# Implementation Experience with Building an Object-Oriented View Management System*

Harumi A. Kuno and Elke A. Rundensteiner
Dept. of Elect. Engineering and Computer Science
Software Systems Research Laboratory
The University of Michigan, 1301 Beal Avenue
Ann Arbor, MI 48109-2122
e-mail: kuno@eecs.umich.edu, rundenst@eecs.umich.edu
fax: (313) 763-1503
phone: (313) 936-2971

August, 1993

Although views have been found to be important mechanisms for database systems, currently no commercially available OODBMS supports view management tools. There are many challenging problems related to view management that must be addressed in the context of object-oriented models: what features are required to support a view system, how to provide updatable views, and how to utilize the complexity of the object-oriented data model for view definition (such as behavioral customization, view hierarchy manipulation, and incorporating virtual classes into a consistent global schema). We solve all of these problems in our implementation of the *MultiView* view management system, which supports updatable views on top of the GemStone OODBMS. The resulting system preserves the functionality of the underlying commercial OODBMS while adding view mechanisms and the features needed to support the view system. Our implementation is general purpose – we provide generic classes defined in Smalltalk that can easily be ported to other OODBMS systems.

**Keywords:** Schema Integration, Meta Schema, View Definition, Data Independence, Object-Oriented Databases, GemStone, Smalltalk.

*This work was supported in part by the NSF RIA grant #IRI-9309076 and the University of Michigan Faculty Award Program. We are also grateful for support from the 1993 NASA Graduate Student Researchers Program.

# 1 Introduction

In relational systems, a view is traditionally defined to be a named, persistent query, i.e., a virtual relation. Relational conceptual schemata are concerned with tables as distinct units (in that tables are independent and related only by means of foreign keys); hence it is trivial to incorporate a virtual table into the global conceptual schema in a relational system by simply adding the new table to the set of existing tables. [1]

Object-oriented schemata are made up of classes arranged in a generalization and decomposition hierarchy. An object-oriented *view schema* functions as a virtual database, and corresponds to an inheritance hierarchy of multiple classes, both actual and virtual, that contain objects of interest to a particular user. Users create virtual classes, which are integrated into a single consistent global schema from which users can select both base and virtual classes to participate in specific view schemata. View schemata and virtual classes provide logical data independence, and offer a means by which data and behavior can be repartitioned, restructured in format, and customized to meet the needs of a particular application.

Object-oriented view technology offers fundamental mechanisms for addressing many important tasks, such as customized tool interfacing to OODBs, interoperability of databases, security (for example by associating access control lists with customized view schemata), and transparent schema evolution. The development of powerful view mechanisms thus represents an important research area.

Creating views in an object-oriented model is not a simple transfer of the relational view solution to the object-oriented model. There are many challenging problems that need to be addressed in the context of this new technology. We must, for instance, re-evaluate how to overcome the view update problem of the relational view mechanism, how to utilize the complexity of the object-oriented data model for view definition (such as behavioral customization, object generation, view hierarchy manipulation, integrating base and virtual classes into a consistent global schema while preserving inheritance semantics). We also have to answer fundamental questions, such as what properties are required from an OODB for the support a view management system.

We present solutions to all these issues in the design of *MultiView*, a view management system which supports the definition of virtual classes through user queries. We have demonstrated the practicality of the *MultiView* approach by implementing the system on top of the GemStone OODBMS using the Smalltalk-like OPAL programming environment. Our system preserves the functionality of the underlying commercial OODBMS while adding view mechanisms and the features needed to support the view system.

While several proposals of object-oriented views have been given in the literature in recent years [3, 9, 10, 17, 21, 22], the large majority of them have not yet been implemented. Furthermore, we are not aware of any commercial OODB currently supporting such general purpose view capabilities as those offered by *MultiView* . The main purpose of this paper thus is to demonstrate the implementation of an actual view management prototype, and to describe its salient features. We expect that this will aid other researchers who wish to construct view systems using object-oriented technology. *MultiView* is unique in that it automatically organizes both base and virtual classes into a single comprehensive global schema graph from which object-oriented views – virtual, possibly restructured, subschema graphs – can be specified in a consistent manner [13]. *MultiView* supports to promotion of method code to the upmost, possibly virtual class, and thus provides for the true (upwards) inheritance of methods for both base and virtual classes.

In this paper, we describe the design and implementation of the *MultiView* prototype system which has been realized using GemStone [2]. In particular, we outline its three-layered architecture, its system classes, its view query language, and its user interface. This work validates the *MultiView* view methodology we introduced elsewhere [13], and also results in general observations about the basic functionalities required from an OODB system for building and supporting a view manager. We anticipate that these experiences will prove useful to other researchers developing object-oriented view support.

The remainder of this paper is organized as follows. In Section 2, we introduce basic object-oriented concepts we use throughout this paper. In Section 3, we outline the *MultiView* approach. In Section 4, we review our system objectives and discuss the decisions we made regarding *MultiView*'s data model. Section 5 presents the implementation of the *MultiView* prototype using GemStone, describing meta-classes, required data structures, and interfaces. We detail an example application using the *MultiView* system in Section 6, compare *MultiView* to related work in Section 7, and conclude with Section 8.

[1] Note that although "conceptual schema integration" is sometimes called "view integration" in the relational model, by "view integration" we mean the syntactic integration of the virtual table into the actual global schema in the database rather than the DBMS-independent semantic task of resolving external views of the data during schema design.

[2] GemStone is the commercial OODB product of Servio Corporation.

# 2 Terminology

## 2.1 Objects, Classes, and Types

An *object instance* (or short, *object*) represents an entity. Anything with distinct existence in objective or conceptual reality can be represented as an object. Each object consists of state (the *instance variables* or attributes of the object) and behavior (the *methods*, or messages, to which the object can respond). Each object has a unique system-generated value-independent *object identifier*, which makes it possible to distinguish between equality and identity, to share sub-objects among complex objects, and to perform updates on common sub-objects.

Methods represent operations that an object can perform. A method consists of a *selector*, which is the name by which the method is invoked, and a block of code specifying the behavior of the method. In the *MultiView* model, two methods are considered to be *equivalent* if they share the same behavior – that is, if their behavior is specified by the same block of code.

A *type* is the library of methods and instance variables available to a given object. In *MultiView*, a class is composed of both a type and an *extent* (set of all the object instances with that type). Every object possesses at least one type, and is thus an instance of at least one *class*. In addition, *MultiView* supports the concepts of both local extent (the collection of all instances of the class itself) and global extent (the collection of the instances of the class itself and all its subclasses).

## 2.2 Base Classes, Virtual Classes, and View Schemata

In order to distinguish between the different meanings associated with the term "view", we use the following terminology throughout this paper. Classes derived via an object-oriented query are referred to as *virtual classes* (as opposed to *base classes*). The classes on which the derivation for a virtual class is based are called its *source classes* (they can be both base or virtual classes). The schema integrating all base and all virtual classes is called the *global schema*. A schema containing a select subset of both base and virtual classes, specified by a user or application, is called a *view schema*. Although *MultiView* distinguishes between the *base classes* that contain actual object instances and the *virtual classes* that contain dynamically computed virtual object instances (computed based on a class derivation query), it organizes them into a single unified global inheritance hierarchy. The original base schema (composed of only the original base classes) is maintained as a special view schema containing all base classes.

## 2.3 Subsumption

In the *MultiView* model, we use the term "subclass" to mean that one class completely subsumes the other. That is, a subclass inherits the complete type description (behavior) of its superclass(es), i.e., all of the methods and attributes contained in the type of the super class must be included in the type of the subclass. Furthermore, a subclass's extent is a strict subset of the extent of its superclass – each instance of the subclass is considered to also be an instance of the superclass. Thus in each context where an instance of the superclass is required, an instance of the subclass is also permitted. [3]

The *MultiView* model differs from GemStone's regarding the issue of method identification. Determination of subsumption is a problematic issue which depends on whether classes are positioned explicitly through user interaction or automatically by the OODBMS application. In systems with single inheritance and those in which classes are explicitly positioned by the user, selectors (method names) can serve alone to register the pedigree of any method. That is to say, a class's *behavior* can be interpreted to be the list of selectors to which it can respond. For example, some languages (including Smalltalk and language derivatives of Smalltalk) identify methods by their selectors [19]. When an object receives a message whose selector does not match the selector of any of the methods belonging to the receiver's class, then the system searches the methods belonging to that class's superclasses, looking for an appropriate selector. *MultiView* preserves this inheritance protocol, even for virtual classes.

