The *MultiView* OODB View System: Design and Implementation [*]

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@umich.edu, rundenst@eecs.umich.edu
fax: (313) 763-1503
phone: (313) 936-2971

Abstract

Views are an established technique for restructuring and repartitioning the format of data, classes, and schemata so that applications can customize shared data objects without affecting other applications' perceptions of the data. The *MultiView* system is one of the first OODB systems to support dynamic and updatable materialized object-oriented database views. *MultiView* is fully functional and is being used for a number of projects. In this paper, we describe our system's architecture, the services it provides, and the decisions we made during our implementation. Although the GemStone system we chose for our implementation base offers many features that greatly aided our implementation, it does not support several key object-model properties that are critical for the realization of our design principles. These fundamental properties include multiple classification, dynamic object-restructuring, and the ability to make dynamic changes to the schema. In this paper, we describe a flexible and powerful technique known as object-slicing that we adopted to construct the *MultiView* object model – this now successfully addresses our requirements. The *MultiView* system is distinguished by a number of unique features, including the incorporation of virtual classes into the global schema as first-class database citizens, support for capacity-augmenting views (virtual classes that add new extrinsic properties or behavior), view materialization strategies that take advantage of object-oriented modeling features, and a graphical interface that is tailored to provide easy access to the *MultiView* system. The resulting system preserves all of the capabilities of the underlying GemStone OODB while providing support for dynamic and updatable materialized object-oriented views.

[*]This work was supported in part by the NSF RIA grant #IRI-9309076, NSF NYI grant #IRI 94-57609, and the University of Michigan Faculty Award Program. We would also like to thank our industrial sponsors, in particular, AT&T, Illustra, and Intel. Harumi Kuno is grateful for support from the NASA Graduate Student Researchers Program.

1 Introduction

Motivation and Background. Views provide logical data independence, and offer a means by which data can be repartitioned and restructured in format. Applications can use views to customize shared data objects, even adding new extrinsic properties and behavior, without affecting other applications' perceptions of the data. Views have been established as an effective method for virtually restructuring data, classes, and schemata so as to meet the needs of specific applications or users; for integrating heterogeneous and distributed systems; for security and access rights restriction; and for achieving interoperability by hiding the idiosyncrasies of component systems to be integrated into one unified, yet federated system [1, 4, 5, 6, 17]. Because views allow a database object to behave differently depending on the context in which it appears, views are a recognized technique for extending a database system with support for subjectivity.

Many research areas can profit from the flexibility provided by object-oriented views. In addition to the traditional use of views for facilitating the sharing of data by applications, current researchers are examining new areas that can specifically benefit from object-oriented views. Barsalou et al. use object-based views to integrate object-oriented and relational databases while preserving update capabilities [4]. Ra and Rundensteiner utilize views to provide transparent schema evolution, preserving existing views through schema change [30]. Because the object-oriented paradigm offers a more powerful model for integration than the relational one, several papers discuss the integration of heterogeneous data repositories via object-oriented views [5, 4, 17].

While a number of researchers have begun to study view mechanisms with regard to object-oriented databases (OODBs) [1, 15, 22, 36, 31, 33], little work has been done regarding implementation issues related to object-oriented views. To the best of our knowledge, commercial OODB systems do not yet support view capabilities. Furthermore, of the few research papers that discuss implementations of OODB view systems, most support only limited functionality at this time. There are several outstanding issues that must be addressed for the successful implementation of an object-oriented view system. First, creating views in an object-oriented model is not a simple transfer of the relational view solution to the object-oriented model. Much of the functionality typically provided by relational databases must be re-evaluated in the context of this new technology—for example, how to overcome the view update problem of the relational view mechanism, and how to utilize the complexity of the object-oriented data model for view definition (such as behavioral customization, view hierarchy manipulation, and property inheritance among view classes). Second, and more critically, to the best of our knowledge, no existing commercial OODB system supports all of the properties we identify as critical for the realization of fully-functional object-oriented views. These properties are quite fundamental, including features such as multiple classification, dynamic object-restructuring, and dynamic changes to the schema. This implies that either the data models of existing OODB systems must be extended, or other solutions must be found to address this problem.

The *MultiView* Approach. Here at the University of Michigan, we have an NSF-funded project developing *MultiView*, a view management system capable of supporting updatable materialized object-oriented views [31]. In the context of the *MultiView* project, we provide solutions to the issues outlined above. In order to facilitate the implementation of our system, we chose to use the commercial GemStone OODB as our base [1]. However, although GemStone offers many features that greatly aided our implementation, neither it nor any other existing commercial OODB system supports several of the key object-model properties that are necessary for the realization of our design principles. We therefore developed an object-slicing representational model, a flexible and powerful technique, that addresses these deficiencies. Using the object-slicing representational model, we were able to construct the *MultiView* object model, which provides all the features required for our view system, on top of GemStone. We ran an experimental evaluation of our system to determine the overhead that can be attributed to our choice of an object-slicing representation paradigm (in terms of both storage costs and performance). Our experiments confirm that the overhead is mixed, ranging from performance gains of 50% to increases in execution time of 200%, depending on the characteristics of the specific access patterns.

Our *MultiView* system is currently fully functional, provides clean object-oriented characteristics, supports dynamic updatable materialized views, and is being used for a number of projects. To the best of our knowledge, *MultiView* is one of the first (and perhaps the only) implemented OODB view system to support incrementally materialized views. Another unique feature of *MultiView* is that (again, to the best of our knowledge) *MultiView* is the only implemented OODB view system that treats virtual classes as first-class database citizens. We integrate both base and virtual classes into a unified global schema. Virtual classes

[1]GemStone is a registered trademark of Servio Corporation

in *MultiView* participate in the actual inheritance hierarchy and thus behave just like base classes. Furthermore, virtual classes in *MultiView* can independently define additional attributes and methods, i.e., they are capacity-augmenting views.

In previous papers we have described individual components of *MultiView*. [31] introduced the founding principles of the *MultiView* view specification methodology as a three-step process. [32] discusses *MultiView*'s strategy for the generation of view schemata, and [33] presents the classification algorithm by which new virtual classes are integrated into the global schema. In [24], we introduce some initial algorithms for incremental propagation of updates for maintaining materialized views in *MultiView*. In this paper we now present a unified overview of the *MultiView* project, focusing primarily on issues related to its system implementation. In particular, we outline the motivation and execution of our design decisions, system architecture, and the *MultiView* system's capabilities. The implementation solutions we describe are portable to other object-oriented systems.

System Overview. We exploit the unique features of the object-oriented paradigm in the design and implementation of the *MultiView* system. As shown in Figure 1, the *MultiView* system consists of a nested-layer architecture.

