Objects and Views*

Serge Abiteboul† Anthony Bonner‡

INRIA, 78153 Le Chesnay, France

Appears in *Proceedings of ACM SIGMOD 1991*, pages 238–247.

Abstract

Object-oriented databases have been introduced primarily to ease the development of database applications. However, the difficulties encountered when, for instance, trying to restructure data or integrate databases demonstrate that the models being used still lack flexibility. We claim that the natural way to overcome these shortcomings is to introduce a sophisticated view mechanism. This paper presents such a mechanism, one which allows a programmer to restructure the class hierarchy and modify the behavior and structure of objects. The mechanism allows a programmer to specify attribute values implicitly, rather than storing them. It also allows him to introduce new classes into the class hierarchy. These *virtual classes* are populated by selecting existing objects from other classes and by creating new objects. Fixing the identify of new objects during database updates introduces subtle issues into view design. Our presentation, mostly informal, leans on a number of illustrative examples meant to emphasize the simplicity of our mechanism.

1 Introduction

Object-oriented databases [3, 14, 20] have been introduced primarily to ease the development of database applications. However, the difficulties encountered when, for instance, trying to restructure data or integrate databases demonstrate that the models being used still lack flexibility. We claim that the natural way to overcome these shortcomings is to introduce a sophisticated view mechanism, i.e., to provide concepts for specifying particular, biased modes of regarding *data* and *operations* on this data. This paper presents such a view mechanism.

The view mechanism is presented in the context of the O_2 model [5]. However, the concepts are general and apply to other models as well. Our work makes extensive use of existing mechanisms. Clearly, the query language [4] is an important component in specifying views. In addition, inheritance and overloading are extremely useful, as are database imports [9] and object creation [2, 15, 13]. New features are introduced such as grouping all objects with similar behaviors, and a parameterized way of grouping objects.

Our main contribution is an analysis of the rich and various aspects of object-oriented views and their incorporation in a coherent framework. The presentation, mostly informal, leans on a number of illustrative examples meant to emphasize the simplicity of our view mechanism. In the formal foundations based on the IQL model [2], which will appear in a subsequent work, *a view is just a query*, as in the relational world. However, in the object-oriented world, the situation is much more intricate, so we must carefully distinguish between various kinds of view specifications.

A view mechanism should allow a programmer to restructure the database and modify the behavior and structure of objects. It should also allow him to specify attribute values implicitly, rather than storing them. This is similar to the concept of a view found in relational and deductive databases. Perhaps more fundamental is the introduction of new classes into the class hierarchy. These *virtual classes* are populated by selecting existing objects from other classes and by creating new objects. We call these new objects *imaginary*, since they have no reality except in the view.

Virtual classes are defined by specifying their population. We provide three facilities for populating a virtual class with existing objects: (*i*) declaring that the virtual class is a superclass of certain existing classes (generalization), (*ii*) declaring that the virtual class contains all objects returned by a given query (specialization), and (*iii*) declaring that the virtual class contains all objects having a certain behavior (behavioral generalization). To populate a virtual class with imaginary objects, the

*This work was partially supported by a PRC BD3 grant of the French government.

†abitebou@inria.inria.fr

‡bonner@db.toronto.edu

programmer specifies a query that returns a set of values. A new object identifier is then attached to each value.

Once a virtual class has been defined, the internal structure and behavior in the class are derived by the system. The class can then be used as any other class. For instance, new methods can be defined and existing methods redefined. As in the presence of multiple inheritance, a problem that arises is *schizophrenia*, that is, in more technical terms, method resolution conflicts. A particular problem raised by imaginary objects is fixing their identity during database updates.

The paper is organized as follows. Section 2 briefly defines the model. Basic tools for importing and hiding data are considered in the Section 3. Virtual classes are discussed in Section 4, and imaginary objects in Section 5. Section 6 contains a conclusion and a brief comparison with previous work.

2 Getting Started

This short section introduces our view mechanism. After briefly describing the database model, we argue that the object-oriented approach is naturally suited to the definition of views. We are then led to blur the distinction between attributes (stored values) and methods (behaviors).

Our starting point is a standard object-oriented database model, the O_2 model [5, 16]. In this model, the database consists of a hierarchy of classes. Each class has an associated type, and every object (oid) in a class has a value of this particular type. Classes also have methods attached to them. Important features of the model include inheritance of types and methods, and method overloading. To keep the presentation simple, we assume in this paper that the value of an object is a tuple. (When the value is not a tuple, i.e., when it is a set or a list, it can be treated as a tuple with a single field.) Each field of the tuple is called an *attribute*. Since we are using the O_2 model, our view definitions use primarily the O_2-query language [4]. However, we are quite liberal with the exact syntax and assume it to be self explanatory. Although we use a particular database model, the concepts described are general and are applicable to other models as well.

Virtual Attributes

Object-oriented systems present the database designer with new choices for structuring data. For instance, should an address be structured as three attributes, *City*, *Street* and *Number*, or as a single complex attribute? There is no definite answer: There are many ways of distributing information among attributes, and no one structure will be best for all applications. It is therefore essential to be able to rearrange the information within an object, so that different views can be tailored to different applications. The following example illustrates the use of our syntax in a simple restructuring.