If, however, classes are defined by users but automatically positioned by the system, then it is important that the positioning algorithm be deterministic and independent of the history of the class's creation. For example, as illustrated in Figure 1, the user must be able to recognize that neither class 1 nor class 2, each

[3] Alternative definitions of the "subclass" relationship exist, for example, in [10] and [5].

with a method named `foo`, is related to class 3, which also possesses a method named `foo`. This would clearly be a problem for an automatic classifier that used only selectors for determining subsumption, since both class 1 and class 2 would appear to subsume the other. In addition, if a virtual class, such as class 4 in Figure 1a, were to be created by intersecting classes 2 and 3, the new class would inherit `foo` from both classes, necessitating that at least one of the `foo` methods be renamed. Despite the renaming, the origins of each `foo` method must be clearly discernable.

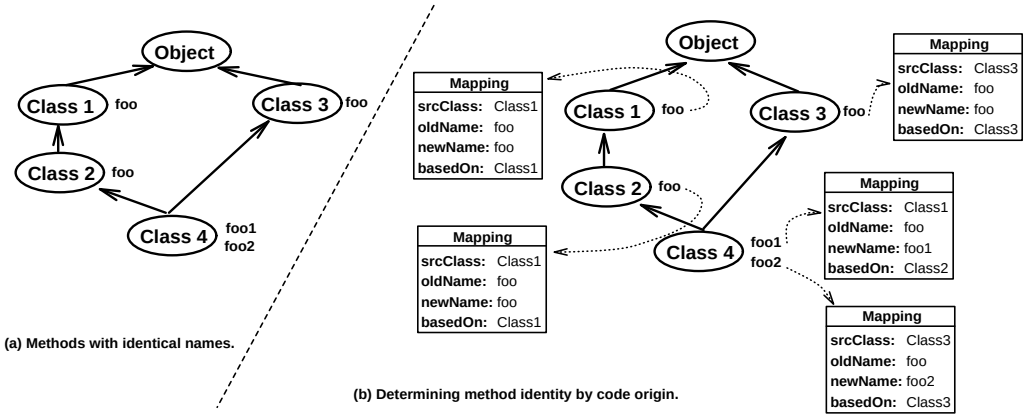


Figure 1: Methods with identical names are not necessarily related.

In its present incarnation, *MultiView* identifies methods by implementation (code block) rather than interface (selector name), with the exception that the positions of base classes relative to each other in the global class hierarchy are set explicitly by the user. For example, in Figure 1b, each method is identified by a structure which maps the method to its origins. In addition, because the classification of new virtual classes can lead to the insertion of virtual classes *above* original base classes, the code blocks associated with a method may be moved upwards in order to preserve the commonly used method for method resolution of downwards inheritance. This solves the performance problem faced by the approach proposed by Abiteboul et al (discussed later), which requires an upwards and downwards search for inheritance. For example, a hide class is a superclass of its source class, but obviously "inherits" its properties form the source class. We would thus have to perform downwards method resolution fo r the hide class.

This basic principle of subsumption using code blocks leads to the following class hierarchy structure: if two classes $C_1$ and $C_2$ share some common property then they must ultimately have inherited it from the same superclass. There must exist a lowest common superclass (LCS) in the class lattice for which this common property is defined. If not, these two properties could, for example, coincidentally share the same selector but correspond to distinct behavior. It is clearly not computable whether or not two methods with different code blocks model the same behavior.

# 3 The *MultiView* Approach

In this section, we outline *MultiView*, our approach for supporting *multiple view schemata* in OODBs [14]. *MultiView* breaks view specification into four subtasks, illustrated in Figures 2 through 4:

1. the derivation of virtual classes via an object-oriented query;
2. the integration of the virtual classes with existing classes into a single consistent global schema graph, maintaining relationships between base and virtual classes [13];
3. the selection of both base and virtual classes from the augmented global schema to participate in named view schemata; and
4. the construction of arbitrarily complex view schemata composed of these selected classes [13].

The separation of the view design process into a number of well-defined tasks has several advantages. First, it simplifies view specification, since each of the tasks can be solved independently from the others. Second, it increases the level of support by allowing for the automation of some of the tasks. The current

implementation of *MultiView* supports all four tasks, as further described below. Tasks 2 and 4 have been successfully automated so that they are carried out automatically when users execute tasks 1 and 3 using the specification language provided by *MultiView*.

## 3.1 Virtual Class Generation

The first task of *MultiView* supports the virtual customization of existing classes by deriving new classes with a modified type description and/or extent. This effectively controls the visibility of data and the access privileges to property functions. For this first prototype, we restrict the query language used for virtual class derivation to be an object-preserving algebra (i.e., queries do not generate objects that require new identifiers) [17]. Virtual classes in *MultiView* are thus automatically updatable [14] [17]. See Section 5.3 for a description of the object algebra operators.

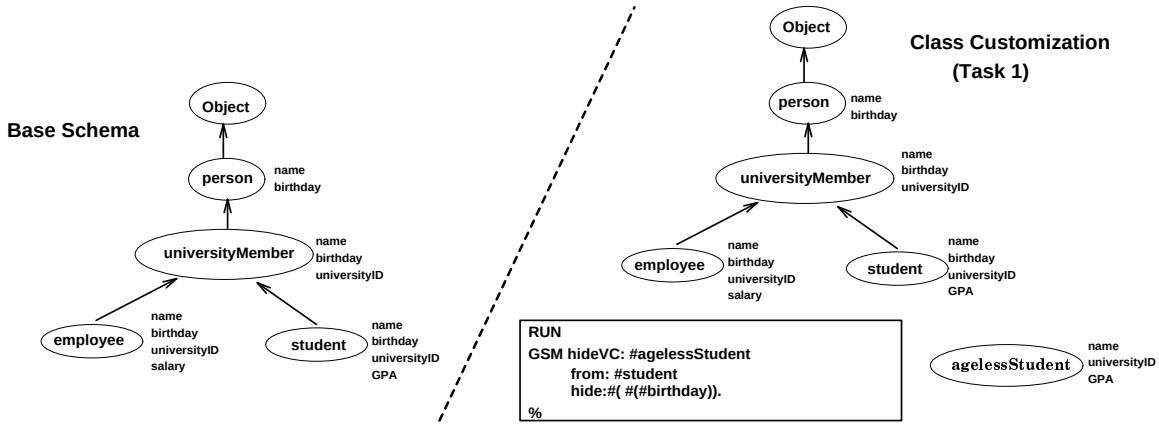


Figure 2: Derivation of virtual classes via a query.

Figure 2 illustrates this first task. The left half shows the original base schema, consisting of the original *person*, *universityMember*, *employee*, and *student* classes. The right half shows the hide query a user would type in if he or she wished to create a new class to hide the `age` attribute from the *student* class. The *MultiView* system uses the query to calculate the type of the *agelessStudent* virtual class. The virtual class now corresponds to a typical base class, except that its extent is computed rather than stored.

## 3.2 Classification

The second task – the creation of a consistent global schema – ensures the explicit capture of all class relationships between stored and derived classes in terms of type inheritance and subset relationships (rather than only between base classes as is typically done in object-oriented databases). *MultiView* integrates new classes by explicitly determining the subsumption relationships between the new virtual class and all other classes in the global schema, as opposed to the partial classification that would result from the examination of the query and the source class alone. This integration is vital for the consistent derivation of semantically correct view schemata, avoids the cost of recomputing these relations upon every request for class relationship information and/or query processing, and is useful for sharing (inheriting) property functions and object instances consistently among classes without unnecessary duplication. Figure 3 illustrates the result of integrating the *agelessStudent* class into the global schema. If virtual classes were not integrated with the base classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph. Automatic classification streamlines view creation by reducing the fourth subtask, the problem of determining the class generalization hierarchy for each of the view schemata, to a simple and efficient graph-theoretic algorithm [13], and more importantly, prevents the introduction of inconsistencies into the global schema.

