their superclasses. These implementing relationships can be summarized as follows.

- An implementation implements the interface to which it is directly bound.
- An implementation implements the superclasses of its directly bound interface.
- An implementation implements all the interfaces of its direct superclasses, only when it does not have the `implements` phrase in its definition.

Thus, implementing relationships are represented formally as follows.

DEFINITION 8. (Implementing relationships: $\trianglerighteq^*$) *If* $\langle m_2, m_1\rangle \in \prec_M^*$ *and* $\langle i, m_1\rangle \in \trianglerighteq^*$, *and there is not* k *s.t.* $\langle k, m_2\rangle \in \trianglerighteq$, *then* $\langle i, m_2\rangle in \trianglerighteq^*$.

4.2. Type system

In our approach, the side effect of implementation relationships is avoided by the type system.

The domains mentioned in the previous sections can be applied as the domains to the typing rules shown in Tables 1 and 2. Table 1 shows general rules for a type system, while Table 2 describes typing rules based on the relationships mentioned in earlier sections. If $+$ under the $\prec_M^*$ is ignored temporarily, then the environment Σ represents $\langle I, M, \prec_I^*, \prec_{M+}^*, \trianglerighteq^*\rangle$. In fact, $+$ in $\prec_{M+}^*$ is introduced to preserve type correctness with the typing rules in Tables 1 and 2.

If $\prec_{M+}^*$ simply denotes $\prec_M^*$, this expression does not cause any type error, which allows a wrong handler (of the type `Zoo_Animal`) for the `Person` object in the previous example.

THEOREM 1. (Type conflict) *If* $\prec_{M+}^*$ *denotes* $\prec_M^*$, *the type system with the typing rules in Table 2 and the domains defined in this section are not correct.*

Proof. For implementations m_1 and m_2 such that $\langle m_2, m_1\rangle \in \prec_M^*$, and interfaces i_1 and i_2 such that $\langle i_1, m_1\rangle \in \trianglerighteq$, $\langle i_2, m_2\rangle \in \trianglerighteq$ and $\langle i_2, i_1\rangle \notin \prec_I^*$, m_2 is a subtype of i_1 by the rule of [I-M-implementing], [M-sub] and [Trans].

However, according to the definition of $\trianglerighteq^*$, $\langle m_2, i_1\rangle \notin \trianglerighteq^*$. So, m_2 cannot be a subtype of i_1 by [I-M-implementing], which contradicts the above statement. □

This stems from the fact that subtyping between implementations is also affected by the subtyping from implementing relationships.

To consider an implementation as a subtype of its superclass only when it can implement all the interfaces

of the superclass, we revise subtyping rules on the implementation hierarchies, by defining $\prec_{M+}$ from $\prec_M$.

DEFINITION 9. (Restricted type relationships between implementations) *For all m_1 and $m_2 \in M$, $\langle m_2, m_1\rangle \in \prec_{M+}$, iff $\langle m_2, m_1\rangle \in \prec_M$, and there exists no k such that $\langle k, m_1\rangle \in \trianglerighteq$.*

$\prec^*_{M+}$ denotes the transitive closure of $\prec_{M+}$.

Although the proof of the general type correctness of the type system is an open problem, the type correctness of the present model can be proved as follows.

THEOREM 2. (Type correctness) *In the type system with the typing rules in Table 2 and the domains in this section including the definition of $\prec_{M+}$, type crash arising in Theorem 1 does not occur.*

Proof. For the m_1, m_2, i_1, and i_2 defined in the proof of Theorem 1, m_2 is not a subtype of m_1 by the definition of $\prec^*_{M+}$, which disables applying the [M-sub] rule in Table 2. Thus, the contradiction in Theorem 1 cannot be made. □

4.3. Related work and discussion

Thor [28, 29], an OODBMS supporting the interface/implementation separation from scratch uses its own language Theta [28, 30]. In Theta, every implementation is forced to have the `implements` keyword explicitly, which means that no implementing relationships are considered except the user-defined bindings. In addition, implementation hierarchies, unlike interface hierarchies, do not affect implementing relationships at all. An implementation in Theta implements only the interface to which it is directly bound. The definition of implementing relationships, denoted by $\trianglerighteq^*$, is as follows.