Example 1. (Merging several attributes.) Suppose that class *Person* has attributes *City*, *Street* and *Zip_Code*, each of type string. The following code merges these three attributes into a single attribute called *Address*:

attribute Address in class Person
  has value [*City: self.City,*
    Street: self.Street,
    Zip_Code: self.Zip_Code].

During execution, the variable *self* is bound to an object in class *Person*, the person whose address is being computed. The expression *self.City* denotes the city in which the person lives. □

This example leads to our first observation: A view mechanism should not force a distinction between attributes (stored values) and methods (procedures). In the example, *Address* has characteristics of both attributes and methods. *Address* is a property of a class, like an attribute, but its value is computed, like a method. Furthermore, the value of *Address* is accessed as though it were stored. For instance, to access Maggy's city and address, we use the same notation: *Maggy.City, Maggy.Address.* (The dot notation here combines both dereferencing—getting the value of an object—and field selection.) Because of this blurring of attributes and methods, our model has only attributes, whose values may be stored or computed. These *virtual attributes* may have zero or more arguments (besides the receiver).

An attribute is defined by a statement of the following form:

attribute A {*of type T*} *in class C* {*has value V*}

where A is the attribute name and V is a procedure. When the *has value* part is missing, the attribute is stored. The type declaration is not compulsory because it is often the case that the type can be inferred by the system. For instance, static type inference determines that attribute *Address* in Example 1 is a tuple of type [*City:string, Street:string, Zip_Code:string*]. As a general rule, the view system should relieve the user of mundane tasks like specifying a type when the type can be inferred.

Since the distinction between stored attributes and methods has been blurred, the same attribute can be stored or computed depending on the class. For instance, the declarations below specify that in class *Employee*, the value of *Address* is stored, and that in class *Manager*, it is computed (even though *Manager* is a subclass of *Employee*). This kind of overloading is possible because attributes and methods are no longer distinguished.

attribute Address in class Employee;
attribute Address in class Manager
  has value self.Company.Address.

So far, we have only slightly modified the standard data model, but this already provides a lot of restructuring capabilities. We can now, for instance, split a complex attribute into several simpler ones, or more generally, restructure several attributes or objects simultaneously. For instance, the attributes

Home: [*Address:U, Telephone:V*]
Office: [*Address:X, Telephone:Y*]

can be restructured as

Addresses: [*Home:U, Office:X*]
Telephones: [*Home:V, Office:Y*]

The following sections complete the view mechanism by providing facilities for hiding part of the database and for creating new classes.

3 Importing and Hiding

In general, there can be *many* databases in a system. In such systems, one database can use data from other databases via *import* statements. A *view* can thus be thought of as a database that imports all its data from other databases. That is, a view has a schema, like all databases, but no proper data of its own. This section describes mechanisms for selecting data for use in a view.

It is convenient to think of a view as being initially empty, and then specify what information from other databases should be made visible. The basic mechanism for this is the *import* [9]. The following simple example, defining a view called *My_View*, illustrates our syntax:

create view My_View;
import all classes from database Chrysler;
import class Person from database Ford.

Semantically, when classes are imported, they become visible together with their subclasses, the objects in the classes, their values and behaviors.

Once data are imported, we often want to hide portions of it. For instance, suppose we wish to hide employee salaries. In a relational database, we might have a relation called *Employee* with attributes such as [*Name,Number,Age,Salary*]. The following select command would then define a view in which salary information is hidden:

create view A_Relational_View
select [*E.Name, E.Number, E.Age*]
from E in Employee.

Besides being cumbersome, this approach presents the disadvantage that the definition of the view must be changed whenever the schema of the *Employee* relation changes, even if all we want to do is hide the *Salary* attribute. More importantly, for object-oriented views, this query is simply incorrect. That is, it does more than just hide salary information; it also hides all attributes defined in all subclasses of *Employee.* For example, let *Manager* be a subclass of *Employee* having the attributes [*Name,Number,Age,Salary,Budget*], where the last attribute indicates a manager's quarterly budget. The above query hides the budget information of all managers, which is not what we intended. Thus, in object-oriented databases, we must introduce an explicit hide command. We use the following declaration:

hide attribute Salary in class Employee.

Semantically, the definitions of *Salary* in class *Employee* and all its subclasses are hidden from the view.

To complete the description of the import-hide mechanism, we observe that, in general, we can build views on top of views on top of views, etc. In practice, we have found that it is often sufficient to import part of a database, define a view on top of it, and then hide (or make private) part of the schema. The rest of the paper therefore assumes that a view definition has the following general structure:

create view My_View;
{ *import and hide specifications* }
{ *class and method definitions* }
{ *hide specifications* }

4 Virtual Classes

An object-oriented view mechanism should be able to provide a user with a class hierarchy that is more appropriate to his needs that the actual hierarchy in the database. Some classes may be hidden and new classes may be introduced. The rest of the paper describes mechanisms for creating new classes, which we call *virtual classes.*

When defining a virtual class, we must define (*i*) its population, (*ii*) its position in the class hierarchy, and (*iii*) the behavior of objects in the class. In our proposal, we specify only the population. Behavior and position in the hierarchy are then derived by the system. The population can be specified in two ways: by selecting objects that already exist, or by creating new ones. This section considers the selection of existing objects, and the next section considers the creation of new ones. This section also introduces two notions, behavioral and parameterized classes, that are not found in existing systems, and that increase the flexibility of the view mechanism.