Note that our approach of providing for the integration of virtual classes (derived using object-oriented queries) into a single unified global schema is distinct from others found in the literature. Existing approaches in the literature either: (1) require the user to specify explicitly the relationship between a virtual class and

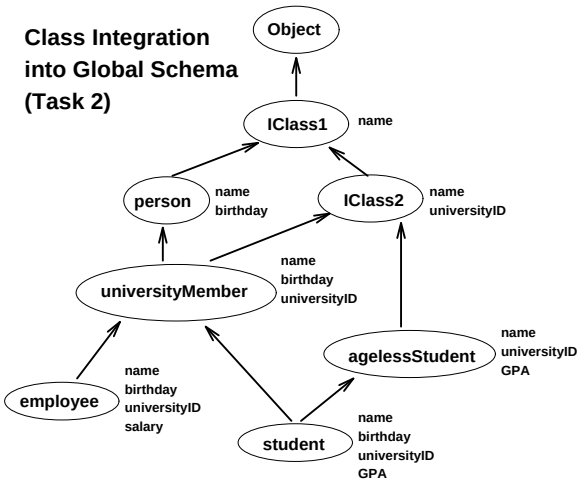


Figure 3: Integration of virtual and existing classes into a single consistent global schema.

existing base classes [22]; or (2) relate a virtual class only with its direct source class via a subclass/superclass relationship [17]; or (3) simply relate a virtual class with its source class via a *derived-from* relationship [3], (4) or with the root of the schema [9, 11].

The first approach is vulnerable to potential consistency problems, since the users might introduce an inconsistency in the schema graph by inserting is-a arcs between two classes not related by a subclass relationship. A solution of verifying the correctness of the relationship in essence would have to provide a capability similar to the automatic classification approach advocated in our system, namely, a means of automatically computing the *subsumes* relationships between pairs of classes. The second approach is prone to misrepresenting the subclass relationships normally represented in a class hierarchy, in particular, because a derived class may not be *is-a* related to its immediate source class. It would at best result in a partial, hence less informative, classification of class extents. The third approach ignores the issue of determining subclass relationships by introducing a parallel *derived-from* relationship hierarchy, which is not very informative in terms of relating different classes and their type descriptions. Note that in all other approaches given above, one would of course also maintain this *derived-from* relationship by keeping the class derivation query (which will be used to recompute the population of the virtual class, whenever needed). Finally, the last approach completely ignores the issue of classification, thus resulting in a flat class structure.

### 3.2.1 Classification Problems and Solutions

Two potential problems which could result during the automatic integration of new virtual classes into an existent class hierarchy are the inheritance mismatch problem in the type hierarchy and the problem of integrating *is-a* incompatible subset and subtype hierarchies into a single class hierarchy. These problems and their solutions are described in more detail in [13] and [14].

Inheritance mismatch is a problem of the type hierarchy which occurs when a new virtual class is created for which there is no existent correct place in the global hierarchy, as illustrated in the bottom half of Figure 2. The *agelessStudent* class cannot be placed directly above any existing classes in the hierarchy because there is no class whose type is a strict subtype of the *agelessStudent* type. Similarly, the *agelessStudent* class cannot be placed directly below any existent class because there is no class whose type is a strict supertype of the *agelessStudent* type. The *is-a* incompatibility problem results when the subtype and subset relationships between two or more classes conflict. For example, when a virtual class's extent may prescribe a place lower in the corresponding set hierarchy than its place in the corresponding type hierarchy, neither can be classified as a direct superclass nor a direct subclass of the other.

Our solution to both problems is to insert additional intermediate classes into the global class hierarchy, as shown in Figure 3. The addition of intermediate classes ensures that a unique, complete, and semantically-correct global schema can be calculated for any configuration of base and virtual classes. This strategy for integrating newly generated classes into the global schema guarantees the closure of the resulting class hierarchy. As described earlier, if two classes $C_1$ and $C_2$ share some common property then they both must

have inherited if from some lowest common superclass (LCS) in the class lattice for which this common property is defined. [13] presents an efficient algorithm for creating a minimal, yet sufficient, set of these intermediate classes. This approach supports true upwards inheritance of method code for both base and virtual classes. This avoids the following two problems: (1) code inherited by virtual classes does not have to be duplicated, which would cause possible maintenance and storage problems, and (2) the relatively simple upwards inheritance strategy does not have to be replaced by a complicated upwards and downwards one. In this paper we validate this method by describing a successful implementation of this approach using GemStone.

### 3.3 View Schema Generation

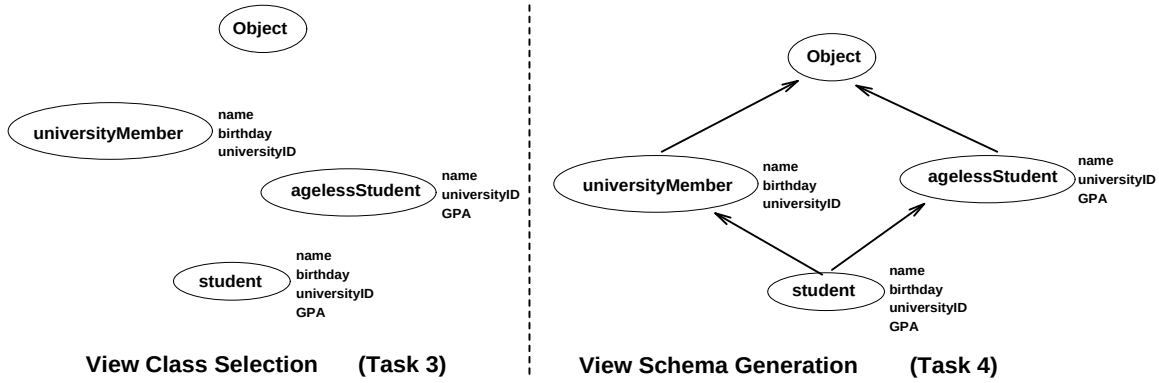


Figure 4: Construction of view schemata.

The third and fourth tasks of the *MultiView* approach are the selection of classes (both base and virtual) from the augmented global schema (illustrated in the left half of Figure 4) and the construction of view schemata composed of these selected classes (illustrated in the right half of Figure 4). *MultiView* uses the augmented global schema graph (generated in task two) for the selection of both base and virtual classes and for arranging these classes in a consistent class hierarchy, called a *view schema*. View schemata represent the virtual restructuring of the *is-a* hierarchy, allowing users to hide and/or highlight classes.

Virtual classes in *MultiView* are completely defined by a class derivation query from which both the type description and the extent of the virtual class are derived. All subclass relationships are calculated a priori for each pair of classes. The result of this evaluation is reflected in the global schema, hence a valid view schema can be derived from a global schema by simply exploiting the *syntactic* graph structure of the global schema rather than by requiring the *semantic* comparison of class specifications for each pair of classes.

We do not support the further modification of this virtual class specification due to its inclusion in a view schema; rather a virtual class will look the same, and exhibit the same behavior, in any of the view schemata in which it is included. This feature of *MultiView* is a significant difference to other approaches. For instance, in [12], the specification of a virtual class (both type and extent) has to be dynamically recomputed for each view schema it is inserted in, since for example the addition of an is-a relationship may add new inherited attributes to the virtual type.

In *MultiView* a view schema is instead defined simply by collecting all virtual classes that are to be made available to a particular user into one schema. While this selection of classes for a particular view (which could be either virtual and base classes) is done explicitly by the user, the generation of view relationships among the set of selected classes of a view schema is automated in the current version of our system.

Given this approach, is-a relationships between completely defined (virtual or base) classes are restricted by the subset and subtype relationships of the classes as defined in Section 4. That is, because inserting arbitrary *is-a* relationships between classes in a view schema may result in an incorrect schema in terms of property inheritance and subset relationships, rather than requiring the manual insertion of *view is-a* arcs by the view definer, we have developed algorithms that automatically augment the set of selected classes with their generalization relationships to generate a *valid* view schema.

Automatic view generation offers numerous advantages, some of which are listed below:

- It simplifies the view specification process for the users by automating tedious tasks.
- It guarantees the consistency of the view schema (i.e., correctness of view query processing).
- It prevents the introduction of redundant subclass relationships into the view (and thus supports a cleaner model of application domains).
- It may reduce execution times for query processing on the view.
- It assures the completeness of the view semantics by guaranteeing the presence of all required subclass relationships (providing maximal information to the user of the view about the class relationships).

