

# A View Environment to Reuse Class Hierarchies in an Object-Oriented Database System

Masatoshi ARIKAWA  
Department of Computer Science  
and Communication Engineering  
Kyushu University  
Hakozaki 6-10-1, Higashi-ku, Fukuoka 812, Japan  
Phone: (+81)92-641-1101 ext. 5400  
Fax: (+81)92-632-5204  
E-mail: arikawa@csce.kyushu-u.ac.jp

## Abstract

Many studies of interactive environments of object-oriented databases are based on extensions to declarative query languages, designed for relational databases. Such environments do not use well the characteristics of the object-oriented databases.

In this paper, we introduce an environment that allows users to construct tailored class hierarchies, which represent both the users' views and the users' queries to the object-oriented databases. In this environment, users express their queries in the form of class hierarchies. The results of the queries are stored as classes of the tailored class hierarchies.

There are three kinds of class hierarchies; *original class hierarchies* (databases themselves), *domain class hierarchies* (collections of domains), and *tailored class hierarchies* (user-made queries). We define the semantics and the constraints of the three kinds of class hierarchies and the basic operations to evolve them, to modify them, to cut them, to copy them, and to paste them.

## 1. Introduction

Better interactive environments that allow end-users to make queries to object-oriented databases are needed[ABDDMZ89]. Most approaches to the interactive

environments are extensions of the declarative query languages designed for relational databases[CDLR89, CDV88]. The environments do not make good use of the characteristics of the object-oriented databases[KN90]. In this paper, we propose an environment which allows users to construct their tailored class hierarchies, which serve as both the users' views and the users' queries to the object-oriented databases. In this environment, users express their queries in the form of class hierarchies. The results of the queries are stored as classes of the user-made class hierarchies, called "tailored class hierarchies"; their edges represent users' queries, their nodes represent users' views. Moreover, we can evolve the hierarchies using three operations: (1)deriving a new class from the current one, (2)combining current classes to a new class, and (3)changing the common conditions of the plural queries at the same time. We can reuse system-supported class hierarchies (called *original class hierarchies*), *domain hierarchies*, and *tailored class hierarchies* as parts of tailored class hierarchies. In other words, we can reuse class hierarchies (i.e. queries) as well as classes (i.e. views). The tailored class hierarchies are also used to manage methods for instances of the classes in them. We define semantics of those hierarchies and the operations to evolve them, modify them, cut them, copy them, and paste them.

Two major concepts in the object-oriented databases are the management of methods as well as data, and the support of the class hierarchies to maintain methods and data-schemata[ABDDMZ89, LRV88]. The concepts are also supported by our view environment. Our approach is different from the approach used in usual object-oriented databases in the several points, which are summarized in Table 1.

**Table 1. Different points between usual object-oriented databases and our view environment**

|                             | Usual Object-Oriented Databases                            | Our View Environment                                      |
|-----------------------------|------------------------------------------------------------|-----------------------------------------------------------|
| <b>Main users</b>           | Programmers                                                | (Advanced) End users                                      |
| <b>The Style of Queries</b> | Programming languages and declarative languages            | Interactive graphical languages                           |
| <b>Class Hierarchy</b>      | System-supported class hierarchies for general purpose use | Tailored class hierarchies for varied users' purpose use  |
| <b>View Function</b>        | Little research is known [TY88].                           | Our environment is designed to support the view function. |
| <b>Reuse</b>                | Classes                                                    | Classes and class hierarchies                             |

Geographic data are composed of various kinds of data, such as geometric data, statistical data, and time-serial data[VISC87]. The geographic data are good experimental data which we can use to prove that our view environment is useful. Research in the area of geographic databases has important applications in administration-management, market research, car navigation systems, or earth sciences[Barr86, LN87], and it is becoming more and more number. An important characteristic of geographic applications is that we have to construct a large and complex queries to geographic databases, since there are many kinds of geographic objects on a map.

Our recent work dealt with constructing the methodologies to present the data selected by queries to the geographic databases[AIK89, KA90]. In this paper, we discuss an environment suitable for making queries to the geographic databases.

Usually, environments for geographic databases are based on the extensions of relational databases, such as additional functions for processing geometric data or drawing pictures in SQL systems[EF88, RFS88]. The query environments based on the SQL language are not appropriate for an interactive use in geographic applications: the language was mainly designed for use in programming languages, and it is not suitable for complex queries.

One of the good points of relational database systems is to provide users with a flexible view functions[Date90]. The results of applying queries to relations are also relations, which can serve as virtual relations (or views). We can apply other queries to the views (virtual relations) to make a new view. Thus in

the relational database systems we can make complex queries incrementally.

Our view environment provides users with more advanced view functions. If the databases and the views are presented as nodes, and the queries are presented as edges between the nodes on a display screen, we can support a visual query language which allows us to make complex queries directly by manipulating the nodes and the edges[NS88].

Section 2 gives an overview of one application of our view environment. Section 3 defines classes and queries. Section 4 defines the semantics of three kinds of class hierarchies which are made up of the classes and the queries. Section 5 introduces the operations to reuse the class hierarchies. Section 6 describes briefly the mechanism to manage methods in tailored class hierarchies, which are user-defined class hierarchies. In Section 7, we give some concluding remarks.