4.1 The Population of Virtual Classes

We consider three ways of populating a virtual class with existing objects. Here are examples of the three:

Specialization: We can define a virtual class called *Adult* consisting of all those persons who are at least 21 years old. In this manner, a database query is used to specify the immediate instances of a class. Observe that in this case, the virtual class has only "immediate instances" and no subclasses.

Generalization: Consider a database called *Navy*. We can define a virtual class called *Ship* that includes the classes *Tanker*, *Cruiser* and *Trawler*, thus specifying its subclasses and its population at once.

Behavioral Generalization: We can define a virtual class called *Printable* that includes all classes that have a *Print* method. Likewise, we can define a virtual class called *On_Sale* that includes all classes that have *price* and *discount* attributes of appropriate signatures. We will see that grouping classes based on behavior provides a lot of flexibility.

With the above examples in mind, we introduce the following notation for declaring the population of a virtual class C:

class C *includes* $\alpha_1, \alpha_2, ..., \alpha_n$

where each α_i is either:

1. the name of a previously defined class,
2. a database query that returns a set of objects, or
3. *like* B, where B is a previously defined class.

Intuitively, the expression *like* B means, "group all classes whose type is at least as specific as the type of B". Such a class may have more attributes than B, but not fewer.

Using this notation, the examples mentioned above are expressed as follows:

class Adult includes
(select P from Person where P.Age $\geq$ *21);*

class Ship includes Tanker, Cruiser, Trawler;

class On_Sale_Spec
has attribute Price of type dollar;
has attribute Discount of type integer;
class On_Sale includes like On_Sale_Spec.

These declarations define the entire population of the virtual classes. In particular, it is not possible for a user to insert an object directly into a virtual class. Thus, a *Ship* object can only be created indirectly, by creating an object of type *Tanker*, *Cruiser* or *Trawler*.

Once a virtual class is defined, it is treated just as any other class. For instance, the programmer can define new attributes (methods) for a virtual class, as illustrated in the next example. This example also illustrates the specification of population by specialization and generalization at the same time.

Example 2. The following code defines a virtual class that has both subclasses and immediate instances:

class Government_Supported includes
Senior, Student,
(select A in Adult where A.Income < 5,000);
attribute Government_Support_Deduction
in class Government_Supported
has value gsd(self).

This code defines the class *Government_Supported*, consisting of seniors, students, and all people who earn less than \$5,000 a year. Our intention is that people in this class are subject to special income tax laws. In particular, they receive a tax deduction called the "government support deduction". (The procedure $gsd(x)$ computes the value of the deduction for person x.) □

A simple extension of the above syntax allows us to define *parameterized classes.* For example, we could define define the class *Adult* to be a function of an age parameter A, as follows:

class Adult(A) includes
(select P from Person where P.Age $\geq$ *A).*

This statement effectively declares infinitely many classes, such as *Adult(20)* and *Adult(21)*, each with a different name and a different population. (Only finitely many of these classes will be non-empty however). We will see that parameterized classes increase the flexibility of a view mechanism.

To conclude this section, we should observe that the emphasis in our approach is on *inference.* The main task of the view designer is to specify the population of the new classes. As we shall see, the system then derives the new class hierarchy and the structure and behavior of the virtual classes. Attribute hiding and overloading allow the designer to modify the default behavior derived by the system.

4.2 Virtual Class Hierarchies

With the mechanisms introduced above, a programmer can reorganize a database by defining hierarchies in which most of the classes are virtual. These hierarchies are derived from the declarations of the virtual classes. The two basic mechanisms for defining such hierarchies are generalization and specialization, as defined above. However, the flexibility and convenience of these two mechanisms is increased by the use of behavioral generalization and parameterized classes. This section describes these processes, first through examples, and then more formally.

Conceptually, a programmer can construct a virtual class hierarchy in two distinct ways: top-down or bottom-up. In the top-down approach, large classes are divided into smaller ones via specialization. (The analogous operation in relational systems is to define a virtual table by selecting tuples from a larger table.) In the bottom-up approach, small classes are combined to form larger classes via generalization. (The analogous operation in relational systems is to define a virtual table as the union of several smaller tables.) The following examples illustrate the construction of virtual class hierarchies in the top-down and bottom-up modes.