The view generation problem can be reformulated as a graph-theoretic problem where we are given a global schema $GS = (V,E)$ and assume that a subset of classes VV $\subseteq$ V of $GS$ has been selected (marked) to belong to the view schema $VS$. The algorithm then determines a set VE of *is-a* edges among classes in $VV$ such that $VS = (VV, VE)$ is a valid view schema [14]. We can apply standard graph algorithms to solve the view generation problem as we proposed elsewhere in [16].

# 4 View Management Modeling Requirements

The *MultiView* data model is similar to the OPAL model used by GemStone, and is fully object-oriented, supporting classes, class methods, object instances, object identifiers, complete encapsulation[4], and many other features [19]. However, *MultiView* adapts GemStone's native data model to concur more closely with the ODMG data model [2] and to incorporate features necessary for the support of updatable views. The ODMG data model distinguishes between class and type, it advises that each class should maintain knowledge of its own extent (the set of objects belonging to the class), and it provides a model for class and type subsumption. *MultiView* adopts all of these recommendations. In addition, *MultiView* also extends the GemStone data model to incorporate features necessary for the support of updatable views, such as dynamic reclassification, multiple inheritance, multiple class membership, and multiple type instantiation [17].

We are designing a view management system with the overall goal that the view schemata function as *virtual databases*, which influenced our modeling choices for which features the underlying object-oriented system must provide:

- whether to support single or multiple inheritance
- whether to support multiple type instantiation
- whether to support multiple class membership
- how to define the relationship between a class, its type, and its extent
- whether to separate class and type hierarchies or to support a single global class hierarchy
- whether to support integration of virtual classes as first-class citizens into the schema graph
- whether or not to support dynamic modification of the class hierarchy
- whether to support automatic or manual placement of a class within a given class hierarchy
- how to determine method equivalence and identity

The support of a view manager requires particular choices for some of these data model questions. In particular, we also have specific sub-objectives:

- Users must be able to create updatable virtual classes using queries.
- Virtual and base classes must be integrated into a single consistent global schema.
- Users must be able to specify view schemata.
- As many routine tasks as possible must be automated.

Because some virtual classes (such as those formed by intersection queries) require the inheritance of properties from more than one class, we need multiple inheritance in order to allow a type to inherit properties directly from any number of supertypes. Because we want objects to be able to possess both virtual and base types, we must support multiple type instantiation and multiple class membership. Because virtual classes

[4] Instance variables (or attributes) cannot be directly manipulated by other classes or methods, but rather must always be accessed using access specific methods defined by the source class. GemStone provides a system-function that lets you automatically generate the typical access functions to get and set the values of instance variables. Thus, when we compare types and classes, we consider the methods of a class to include the access methods for attributes associated with its type description (as opposed to comparing attributes).

derive both their types and extents from base classes via queries, we must define the concept of class to incorporate both type and extent. Because we want virtual classes to inherit properties just like base classes, we must integrate them into a single global generalization hierarchy. However, because we want to be able to derive virtual classes over the lifetime of the database, we must be able to reclassify classes dynamically. Furthermore, objects must be able to *dynamically* take on new types and new class membership. Because we would like a deterministic classification algorithm, we need a definition for subsumption that applies to both base and virtual classes. Finally, as discussed in Section 2, our need for a deterministic positioning algorithm affected our technique for method-identification in the context of our datamodel.

# 5 Implementation of the *MultiView* Prototype

Ideally, we would have liked to implement *MultiView* on top of an object-oriented system that supports the key properties listed above. However, most available systems do not support the majority of these features. We chose to use Servio Corporation's GemStone OODBMS rather than to implement *MultiView* from scratch because it provides a rich object-oriented data model with supporting tools. Despite the differences between the GemStone and *MultiView* data models, GemStone offers key features which were extremely useful in the implementation of *MultiView*. Besides the typical database functionalities, such as persistence, database programming language support, and composite objects, GemStone features include:

- GemStone provides automatic, system-maintained object identity.
- GemStone treats everything in the system, including code blocks and classes, as objects.
- GemStone offers a number of programming language interfaces, such as C, C++, and Smalltalk, which facilitate the development and integration of a graphic interface.
- GemStone permits access to the source code for most methods, whether system or user defined.

As discussed in Section 4, the *MultiView* data model differs from GemStone's in a number of fundamental ways. Our implementation had to reconcile the differences between the properties needed to support a view system and GemStone's data model, which are summarized below:

- GemStone does not maintain explicit extents to collect all instances of a type, which is needed for the specification of select virtual classes.
- GemStone does not support multiple type instantiation, which is a required characteristic for view support if a given object is to possess both the base class's type and the virtual class's type.
- GemStone does not support multiple class membership for objects, which is necessary if an object is to be considered to be an instance of both a select virtual class and the source base class.
- GemStone does not support multiple class inheritance, which is needed because some classes, such as those defined by intersection queries, inherit methods from multiple superclasses.
- Schema evolution in GemStone is severely restricted for classes with instances, which would prevent the classification of virtual and base classes into a single global schema.

In the following section, we describe the architecture we have designed and built on top of GemStone in order to implement *MultiView*. This implementation approach successfully addresses the data model differences discussed above.

## 5.1 *MultiView* Class Architecture

The *MultiView* system can be integrated with Smalltalk-based systems by adding the set of generic *MultiView* system classes at the user level. Because we must keep track of both virtual and base classes, along with maintaining extent and type information, *MultiView* represents database objects using three disjoint levels of constructs illustrated in Figure 5:

- The *meta-schema classes* of *MultiView* are used to hold information about the classes in the global and view schemata and the relationships between the classes (top level).
- The *schema objects* – the global schema, the view schemata, and the classes contained in the user-visible schemata of *MultiView* – are each represented by an instance of some meta-schema class (middle level). Each class knows about its superclass/subclass relationships through the `subs` and `supers` instance variables of its metaclass instance.

- Finally, the object instances of *MultiView* (the base as well as the virtual objects) are maintained at the bottom *data level* underlying the meta-schema and schema classes.

In GemStone terminology, *MultiView* metaclasses are implemented as GemStone classes, *MultiView* user-defined classes are represented by GemStone object instances (with the base classes also captured in parallel by GemStone classes), and the *MultiView* object instances are implemented by GemStone object instances.

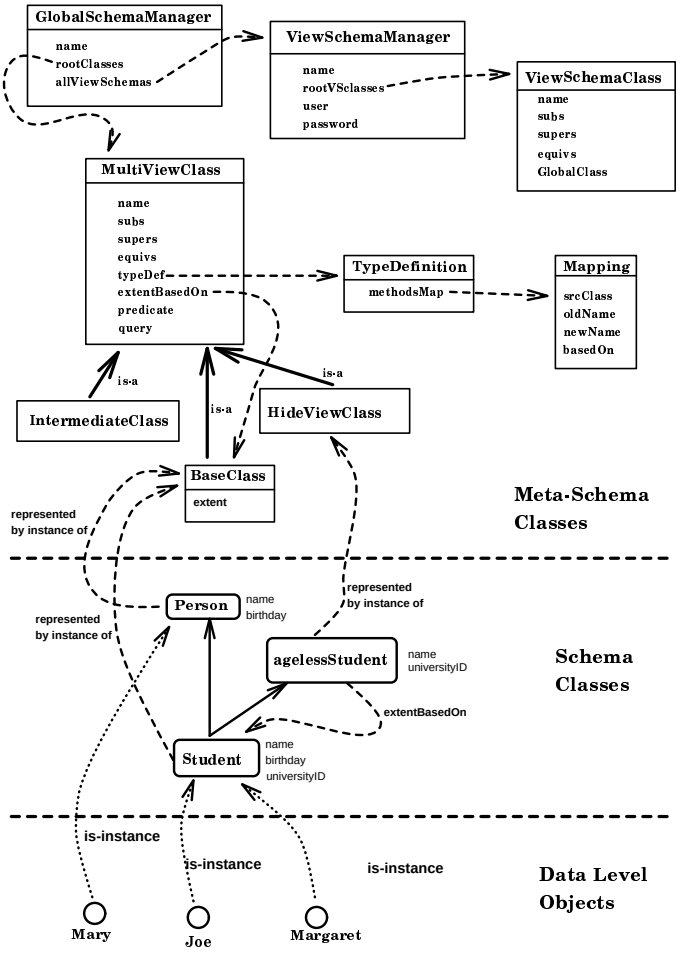


Figure 5: *MultiView* System Architecture.