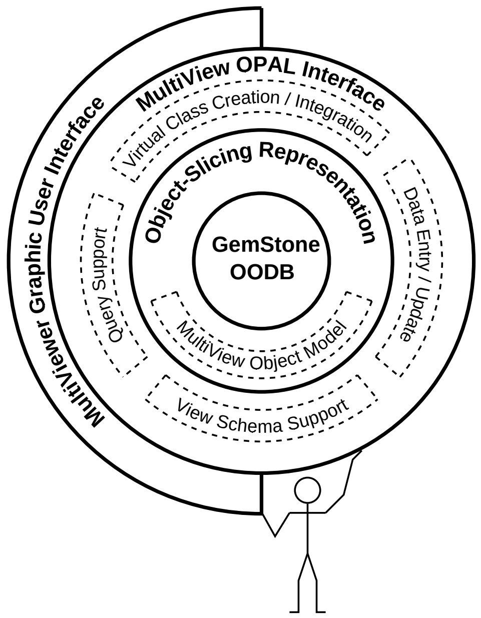


Figure 1: The Nested Layer Architecture of the *MultiView* System Design.

- The innermost layer of the figure represents the underlying GemStone database system kernel. GemStone provides many of the basic functionalities for our system, including persistence, database programming language support, and transaction management.

- The second layer illustrates how we overlay an object-slicing representation on top of the GemStone architecture in order to implement the fundamental object model needed for view support. We identified features that must provided by an object model for the flexible support of object-oriented views according to pure object-oriented principles. Like all the other OODB systems we surveyed, the GemStone OODB underlying our system does not support all of the properties required by our model. We hence designed the object-slicing representational layer, which resolves the significant differences between the underlying system and our object model. Object-slicing allows us to provide all of our required object model properties, including multiple classification, multiple inheritance, the dynamic restructuring of objects and classes, and dynamic changes to the class hierarchy.

- The third layer of the architecture represents *MultiView*'s support for materialized views and virtual schemata. Our implementation provides a Smalltalk interface that supports the creation of virtual classes and the automatic integration of the base and virtual classes into a consistent and correct global schema, the creation and customization of view schemata, the entry and update of data, and the retrieval of data via user queries.

- Finally, *MultiView*'s graphic user interface provides access to the system in an intuitive and user-friendly fashion. The *MultiView* graphic user interface displays global and view schemata in visually intuitive graphic windows, facilitates the creation of new virtual classes and view schemata, and provides an interface for querying class extents and data entry. The current version of the *MultiView Viewer*, or short, the *MultiViewer*, is written in Tk/Tcl using C.

As represented by the anthropomorphic stick-figure below the diagram of system components, users can access the *MultiView* system either through the graphic user interface or through *MultiView*'s Smalltalk interface. The *MultiView* system as described in this paper is fully functional. We have used *MultiView* to implement the University of Wisconsin's OO7 benchmark suite schema, populated the database with OO7's 10,000 object parts-assembly example, and have tested *MultiView* using both OO7 benchmark queries and queries of our own design. A summary of our performance evaluation, as well as of usage of the *MultiView* prototype in other projects, will be presented in later sections of this paper.

Outline of this Paper. In the remainder of this paper, we describe the object model, techniques, and algorithms underlying our implementation of *MultiView*. Our work should be of interest to researchers studying view systems. In addition, because many aspects of our experience, for example our use of an object-slicing technique, can be applied to the general support of object model features such as multiple classification and type-specific behavior, other implementors seeking to support subjectivity will also benefit from this work. We begin by discussing related work. The goals and object model of the *MultiView* project are set forth in Section 3. We present our approach of realizing object model features required for the support of our goals in Section 4, then discuss each of the components of our implementation in Sections 5 through 7. Finally, we conclude in Section 8 with a summary of the implementation status of *MultiView*, and a discussion of our contributions and future work.

2 Related Work

Relational Versus Object-Oriented Views. View mechanisms have been extensively studied for the relational model by [7, 13, 2, 16]. Although we can benefit from this work, there are some significant differences between relational and object-oriented views.

- Tables in relational databases (RDBs) are arranged in a "flat" fashion in the schema. There is no information about the subset or subtype relationships contained between tables in the schema. Both virtual and base classes in the *MultiView* model are arranged in an *integrated generalization hierarchy*. Information about the relations between the types and extents of OODB classes can thus be easily recognized by database users. Furthermore, update propagation techniques can be designed to exploit the object membership information that is implicit in the hierarchical structure in order to terminate propagation as soon as appropriate.

- Views in RDBs refer to virtual tables that are formed by applying query operations to base tables. Because an OODB schema is organized into a generalization hierarchy, OODB views include both virtual classes (formed by applying query operations to base classes) and virtual schema (formed by applying query operations to a base or view schema).

- Attributes in RDBs are identified only as the names of columns in tables. If two relational tables each have an attribute with the same name, there is no guarantee that the two attributes refer to the same property. Each property in the *MultiView* model is defined at a single class and using property inheritance shared by other classes. Database users can thus unambiguously identify whether or not two properties represent the same instance variable or method. In addition, when a predicate term in a virtual class's query invokes a given property, we can identify exactly which class defines that property, enabling us to *localize the effects of updates*.

- Attributes in RDBs are associated with simple values. Although the simple value can be a foreign key referring to a row in another table, RDB attributes cannot contain sets or other complex data structures. Instance variables in OODBs, on the other hand, can contain references to actual objects or sets of objects. OODBs thus contain *aggregation relationships* that can be used in view formation queries.

- The values of relational attributes are updated using generic queries, and thus update procedures must be implemented for the general case. On the other hand, because OODBs support *encapsulation*, updates to attributes are localized to specific accessing methods that are themselves defined at a single identifiable class. We can embed our update propagation methods into the accessing methods for the properties referenced by the predicates of materialized virtual classes. The accessing methods then serve as *update triggers*.

- Data in RDBs exists as values in tables. Thus when virtual tables in RDBs are materialized, the materialized tables typically contain *copies of data values* from the base tables [2]. On the other hand, data in OODBs exists as objects that can be uniquely identified and then accessed using unique object identifiers (oids). Thus when virtual classes in OODBs are materialized, the materialized classes can contain references to the relevant objects rather than value duplicates (copies) of the original objects. This not only saves space, but also significantly simplifies the view update problem. Furthermore, regardless of which properties are "hidden" by a "project" query, objects belonging to a virtual class can be easily associated with their base incarnations. In fact, this provides us with a basis for the arbitrary restructuring of an object's look and feel without compromising its identity.

Object-Oriented View Approaches. In recent years, several proposals for object-oriented view systems have appeared in the literature [1, 6, 15, 26, 36, 39]. *MultiView* differs from other view systems in that it does not simply adopt assumptions made by current OODB architectures; instead we re-examine key features required as foundation for views. For example, we overcome the stringent constraint that each object belongs to one and only one most-specific type—which is an unreasonable assumption for view systems. Rather than handle an object as being of a single fixed type, a *MultiView* object is distributed among multiple object-slicing implementation objects. The Iris functional database system resembles our system in that, being built on top of a relational engine, it distributes data over several relational tables [11]. Iris does not support view mechanisms, and does not address issues of classification, inheritance for virtual classes, etc.