## 2. Presentation of an Application of Our View Environment

We present an application of our view environment using the example shown in Figure 1. This example shows how to make a tailored class hierarchy step by step. Each subfigure represents the state of the evolution of the tailored class hierarchy.

- (1) This is the initial state of a tailored class hierarchy. A rectangle icon named "Map" represents a tailored class hierarchy and serves as its root class. Two small black rectangles in the icon denote the buttons; the upper button for setting the conditions of a query, the lower button for executing other methods.
- (2) Two attribute domains are set: the "Area" is "Fukuoka City", the "Time" is "Now".
- (3) Two classes, "Coastline" and "Railway", are selected from the menu of available databases.
- (4) The tailored class hierarchy evolves by adding two classes. The two subclasses inherit the attribute domains from the root class "Map". The class "Coastline" is thus the class of "present Coastlines in Fukuoka".
- (5) A new class, "School", is selected from the menu.
- (6) The class "School" is added to the tailored class hierarchy. The attribute domain "Category" of the class "School" is set to be "Elementary School" and "Junior

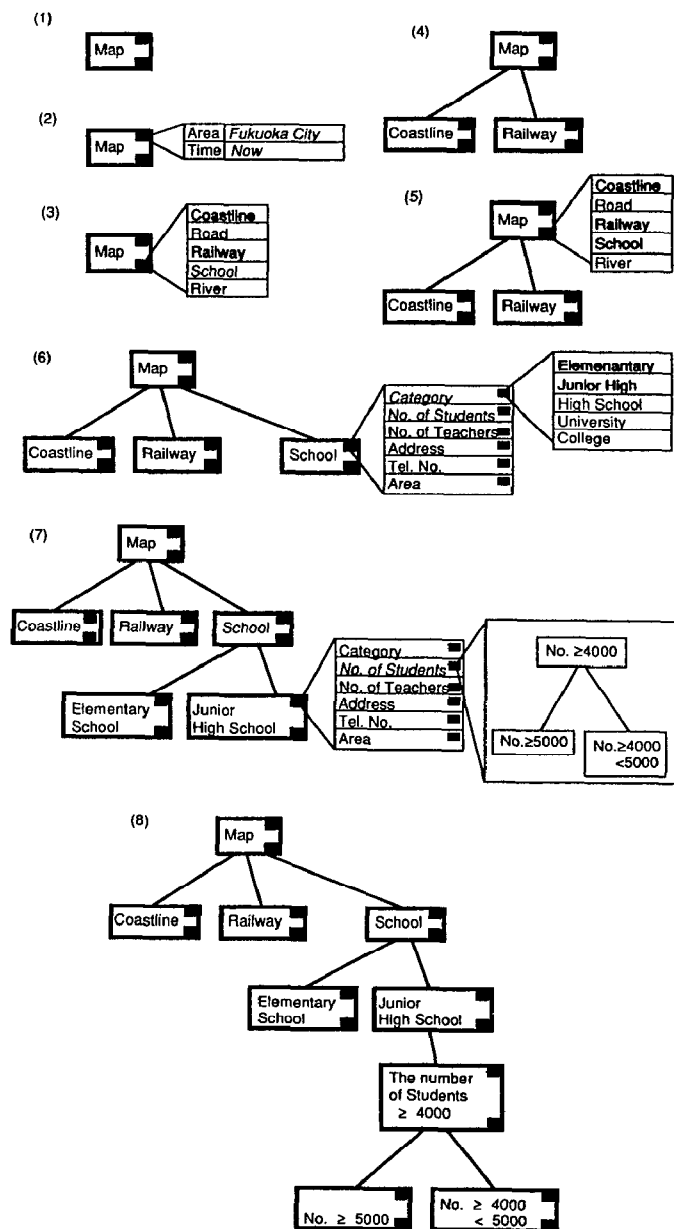


Figure 1. A Tailored Class Hierarchy (II)

High School” using the pop-up menus, in which two kinds of metadata are used; the metadata are the attribute names of the classes or the domain hierarchies of the attributes.

(7) Two classes, “Elementary School” and “Junior High School”, are added to the hierarchy as children of “School”. The class “Junior High School” is the class of “current Junior High School in Fukuoka”. A domain sub-hierarchy is associated with an attribute “No. of Students” of the class “Junior High School”.

(8) The tailored class hierarchy evolves by adding a class sub-hierarchy, in which the root is “Junior High School with more than 4000 students” and its children are “Junior High School with more than 4000 and less than 5000” and “Junior High School with more than 5000 students”.

Thus we can derive new classes from any classes. We can also change plural classes at the same time. For example, if the attribute domain “Area” of the root class is changed from “Fukuoka City” to “Kumamoto City”, we obtain another map composed of the same taxonomical classes as the previous one. Furthermore, this environment supports reuse of the tailored class hierarchies as part of queries; users can make a copy of partial or all parts of the hierarchies, and then add it to the other hierarchies.

We can also set output properties, such as color, font, and line size, to the component classes of the tailored class hierarchy. (Each component class can be considered to be a layer of a map.)