Example 3. (Top-down construction of a virtual class hierarchy.) Starting from the class *Person*, we first define two subclasses, called *Adult* and *Minor*. We then define a subclass of *Adult*, called *Senior*, and a subclass of *Minor*, called *Adolescent*.

class Adult includes
  (select P from Person where P.Age $\geq$ *21);*
class Minor includes
  (select P from Person where P.Age $<$ *21);*
class Senior includes
  (select A from Adult where A.Age $\geq$ *65);*
class Adolescent includes
  (select M from Minor where M.Age $\geq$ *13).*

□

Example 4. (Bottom-up construction of a virtual class hierarchy.) Suppose the database contains the classes *Tanker*, *Trawler*, *Cruiser*, and *Frigate*. We define a *Merchant_Vessel* to be a *Tanker* or a *Trawler*, and we define a *Military_Vessel* to be a *Cruiser* or a *Frigate*. We then define a *Boat* to be either a *Merchant_Vessel* or a *Military_Vessel*.

class Merchant_Vessel includes Tanker, Trawler;
class Military_Vessel includes Frigate, Cruiser;
class Boat includes Merchant_Vessel,
Military_Vessel.

□

Examples 3 and 4 illustrate the two simplest modes of constructing a virtual class hierarchy. More generally, these two modes can be combined. This is partly illustrated in Example 2, where the class *Government_Supported* is defined by combining the classes *Student* and *Senior*, in bottom-up fashion, and by selecting objects from the class *Adult*, in top-down fashion.

Increasing the Flexibility

We have just seen how to build virtual class hierarchies by using generalization and specialization. We can increase the flexibility and convenience of these constructions by employing behavioral and parameterized classes. In particular, behavioral classes increase the flexibility of bottom-up constructions, and parameterized classes increase the flexibility of top-down constructions. This section presents an example of each.

When using generalization to build a virtual class hierarchy, we have to name all the classes that we want to group. However, it is sometimes more convenient to group classes by their properties, instead of by their names. We saw an example of this in the definition of the class *On_Sale*. If the only objects for sale were cars, houses and companies, we could define the class *On_Sale* as follows:

class On_Sale_Bis includes Car, House, Company.

The definitions of *On_Sale* and *On_Sale_Bis* would then be equivalent and be treated as such by the type system. However, the behavioral definition of *On_Sale* is more flexible than that of *On_Sale_Bis*. In particular, the introduction of a class *Boat* (with appropriate price and discount attributes) would require the programmer to change the definition of the class *On_Sale_Bis*. This is not needed with the behavioral definition.

Similarly, parameterized classes increase the flexibility of top-down constructions. When using specialization to build a virtual class hierarchy, we have to define each subclass individually. However, it is sometimes more convenient to partition a class into subclasses according to a parameter. For instance, the following declaration partitions the class *Person* according to address, producing one subclass for each country:

class Resident(X) includes
  (select P from Person
  where P.Address.Country = X).

Thus, *Resident(USA)* and *Resident(France)* represent two distinct subclasses of *Person*. This is certainly more convenient than providing a separate class declaration for each country. Furthermore, as countries are removed from the database or added, classes automatically disappear or are created and integrated into the class hierarchy.

Inferring the New Class Hierarchy

The examples above show that a virtual class has natural subclasses and superclasses. They also suggest how a class hierarchy can be inferred from the definitions of the virtual classes. This section develops this idea further and makes it precise.

Recall that virtual classes are defined to include entire classes as well as objects selected from classes. In Example 2, the class *Government_Supported* was defined to include the classes *Student* and *Senior* and objects selected from the class *Adult*. In general, suppose that a virtual class C is defined to include classes $C_1 ... C_k$ as well as objects selected from classes $C_{k+1} ... C_n$. We define the subclasses and superclasses of C as follows:

1. if D is a superclass of $C_1 ... C_n$, then D is also a superclass of the virtual class C; and

2. each C_i is a subclass of C for $1 \leq i \leq k$.

The new class hierarchy can be computed from these two rules using standard type inference techniques.

For example, consider a variation of Example 4. Suppose that the database has a class called *Ship* with four subclasses, *Tanker*, *Trawler*, *Frigate* and *Cruiser*. Now define two virtual classes as follows:

class Merchant_Vessel includes Tanker, Trawler;
class Military_Vessel includes Frigate, Cruiser.

These declarations effectively combine the four subclasses of *Ship* into two larger classes, *Merchant_Vessel* and *Military_Vessel*. By rule (1), all objects in these two classes are also in the class *Ship*, so *Ship* is a superclass of the two virtual classes *Merchant_Vessel* and *Military_Vessel*. By rule (2), *Merchant_Vessel* becomes a new superclass of *Trawler* and *Tanker*, and *Military_Vessel* becomes a new superclass of *Cruiser* and *Frigate*. (In fact, they become direct superclasses).

Observe that such definitions allow virtual classes to be inserted in the middle of a class hierarchy. E.g., the virtual class *Merchant_Vessel* is inserted between the upper class *Ship* and the lower classes *Tanker* and *Trawler*. As the class hierarchy evolves and becomes more complex, this facility can help organize data. For example, we expect that in a real database, the class *Ship* would eventually acquire many new subclasses, such as *Destroyer*, *Air_Craft_Carrier*, *Battle_Ship*, and *Mine_Sweeper*, as well as *Ocean_Liner*, *Freighter*, *Ferry*, *Barge* and *Gondola*. To keep the class hierarchy manageable, we would like to insert new, intermediate classes between *Ship* and its proliferating subclasses. Creating virtual classes such as *Merchant_Vessel* and *Military_Vessel* does exactly this, providing the user with a more organized view of the data.