Most previous work regarding view systems for OODBs focuses on view formation to the exclusion of view incorporation. Note that our approach of providing for the integration of virtual classes into a single unified global schema is distinct from others found in the literature. Existing approaches either: (1) require the user to specify explicitly the relationship between a virtual class and existing base classes [18, 41]; or (2) relate a virtual class only with its direct source class via a subclass/superclass relationship [36]; or (3) simply relate a virtual class with its source class via a *derived-from* relationship [6], (4) or with the root of the schema [15, 19].

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.

O2 Views [26] [35], based on Abiteboul and Bonner [1], is the first and only commercial implementation of an object-oriented view management system, currently realized. The O2 Views approach does include the integration of view classes into a view schema, but rather than supporting a global class hierarchy and migrating properties, it instead daisy-chains views and "bases."

Scholl et al. [36, 37] have developed an object-preserving algebra to define virtual classes and thus achieve updatable views. Their system, named Cocoon, has been implemented on top of a nested relational

[2]Note that some materialized RDB views may be implemented using techniques such as view indices that resemble membership materialization.

model. The *MultiView* object-algebra is similar in flavor to Cocoon's. However, Cocoon does not support the classification of view classes into a global schema, the automatic generation of complete view schemata, the implementation of capacity-augmenting views, nor the incremental support of *materialized* views.

Others define view schemata through the manipulation of the object schema graph rather than solely by query languages. Tanaka et al. [41] propose that view schemata be defined by manually manipulating the edges in the global schema graph. Kim also uses DAG rearrangement for view schema definition [18]. 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.

View Materialization Research. Relational and object-oriented systems share a common motivation for view materialization: the goal of improving query performance. Although not much work has been done regarding OODB view materialization, object-oriented and relational systems both must address a number of common issues when designing a view materialization system, such as when to evaluate updates and how much to update. Our research on view materialization in OODBs borrows several techniques from the relational arena. [7, 8] tests modified tuples to see if they fulfill view predicates, thereby detecting irrelevant and autonomously computable updates. This resembles our solution of filtering irrelevant updates by exploiting the generalization hierarchy and the derivation structure. The system provided by [10] performs incremental view maintenance using production rules that are triggered by update operations. Similarly, we override generic-update operations with type-specific update operators for virtual classes.

However, there are significant differences between view materialization in OODBs and relational systems. In relational systems a view is a virtual table, so a materialized view consists of stored values/rows. In OODBs, a view is a schema containing both base and virtual classes, so an object-oriented materialized view consists of a schema in which some of the "virtual" classes contain actual stored objects. Our OODB view approach supports membership materialization, meaning that we store references to the objects rather than copies of them, while the relational model typically duplicates data on view materialization. Object-identity eliminates the duplicate row problem of relational views in the context of our object-preserving model. Finally, OODB support for encapsulation and object-identifiers significantly eases the implementation of triggered incremental updates and the update of path query views.

As described in this paper, our approach incorporates several aspects (such as encapsulation and inheritance) unique to object-oriented view technology. For instance, because there is a unique point of inheritance for each property in the database, any modification to the value of an instance variable will take place at a pre-determined class's implementation object independent of through which base or virtual class the update request was specified. Also, because an instance variable's update method is always stored at the same location as the instance variable itself, it is a simple matter to determine which selection class should register where. Thus, when an object is updated, the update method triggers a *notification* function that informs all virtual classes that have *registered* with the class of the update to the object.

Only a few published papers address issues of view materialization in OODBs. [14] provide a view materialization model in which updates are propagated by use of change files, representing histories of design sessions. However, [14] duplicate objects (including identifiers) for virtual classes rather than merely storing references to objects. [20] address maintaining consistency for a particular type of join class formed along an existing path in the aggregation graph. Our work instead focuses on the exploitation of the structure of the schema hierarchy and derivation dependency graph in order to reduce update propagation. Our research is also unique in studying incremental updates in the context of the object-slicing paradigm.

Role Modeling Systems. *MultiView* uses an object-slicing mechanism to address the object model requirements underlying the support of object-oriented views. These requirements include multiple classification and dynamic object migration. The flexibility offered by the object-slicing approach naturally lends itself to implementing role systems: object-slicing's implementation objects can easily be adapted to represent the various roles of objects in a role system [12]. Although the object-slicing techniques underlying the current implementation of *MultiView* can be compared to mechanisms used in *role modeling* approaches [12, 27], no other research in the literature discusses the application of the object-slicing paradigm regarding the support of object-oriented views. In *role modeling* systems, objects dynamically gain and lose multiple interfaces (a.k.a. *roles*) throughout their lifetimes. These roles can be compared to the implementation objects of an object-slicing implementation, in that both permit objects to belong to multiple classes and change types dynamically. In some sense, accessing an object through one of its implementation objects is like accessing an object while it is playing one of its roles. However, role systems and views systems have different goals. Role systems strive to increase the flexibility of objects by enabling them to dynamically change types and class membership at the level of individual object-instances. Such changes are done explicitly by user request, and on an object-by-object basis. View systems, on the other hand, enable users to restructure the types and class membership of complete classes—based on content-based queries.

In [38], Sciore proposed an object specialization approach, in which a real world entity is modeled by multiple objects arranged in an object hierarchy. These object hierarchy objects inherit from each other, enabling each individual entity object to decide its own inheritance hierarchy. Although our implementation objects resemble object hierarchy objects in that they inherit from each other, objects in our implementation always conform to the existing global class hierarchy. This single-point-of-inheritance allows us to optimize view update propagation.

Unlike many role systems, which allow object hierarchies to exist independently from class hierarchies [38], objects in our model always conform to the existing global class hierarchy. In short, if an object possesses an implementation object of a given class's type, it must also possess an implementation object for every class that is a superclass of that given type. This achieves an efficient and uniform inheritance scheme. Also, unlike many role systems, in our implementation conceptual objects can be associated with at most one implementation object of a given type [12]. Role systems do not deal with the issues of virtual class derivation, classification, nor with method promotion.

The role system proposed by Gottlob et al. was implemented using techniques similar to object-slicing [12]. This system, like ours, is implemented in Smalltalk by overriding the `doesNotUnderstand:` method. The difference between [12] and our implementation is that [12] is a role system while we are developing a view system. Also, the [12] system does not permit the derivation of new virtual classes, and thus does not address issues related to view management.

Deputy Mechanisms. The *Deputy Mechanisms* proposed by Peng and Kambayashi unify the concepts of object views, roles, and migration in the form of deputy objects and deputy classes [28]. In the deputy mechanism paradigm, view objects are treated as roles of database objects. The deputy mechanism object model is probably the work most closely related to *MultiView*'s. In particular, the similarities of the two systems include the following: Both support capacity-augmenting views, dynamic classification, and multiple classification. Neither duplicate the state of an object when representing it as a virtual object. And finally, both support update propagation to materialized virtual/deputy classes.