### 3. Components of Class Hierarchies

#### 3.1 Classes and Instances

An *instance* is a set of data that represents an entity. The data comprising the set are called *attribute values*. For example, entities “Kyushu University”, “Narita Airport”, and “Mt. Fuji” are represented by instances. An instance is represented by an unique *instance-id*. Examples of attribute values of an instance are “Kyushu University” as its name, “Hakozaki 6-10-1 Fukuoka” as its address, and a certain polygonal data as its area. The attribute value  $a_j(I)$  denotes the value of the attribute “ $a_j$ ” of the instance having its instance-id “ $I$ ”. Examples of the attributes are “Name”, “Address”, and “Area”. The instances are categorized by *classes*.

The classes allow us to construct and operate complex structured databases easily. The class is also represented by a unique *class-id*, and has some class values, such as “University” for its class name and “[Name, Address, Area, The Number of Students]” for common attributes of its instances. (We use an *id* instead of something to be represented by it. For example, the class  $x$  means the class having its class-id  $x$ .) All instances belong to at least one class. A set of the classes to which an instance  $x$  belongs is denoted by

$class(x)$  (= a set of class-id's). Conversely a class  $x$  is considered to be a set of the instances, denoted by  $instance(x)$ . Thus we treat complex objects[BK86] in this paper.

Each *class domain* is denoted by  $domain(class-id)$  (= a set of tuples of (*attribute*, *attribute-domain*), that is called a *multi-attribute domain*). For example, the attribute-domain of the attribute "Age" of the class "Human" is [0, 150] which means a set of numbers between 0 and 150. The other example is that the attribute-domain of the attribute "Area" of the class "Elementary School in Tokyo Prefecture" is a set of polygonal regions, which may be comprised of some disjoint domains, such as land regions and island regions. An example of the class domain is that the domain of a class "University" is a set  $\{(Name, Text), (Administration, \{Private, The Ministry of Education, City, The Defence Agency, The Ministry of International Trade and Industry, \dots\}), (Location, Region-in-Japan)\}$ . ("Text" means a set of all combinations of characters, and "Region-in-Japan" means a set of all regions within Japan.) A constraint between classes and their instances is that all attribute values of all instances must be within the class domains of their correspondent classes.

### 3.2 Selection Queries and Derived Classes

We will discuss *selection queries* and their resultant classes, called *derived classes*. We define an inclusion relation ( $\supset$ ) between two multi-attribute domains as follows.

$a_1, a_2$ : attribute  
 $d_1, d_2$ : attribute-domain  
 $MultiAttriDom_1, MultiAttriDom_2$ : multi-attribute domain

$MultiAttriDom_1 \supset MultiAttriDom_2$   
 $\Leftrightarrow (\forall (a_1, d_1) \in MultiAttriDom_1) (\forall (a_2, d_2) \in MultiAttriDom_2) ((a_1 = a_2 \wedge d_1 \supset d_2) \vee (a_1 \neq a_2))$

A selection query to a database is denoted by " $SQ(C-id, SpecifiedDomain)$ " which returns a set of instances. A *SpecifiedDomain* is a multi-attribute domain to set the conditions of the selection query. The semantics of  $SQ(C-id, SpecifiedDomain)$  is as follows:

$SQ(C-id, SpecifiedDomain) = \{I-id \mid (I-id \in instance(C-id))$   
 $\wedge (\forall (a, d) \in SpecifiedDomain) (a(I-id) \in d)\}$   
 ( A constraint:  $domain(C-id) \supset SpecifiedDomain$  )

The set of instances resulted from the  $SQ(C-id, SpecifiedDomain)$  is considered to be a new class, called a *derived class*. The classes that are not derived from any classes are called *original classes*. Usual classes correspond to views of databases. The original class correspond to databases themselves. The query of a derived class "DC" is denoted by  $Q-id(DC)$ , which is a *query-id*. The components  $C-id$  and  $SpecifiedDomain$  of a selection query  $SQ$  are denoted by  $C-id(SQ)$  and  $SpecDom(SQ)$  respectively. The domain of a derived class "DC" is defined as  $domain(C-id(Q-id(DC))) \cap SpecDom(Q-id(DC))$ . The intersection operator ( $\cap$ ) between two class domains is defined as follows.

$X, Y$  : multi-attribute domains  
 $X \cap Y =$   
 $\{(a, d) \mid (\exists (a_X, d_X) \in X) (\exists (a_Y, d_Y) \in Y) (a_X = a_Y \wedge a = a_X \wedge d = d_X \cap d_Y)$   
 $\vee (\exists (a_X, d_X) \in X) (\forall (a_Y, d_Y) \in Y) (a_X \neq a_Y \wedge a = a_X \wedge d = d_X)$   
 $\vee (\forall (a_X, d_X) \in X) (\exists (a_Y, d_Y) \in Y) (a_X \neq a_Y \wedge a = a_Y \wedge d = d_Y)\}$

### 3.3 Union Queries and Virtual Parent Classes

In cases such as changing the scale of maps and the time of newspapers, we need functions to change plural derived classes at the same time. Especially in the case of a map, all (geographic) component classes in the map have geometric attributes which express the places they occupy. The geometric attributes have their domains to specify current or available geometric regions. When the display area of the map is changed, the domains of the geometric attributes of all the component classes of the map must be changed. To make such change operations easy, we introduce *union queries*. Union queries are made for two functions: the first one for changing the domains of the common attributes of the plural classes, the second one for making unions of classes. Before defining the union queries, we introduce the partial-assignment operation ( $\oplus$ ) and the union operation ( $\cup$ ) between two multi-attribute domains. These are defined as follows.