For example, in Figure 5, the *student* class is a class in the example application schema (illustrated in the middle section of the figure). It is represented in the *MultiView* system by an instance of the *BaseClass* meta-schema class. This meta-schema instance also has associated with it the instances belonging to the extent of the *student* class. Similarly, the *agelessStudent* schema class (formed by a hide query) is represented by an instance of *HideViewClass* meta-schema class. The *agelessStudent* class's extent is based upon the *student* class, and thus the `extentBasedOn` instance variable of the *agelessStudent* class points to the *student* class.

There are three basic types of classes in the *MultiView* system:

1. All meta-classes that represent application classes, such as `HideViewClass` and `BaseClass`, are subclasses of the `MultiViewClass` class.
2. The `GlobalSchemaManager` and `ViewSchemaManager` classes provide an interface to the view management system.
3. Attributes of classes, such as their types, methods, and predicates, are represented by instances of component classes such as the `TypeDefinition`, `Mapping`, and `Predicate` classes.

### 5.1.1 The MultiViewClass Meta-Class

Since we need to explicitly collect the extents for classes in a view system, every class that participates in the global schema is represented internally by an instance of some subclass of the meta-schema class *MultiViewClass*. The subclasses of *MultiViewClass* such as *BaseClass*, *ViewClass*, and *HideViewClass*, represent the various types of classes, namely, either base or virtual classes,

```
Object subclass:          #MultiViewClass
          instVarNames:      #( #typeDef
                                #extentBasedOn
                                #query
                                #predicate #extendedPredicate
                                #subs #supers #equivs )
          classVars:         #( )
          inDictionary:      UserGlobals
          constraints:       #[ #[ #typeDef, TypeDefinition ],
                                #[ #extentBasedOn, Array],
                                #[ #query, String],
                                #[ #predicate, PredicateSet],
                                #[ #extendedPredicate, PredicateSet] ].
```

*TypeDefinition* is a set of methods to which objects belonging to the class can respond. *ExtentBasedOn* is the set of classes on which the MultiViewClass is based [5]. For example, in Figure 5, the extent of the *agelessStudent* class is based on the *student* class, since the former has been derived from the latter using a hide query. *Query* is a stored copy of the query defining the virtual class. For example, as shown in Figure 6, the *agelessStudent* class could have been created using a query to hide the `birthday` method from the *student* class. *Predicate* is a collection of the predicates originally defining the extent of the virtual class in the query. The *ExtendedPredicate* is the set of all predicates that affect the extent (i.e., the *ExtendedPredicate* is the union of the predicate set of the class and all the predicates of the classes from which the class is derived). We incorporate extent directly within class definition in MultiView (as is done in numerous other approaches, [6] and is necessary for the extent of virtual classes to be calculated). Base classes are hence represented by a subclass of *MultiViewClass* with the added instance variable of *Extent* containing the set of instances belonging to the class. *MultiView* defines a `create` method to replace `new`, so that when a new object is created in *MultiView*, it is automatically added to its base class's extent. Note that the extent of a virtual class is computed at the time of reference rather than stored.

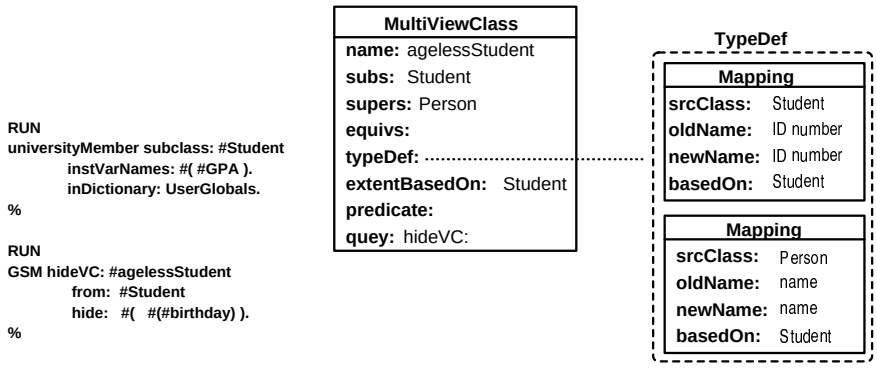


Figure 6: Type Definition of the *agelessStudent* class.

[5] These mappings correspond to the *derived-from* relationships between virtual and source classes, forming a *derived-from* class hierarchy orthogonal to the generalization hierarchy, which can also be found in other view approaches ([3]).

### 5.1.2 TypeDefinition and MethodMapping classes

In Section 4, we described *type definition* as the set of methods available to members of a particular class. However, because *MultiView* uses the underlying GemStone database as a "black box" on which to implement a view management system, and also because (as explained in Section 4) in the case of virtual classes *MultiView* identifies methods by their implementation rather than selector, methods are represented by *method mapping* structures within the type description.

```
Object subclass:      #TypeDefinition
       instVarNames:     #( #MethodMapSet)
       . . .

Object subclass:         #MethodMapping
       instVarNames:     #( #srcClass #oldName #newName #basedOn)
       classVars:        #( )
       inDictionary:     UserGlobals
       constraints:      #[ ] .
```

The MultiViewClass subclasses use instances of the meta-class Mapping to represent the relationships between methods of base and virtual classes for the type definition of each database class. In the first prototype implementation, each instance of the Mapping class provides one-to-one mappings between methods and their origins. Each mapping keeps track of the GemStone class where the source code for the method/attribute is based (*srcClass*), the selector (name) of the method/attribute in the GemStone class where it is defined (*oldName*), the current selector of the method/attribute in the virtual class (*newName*), and a reference to the class from which the method came (*basedOn*). As virtual classes are created, the mappings "daisy-chain" through the inheritance graph to ensure that the methods are associated with correct code blocks. Figure 6 illustrates the internal type description of the *agelessStudent* class from Figure 5. Note that each method of the class is represented by an instance of the *Mapping* class.

As discussed earlier, in orer to avoid this overhead of "daisy-chaining" and to support true dynamic method resolution, we are now augmenting our system to actually move code blocks upwards.

### 5.1.3 GlobalSchemaManager and ViewSchemaManager classes

Access to all classes that make up a particular database is maintained by a single instance of the meta-class GlobalSchemaManager (Figure 5, top). The GlobalSchemaManager type has instance variables to store information about the classes contained in the global schema, including a pointer to the root of the database, and pointers to all of the view schemata formed upon the database.

```
Object subclass:         #GlobalSchemaManager
       instVarNames:     #( #name #allVSM #roots)
       classVars:        #( )
       inDictionary:     UserGlobals
       constraints:      #[ #[ #name, Symbol ] ].

Object subclass: #ViewSchemaManager
          instVarNames: #( #name #allVSC #user #password)
          classVars: #()
          inDictionary: UserGlobals
          constraints: #[ #[ #name, Symbol ] ].
```

Each view schema, or user-selected subschema graph, is represented by a single instance of the meta-class ViewSchemaManager. An instance of the meta-class ViewSchemaClass is created to represent every class that participates in a particular view schema. Besides pointing to the view schema classes, the ViewSchema-Manager meta-class also maintains a list of users who are permitted to access, add, and remove the classes associated with a given view schema, and a password to restrict who can delete the view schema.

## 5.2 *MultiView* System Methods

The various methods used for the specification, creation, classification, and manipulation of virtual and base classes can be divided into four groups: (1) those that treat the schema as a whole are associated with the GlobalSchemaManager class, (2) those that deal specifically with classes on an individual basis are associated with the MultiViewClass class and its subclasses, (3) those used to specify and manipulate view sub-schemata formed from the global schema belong to the ViewSchemaManager class, and (4) those used for data-definition and data-manipulation at the object instance level. These last correspond to all regular OPAL methods provided by the GemStone system.

For example, the GlobalSchemaManager type includes system methods to manipulate classes within a database and user-interface methods that provide a front-end to system methods, allowing users to add, define, and access both base and virtual database classes, process user queries, and place a new class within the global schema. On the other hand, the MultiViewClass is responsible for the methods used to determine the subsumption relationship between one class and another via the comparison of predicates and types.