There are some significant differences between the two systems, however. *MultiView* treats virtual classes as first-class database citizens, and thus integrates virtual and base classes into a unified global schema, whereas Deputy classes are not integrated into the global schema. Virtual classes in *MultiView* participate in the actual inheritance hierarchy and thus behave just like base classes. *MultiView* supports the automatic generation of view schemata composed of selected base and virtual classes, while the Deputy system does not support view schemata. Although both the Deputy and *MultiView* systems are based on Smalltalk, *MultiView* supports multiple inheritance. Deputy objects and source objects have independent object-identifiers, whereas a view object and a source object in *MultiView* share a conceptual object-identifier in addition to individual implementation object-identifiers. Finally, *MultiView* supports optimized incremental materialized view maintenance algorithms that exploit the integrated class hierarchy structure.

3 Object Model of *MultiView*

3.1 Goals of the *MultiView* Project

The purpose of the *MultiView* project is to build, implement, and evaluate a system for OODB view support. The goals of the project include the following:

- Users should be able to create customized virtual classes at any time.
- Users should be able to query and update both base and virtual classes.
- Users should be able to create and modify customized virtual schemata at any time.
- Virtual classes and schemata should be first-class citizens of the database.

By first-class database citizen, we mean that virtual classes and schemata should look and feel like base classes and schemata. Virtual and base classes should be fully integrated into a global class hierarchy in terms of both type and extent[3]. This implies that virtual classes should fully participate in the inheritance scheme

[3]Although we combine type and extent for the sake of consistency between virtual and base classes, our model can easily be extended to support models that separate type and extent.

in such a way that virtual classes can act as a point of inheritance for properties, and also that the extents of both virtual and base classes should be kept consistent with the class hierarchy. In addition, *MultiView* supports *capacity-augmenting* virtual classes. First introduced by [6], a capacity-augmenting virtual class is a virtual class that includes instance variables that are not derived from the source classes of the virtual class. Finally, the user should be able to select both base and virtual classes from the global schema at any time and add them to view schemata. The *MultiView* system should automatically integrate these selected classes into a consistent and correct view schemata.

In the following subsections, we give a description of an object model that fulfills the *MultiView* project goals. We first formalize our basic concepts and principles, then extend this initial description to include virtual classes and schemata and materialized views.

3.2 Basic Concepts and Principles

An **object instance** (or short, **object**) represents an entity. Anything with distinct existence in objective or conceptual reality can be represented as an object. Let O be an infinite set of object instances. Each object $O_i \in O$ consists of state (the **instance variables** or attributes of the object) and behavior (the **methods**, or messages, to which the object can respond). Because our model is object-oriented and assumes full encapsulation, any modification to an instance variable of an object in our system must be accomplished by means of an **accessing method**. An **accessing method** is a method that modifies or retrieves an instance variable. Together, the **methods** and **instance variables** of an object are referred to as its **properties.** In addition, 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.

Objects are grouped into sets of similar objects that share a common structure and behavior. The term **class** denotes this specification of a common structure and behavior. Let C be the set of all classes in a database. A **class**, $C_i \in C$, has a unique class name, a **type**, and a set membership (known as the class's **extent**), as defined below [4]. We use the term **type** to indicate the set of applicable property functions shared by all object-instances of the class. The set of property functions defined for class C_i is denoted as **properties**(C_i).

Definition 1 (type) *A* ***type*** *is a tuple,* ***<InstanceVariables, Methods>****, where InstanceVariables is the set of attributes (instance variables) possessed by the type and Methods is the set of* **methods** *defined by the type. We refer to the the type associated with a class, $C_i \in C$ by* ***type(C_i)****.*

Figure 3 shows the class that *MultiView* would generate based on the class definition given in Figure 2, creating a new class named *Person* that has instance variables (and accompanying accessing methods) to store a Person's birthyear and name. As shown in Figure 3, the *Person* class's type contains the instance variables `birthyear` and `name`, retrieval methods `birthyear` and `name`, and assignment methods `birthyear:` and `name:`.

Definition 2 (subtype, supertype) *For two classes C_i and $C_j \in C$, C_i is called a* ***subtype*** *of C_j, denoted by $C_i \preceq C_j$ if and only if (****properties(C_i)*** $\supseteq$ ***properties(C_j)****). C_j is called a* ***supertype*** *of C_i if and only if C_i is a subtype of C_j. All properties defined for a supertype are* ***inherited*** *by its subtypes.*

Definition 3 (extent) *The* ***extent*** *(also called* ***set-membership****) of a class $C_i \in C$ is the collection of object instances that belong to that class. The membership of an object instance, o, in the extent of a class, C_i, is denoted by $o \in C_i$. We refer to the set of objects that are implemented as* ***direct instances*** *of C_i (objects whose most specific type is C_i) as* ***LocalExtent(C_i)****, or short,* ***extent(C_i)****. We call the collection of all objects that possess a class's type the* ***GlobalExtent(C_i)****.*

For example, we can create a new instance of the Person class and assign values to its birthyear and name instance variables (Figure 4). This instance is automatically added to the extent of the *Person* class, as shown in Figure 5.

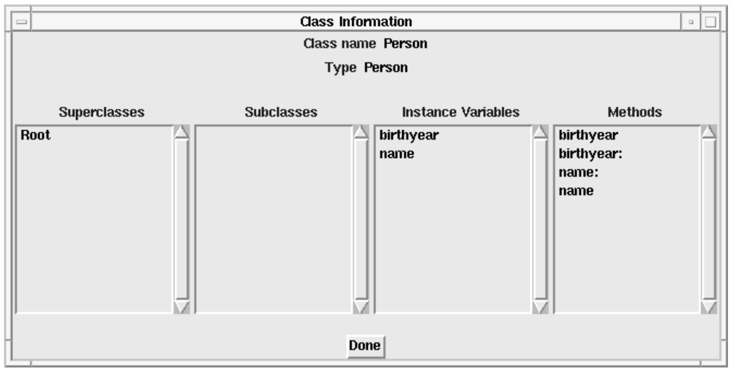
Definition 4 (subset, superset) *For two classes C_i and $C_j \in C$, C_i is called a* ***subset*** *of C_j, denoted by $C_i \subseteq C_j$, if and only if $(\forall o \in O)((o \in C_i) \Rightarrow (o \in C_j))$. Thus* ***GlobalExtent($C_i$)*** $= \bigcup$ ***LocalExtent(C_j)*** *$\forall C_j \subseteq C_i$. C_j is called a* ***superset*** *of C_i, denoted by $C_j \supseteq C_i$, if and only if $C_i \subseteq C_j$.*

[4]We associate both **type** and **extent** with our concept of **class**. Although there is no general agreement on whether or not classes in OODBs should incorporate their own extents rather than requiring users to maintain their own collections of class-instances, several systems follow this philosophy, including Orion and the system proposed by H. J. Kim [18]. Furthermore, the proposed ODMG standard [3] recently formulated by several key OODB vendors also follows this approach.

```
run
BaseClass createSubclass: #Person
    supers: #[ Root ]
    instVarNames: #( #birthyear #name )
    constraints: #[ #[birthyear, Integer],
                   #[name, String] ]
%

run
Person compileAccessingMethodsFor:
            #( #birthyear #name).
%
```

Figure 2: Code to create a *Person* class.