$X, Y$  : multi-attribute domains

$$X \oplus Y = \{(a, d) \mid (\exists (a_X, d_X) \in X)(\exists (a_Y, d_Y) \in Y)(a_X = a_Y \wedge a = a_X \wedge d = d_Y) \vee (\exists (a_X, d_X) \in X)(\forall (a_Y, d_Y) \in Y)(a_X \neq a_Y \wedge a = a_X \wedge d = d_X)\}$$

$$X \cup Y = \{(a, d) \mid (\exists (a_X, d_X) \in X)(\exists (a_Y, d_Y) \in Y)(a_X = a_Y \wedge a = a_X \wedge d = d_X \cup d_Y) \vee (\exists (a_X, d_X) \in X)(\forall (a_Y, d_Y) \in Y)(a_X \neq a_Y \wedge a = a_X \wedge d = d_X) \vee (\forall (a_X, d_X) \in X)(\exists (a_Y, d_Y) \in Y)(a_X \neq a_Y \wedge a = a_Y \wedge d = d_Y)\}$$

The union query is denoted by “ $UQ(a \text{ set of } C\text{-id's, SpecifiedDomain})$ ” which returns a set of instances. The “ $a \text{ set of } C\text{-id's}$ ” is a set of classes.  $C\text{-id}(UQ)$  for an union query  $UQ$  denotes  $a \text{ set of classes}$ , while  $C\text{-id}(SQ)$  for a selection query  $SQ$  denotes *only one* class. The resultant class of the union query is called a *Virtual Parent Class*. It is defined as follows.

$VPC$ : virtual parent class

$(X_i, X_j \in C\text{-id}(Q\text{-id}(X_i)), X_i \neq X_j, 1 \leq i, j \leq n, i \neq j,$

$n = \text{the number of elements in } C\text{-id}(Q\text{-id}(X_i)))$

$\text{domain}(X_i) = \text{SpecDom}(Q\text{-id}(X_i)) \oplus \text{SpecDom}(Q\text{-id}(VPC))$

$\text{domain}(VPC) = \cup_{i=1}^n \text{domain}(X_i)$

; this “ $\cup$ ” means an union of multi-attribute domains

$\text{instance}(VPC) = \cup_{i=1}^n \text{instance}(X_i)$

; this “ $\cup$ ” means an union of sets

Virtual parent classes can be child classes of another virtual parent class, if there is no cyclic relation between these virtual parent classes.

The classes  $C\text{-id}(Q\text{-id}(X))$  for a virtual parent class  $X$  is called a *virtual child class*. The changes of the virtual child classes influence those of their virtual parent classes. On the other hand, Classes derived from virtual parent classes are called *derived child classes*.

If the domains of virtual parent classes are changed, the domain of their two kinds of child classes are both changed successively. We define sets of instances which correspond to a given virtual parent class, its virtual child classes, and its derived child classes as follows.

$VPC$ : a given virtual parent class

$VCC$ : one of the virtual child classes

$DCC$ : one of the derived child classes

$UQ_I$ : the query-id of the union query of  $VPC$

$SQ_V, SQ_D$ : the query-id's of selection queries of  $VCC$  and  $DCC$  respectively

$$1. \text{instance}(VCC) = SQ(C\text{-id}(SQ_V), \text{SpecDom}(SQ_V) \cap \text{SpecDom}(UQ_I))$$

$$2. \text{instance}(VPC) = UQ(C\text{-id}(UQ_I), \emptyset); \text{ after } 1$$

$$(\text{SpecDom}(UQ_I), \text{SpecDom}(UQ_I))$$

$$3. \text{instance}(DCC) = SQ(VPC, \text{SpecDom}(SQ_D)); \text{ after } 2$$

## 4. Tailored Class Hierarchies as Complex Views

We introduce *tailored class hierarchies* as complex views. The tailored class hierarchies provide us with a view environment to make long and complex queries incrementally. The tailored class hierarchy consists of classes and queries. There are four kinds of component classes: a *root class*, a *first derived class*, a *new derived class* and a *new root class*. Figure 2 illustrates these classes briefly. They are defined as follows.

(1) The *root class* is unique in the tailored class hierarchy; it does not have any parent classes, and it is a virtual parent class. Its child classes are first derived classes or new root classes.

(2) The *first derived classes* are the classes directly derived from original classes (that is databases) by selection queries or union queries. The component classes on the path between the root class and a leaf class of the hierarchy must include exactly one first derived class.