## 5.3 The *MultiView* User Interface

Below, we describe the chief methods that form the user interface to *MultiView* (summarized in Figure 7):

### 5.3.1 Initialization and Base Class Operators

The user initializes an instance of the GlobalSchemaManager class to start up the *MultiView* interface to GemStone. All new base and virtual classes are initialized and added to the database using the GlobalSchemaManager instance, as are instances of base classes. Note that because the base classes are created using OPAL (and GemStone's OPAL supports only single inheritance), they cannot be originally declared as a subclass of multiple classes.

**createGSM:** – creates an instance of the *GlobalSchemaManager* meta class.
**baseVC:** – initializes and adds a base class to the GSM as an instance of the *BaseClass* meta class.
**addBVCInstance:** –adds an instance to the extent of the corresponding base class.
**removeBVCInstance:** – removes an instance from the corresponding base class.
**getVC:** – retrieves a class based on its name symbol.
**do:** – iteration interface for a class's extent.
**extent** – returns the extent of a class.

### 5.3.2 View Schema Operators

Users can create any number of individualized view schemata by creating view schema managers and adding base and virtual classes to them. Each view schema has its own access control list, which regulates which other GemStone users are allowed to manipulate the schema.

**createVSM:password** – creates a new instance of *ViewSchemaManager* and stores the password.
**removeVSM:password** – allows users who know the appropriate password to remove an instance of *ViewSchemaManager* from the global schema.
**addVSC:to:** adds an instance of *ViewSchemaClass* representing a class from the global schema to the specified ViewSchemaManager instance. Only users who are in the access control list of the *ViewSchemaManager* instance are able to execute this method.
**removeVSC:from:** removes a class from the specified *ViewSchemaManager* instance. Only users who are in the access control list of the *ViewSchemaManager* instance are able to execute this method.
**adduser:to:password:** adds a new userId to the access control list.
**removeuser:to:password:** removes a userId from the access control list.

### 5.3.3 Virtual Class Operators

The following operators are used to create virtual classes. Currently all view operators are object-preserving, but we are in the process of investigating the issues involved with the support of object-generating operators.

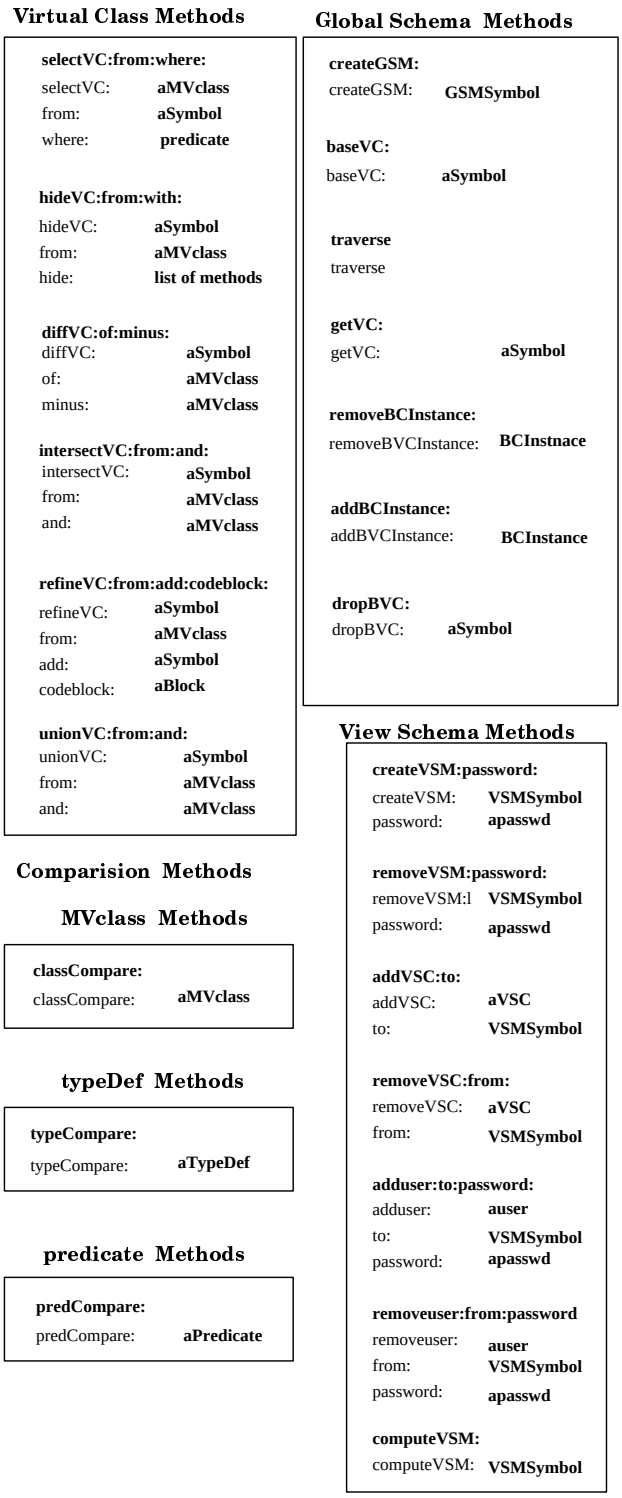


Figure 7: *MultiView* User Interface

**selectVC:from:where:** – creates an instance of *SelectViewClass*. A virtual class formed by a select query has a type definition that refers directly to the type definition of the origin class. In addition, it maintains a structured list of the predicates used to determine the class's extent. The format of the selection is a single free variable GemStone block, i.e. a conjunctive collection of predicates of the form: `(<single method> <comparator> <value>)`.

**hideVC:from:hide:** – creates an instance of *HideViewClass*. The *hide* clause is an array of methods to be excluded from the new class. The members of the *hide* clause array take the form of a one symbol array for method exclusion. For example, in order to hide the methods x and z, one would use the following *hide* clause *#( (#x) (#z) )* .

**refineVC:from:add:codeblock:** – creates an instance of *RefineViewClass*, which adds a new method (as opposed to instance variable) indicated in the specified block of code to the ViewClass. The new method must be defined in terms of existing methods.

**intersectVC:from:and:** – creates an instance of *IntersectViewClass*. An intersect view's type definition is the union of the types of the two origin classes, and the new class's extent is the intersection of the extents of the origin classes.

**diffVC:of:minus:** – creates an instance of *DifferenceViewClass*. A virtual class formed from a difference query has a type definition that points directly to the type definition of the first origin class, and the new class's extent is the extent of the first origin class minus the extent of the second origin class.

**listClasses** – returns all of the classes in the global schema in depth-first search order, listing each class's subclasses and superclasses.

### 5.4 Discussion

Despite the difference in data models, the experience of working with GemStone was a positive one because of features such as GemStone's support for object identity and the way in which GemStone treats everything in the system as an object. Although it may have been preferable to have completely integrated *MultiView* and GemStone source code into a seamlessly unified system, *MultiView* has been implemented on top of GemStone's OPAL in such a way as to preserve the functionality of GemStone. Until commercial systems begin to support full view management capabilities, portable view management systems such as *MultiView* offer a realistic and practical alternative.

As a consequence of supplying the properties needed to support a view management system, *MultiView* extends the GemStone data model to support multiple type instantiation, multiple class membership, and multiple inheritance. Because *MultiView* supports automatic classification of virtual and base classes into view and global schemata, *MultiView* adds dynamic reclassification to GemStone. *MultiView* also maintains explicit extent for classes to collect all instances of types. Note that *MultiView* users can still apply all relevant GemStone functions to objects within *MultiView*.

The current system allows users to create and delete virtual classes; to create, define, manipulate, and delete individualized view schemata; and to update both base and virtual object instances. Although the current authorization implementation provides rudimentary authentication security (associating a password and access control list with each view schema), because authentication was not the focus of our project, the authorization mechanisms are not very robust.

Finally, because some virtual classes (such as those formed by hide queries) are the superclasses of the classes on which they are defined, we support the transfer of method code blocks. This reorganization allows the projected methods to belong to the virtual superclass and be inherited by the base subclasses, i.e., the regular downwards inheritance scheme is preserved.