Also observe that this definition allows a virtual class to have multiple superclasses. For example, consider the virtual class *Rich&Beautiful* defined as follows:

class Rich&Beautiful includes
(select P from Rich where P in Beautiful).

The type system detects that every object in this class is both in *Rich* and in *Beautiful*. It infers, therefore, that *Rich* and *Beautiful* are both superclasses of *Rich&Beautiful*. Therefore, assuming that *Rich* and *Beautiful* are not comparable, this definition introduces multiple inheritance into the schema.

To conclude this section, we point out that virtual classes can overlap in many ways. For instance, we can categorize people by many criteria (age, wealth, height, etc.), defining a virtual class hierarchy for each such criterion. In principle, all of these virtual classes can overlap, and they can overlap in many combinations. For example, some people may be in the classes *Rich*, *Young* and *Beautiful*, whereas others may be in the classes *Poor*, *Old* and *Ugly*. In general, given n virtual classes, they may overlap in $O(2^n)$ different ways. A programmer may choose to define some of these overlaps as classes, like the class *Rich&Beautiful* defined above. However, as a practical matter, we do not require that all possible overlaps be defined as classes. Thus, in our framework, an object may simultaneously belong to several incomparable virtual classes.

Implementation Issues

Since an object may be a member of many classes simultaneously, several implementation issues immediately arise. In considering these issues, we first recall that even in the absence of a view mechanism, objects may belong to several classes, because of the *isa* relation. That is, an object created in class C is also a member of each superclass of C. We say that the object is *real* in C and *virtual* in each superclass of C. To deal with this situation, most object-oriented data models adopt the following rule (see e.g., [6, 5]):

Unique Root: An object is real in only one class.

As a consequence of this rule (and in the absence of views), to find the code for a method of a particular object, it suffices to "climb" the class hierarchy until a class is found that provides the code. That is, the following rule is usually adopted:

Upward Resolution: The resolution of a method in class C is to be found in a superclass of C.

These two rules have important consequences for the implementation of object-oriented database systems. On the one hand, experience has shown that the unique root rule increases the efficiency and decreases the complexity of data-intensive systems [6, 5]. The reason is that under this rule, the structure of an object is fixed: It has a fixed set of attributes and it can be stored uniformly along with similar objects. On the other hand, the upward resolution rule facilitates the resolution of methods: In principle, it suffices to climb the class hierarchy dynamically. (In practice, static method resolution is preferred, but this is complicated by multiple inheritance.)

We adopt the unique root rule here. That is, although an object may belong to many virtual classes, we insist that it belongs to exactly one real class. Under this restriction, efficient storage and retrieval are still feasible. However, because of the view mechanism, and in particular, because of specialization, an upward traversal in the class hierarchy is no longer sufficient to resolve methods. That is, the upward resolution rule no longer applies. Indeed, efficient resolution of methods is a subtle issue (similar to the case with multiple inheritance), which we discuss further in the next section.

4.3 Attributes

From the declaration of a virtual class, we can infer the type of the class, that is, we can infer its attributes and their types. There are two basic mechanisms by which virtual classes acquire attributes: (*i*) by inheriting the attributes of its superclasses, as all classes do, and (*ii*) by inheriting attributes that are common to all the objects in the class.

Once we infer the superclasses of a virtual class, it inherits the attributes of the superclasses in the standard way. Thus, in Example 2, since *Person* is a superclass of *Senior*, *Student* and *Adult*, then *Person* is also a superclass of *Government_Supported*. Thus, *Government_Supported* inherits all the attributes of *Person*, such as *Name*, *Address* and *Birth_Date*, as well as the behavior of *Person*.

In addition, a virtual class may acquire attributes that are common to all objects in the class. For instance, in Example 4, if *Tanker* and *Trawler* both have an attribute called *Cargo*, then the class *Merchant_Vessel* will inherit it. Similarly, the class *Military_Vessel* will inherit the attribute *Armament* from its subclasses. To make this idea precise, suppose the following are true:

1. the virtual class C is defined to include the classes $C_1 ... C_k$ as well as objects selected from the classes $C_{k+1} ... C_n$;
2. each C_i has an attribute called A, for $1 \leq i \leq n$; and
3. the types of attribute A in classes $C_1 ... C_n$ have a least upper bound τ.

Then, the virtual class C also has an attribute called A, and the type of A in C is τ. (Otherwise, A is undefined in C.) This acquisition of structure or behavior from subclass to class is called *upward inheritance*, to distinguish it from the standard *downward inheritance* from class to subclasses. Again, all that is involved here is standard type checking techniques.

At the end of Section 4.2, we mentioned that method resolution in virtual classes resembles method resolution when multiple inheritance is allowed. We can pursue this direction a little further now that we have defined how virtual classes inherit attributes and methods.