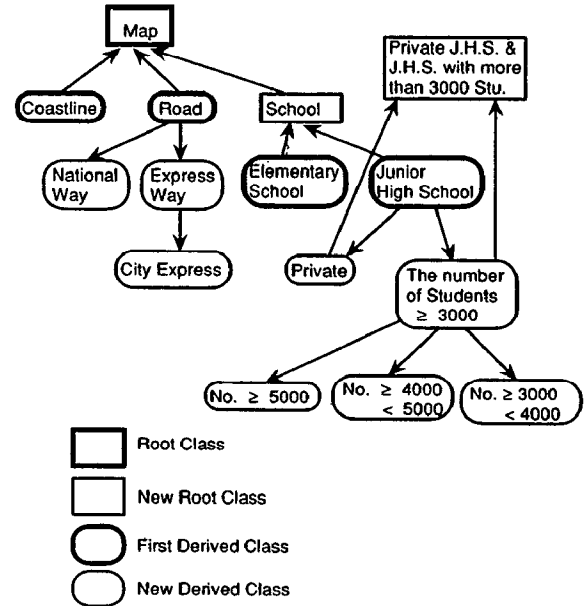


Figure 2. A Tailored Class Hierarchy

(3) All descendant classes of first derived classes are derived classes, and are called *new derived classes*.

(4) All component classes on the path between the root class and a first derived class are *new root classes* which have their virtual parent classes. The new root classes are virtual parent classes. All component classes, except for the root class, can become virtual child classes of the new root classes. The new root classes may be considered as root classes of sub-hierarchies in the tailored class hierarchy. Thus the sub-hierarchies in the hierarchy can evolve by deriving new classes from the new root classes.

The parent-child relationships between the component classes are defined by their inclusion relations: the class domains of parent classes must include the class domains of child class. In Figure 2 the parent-child relationships are expressed with the locations of the classes, where the parents are positioned higher than their children, and with the edges between them.

Let's consider an example of making a tailored class hierarchy for a map application. "Map" is the name of the root class, which is a representative class in the tailored class hierarchy. The domain of the union query of the class "Map" is used to assign common conditions to all component classes in the tailored class hierarchy, such as "Fukuoka City" for the area and "1945" for the time. Usually, first derived classes, such as "River" and "Administration Border Line", are sets of instances imported from existing original classes. New derived classes are made by narrowing the domains of classes. For instance, we derive a new derived class "City Border Line" from a first derived class "Administration Border Line". If we change the domain of the virtual parent class "Elementary School and Junior High School" which are resulted from an union query of "Elementary School" and "Junior High School", the domains of "Elementary School" and "Junior High School" are also changed according to the semantics of the union query.

## 5. Reuse of Class Hierarchies

It is tiresome to make a large and complex tailored class hierarchy by primitive operations, that is selection queries and union queries. We introduce

macro operations to reuse class hierarchies as well as classes. We propose three kinds of class hierarchies for a view environment; *domain hierarchies*, *original class hierarchies*, and *tailored class hierarchies*. The domain hierarchies and original hierarchies are special tailored class hierarchies. The sub-hierarchies copied out of these class hierarchies can be added to the tailored class hierarchies unless the operations do not violate the constraints among classes of the hierarchies described in the Section 3 and 4.

### 5.1 Operations to Reuse Domain Hierarchies

The domain hierarchies are considered as class hierarchies that have no instances. An example of a domain hierarchy is shown in Figure 3. A domain hierarchy is composed of only derived classes, called *domain classes*, and selection queries between them. The domain of the root class of the domain hierarchy includes the domain of any descendant class.

There are various ways to specify sub-hierarchies within a domain hierarchy. One of them is to picking up all the domain classes of user's interesting sub-hierarchy from the domain hierarchy on the screen by the mouse.

A *Copy* operation  $Copy(H)$  creates and returns a new hierarchy ( $H_2$ ) with the same structure of a given hierarchy  $H$ . The  $H_2$  contains new class-id's, new query-id's, the same contents of classes, and the same queries as the source hierarchy  $H$ .