Class Information
Class name Person
Type Person

Superclasses	Subclasses	Instance Variables	Methods
Root		birthyear	birthyear
		name	birthyear:
			name:
			name

Figure 3: The *Person* class has instance variables and accessing methods for `name` and `birthyear`.

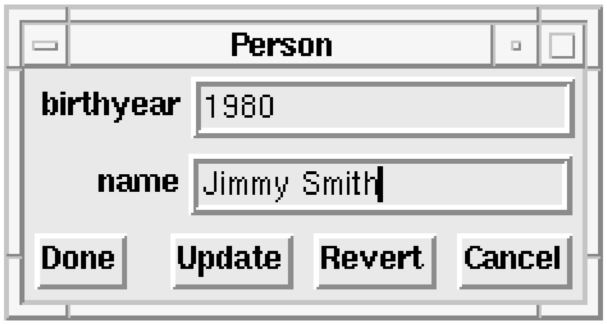

Person
birthyear 1980
name Jimmy Smith

Figure 4: Using the *MultiView* graphic interface to create a new instance of *Person*.

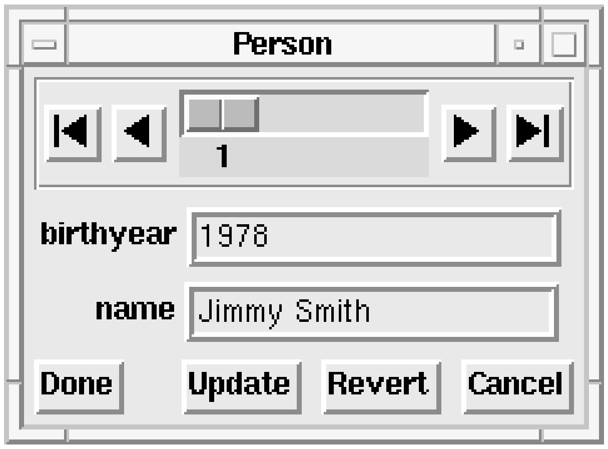

Person
1
birthyear 1978
name Jimmy Smith

Figure 5: The *Person* class's extent now contains the new instance.

Definition 5 (subclass, superclass) *For two classes C_i and $C_j \in C$, C_i is called a **subclass** of C_j, denoted by C_i* is-a *C_j, if and only if $(C_i \subseteq C_j)$ and $(C_i \preceq C_j)$. C_i is a **direct subclass** of C_j if $\nexists C_k \in C$ s.t. $k \neq i \neq j$, C_i is a **subclass** of C_k, and C_k is a **subclass** of C_j. Similarly, C_j is called a **superclass** of C_i if and only if C_i* is-a *C_j, and C_j is a **direct superclass** of C_i if $\nexists C_k \in C$ s.t. $k \neq i \neq j$, C_j is a **superclass** of C_k, and C_k is a **superclass** of C_i.*

Definition 6 (object schema) *An **object schema** is a rooted directed acyclic graph $G = (V, E)$, where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class $C_i \in C$, while E corresponds to a binary relation on $V \times V$ that represents all direct* is-a *relationships between all pairs of classes in V. In particular, each directed edge $e \in E$ from C_1 to C_2, denoted by **edge**(C_1, C_2), represents the relationship C_1* is-a *C_2. Two classes, $C_i, C_j \in C$, share a common property if and only if they inherit it from the same superclass. The designated root node, representing the **Object** class, is a class defined to have a global extent of all database object instances and an empty type description.*

Figure 7 shows the result of executing the code shown in Figure 6 in *MultiView*, thereby defining the *Student* and *Staff* classes as subclasses of our original *Person* class. *MultiView* will update the schema incorporating each new class into the global schema.

3.3 Views in OODBs

Now we extend the basic definitions given above to include the concept of virtual classes and virtual schemas. In the relational model, a **view,** or **virtual table**, is defined to be a **named, stored query**. Similarly,

```
run
BaseClass createSubclass: #Student
    supers: #[ Person ]
    instVarNames: #( #gpa )
    constraints: #[ #[gpa, Number] ]
%
run
Student compileAccessingMethodsFor: #( #gpa ).
%

run
BaseClass createSubclass: #Staff
    supers: #[ Person ]
    instVarNames: #( #title )
    constraints: #[ #[title, String] ]
%
run
Staff compileAccessingMethodsFor: #( #title ).
%
```

Figure 6: Code defining *Student* and *Staff* as subclasses of the *Person* class.

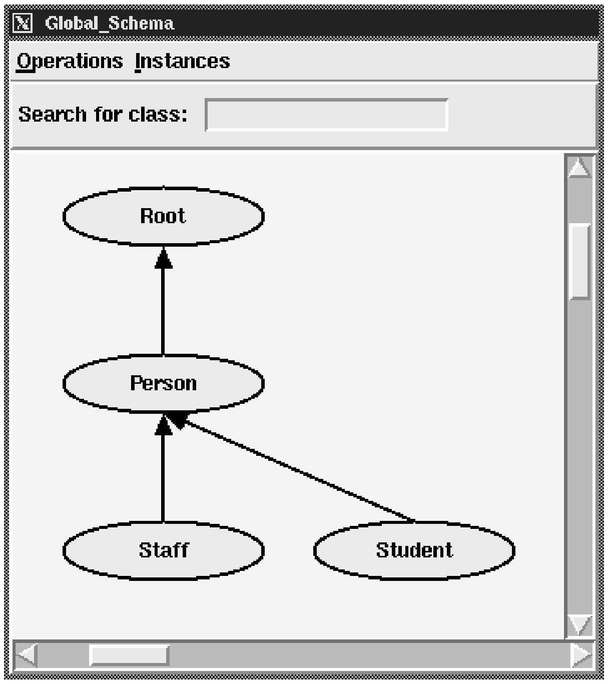


Figure 7: The new classes are automatically integrated into the Global Schema.

our object-oriented model permits the application of a query operator to a **source class** (or classes) that restructures the source class's type and/or extent membership in order to form a **virtual class** with a type and extent derived from its source class(es).

3.3.1 View-defining Queries

Let Q be the set of all possible queries. We constrain a **query** $Q_i \in Q$ used to define a virtual class to correspond to a single algebra operation. Figure 8 displays the view-forming queries currently supported by our model. If a complex query is specified by nesting algebra operators, then each intermediate algebra operator generates a separate virtual class [31].

Definition 7 *A* ***query*** $Q_i \in Q$ *includes the following components: <****QueryType,SourceClasses, TypeModifiers, MemberPredicates, ValuePredicates****>, where QueryType* $\in$ *–Select, Hide, Refine, Union, Intersect, Difference"; SourceClasses* $\in C$ *is an ordered list of the classes from which the class is derived (that is, the classes to which the query is applied); TypeModifiers defines the derivation relationship between the type of the virtual class and that of the source class from which it is derived; MemberPredicates is an ordered list of the class membership requirements that apply to instances of the virtual class, listed in the same order as the classes in SourceClasses to which they correspond; and ValuePredicates is the set of simple value predicates that constrain the extent of the virtual class. We refer to the query that defines a virtual class,* $VC_i \in VC$*, as* ***query****(VC_i).*

Currently we support the object algebra shown in Figure 8 for the derivation of virtual classes. These queries allow us to determine the methods, instance variables, and extent of the virtual classes.