The typical setting of a resolution conflict is a class C with two direct superclasses C_1 and C_2, with a method m defined in each. If m is not redefined in C, then an object in class C, on receiving message m, doesn't know how to react. We call this behavioral problem *schizophrenia*, in the sense that the receiver doesn't know which personality to choose. In the context of views, the overlapping of classes leads to the same problem. For example, suppose the classes *Rich* and *Senior* both define a *Print* method. Then, when a print message is sent to an object that is in both *Rich* and *Senior*, we have an instance of schizophrenia. There have been many solutions proposed for solving multiple inheritance conflicts from forbidding schemas with conflicts, to explicitly assigning levels of priority, or using priorities based on creation time.

A view system should not strictly disallow schizophrenia, but should provide a default instead (even a meaningless default). As we saw earlier, there are potentially 2^n ways in which n classes can overlap. Most of these overlaps, however, are not likely to represent meaningful classes. If there are only a few cases of overlap to consider, inheritance conflicts can be resolved by assigning a class name to overlapping classes (like the class *Rich&Beautiful* defined above). One can then redefine the conflicting methods in the new class. However, this explicit conflict resolution becomes cumbersome or infeasible when there are too many cases to consider.

5 Imaginary Objects

Section 4 showed how to populate a virtual class by selecting objects that already exist in the database. This section shows how to populate a virtual class by creating new objects. We say that such objects are *imaginary*, since they exist in the view, but not in the database. Object creation has been considered in a number of papers (e.g., [2, 15]). Our semantics of identity for created objects is in the style of [13].

Imaginary objects have several important applications:

- Creating an object-oriented view of a relational database. Typically, this means creating new objects from database tuples.
- Combining small objects into larger aggregate objects.
- Decomposing large objects into several smaller objects.
- Sophisticated restructurings that turn objects into values and values into objects.

To create imaginary objects, we use a simple extension of the notation introduced in Section 4 for defining virtual classes. For instance, suppose the database has a class called *Person* with attributes *Name*, *Sex*, *Spouse*, and *Children*. We would like to view the data not as a collection of people, but as a collection of families. We shall therefore create a virtual class called *Family* with two attributes, *Husband* and *Wife*. To populate this class, we must specify the values of the attributes for each object. The following query does exactly this:

select [*Husband:H, Wife:H.Spouse*]
from H in Person
where H.Sex = 'male'.

The result of this query is a set of tuples where each tuple represents the attributes of some family. Furthermore, this set represents *all* the families in the database. Given the result of the above query, we must convert each tuple into an object of type *Family*, creating a new identifier for each object. To do this, we extend our syntax for declaring virtual classes, adding the keyword *imaginary*:

class Family includes imaginary
(*select* [*Husband:H, Wife:H.Spouse*]
from H in Person
where H.Sex = 'male'.

This declaration does three things: (*i*) it declares *Family* to be a virtual class, (*ii*) it specifies the population of class *Family*, and (*iii*) by static type inference, it declares that class *Family* has two attributes, *Husband* and *Wife*, both of type *Person*. We call *Husband* and *Wife* the *core* attributes of class *Family*.

When a virtual class contains imaginary objects, as *Family* does, we call it an *imaginary class*. Imaginary classes are treated like any other class. In particular, we can declare virtual attributes for them. Thus we can define a virtual attribute called *Children* for class *Family*, as follows:

attribute Children in class Family has value
(select P from Person
where P in self.Husband.Children
or P in self.Wife.Children).

We have distinguished above between the core and virtual attributes of an imaginary class. So far, however, there appears to be little difference between them. For instance, the attribute *Children* could easily have been declared as a core attribute of class *Family*, instead of a virtual attribute. However, we will see in the next section that there is a crucial difference when updates are considered.

The example of families uses imaginary objects to combine small objects into larger ones. As mentioned in the introduction of this section, imaginary objects have other applications. This is illustrated next.

Example 5. (Transforming values into objects.) Suppose that the database *Staff* has a class called *Person* whose attributes include *City*, *Street* and *Number*, among others. We define a view in which these three attributes are replaced by a single attribute called *Address*, whose value is an object. The main advantage of this modification is that addresses can now be shared:

```
create view Value_to_Object;
    import class Person from database Staff;
    class Address includes imaginary
        (select [City: P.City,
                 Street: P.Street,
                 Number: P.Number]
         from P in Person);
    attribute Address in class Person has value
        (select the A in Address
         where A.City = self.City
         and A.Street = self.Street
         and A.Number = self.Number);
    hide attributes City, Street, Number
        in class Person.
```

(Note the use of the expression "*select* the *A in Address...*," indicating that a query must return a single element and not a set.) □

To conclude this section, we consider one more application of imaginary objects. This application addresses a problem faced by designers of object-oriented systems: deciding what information to represent as objects, and what to represent as attributes. For example, in designing a database about people and addresses, we could define a class called *Person* with an attribute called *Address*. Alternatively, we could define a class called *Address* with an attribute called *Occupants*, listing the names of all people who live at that address. Both organizations represent the same information, although the former emphasizes people, whereas the latter emphasizes addresses. In general, no one choice of classes is ideal for all applications. The mechanism of imaginary objects allows the programmer to adapt the database reality to the needs of the user. The main step in this transformation is the conversion of values/attributes into imaginary objects.