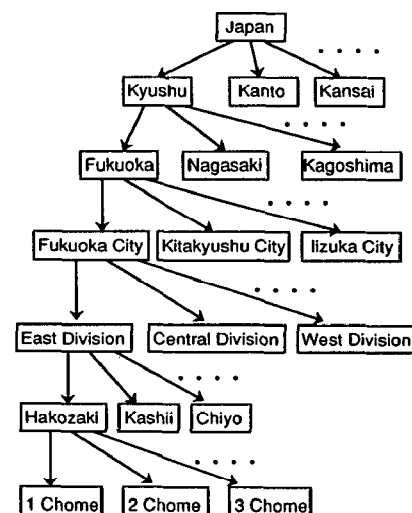


Figure 3. A Domain Hierarchy

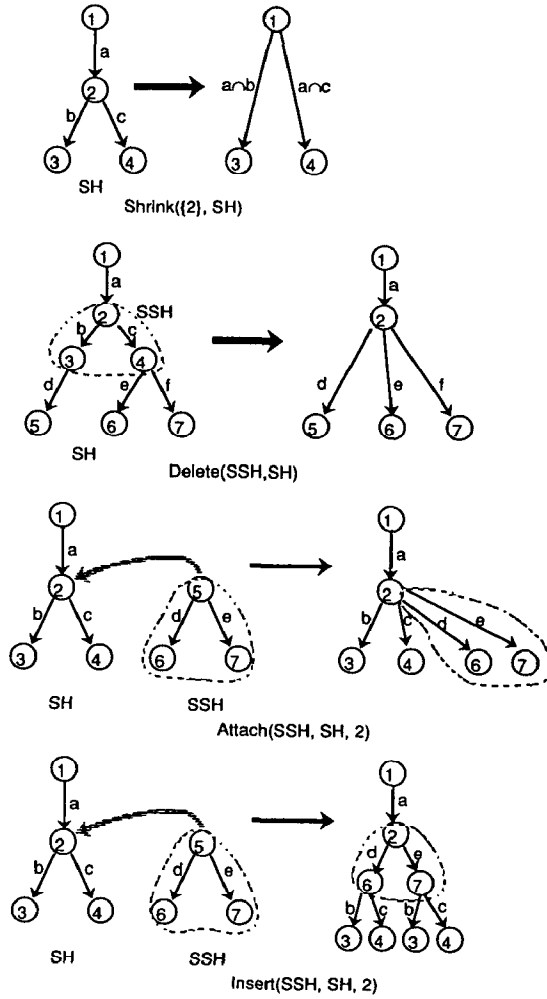


Figure 4. Operations to Modify Domain Hierarchies

We need to modify the (copied) sub-hierarchies to fit our aims. We introduce four basic operations, *Shrink*, *Delete*, *Attach*, and *Insert* (See Figure 4.) The labels on the edges of the sub-hierarchies in Figure 4 represent the SpecifiedDomains of the selection queries correspondent to the edges. Those operations are defined as follows.

(1) The *shrink* operation shrinks paths by removing classes on the paths from the sub-hierarchy. The operation causes no change of classes in the sub-hierarchy other than the removing classes. The shrink operation,  $Shrink(C, SH)$ , to remove a set of classes  $C$  from a sub-hierarchy  $SH$ , is defined as follows.

$C$ : a set of classes (=class-id's) to be removed, which are included by  $C-id(SH)$  and includes neither the root nor leaves of the sub-hierarchy  $SH$ .  
 $C-id(SH)$ : a set of all component classes (=class-id's) which belong to the sub-hierarchy  $SH$   
 $Q-id(SH)$ : a set of all queries (=query-id's) which belong to the sub-hierarchy  $SH$

1. For  $c$  In  $C$ 
  - For  $q$  In  $\{q \mid C-id(q)=c\}$ 
    - Begin
      - $SpecDom(q) = SpecDom(q) \cap SpecDom(Q-id(c))$
      - $C-id(q) = C-id(Q-id(c))$
    - End
2.  $C-id(SH) = C-id(SH) - C$
3. For  $c$  In  $C$   $Q-id(SH) = Q-id(SH) - \{Q-id(c)\}$

(2) The *delete* operation,  $Delete(SSH, SH)$ , to delete a sub-sub-hierarchy  $SSH$  from a sub-hierarchy  $SH$ , is defined as follows.

$Root(H)$ : the root class of a hierarchy  $H$   
 $Leaves(H)$ : the set of all leaf classes of the hierarchy  $H$   
 $SH, SSH$ : a sub-hierarchy

1. For  $c$  In  $Leaves(SSH)$ 
  - For  $q$  In  $\{q \mid c=C-id(q)\}$   $C-id(q)=Root(SSH)$
2.  $C-id(SH) = C-id(SH) - C-id(SSH) \cup \{Root(SSH)\}$
3.  $Q-id(SH) = Q-id(SH) - Q-id(C-id(SSH))$

(3) The *attach* operation,  $Attach(SSH, SH, cc)$ , to attach a sub-sub-hierarchy  $SSH$  to a class  $cc$  of a sub-hierarchy  $SH$ , is defined as follows.

- $cc$ : the current class which is a class of  $SH$
1. For  $q$  In  $\{q \mid Root(SSH)=C-id(q) \wedge q \in Q-id(SSH)\}$ 
    - $C-id(q)=cc$
  2.  $C-id(SH) = C-id(SH) \cup C-id(SSH) - \{Root(SSH)\}$
  3.  $Q-id(SH) = Q-id(SH) \cup Q-id(C-id(SSH))$

(4) The *insert* operation,  $Insert(SSH, SH, cc)$ , which inserts a sub-sub-hierarchy  $SSH$  below an inner class  $cc$  of a sub-hierarchy  $SH$ , is defined as follows.

- $SH, SSH, Desc-cc$ : a sub-hierarchy  
 $Descendant(c, H)$ : the sub-hierarchy below the class  $c$  in the hierarchy  $H$   
 $cc$ : the current class where a sub-sub-hierarchy is inserted below
1.  $Desc-cc = Copy(Descendant(cc, SH))$
  2.  $Delete(Descendant(cc, SH))$
  3.  $Attach(SSH, SH, cc)$
  4. For  $c$  In  $Leaves(SSH)$   $Attach(Copy(Desc-cc), SH, c)$

The sub-hierarchies can be added to the tailored class hierarchies using the “attach” and “insert” operations. The “insert” operations are useful to make the same sub-hierarchies below some different derived classes.

## 5.2 Operations to Reuse Original Class and Tailored Class Hierarchies

The reuse of domain hierarchy means the reuse of queries. On the other hand, the reuse of the original class hierarchies and the tailored class hierarchies means the reuse of sets of instances in addition to queries.

The tailored class hierarchies have the connections to the original class hierarchies with the queries of the first derived classes in them. Parts of the original classes can be copied and added to the tailored class hierarchies as the first derived classes.