Now suppose we were to use a *select* query upon class *Person* to define a new virtual class, *YoungPerson*: **createSelectClass:** `YoungPerson` **query:** `[ :person | person birthyear > 1975]`. We can derive all aspects of the new virtual class from this query. The new class's name will be *YoungPerson*, its type will have the same properties as that of the *Person* class, **SourceClasses**(*YoungPerson*) will contain only the *Person* class, **MemberPredicates**(*YoungPerson*) = ($\in$ *Person*), and **ValuePredicates**(*YoungPerson*) will contain only the simple predicate `(birthyear > 1975)`. After the *YoungPerson* class has been created, *MultiView* will then integrate the new virtual class into the class hierarchy and add the appropriate edge ($YoungPerson, Person$) to the object schema's edge set. The new object schema is shown in Figure 10.

hide	syntax	<*virtual*-class>) := (<*source*-class>) **createHideClass:** [<new-class-name>] **hideMethods:** [<prop-functions>])
	semantics	**type**(<*virtual*-class>) := {p $\in$ P $\mid$ p$\in$**properties**(<*source*-class>) $\wedge$ p$\notin$<prop-functions>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*source*-class> $\preceq$ <*virtual*-class> <*source*-class> $\subseteq$ <*virtual*-class> <*source*-class> *is-a* <*virtual*-class>
refine	syntax	<*virtual*-class> := (<*source*-class>) **createRefineClass:** [<new-class-name>] **withProperties:** [<prop-function-defs>]
	semantics	**type**(<*virtual*-class>) := {p$\in$P $\mid$ p$\in$**properties**(<*source*-class>) $\vee$ p$\in$<prop-function-def>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*virtual*-class> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
select	syntax	<*virtual*-class> := (<*source*-class>) **createSelectClass:** (<*new-class-name*>) **query:** (<predicate>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class>) **extent**(<*virtual*-class>) := {o$\in$O $\mid$ o$\in$<*source*-class> $\wedge$ <predicate>(o)=true}
	class rels	<*virtual*-class> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
union	syntax	<*virtual*-class> := (<*source*-class1>) **createUnionClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) $\sqcap$ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o$\in$O $\mid$ o$\in$<*source*-class1> $\vee$ o$\in$<*source*-class2>}
	class rels	<*source*-class1> $\preceq$ <*virtual*-class> $\wedge$ <*source*-class2> $\preceq$ <*virtual*-class> <*source*-class1> $\subseteq$ <*virtual*-class> $\wedge$ <*source*-class2> $\subseteq$ <*virtual*-class> <*source*-class1> *is-a* <*virtual*-class> $\wedge$ <*source*-class2> *is-a* <*virtual*-class>
intersect	syntax	<*virtual*-class> := (<*source*-class1>) **createIntersectClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) $\sqcup$ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o$\in$O $\mid$ o$\in$<*source*-class1> $\wedge$ o$\in$<*source*-class2>}
	class rels	<*virtual*-class> $\preceq$ <*source*-class1> $\wedge$ <*virtual*-class> $\preceq$ <*source*-class2> <*virtual*-class> $\subseteq$ <*source*-class1> $\wedge$ <*virtual*-class> $\subseteq$ <*source*-class2> <*virtual*-class> *is-a* <*source*-class1> $\wedge$ <*virtual*-class> *is-a* <*source*-class2>
difference	syntax	<*virtual*-class> := (<*source*-class1>) **createDifferenceClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) **extent**(<*virtual*-class>) := {o$\in$O $\mid$ o$\in$<*source*-class1> $\wedge$ o$\notin$<*source*-class2>}
	class rels	<*virtual*-class> $\preceq$ <*source*-class1> <*virtual*-class> $\subseteq$ <*source*-class1> <*virtual*-class> *is-a* <*source*-class1>
join	syntax	<*virtual*-class> := (<*source*-class1>) **createJoinWith:** (<*source*-class2>) **named:** (<*new-class-name*>) **withMethods:** (<*aMethodArray*>) **joinAtt1:** (<*property*>) **joinAtt2:** (<*property*>)
	semantics	**type**(<*virtual*-class>) := newly-generated type **extent**(<*virtual*-class>) := newly-generated extent
	class rels	<*virtual*-class> *is-a* <*Root*>

Figure 8: The Object Algebra Operators: Syntax, Semantics and Class Relationships.

```
run
Person createSelectClass:
                 #YoungPerson
  query:

  [ :a | a birthyear > 1975 ].
%
```

Figure 9: *MultiView* definition of a new *YoungPerson* virtual class.

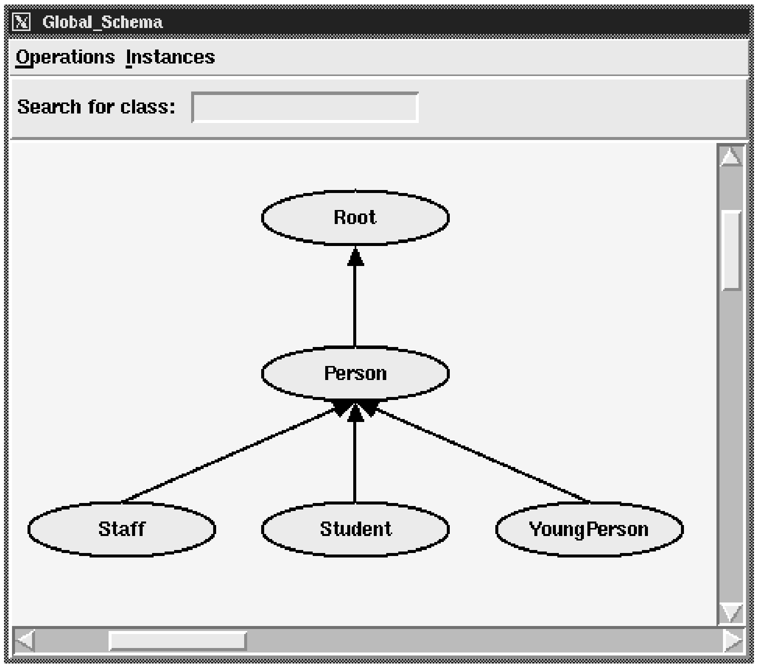


Figure 10: *MultiView* automatically integrates the new virtual class into the schema.

3.3.2 Base Classes, Virtual Classes, and View Schemata

Let C be the set of all classes in the database. C is partitioned into two sets – BC is the set of all base classes in the database, and VC is the set of all virtual classes in the database. **Base** classes are defined during the initial schema definition. **Virtual** classes can be defined and added dynamically to the schema throughout the lifetime of the database. They are defined by the application of a query operator to a *source class* (or classes) that restructures the source class's type and/or extent membership.