## 6 A Demonstration Application

Figure 8 illustrates the view creation process in *MultiView*, using a few queries from an example that has been successfully run on the system. The left side of each section of the figure depicts a graphic representation of the global and view schemata, while the right hand side contains a boxed partial transcript of the corresponding *MultiView* session. Figure 8.a portrays the original global schema, to which the user has added the base classes *person*, *universityMember*, *employee*, and *student*. Figure 8.b shows the global schema after the user has created the virtual class *agelessStudent* by hiding the `birthday` method from the *student* class. Note that two intermediate classes needed to be formed in order to integrate the *agelessStudent* class. This also resulted in code movement; for instance, the `name` method is now defined in the *IClass2* so

that it can be inherited by the *universityMember* class as well as by the new virtual class, *agelessStudent*. Due to the method and extent mapping described earlier, the resulting global schema is closed – new virtual classes can be specified against the *agelessStudent* class, and updates made to instances of the *agelessStudent* class will propagate back to the original objects. At any point during this process, the user is free to create any number of view schemata from the global schema. Figure 8.c shows the commands the user would type to create and populate a view schema showing only the student-related classes.

Now, suppose that a user applied the following command (in this case invoking the `name:` update method) to the *agelessStudent* class.

```
(GSM getVC: #agelessStudent) do:
        [ :aStudent |
              ((aStudent universityID) = '5555555') ifTrue:
                  [ aStudent name: 'Gordon Bennet' ].  ].
```

In Opal syntax, this applies an update operation to the virtual class *agelessStudent*, changing the name of the *agelessStudent* with the universityID of '5555555' to the new name 'Gordon Bennet.' Figure 6 shows the type definition of the *agelessStudent* class, from which the Mapping for the `name:` method is retrieved. The association is then made to the code block for the `name:` method, and the update is performed upon the original object. In this example (see Figure 8.c), the update will be performed on the base objects in the *Student* class, e.g., on `e3`, `e4`, or `e5`.

# 7 Related Work

In recent years, a number of other researchers have written papers on object-oriented views. Below, we describe the relationship of some of this work to our approach based on the comparison table in Figure 9. The rows of the table refer to the different view approaches being compared, generally listing the first view-related paper by the respective first authors. The first row is our *MultiView* approach [13] [14] [16] [15]; row two corresponds to the view approach proposed by Abiteboul and Bonner [1]; row three represents Bertino's view mechanism [3]; row four covers the *Polyview* system developed by Gilbert and Bic [8]; row five refers to Heiler and Zdonik's *FUGUE* model [9]; row six refers to Shilling and Sweeney's three step approach [20]; row seven represents the *COCOON* project by Scholl, Schek, Laasch, and Tresch [18] [17]; and row eight corresponds to the schema virtualization work done by Tanaka, Yoshikawa, and Ishihara [22].

The columns of Figure 9 represent the following criteria for comparing view management systems:

- The terms *object preserving* (along with *value generating* and *object generating*), as used by [18], refer to the closure property of query models – whether queries result in a collection of objects identical to the original database objects, a set of data values abstracted from the database objects, or a collection of newly created objects. It is an unresolved issue at this point whether object generating view definition languages can automatically solve the view update problem.
- Unlike relational databases, in which most views are not updatable [7], object-oriented views are potentially *updatable*. Two reasons why object-oriented systems potentially permit updatable views are that objects have unique, system-generated object identifiers, and class-specific methods are associated with each object. For example, because the *COCOON* system [18] [17] is object preserving, operations performed upon objects in views automatically take effect upon the actual objects on which the virtual objects are based, and similarly, updates on base objects are reflected in views. This column indicates whether or not the system offers updatable views.
- Some systems (such as [8] and [20]) associate multiple protocols (interfaces) with each class rather than having objects belong to multiple (possibly virtual) classes. We call this alternative to the traditional multiple inheritance data model *multiple protocols*. Such an approach would require considerable extensions to the typical object model, and as indicated in the final column, no implementation of this approach has been attempted. This column shows whether or not the system features multiple protocols.
- *MultiView* is one of several systems advocating the *integration of virtual classes into a global schema*. However, in most other systems, such as [22], the integration is manual, rather than automatic. Other approaches do not generate a global schema, but instead import individual classes into various view class hierarchies [1], or support only partial classification [17] [3] [9] [11] (as discussed in Section 3).

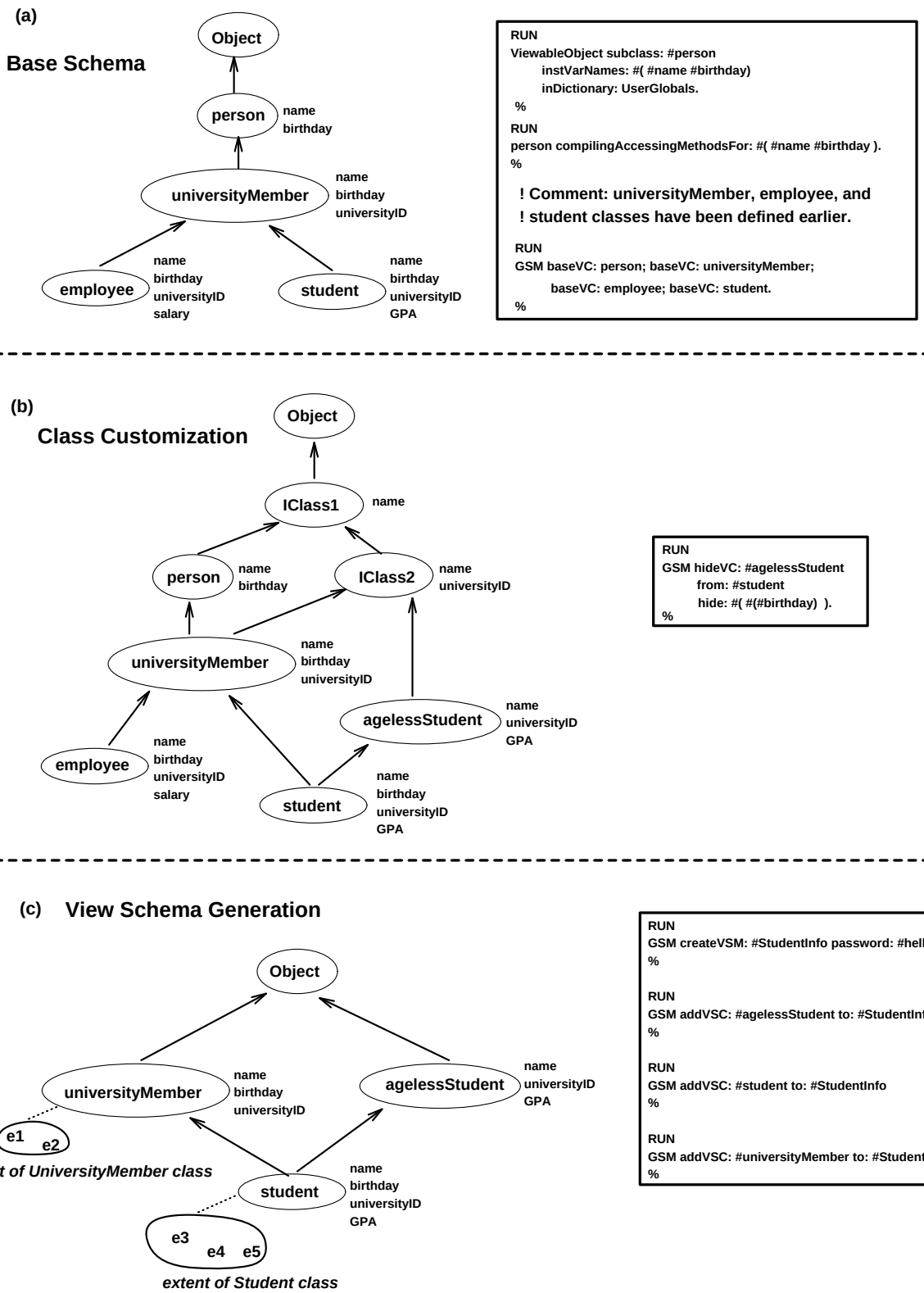


Figure 8: Example Session

| | object preserving | view updatability | multiple protocols | virtual classes integrated into global schema | inheritance hierarchy for view schema | upwards relocation of method code | closure checking for views | underlying object model | implementation status |
|---|---|---|---|---|---|---|---|---|---|
| MultiView '92 | yes | yes* | no | yes, automatic | yes, automatic | yes | yes | GemStone* | yes |
| Abiteboul '91 | no | yes* | no | no | yes, partial classify | no | no | O2 | planned |
| Bertino '92 | optional | optional | no | no | no | no | no | unknown O2? | unknown |
| Gilbert '91 | yes | yes | yes | yes (implicit) | not applicable | no | no | custom (simplistic) | no |
| Heiler '90 | no | yes* | no | no | no (indicated as issue) | no | yes | Fugue Model | unknown |
| Scholl '91 | yes | yes** | no | partial | yes | no | no | COCOON | yes |
| Shilling '89 | yes | limited | yes | no hierarchy | not applicable | no | no | custom | no |
| Tanaka '88 | no | not clear | no | yes, manual | yes, manual | no | no | Smalltalk | partial? |

yes* -- user -supplied methods  yes** -- generic operators  GemStone* -- extended

Figure 9: Related Work

The entries in this column indicate the degree to which virtual classes are integrated into the global schema.