DEFINITION 10. (Implementing relationships in Theta: $\trianglerighteq^*$) *In Theta, for all $\langle i, m\rangle$ in $IN \times IM$, $\langle i, m\rangle \in \trianglerighteq^*$ iff $\langle i, m\rangle \in \trianglerighteq$.*

According to the definition, $\prec^*_{M+}$ is defined as follows.

DEFINITION 11. (Restricted type relationships between implementations in Theta) *In Theta, for all $\langle m_2, m_1\rangle \in \prec^*_M$, $\langle m_2, m_1\rangle$ is in $\prec^*_{M+}$ iff there is i_2 and i_1 s.t. $\langle m_2, m_1\rangle \in \prec^*_M$, $\langle i_1, m_1\rangle \in \trianglerighteq$, $\langle i_2, m_2\rangle \in \trianglerighteq$ and $\langle i_2, i_1\rangle \in \prec^*_I$.*

However, in this approach, non-database classes and the implementations without the `implements` phrase are disallowed for use as subtypes of their superclasses [28, 29].

THEOREM 3. (Types of non-database classes in Theta) *If the base language has their own subtyping rules like C++, non-database classes cannot be involved in the type system of Theta, without type correctness.*

Proof. Since non-database classes do not have the implementation phrase, it is impossible for the non-database classes to satisfy the condition in Definition 11. □

In our approach, all the implementations including the ones without the `implements` phrase and the non-database classes can be subtypes of their superclasses, whenever they satisfy the conditions mentioned earlier. In addition, the following property enables our approach to be used as DBPLs based on the general purpose OOPLs.

THEOREM 4. (Type systems for non-database classes) *In our approach, the type system in this paper follows the subtyping rules in the base languages, in the non-database programs without interface/implementation separation.*

Proof. Since the non-database classes are regarded as implementations in our approach, they follow the subtyping rules for implementations. However, none of them have the `implements` phrase, so their subtyping is identified by the user-defined class hierarchies of Definition 9. Thus, the type system in non-database programs follows that of the base language, since it is assumed that the user-defined hierarchies identify the type system in the base language. □

Accordingly, Theta is not appropriate for database application programs in general purpose languages such as C++, because usual OOPL classes, which are non-database classes, cannot be used as types in the programs. Thus, our approach is better than Theta for database programming in general-purpose OOPLs like C++. Moreover, Theta is so dedicated to a specific system that it lacks user-friendliness which compared with C++.

In contrast to our approach, in most interface/implementation separation supports in programming languages, implementations do not identify types [2, 31, 32, 33, 34, 35]. Even in the languages that allow the types identified by implementations, implementation hierarchies do not affect subtyping [28, 30]. Such type systems are so simple that it is not necessary to elaborate on the effects of the semantics of implementation relationships on their typing rules in this section. POOL-I [36] and Java [1] are the exceptions we have found, which allow both interfaces and implementations to distinguish types and both interface hierarchies to identify subtyping. Java [1] allows all the relationships inferred from bindings identifying subtyping, which results in the side-effect mentioned earlier. In POOL-I [36] signatures of interfaces and implementations, instead of their names and user-defined relationships, distinguish types, subtyping and implementation relationships. One of the distinguishing features of POOL-I is a collection of identifiers, called *property*, which is augmented to the specification of an interface or an implementation by the programmer, and gives more information than the signature alone. Thus, by defining properties appropriately, the type system in this paper is also achieved. However, in this case, defining properties is so complicated that it will considerably increase the user's burden. For example, to make user-defined inheritance identify the subtyping, the identifiers in the properties of the roots have to be defined in their descendants, and all leaf nodes in the hierarchy have all the identifiers that their direct/indirect superclasses have. It also requires the complicated combination of a massive number of identifiers in order that properties make user-defined bindings, and that implementation relationships be