The original class hierarchy is composed of three kinds of classes: original classes, derived classes from the original classes, and virtual parent classes based on the original classes (See Figure 5). The queries of the virtual parent classes in the original class hierarchies are supposed to have their empty SpecifiedDomains, that is their SpecifiedDomains are empty sets, since we do not need the function of changing globally the domains of all descendant classes for the original class hierarchies.

Every path from the root class to leaves of the original class hierarchy must have exactly one original class. The ancestors of the original classes are all virtual parent classes. The descendants of the original classes are all derived classes.

First derived classes have their queries describing how they are made of the original classes. The queries are called the *first queries* of the first derived classes.

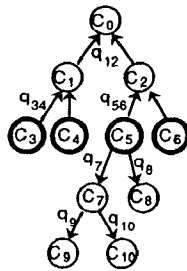


Figure 5. An Original Class Hierarchy

We use Figure 5 to explain the way to copy sub-hierarchies from an original class hierarchy and paste them to a tailored class hierarchy. If we use  $C_5$  as the first derived class, its first query is  $SQ(C_5, \emptyset)$ . If we use  $C_9$  as the first derived class, its first query is  $SQ(C_5, SpecDom(q_7) \cap SpecDom(q_9))$ . If we use  $C_0$  as the first derived class, its first query is  $UQ(\{C_3, C_4, C_5, C_6\}, \emptyset)$ . (The SpecifiedDomains of  $q_{12}$ ,  $q_{34}$  and  $q_{56}$  are all empty.) If we make a copy a sub-hierarchy  $\{C_7, C_9, C_{10}\}$  and paste it to a tailored class hierarchy,  $C_7$  becomes a first derived class, and  $C_9, C_{10}$  become new derived classes. If we use a sub-hierarchy  $\{C_0, C_1, C_2\}$ ,  $C_1$  and  $C_2$  become first derived classes and  $C_0$  becomes a new root class. The same copy and paste operations are feasible in tailored class hierarchies.

In addition to the operations to modify the class hierarchies composed of selection queries, we also introduce the basic operations on virtual parent class hierarchies composed of union queries, that is *Uni-delete*, and *Uni-attach* operations (The shrink and insert operations to the virtual parent class hierarchies, which are prepared for the domain hierarchies, may be nonsense.). The definitions of those operations are as follows.

- (1) The *Uni-delete*( $VPC, SH$ ) to delete a set of the virtual parent classes  $VPC$  from a sub-hierarchies  $SH$  which are composed of virtual parent classes. It is defined as follows.

$VPC$ : a set of virtual parent classes to be deleted, which are included by  $C-id(SH)$  and is not the root  
 $SH$ : a sub-hierarchy of virtual parent classes  
 $C-id(SH)$ : a set of all component classes which belong to the sub-hierarchy  $SH$   
 $Q-id(SH)$ : a set of all queries which belong to the sub-hierarchy  $SH$   
 $C-id(uq)$ : a set of all virtual child classes of the union query  $uq$   
 $Q-id(c)$ : the union query of the virtual parent class  $c$

1. For  $vpc$  In  $VPC$

For  $uq$  In  $\{uq \mid vpc \in C-id(uq)\}$

$C-id(uq) = C-id(uq) \cup C-id(Q-id(vpc)) - \{vpc\}$

2.  $C-id(SH) = C-id(SH) - VPC$

3. For  $vpc$  In  $VPC$   $Q-id(SH) = Q-id(SH) - Q-id(vpc)$

- (2) The *Uni-attach*( $SSH, SH, cc$ ), to attach a sub-sub-hierarchy  $SSH$  composed of virtual parent classes to a virtual parent class  $cc$  of a sub-hierarchy  $SH$ , is defined as follows.

$Root(H)$ : the root class of a virtual parent class hierarchy  $H$   
 $SH, SSH$ : a sub-hierarchy of virtual parent classes  
 $cc$ : the current class to which a sub-hierarchy is attached

1.  $C-id(Q-id(cc)) = C-id(Q-id(cc)) \cup C-id(Q-id(Root(SSH)))$
2.  $C-id(SH) = C-id(SH) \cup C-id(SSH) - \{Root(SSH)\}$
3.  $Q-id(SH) = Q-id(SH) \cup Q-id(SSH) - \{Q-id(Root(SSH))\}$

## 6. Management of Methods in the Tailored Class Hierarchies

The current trendy is to treat data and methods together as objects. This is an important idea of object-oriented approach. Researchers in the field of the object-oriented databases have focused on the methodology to manage methods in addition to the data.

The tailored class hierarchies introduced in this paper should also manage the methods. In the other words, the classes (as views) should also manage their methods.

The methods are classified into (1) the methods for building the tailored class hierarchies and (2) the methods for presenting instances of the classes. Examples of the methods (1) are deriving new classes, using tools to change the class domain (e.g. changing the domain of an attribute "Area" to enlarge a map by setting a window on it as a part to be enlarged), and using the domain hierarchies to evolve the tailored class hierarchies. Examples of the methods (2) are selecting attributes to be displayed, setting presentation attributes such as color, line width, line type, fill pattern, character font, and moving effect.