Just like our original class definition, each database class (whether virtual or base) $C_i \in C$ has a unique class name, a type description, and a set membership. In addition, in order to accommodate the support of virtual classes, each class C_i also maintains a set of all virtual classes $VC_j \in VC$ s.t. $C_i \in$ **SourceClasses**(VC_j). We use the corresponding notation VC_j **derived-from** C_i to characterize the relationship $C_i \in$ **SourceClasses**(VC_j).

Let VC be the set of all virtual classes. Like members of the set of base classes, a **virtual class** $VC_i \in VC$ has a unique class name, a type description, and a set of the virtual classes derived from VC_i. In addition, because the set membership of VC_i is **derived** using its derivation query, VC_i also includes the query from which it is derived, denoted by **query**(VC_i), as defined in Definition 7.

Definition 8 (Derived-from Sub-Graph) *We define the **Derived-from Sub-Graph** of a class $C_i \in C$ to be a schema $DS(C_i) = (DV, DE)$ containing all the classes that are either directly or indirectly derived from C_i:*

1. *C_i is the root of the DS.*

2. *$DV \subseteq V$,*

3. *$\forall V_j \in V$ s.t. V_j corresponds to virtual class $VC_j \in VC$, $DV_j \in DV$ if and only if VC_j **derived-from** C_i or $\exists V_k \in DV$ s.t. VC_j **derived-from** VC_k.*

4. *Each edge $DE_i =<DV_i,DV_j> \in DE$ corresponds to a **direct derivation relationship** between two classes C_i and $C_j \in C$ with corresponding nodes DV_i and $DV_j \in DV$, meaning that C_j **derived-from** C_i.*

In Definition 6, we define an object schema as a database schema containing all of the classes in the database. We now extend this definition to define a **global schema** as a database schema containing

```
run
   YoungCultureView
       addVC: YoungPerson
       withName: #GenerationXer
%
run
   YoungCultureView
       addVC: Student
       withName: #Student
%
```

Figure 11: Code to add classes to the *YoungCulture* virtual schema.

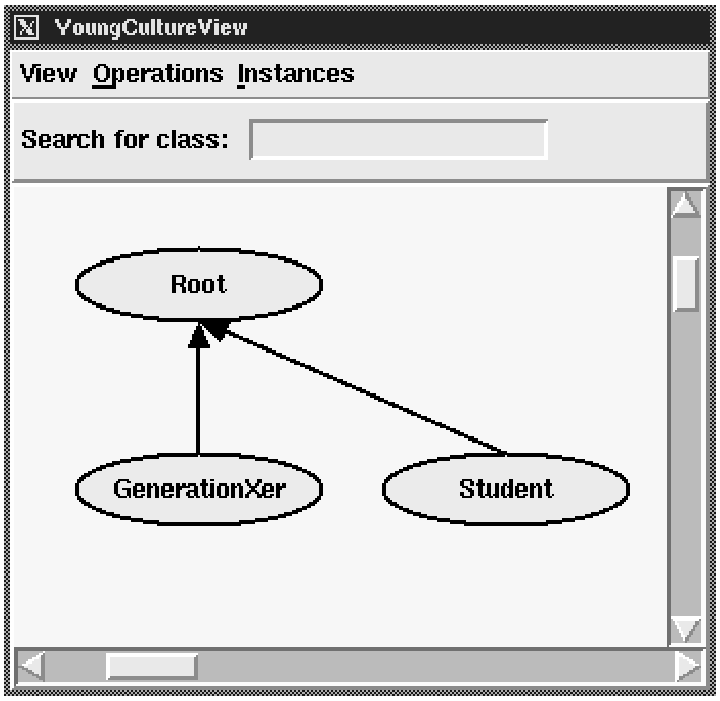


Figure 12: The *YoungCultureView* schema contains a selected set of classes organized into a schema by *MultiView*.

all of the database classes, both base and virtual, integrated into a consistent generalization hierarchy; a **base schema** as a database schema containing all base classes; and a **view schema** as a database schema containing a user-selected subset of the classes (either base or virtual) from the global schema. A view management system must support a flexible classification mechanism in order to maintain this global class hierarchy. For example, the system must be able to make dynamic changes to the class hierarchy, possibly inserting a new class between two existing classes without affected the types or extents of previously-existing classes in the hierarchy. We have proposed elsewhere [34, 31] algorithms and techniques by which the global class hierarchy can be maintained.

Virtual classes are often simply called "views," but we interpret an object-oriented view to imply a complete **view schema**, which is a user-selected subgraph of the global schema.

Definition 9 (view schema) *Given a global schema $GS = (V, E)$, a* ***view schema****, VS, is defined to be a schema $VS = (VV, VE)$ with the following properties:*

1. *VS has a unique view identifier denoted $< VS >$,*
2. $VV \subseteq V$
3. $VE \subseteq$ transitive-closure(E)

For example, we can create a new view schema, named *YoungCultureView*, and select the *YoungPerson* and *Student* classes to participate in that schema under the names *GenerationXer* and *Student*. The *MultiView* system will automatically build the class hierarchy for the new view schema. Figure 12 shows the resulting view schema.

Because view schemata have all the properties of the global schema, and because classes can participate in view schemata under assumed names, different applications can use their own views of a shared database.

3.3.3 View Materialization

Typically, the contents of a view are not stored, but are instead derived using the stored query defining the view. The term **view materialization** means that results of the query that defines a view are actually maintained in the view's extent, as opposed to being computed on demand. Materialized views have already been established in the relational context as offering improved query times. We define a **materialized virtual class** as a virtual class that stores either copies of or references to data instances in its extent rather than computing its extent upon access. The extent of a materialized virtual class must be maintained in a consistent state with respect to updates by the database system.

4 *MultiView* Design Principles

In the following sections we describe and discuss our system design and its components, as illustrated by the concentric circles of Figure 1. We begin in this section by reviewing the properties required for the support of our object model's features. We next review the properties provided by the underlying GemStone system. Finally, we compare the required properties to those that are provided.

4.1 Properties Required for View Support

MultiView's object model includes some fundamental design principles. These principles require the underlying database system to support certain properties. Figure 13 outlines the design principles and the properties needed to support them.

Design Principles / Required Properties	multiple classification	multiple inheritance	dynamic class restructuring	dynamic object restructuring	flexible inheritance system
class has type and extent	✓				
dynamic class creation			✓		✓
integrated class hierarchy	✓	✓		✓	✓
virtual class first-class database citizen	✓	✓	✓	✓	✓
objects gain and lose types dynamically				✓	
objects have both base & virtual types	✓			✓	

Figure 13: Properties required for view support

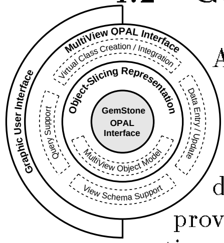
- **A class has a type and an extent.** If an object qualifies for membership in two classes, it must belong to the extent of both classes, whether or not any subsumption relationship exists between the two classes. Our system must therefore support *multiple classification.*

- **Users should be able to create virtual classes dynamically, customizing both type and extent.** If the new virtual class's type is a customization of an existing class's type, then the correct position in the global type hierarchy might be between two existing classes. The underlying inheritance implementation must therefore be flexible enough to support the *dynamic reclassification* of database classes. Furthermore, if the new virtual class is to serve as the point of inheritance for a new property, that property could be relocated to the new class. Our system must therefore support *dynamic class restructuring.* This dynamic class restructuring includes support for the possible migration of state as properties are moved from class to class.