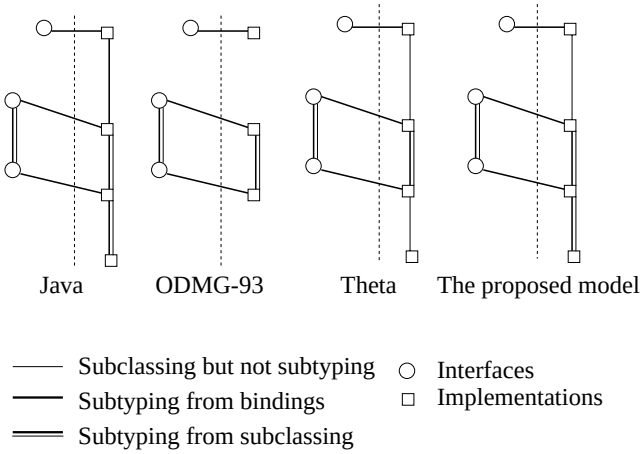

FIGURE 4. Comparisons among the semantics of Java, ODMG-93 and our approach.

able to identify subtyping without the side-effects mentioned in this paper.

The comparisons among the semantics of implementation relationships in Java, ODMG-93, Theta and our proposed separation mechanism are provided in Figure 4. Java is a general purpose language and uses implementations as independent type units, but it uses the naive definition of implementation relationships, which is inappropriate for OODBMSs. The ODMG-93 object model supports class separation, but interfaces and implementations are mapped only on a one-to-one basis in a given program. Our approach presents some restriction on the implementation relationships in order to avoid side-effects. In Theta, the inheritance between the implementations can identify subtyping only when their bound interfaces identify subtyping, which is not appropriate for database application programs based on a general purpose language like C++.

5. CONCLUSION

In this paper, we propose a preprocessing-based approach with its object model for OODBPLs with interface/implementation separation. We define the syntax and the semantics of the interface/implementations definitions in such a way that interfaces represent the users' view of the database schema classes. Interfaces preserve the self-containment property in order to allow the users to access the objects of the class only through the interfaces. The interfaces/implementations identify types in the type system; distinct hierarchies in interfaces and in implementations identify subtyping. This paper suggests a way of subtyping based on the user-defined bindings between interfaces and implementations without side-effects, which is especially useful for database applications where the extents of classes are automatically obtained from the database type system. Although there may be other approaches to achieve interface/implementation separation, we believe that ours is a most reasonable way to the seamless integration of separation semantics and general purpose languages in database programming.

Currently, we realize our approach here on an OODBMS named SOP(*S*NU *O*O-DBMS *P*latform) [5, 37], which was developed between 1992 and 1996 at Seoul National University, based on the ODMG-93 model. We are planning to extend our model to cover class templates.

ACKNOWLEDGEMENT

This work is supported by Brain Korea 21 Project.

REFERENCES

[1] Arnold, K. and Gosling, J. (1996) *The Java Programming Language*. Addison-Wesley, Reading, MA.

[2] Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. (1987) Distribution and abstract types in emerald. *ACM Comput. Surveys*, **19**, 105–190.

[3] Cox, B. J. and Novobilski, A. J. (eds) (1991) *Object-Oriented Programming—An Evolutionary Approach* (2nd edn). Addison-Wesley, Reading, MA.

[4] DEC, HP, HyperDesk, NCR, O Design, and SunSoft (1997) *The Common Object Request Broker: Architecture and Specification*. OMG Group.

[5] Cho, E., Han, S. Y. and Kim, H. J. (1997) A new data abstraction layer required for OODBMS. In *Proc. Int. Database Eng. Applic. Symp.*, Montreal, Canada, pp. 144–148.