5.1 Assigning Identifiers to Imaginary Objects

To create new objects, the view mechanism creates new object identifiers (oid's) and assigns them to objects. (We consider here the logical viewpoint; it doesn't *have to* be like this in the implementation.) This requirement leads to subtle problems since it can be difficult to fix the identity of an imaginary object.

At the implementation level, suppose that the oid's of the imaginary objects are computed (and recomputed) on demand, like a relational view. How can we guarantee that an imaginary object is assigned the same oid at each invocation? This problem arises in the simplest cases. For instance, consider the following query: *select F from Family*. Each time this query is posed to the system, it generates a set of families and assigns an oid to each one. How can we ensure that each family receives the same oid every time the query is invoked? This problem becomes crucial once we consider joins and intersections. For instance, consider the following, seemingly equivalent queries:

```
select F from Family
where F.Size > 5
and F.Father.Age < 25

select F from Family
where F.Size > 5
and F in (select F from Family
          where F.Father.Age < 25).
```

With the first query, we obtain as many objects as families satisfying the criteria. With the second query, the result is implementation dependent, and we may obtain an empty set if new oid's are generated each time we use *Family*.

In this example, it is obvious that we want to use the same set of objects each time *Family* is used. In general, however, a virtual class may be modified by database updates, and it is not clear how imaginary objects maintain their identity as the database evolves. The rest of this section addresses this issue.

When defining imaginary objects, it is convenient to think of the definition as a function mapping tuples to oid's. The problem now becomes clear: It is the problem of updating this function when the database is updated. (In this sense, it generalizes the traditional problem of materialized views.) From the logical point of view, we choose a simple solution. The population of an imaginary class C is defined by a query that returns a set of tuples. For each tuple t returned by the query, we use the expression $C(t)$ to denote the oid assigned to t. From an implementation point of view, there could be a table giving the mapping between the tuples and oid's. In this way, we are guaranteed that the same tuple will be assigned the same oid each time the class C is invoked. (Note that a tuple will generate a different oid when used in a different class.)

This approach is rather simple. Even so, there are logical issues that a view designer should be aware of. The core attributes play a central role, because when the value of a core attribute changes, the system assumes that it is dealing with a new object. One must therefore choose the core attributes carefully, since updating them can affect object identity. Indeed, the core attributes should be thought of as being somewhat immutable.[1]

In Example 5, the core attributes are well chosen. The intent in the example is that each address is represented by a distinct object, with a distinct identifier. In particular, when a person changes his address, we expect his new address to be represented by a new object. The code in Example 5 specifies exactly this. For instance, when Maggy moves out of 10 Downing Street, the attribute *Address* of object Maggy will point to a different object (and if necessary, a new object will be generated). On the other hand, the object corresponding to 10 Downing Street may still be used in other parts of the view. Thus, the definition meets the intuition about addresses.

Example 6. (A poorly designed view.) Consider a database for a life insurance company. To represent insurance policies, the database has a class called *Policy* with attributes such as *Policy_Number*, *Coverage*, *Cost*, *Name*, *Address*, *Age* and *SS#*. We define a view in which insured persons are represented as objects. Thus, we create a virtual class called *Client* with attributes *Name*, *Age*, *Address*, *SS#* and *Policy*. In class *Policy*, the four attributes *Name*, *Age*, *Address* and *SS#* are replaced by the single attribute *Person*. The following code defines the view:

```
create view My_Clients;
    import class Policy from database Insurance;
    class Client includes imaginary
        (select [Name: P.Name,
                 Age: P.Age,
                 SS#: P.SS#,
                 Address: P.Address,
                 Policy: P ]
         from P in Policy);
    attribute Person in class Policy
        has value (select the C from Client
                   where C.Policy = self);
    hide attributes Name, Age, Address, SS#
        in class Policy.
```

Now suppose that Maggy's address is updated. In the view, a new client object with a new identifier will be created, and Maggy's policy will refer to this new object. In other words, as far as the system is concerned, Maggy before moving and after moving are two different clients. □

The problem in Example 6 is that the core attributes of class *Client* are poorly chosen. By including *Address* as a core attribute, we effectively declare that *Address* is crucial to the identity of a client, just as *Street* and *City* are crucial to the identity of an address. It would be better to define *Address* as a virtual attribute of *Client*, just as *Children* was defined as a virtual attribute of *Family* in an earlier example.

6 Conclusions and Comparison with Previous Work

The problem of views in a semantic or object-oriented database model has been considered in a number of papers. For instance, hiding the distinction between stored attributes and computed properties has been a prominent feature of several systems (e.g., [10, 8]); some interesting ideas on views in the Daplex world are mentioned in [8]; and a limited view mechanism is described for Orion in [15], and for the Fugue Model in [11]. These proposals introduce interesting ideas but do not provide the flexibility and the generality that one would expect from a view mechanism. In particular, the creation of new classes is too limited and relies too heavily on explicit specifications given by the user. Furthermore, object creation, when considered, has an unclear semantics.