Methods have some arguments, which are called *method attributes*, and are defined as tuples of (*method-id*, *method-attribute*, *value*). A set of the *method-id*'s of a class  $c$  and the *value* of a *method-attribute*  $ma$  of a method  $m$  of it are denoted by  $Methods(c)$  and  $MAV(c, ma, m)$ , respectively. It is tiresome to set values to many method-attributes of all classes in the tailored class hierarchies. We introduce an inheritance mechanisms of methods and method-attribute values for the tailored class hierarchies as well as the ones of the data schema and the domains of the data which are described in Section 3, 4 and 5. When we create new derived classes, their methods and their method-attribute values of the new derived classes are all the same as those of its parent class. We can change the method-attribute values in any classes. Such

changes of the method-attribute values of the classes usually affect all their descendants. The default (or standard) method-attribute values are considered to be set in the original class hierarchies. The original class hierarchies (and their classes) serve as basic templates of the general taxonomy of the data, the methods, and the default method attribute values.

## 7. Concluding Remarks

It is important for users to have good environments to form their queries to databases, process the data, and manage methods applicable to the data. We introduced a new-style view environment for object-oriented database systems, in which users can evolve tailored class hierarchies (i.e. user-defined class hierarchies, such as long and complex queries) and reuse existing class hierarchies, that is *original class hierarchies* (i.e. databases), *domain hierarchies* (i.e. collections of domains), and *tailored class hierarchies* (i.e. queries).

From the viewpoint of semantic data models[HK88, SS77], our view environment provides users with facilities for utilizing IS-A and IS-PART-OF relationships in their complex views or queries. The IS-A and IS-PART-OF relationships correspond to the selection queries and union queries in our environment.

Many information products, such as maps, newspapers, and TV programs, have been provided and distributed as mass products, because of the limitation of their media and their production technologies. If we apply the view environments, which are introduced in this paper, to the areas of information products, we obtain a new, personal style of information products that are results from users' queries to databases.

## Acknowledgements

I thank Professor Yahiko KAMBYASHI (Kyoto University) for his helps and advice.

## References

- [ABDDMZ89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database System Manifesto," The First International Conference on Deductive and Object-Oriented Databases, 1989, pp. 40-57.

- [AIK89] H. Aonuma, H. Imai, and Y. Kambayashi, "A Visual System of Placing Characters Appropriately in Multimedia Map databases," in *Visual Database Systems*, North-Holland, 1989, pp. 525-546.
- [Barr86] R. Barr, "Automated Cartography and Geographical Information Systems," in *Advances in Computer Graphics II*, Springer-Verlag, 1986, pp. 29-53.
- [BK86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," ACM PODS Conference, 1986, pp. 53-59.
- [CDLR89] S. Cluet, C. Delobel, C. Lecluse, and P. Richard, "Reloop, an Algebra Based Query Language for an Object-Oriented Database System," The First International Conference on Deductive and Object-Oriented Databases, 1989, pp. 294-313.
- [CDV88] M. Carey, D. DeWitt, and S. Vandenberg, "A data model and query language for Exodus," Proc. of ACM Sigmod Conf., 1988.
- [Date90] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Co., 1990.
- [EF88] M. J. Egenhofer and A. U. Frank, "Toward a Spatial Query Language: User Interface Considerations," Proc. of the Fourteenth International Conference on Very Large Data Bases, 1988, pp. 124-133.
- [HK88] R. Hull and R. King, "Semantic Database Modeling: Survey, Application, and Research Issues," ACM Computing Surveys, Vol. 19, No. 3, 1988, pp. 201-260.
- [KA90] Y. Kambayashi, M. Arikawa: "Semantic Overview Functions for Geographic Databases," Proceedings of the Far-East Workshop on Future Database Systems,, 1990, pp. 192-220.
- [KN90] M. Kracker and E. J. Neuhold, "Schema Independent Query Formulation," in *Entity-Relationship Approach*, Elsevier Science Publishers B.V. North-Holland, 1990, pp. 387-401.
- [LN87] U. W. Lipeck and K. Neumann, "Modelling and Manipulating Objects in Geoscientific Databases," in *Entity-Relationship Approach*, Elsevier Science Publishers B.V. (North-Holland), 1987, pp. 67-86.
- [LRV88] C. Lecluse, P. Richard, and F. Velez, " $O_2$ , an Object-Oriented Data Model," ACM SIGMOD Conference, 1988, pp. 424-433.
- [NS88] N. C. Shu, "Languages for Handling Visual Information" (in *Visual Programming*), Van Nstrand Reinhold Co. Inc., 1988, pp. 109-123.
- [RFS88] N. Roussoupoulos, C. Faloutsos and T. Sellis, "An Efficient Pictorial Database System for PSQL," IEEE Transactions on Software Engineering, Vol. 14, No. 5, 1988, pp. 639-650.
- [SS77] J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation and Generation," ACM Transactions on Databases, Vol.2, No. 2, June 1977, pp. 105-133.
- [TY88] K. Tanaka, M. Yoshikawa, and K. Ishihara, "Schema Virtualization in Object-oriented Databases," Proc. of IEEE 4th Annual Int. Conf. on Data Engineering, pp. 23-30, 1988.
- [VISC87] Visualization in Scientific Computing, ACM SIGGRAPH - Computer Graphics, Vol. 21, No. 6, 1987.