[6] Brodnik, A. and Xiao, H. (1992) *Typing in OODBS*. Technical Report, University of Waterloo.

[7] Agrawal, R. and Gehani, N. H. (1989) ODE (object database and environment): the language and the data model. In *Proc. ACM SIGMOD Conf. on Management of Data*, Portland, OR, pp. 36–45. *ACM SIGMOD Record*, **18(2)**.

[8] Agrawal, R. and Gehani, N. H. (1989) Rationale for the design of persistency and query processing facilities in the database programming language O++. In *2nd Int. Workshop on Database Programming Languages*, Portland, OR, pp. 25–40.

[9] Atkinson, M. P. and Buneman, O. P. (1987) Types and persistence in database programming languages. *ACM Comput. Surveys*, **19**, 105–190.

[10] Atwood, T. (1990) Two approaches to adding persistence to C++. In *4th Int. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, pp. 369–383.

[11] Obj (1994) *Objectivity/DB: Getting Started with C++*. Objectivity Inc.

[12] Stroustrup, B. (ed.) (1991) *The C++ Programming Language Second Edition*. Addison-Wesley, Reading, MA.

[13] Richardson, J. E., Carey, M. J. and Schuh, D. T. (1989) *The Design of the E Programming Language*. Technical Report No 824, Computer Science Department, University of Wisconsin-Madison.

[14] Cook, W. R. (1990) Inheritance is not subtyping. In *Proc. SIGPLAN Conf. on Principle of Programming Languages*, San Francisco, CA, pp. 125–135.

[15] Albano, A., Cardelli, L. and Orsini, R. (1985) Galileo: a strongly typed, interactive conceptual language. *ACM Trans. on Database Systems*, **10**, 230–260.

[16] Albano, A., Cardelli, L. and Orsini, R. (1986) *Galileo Reference Manual, VAX/UNIX Version 1.0*. Servizio Editoriale Universitario di Pisa.

[17] Cattell, R. G. G. (1993) *Object Database Standard: ODMG-93*. OMG Group. Morgan Kaufmann.

[18] ION. (1996) *Orbix+ObjectStore Adapter*. IONA Technologies Ltd.

[19] Kilic, E. *et al.* (1995) Experiences in using CORBA for a multidatabase implementation. In *Proc. of the 6th Int. Conf. on Database and Expert System Applications (DEXA)*, London.

[20] Reverbel, F. (1996) *Persistence in Distributed Object Systems: ORB/ODBMS Integration*. PhD Thesis, University of New Mexico.

[21] Stroustrup, B. (ed.) (1997) *The C++ Programming Language Third Edition*. Addison-Wesley, Reading, MA.

[22] Balter, R., Lacourte, S. and Riveill, M. (1994) The Guide language. *Comp. J.*, **37**, 519–530.

[23] Hagimont, D. *et al.* (1994) Persistent shared object support in the guide system: evaluation and related work. *ACM SIGPLAN NOTICE*, **29(10)**, 129–144.

[24] Topper, A. (1994) Object-oriented COBOL standard. *Object Magazine*, **3**, 39–41.

[25] Connor, R., Brown, A., Cutts, Q. and Dearle, A. (1990) Type equivalence checking in persistent object systems. In *Proc. of the 4th Int. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, pp. 154–167.

[26] Ellis, M. A. and Stroustrup, B. (eds) (1990) *The Annotated C++*. Addison-Wesley, Reading, MA.

[27] Lamb, C., Landis, G., Orenstein, J. and Weinreb, D. (1991) The ObjectStore database system. *Commun. ACM*, **34**, 64–77.

[28] Liskov, B. (1993) Specifications and their use in defining subtypes. *ACM SIGPLAN NOTICE*, **28,** 16–28.

[29] Liskov, B. *et al.* (1996) Safe and efficient sharing of persistent objects in Thor. *ACM SIGMOD Record*, **25**, 318–329.