In contrast, the present paper provides a general framework for view definition from a language perspective based on a clear semantics. The semantic issues raised by the approach are covered in depth. Furthermore, the view mechanism was built with a number of simple ideas in mind:

1. *A view should be treated as a database.* e.g., A virtual or imaginary class is usable as any other class.
2. *A view should be easily constructed.* e.g., We use the query language as much as possible.

[1] One can imagine more sophisticated approaches in which an object preserves its identity when its core attributes change. Suppose that we do so. Now, suppose that because of an update, two tuples become identical. Should we merge the two corresponding objects? This leads to *object merging*. Similarly, one can find examples that lead to *object splitting*.

3. *The spirit of object-orientation should be kept.* e.g., The semantics of import and hide relies primarily on inheritance.

4. *The user should be relieved of mundane tasks.* e.g., After he specifies the population of a class, the system derives the type and behavior of the class.

5. *More flexibility should be provided.* e.g., We have removed the distinction between attributes and methods, and we have introduced new features such as behavioral and parameterized classes.

Problems related to object-oriented views have been considered in several papers. For instance, a query language is a core component of a view mechanism, and a number of query languages have been proposed for object-oriented databases (e.g., [4, 19, 10]). Orthogonal issues are covered in other papers. The use of views for integrating databases is considered in [12]. The important issue of the interaction between user and system through the use of views is treated in [17]. The problem of authorization, which is clearly relevant to information hiding, is considered in depth in [18]. Finally, we note that important issues such as materialized views and view updates, which have been extensively studied in the relational model, acquire a new dimension in the context of objects.

Acknowledgments: François Bancilhon is thanked for getting the first author interested in this virtual topic, Peter Buneman, Dave Maier, Jean-Claude Mamou and Claudia Medeiros for discussions on views, and Guy Ferran for explaining the import of O_2. Peter Buneman also managed to convince the first author of the importance of behavioral generalization. The second author would like to thank Serge Abiteboul for arranging time and facilities at INRIA, without which, this paper would not have been possible.

References

[1] S. Abiteboul, Virtuality in Object-Oriented Databases, *proc. VIèmes Journées Bases de Données Avancées, 1990.*

[2] S. Abiteboul and P. Kanellakis, Object Identity as a Query Language Primitive, proc. ACM SIGMOD 1989, to appear in *Journal of the ACM.*

[3] F. Bancilhon. Object-Oriented Database Systems. *Proc. ACM conf. Principles of Database Systems (PODS),* pages 152–162, 1988.

[4] F. Bancilhon, S. Cluet and C. Delobel, Query Languages for Object-Oriented Database Systems: The O_2 Proposal, *proc. Intern. Work. on Data Base Programming Languages 2*, 1989.

[5] F. Bancilhon, C. Delobel and P. Kanellakis ed., *The O_2 Book*, 1989, in preparation.

[6] J. Banerjee et al., Data Model Issues for Object-Oriented Applications, *ACM Trans. on Office Information Systems*, 1987.

[7] J. Banerjee, W. Kim. K.-C. Kim, Queries in Object-Oriented Databases, *proc. Data Engineering conf.*, 1988.

[8] U. Dayal, Queries and Views in an Object-Oriented Data Model, *Intern. Work. on Data Base Programming Languages 2*, 1989.

[9] G. Ferran, the import mechanism in O_2, private communication.

[10] D.H. Fishman, D. Beech, et al, IRIS: An Object-Oriented Database Management System. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 216–226. Morgan Kaufmann, 1990.

[11] S. Heiler and S. Zdonik, Object Views: Extending the Vision, *proc. Data Engineering conf.*, 1990.

[12] M. Kaul, K Drosten, E.J. Neuhold, ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views, *proc. Data Engineering conf.* 1990.

[13] M. Kifer and J. Wu, A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited), *proc. ACM conf. Principles of Database Systems (PODS)*, 1989.

[14] W. Kim, *A Foundation for Object-Oriented Databases.* Technical Report, MCC, 1988.

[15] W. Kim, A Model of Queries for Object-Oriented Databases, *proc. 15th Conf. on Very Large Databases*, 1989.

[16] C. Lecluse and P. Richard, the O_2 Database Programming Language, *proc. 15th Conf. on Very Large Databases, Amsterdam*, 1989.

[17] C. Medeiros and J.-C. Mamou, Interactive Manipulation of Object-Oriented Views, *proc. Data Engineering conf.*, 1991.

[18] F. Rabiti, E. Brtino, W. Kim, D. Woelk, A Model of Authorization for Next-Generation Database Systems, to appear in *ACM Transactions on Database Systems.*

[19] G.M. Shaw, S.B. Zdonik, An Object-Oriented Query Algebra, *proc. Intern. Work. on Data Base Programming Languages 2*, 1989.

[20] *IEEE Transaction on Knowledge and Data Engineering*, Special issue on database prototype systems, Ed. M. Stonebraker, 2:1, 1990.