- *MultiView* is the only system of those listed which features the *upwards relocation of method code*. In this context, the virtual classes are integrated into the global hierarchy and participate in the same inheritance scheme as the base classes, including dynamic upwards-resolution.
- Unlike relational views, which are queries resulting in virtual tables, object-oriented views can be thought of as collections of virtual classes, with each virtual class having its own set of methods defined. Whether or not these collections are composed into *view schemata supporting their own inheritance hierarchy* is an open issue. For example, [9] consider views to be simple collections of virtual classes, while others consider the views to be sub-schema graphs.
- Those systems that support the definition of view schemata in addition to virtual class creation should provide *view schema closure checking* to ensure that classes referenced by the classes participating in the view schema are visible with respect to the view.
- The term *underlying object model* indicates the system on which each view system is based. Some of these systems are based upon existing or commercial systems, while others are designed from scratch. For example, Scholl et al [18] have developed their own system, COCOON, while Tanaka et al [22] base their object model upon Smalltalk.
- As far as we know, no commercial system provides a general view management system for their object-oriented database system, and most of the systems proposed in academia have not yet been implemented. The final column indicates the *implementation* status of the various systems, so far as we could gather from the published literature.

We can compare the various approaches by classifying them into those that focus on view formation via a query language, through user-manipulation of the object schema graph, and by supporting multiple protocols. Most previous work regarding view systems for OODBs focuses on view formation to the exclusion of view incorporation. In fact, most researchers have focused on how query languages can be used to support the definition of virtual classes [9, 17, 22, 1, 14, 3]. In their discussion of *FUGUE*, Heiler and Zdonik [9] propose that the query language of *FUGUE* can be used for the specification of object-oriented virtual classes. They do not investigate either the issues of classification nor the re-use of methods. On the contrary, they require the view definer to manually enter the methods to be associated with a newly derived class, rather than deriving them automatically whenever possible, as done in our system. Abiteboul and Bonner [1] mention the integration of select classes into a view schema, but choose to enable selective upward versus downwards inheritance rather than creating intermediate classes and propagating methods upwards. To the best of our knowledge, an implementation of their approach using the O2 system is planned, but is still in progress.

Scholl and Schek's [18] work on views comes closest to our work; they have also developed a prototype of their approach [17]. They suggest use of an object-preserving subset of their algebra to define virtual classes and thus achieve updatable views. However, they do not address the classification of virtual classes into a global schema or the automatic generation of complete view schemata.

Others define view schemata through the manipulation of the object schema graph rather than solely by query languages. Tanaka et al. [22] propose that view schemata be defined by manually manipulating the edges in the global schema graph. [10] also uses DAG rearrangement for view schema definition. Such DAG manipulation approaches must deal with the issues of (1) possibly introducing inconsistencies into the view schema due to human error and of (2) unintentionally modifying the semantics of a virtual class due to side effects of graph manipulation. For example, in [12], the addition of an is-a relationship may add new inherited attributes to the virtual type, so the specification of a virtual class (both type and extent) is dynamically recomputed for each view schema in which it is inserted.

There are also some proposals on supporting multiple protocols (interfaces) for each class [8, 20]; such an approach would require however considerable extensions to the typical object model. To the best of our knowledge, no implementation of this approach has been attempted as of now. A comparison of this multiple protocol approach versus derived classes using a query language would be an important contribution to future object-oriented view research.

# 8 Contributions and Future Work

In this paper, we have described the implementation issues we faced when building a prototype of the *MultiView* object-oriented view management system using commercially available OODB technology. *MultiView* system classes extend GemStone's Opal model to include multiple inheritance, multiple type instantiation, multiple class membership, dynamic type changes, and explicit mappings between virtual and base methods. We discuss the impact of inheritance models on the implementation of classification. We also describe the various system metaclasses used in the implementation and the methods associated with them, and present the current user interface of the *MultiView* prototype. A unique feature of *MultiView* is that it automatically integrates newly derived virtual classes into a consistent global class hierarchy, thus simplifying the task of deriving complete customized view schema graphs to the simple selection of virtual classes.

The system as described in this report is functional. Note that the example described in Section 6 has been successfully run through the *MultiView* prototype. Much of the implementation code is general-purpose, so that it can be easily extended to serve as a view manager on top of another OODB. We are currently evaluating our prototype system using more extensive example applications.

The system is currently functioning as a test bed for the exploration of various issues, such as view materialization, view usage, etc. This prototype of *MultiView* was carried out with an eye towards simplicity and functionality rather than efficiency. A future version could examine various methods of optimizing view creation and classification, perhaps by pre-calculating intermediate classes or by optimizing queries. We also want to further examine the issues involved with implementing the system using object-splitting, with adding constraints to virtual classes (for example, constraining the insertion of objects into a virtual class), and with the calculation of query subsumption.

# 9 Acknowledgements

We thank Douglas Lee Moore, Trent Jaeger, and Alexandre Eichenberger, who helped design the meta-architecture and implement an early prototype of the current *MultiView* implementation; and also to Chris Ma, who helped implement the view schema generator.

# References

[1] S. Abiteboul and A. Bonner. Objects and views. *SIGMOD*, pages 238–247, 1991.

[2] T. Atwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade. The odmg object model. *Journal of Object Oriented Programming*, pages 64–69, June 1993.

[3] E. Bertino. A view mechanism for object-oriented databases. In *3rd International Conference on Extending Database Technology*, pages 136–151, March 1992.

[4] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *American Association for Artificial Intelligence Conference*, pages 34–37, 1984.

[5] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types – Lecture Notes in Computer Science, 173*, pages 51–67. Springer, 1984.

[6] R. G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Company, Inc., January 1992.

[7] C. J. Date. *An Introduction to Database Systems*, volume I. Addison-Wesley Systems Programming Series, Reading, Massachusetts, 1990.

[8] J. P. Gilbert. Supporting user views. *Computer Standards and Interfaces*, 13:293–296, 1991.

[9] S. Heiler and S. B. Zdonik. Object views: Extending the vision. *IEEE Data Engineering Conference*, February 1990.

[10] H. J. Kim. *Issues in Object Oriented Database Systems.* PhD thesis, University of Texas at Austin, May 1988.

[11] W. Kim. A model of queries in object-oriented databases. In *Proceedings of the International Conference on Very Large Databases*, pages 423–432, August 1989.

[12] M. M. A. Morsi, S. B. Navathe, and H. J. Kim. A schema management and prototyping interface for an object-oriented database environment. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Object Oriented Approach in Information Systems*, pages 157–180. Elsevier Science Publishers B. V. (North Holland), 1991.

[13] E. A. Rundensteiner. A class integration algorithm and its application for supporting consistent object views. Technical Report 92-50, University of California, Irvine, May 1992.

[14] E. A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *18th VLDB Conference*, 1992.

[15] E. A. Rundensteiner. Design tool integration using object-oriented database views. In *IEEE Int. Conf. on Computer-Aided Design*, November 1993.

[16] E. A. Rundensteiner. Tools for view generation in oodbs. In *ACM 2nd Int. Conf on Information and Knowledge Management (CIKM'93)*, November 1993.

[17] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proceedings of the Second DOOD Conference*, December 1991.

[18] M. H. Scholl and H. J. Schek. Survey of the cocoon project. *Objektbanken fur Experten*, October 1992.

[19] Servio Corporation. *Programming in OPAL*, version 2.5 edition, August 1991.

[20] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *OOPSLA*, pages 353 – 361, October 1989.

[21] L. A. Stein and S. B. Zdonik. Clovers: The dynamic behavior of types and instances. Technical Report CS-89-42, Brown University, November 1989.

[22] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema virtualization in object-oriented databases. *International Conference on Data Engineering*, February 1988.