[30] Myers, A. C. (1995) Bidirectional object layout for separate compilation. *ACM SIGPLAN NOTICE*, **30(10)**, 124–139.

[31] Black, A. P. and Hutchinson, N. (1991) *Typechecking Polymorphism in Emerald*. Technical Report, DEC and UCB.

[32] Brancha, G. and Cook, W. (1990) Mixin-based inheritance. *ACM SIGPLAN NOTICE*, **25(10)**, 303–311.

[33] Cardelli, L., Donahue, J. and Glassman, L. (1992) Modula-3 language definition. *ACM SIGPLAN Notices*, **8**, 15–42.

[34] Raj, R. K. *et al.* (1989) *The Emerald Approach to Programming*. Technical Report 88-11-01, University of Washington.

[35] Freeman, S. (1995) Partial revelation and Modula-3: importing only necessary class features. *Dr Dobb's Journal*, **20**, 36–42.

[36] America, P. (1990) A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN NOTICE*, **25**, 161–168.

[37] Ahn, J. and Kim, H. J. (1997) Seof: an adaptable object prefetch policy for object-oriented database systems. In *Proc. of the 13th Conf. on Data Engineering*, Birmingham, UK, pp. 4–13.

[38] Emmerich, W., Kroha, P. and Schafer, W. (1993) Object-oriented database management systems for construction of CASE environment. In *Proc. of the 4th Int. Conf. on Database and Expert System Applications (DEXA)*, Prague, Czech Republic.

[39] Bancilhon, F., Delobel, C. and Kanellakis, P. (1991) *Object-Oriented Database System—The Story of O_2*. Morgan Kaufmann Publishers.

APPENDIX A

An example: CASE repositories

The following is an example of our approach for CASE repositories. The class `increment` represents the subtrees of the abstract syntax graph for a document, which are units of manipulation of the user interface. An example of an increment is a definition of the function `f` with its identifier and parameter list. It is determined by the grammar of the language in which the document is written.

This example is originally from [38] using O_2 [39]. Here we begin with the transformation of the codes into ODMG C++ binding ODL codes. Each terminal and non-terminal symbol of the grammar is translated into a class.

```
class increment {
        Ref<increment>  father;
    public:
        increment(Ref<increment> f);
        Ref<increment>  get_father;
        void set_father(Ref<increment> f);
};
```

Symbols that appear on the right-hand side of an alternative production are transformed into subclasses of the classes representing the symbols on the left-hand sides. When the grammar rule is as follows,

```
parameterlist : identifier| parameter | parameter_list
parameter : cbv
```

the classes are defined thus [38].

```
class parameter_list : public increment{
        List<parameter> pl;
    public:
        parameter_list(Ref<increment> f);
        void add_parameter(Ref<parameter> par);
        void delete_parameter(Ref<parameter> par);
        void insert_parameter(Ref<parameter> par);
        boolean parse(String t; Ref<parameter_list> pl);
        String unparse;
};

class parameter : public increment{
        Ref<identifier> name;
        Ref<identifier> type;
    public
        boolean expand_name(String t);
        boolean expand_type(String t);
        boolean change_name(String t);
        boolean change_type(String t);
        boolean parse(String t, Ref<parameter> p);
        String unparse;
};

class identifier : public increment {
        String value;
    public:
        boolean scan(String t);
        String unparse;
};
```

However, we notice that a function parameter list in a declaration is similar to a function parameter list in a function implementation and a template parameter list in a

definition of a template type. For example, let us consider the case in which a user may want to change the list manipulation of all the parameter lists into a variable length manipulation. In ODMG C++ binding, we have to modify all the method definitions in each definition of the class `parameter_list`, `class parameter_list_decl`, class `template_parameter_list_in_decl` and class `template_parameter_list`. It is also impossible to make the class `parameter_list_in_decl` a subclass of the class `parameter_list` for code sharing, since it destroys the model represented by the schema where a superclass and its subclass mean the symbol on the left-hand side and right-hand side of a production rule, respectively.

In our approach, such modification can be done more elegantly. With interface/implementation separation, the above schema definitions are changed into the ones without private data, and we have another class hierarchy for implementation of the classes. First, the schema definitions in our approach are as follows.

```
persistent class parameter_list : public increment{
        parameter_list(Ref<increment> f);
        void add_parameter(Ref<parameter> par);
        void delete_parameter(Ref<parameter> par);
        void insert_parameter(Ref<parameter> par);
        boolean parse(String t; Ref<parameter_list> pl);
        String unparse;
};

persistent class parameter : public increment{
    // without the declaration of name and type
    ...
};

persistent class identifier : public increment {
    // without the declaration of value
    ...
};

persistent class parameter_list_in_decl :public incre-
ment {...};
persistent class template_parameter_list : public incre-
ment {...};
persistent class template_parameter_list_in_decl : pub-
lic increment{...};
```

The implementations of these schema are as follows. First, we define an implementation `Parameter_List_manipulations` for the list manipulation.

```
class Parameter_List_manipulations{
    List<increment> pl;
    Parameter_List(Ref<increment> f){...}
    void add_parameter(Ref<parameter> par){...}
    void delete_parameter(Ref<parameter> par){...}
    void insert_parameter(Ref<parameter> par){...}
};
```

Then, we make the implementations of the mentioned schema classes derived from it.

```
// for list implementation
class ImplList_parameter_list:
            Parameter_List_manipulations{
    implements parameter_list;
    boolean parse(String t; Ref<parameter_list> pl) {...}
    String unparse;
};

class ImplList_parameter_list_in_decl:
            Parameter_List_manipulations{
    implements parameter_list_in_decl;
     ...
};

class ImplList_template_parameter_list:
            Parameter_List_manipulations{
    implements template_parameter_list;
     ...
};

class ImplList_template_parameter_list_in_decl:
            Parameter_List_manipulations{
    implements parameter_list_in_decl;
     ...
};
```

The modification of the implementations are simply done by the addition of new implementations. Let us consider another implementation `Parameter_Varray_manipulations` for the array manipulation.

```
class Parameter_Varray_manipulations{
    List<increment> pl;
    Parameter_List(Ref<increment> f){...}
    void add_parameter(Ref<parameter> par){...}
    void delete_parameter(Ref<parameter> par){...}
    void insert_parameter(Ref<parameter> par){...}
};
```

Then, by deriving from it, we add the implementations of the mentioned schema classes, as follows.

```
// for list implementation
class ImplVarray_parameter_list:
                Parameter_Varray_manipulations{
    implements parameter_list;
    boolean parse(String t; Ref<parameter_list> pl) {...}
    String unparse;
};

class ImplVarray_parameter_list_in_decl:
                Parameter_Varray_manipulations{
    implements parameter_list_in_decl;
     ...
};

class ImplVarray_template_parameter_list:
                Parameter_Varray_manipulations{
    implements template_parameter_list;
     ...
};

class ImplVarray_template_parameter_list_in_decl:
        ImplVarray_template_parameter_list {
    implements template_parameter_list_in_decl;
     ...
};
```

So there exist two distinct implementations for the class `parameter_list`; `ImplList_parameter_list` and `ImplVarray_parameter_list`. These behave as two

versions of implementations of the class. Similar to this are the cases for `ImplList_parameter_list_in_decl`, `implList_template_parameter_list` and `Impl List_template_parameter_list_in_decl`.

Note that the class `ImplVarray_template_parameter_list_in_decl` is inherited from `ImplVarray_template_parameter_list` instead of `Parameter_Varray_manipulations`. In this case, although `ImplVarray_template_parameter_list_in_decl` is a subclass of `ImplVarray_template_parameter_list`, the instance of the `ImplVarray_template_parameter_list_in_decl`, which is explicitly bound to `template_parameter_list_in_decl`, is not an element of `template_parameter_list` by the definition of implementation relationships.