- **Both virtual and base classes should be integrated into a correct and consistent hierarchy.** As detailed in [32],this integration requires both multiple classification and multiple inheritance. For example, if a virtual class is formed by applying a union query to two base classes, then the newly-formed union class should be classified as a superclass of both base classes, which should continue to be subclasses of any other classes from which they inherit properties. In addition, because new classes will have to be integrated dynamically, the inheritance implementation must be flexible and the underlying system must support *dynamic class restructuring*.

- **Virtual classes should be treated as first-class database citizens.** This means that virtual classes should participate in the upwards-inheritance mechanism of the schema, and that the system should support *capacity-augmenting virtual classes*.
- **Objects should be able to gain or lose types dynamically.** This feature requires support for the *dynamic restructuring of objects* because an object could gain or lose state depending on which types it possesses.
- **Objects should possess both virtual and base types.** For example, if an object qualifies for membership in two select classes, then it should be a member of both—regardless of whether or not any subsumption relationship exists between the select classes. The system must therefore support *multiple classification* and, as explained above, the *dynamic restructuring of objects*.

4.2 GemStone Kernel



As illustrated by the innermost circle of Figure 1, *MultiView* is built on top of Servio Corporation's GemStone OODBMS using GemStone's Smalltalk-like OPAL programming environment. We chose to use the GemStone OODBMS rather than implement *MultiView* from scratch because GemStone provides a rich object-oriented data model with supporting tools. Despite significant differences between the GemStone and *MultiView* data models, GemStone offers key features that proved extremely useful in the implementation of *MultiView*. Besides the typical database functionalities, such as persistence, database programming language support, and transaction management, features of GemStone 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.

4.3 Differences Between *MultiView* and GemStone

Our implementation made a number of extensions to the GemStone system that were necessary in order to support views. For example, because GemStone does not maintain explicit extents to collect all instances of a type, which is needed for the specification of select virtual classes, *MultiView* extends the GemStone concept of class to include an extent. We add system methods to automatically add objects to the appropriate extents upon creation, and maintain the extents of virtual classes upon updates.

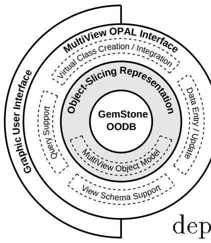
More significantly, the goals of *MultiView* require that the underlying object model support certain properties, as discussed in Section 4.1. Many of these properties are not supported by GemStone's data model. The most important of these properties are:

- GemStone does not support *multiple classification*, which is a required characteristic for view support in *MultiView*.
- GemStone does not support *multiple inheritance*, which is necessary both in order to preserve our single-point-of-inheritance property and also the participation of virtual classes as first-class citizens in an upward-inheritance mechanism.
- *Class restructuring* in GemStone is severely restricted for classes with instances, which would prevent the manipulation of the schema hierarchy necessary for the integration of virtual and base classes into a single global schema.

- GemStone supports object restructuring only in terms of migrating objects from one class to another. Furthermore, an object can only belong to one class, and migration can only take place between two classes that share a class history.
- GemStone's native inheritance schema does not support the independent restructuring of class inheritance relationships that is necessary if virtual classes are to participate fully in the inheritance schema.

Multiple classification is particularly necessary in a capacity-augmenting view system, because an object can be an instance of multiple virtual classes (as well as its base class) regardless of whether or not any subsumption relationships exist between those classes [5]. To the best of our knowledge, current OODB systems do not support multiple classification — with the exception of IRIS [11], which is a functional database system that uses a relational database as a storage system, to store data from one object across many relations [6]. Inspired by this approach we have developed an object-slicing technique to address these advanced features in the context of the OODB technology [29]. In the following section, we describe our object-slicing technique.

5 Object-Slicing Architecture



We used a flexible and powerful technique called object-slicing to construct the *MultiView* object model on top of GemStone. In object-slicing, a real-world object corresponds to a hierarchy of **implementation objects** (one for each class whose type the object possesses) linked to a **conceptual object** (used to represent the object-itself) rather than associating one implementation with each conceptual object as is commonly assumed in OODB systems [25]. For example, Figure 14 depicts a schema composed of two base classes, *Cat* and *HouseCat*, and two virtual classes, *HeavyCat* (derived from a selection query upon the *Cat* class) and *DietingHeavyCat* (derived by refining *HeavyCat* to add a new instance variable, `diet`). The right-hand side of the figure illustrates a single instance of the *HouseCat* class with instance variables `height = 9"`, `weight = 17lbs`, and `owner = Fred`. The instance fulfills the membership requirements of *HeavyCat* (weight $>$ 15 lbs), and thus it belongs to both the *HeavyCat* and the *DietingHeavyCat* virtual classes. Note that the state of the object is maintained by the implementation objects of the classes defining each property. Because the instance belongs to *DietingHeavyCat*, it now carries a data value for the `diet` attribute.

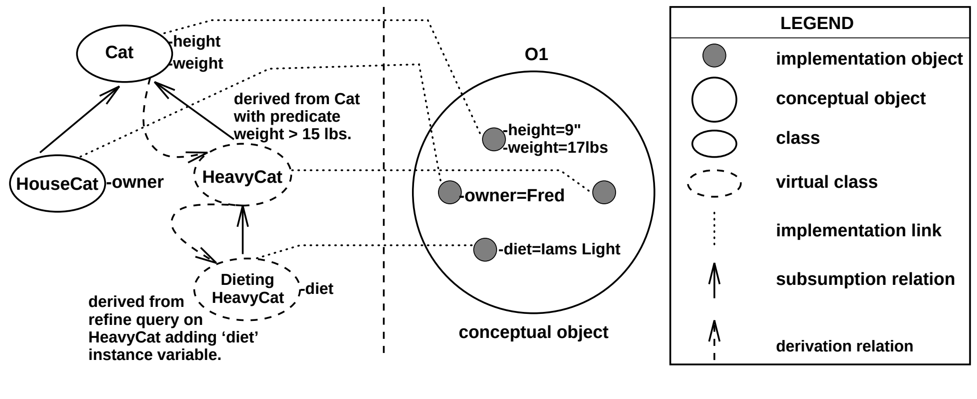


Figure 14: Example of object-slicing.

[5]While regular view systems (i.e., that do not support capacity-augmenting views) also must permit an object to be an instance of multiple virtual classes (in addition to its base class), note that virtual classes do not carry any additional stored data – and it is thus trivial to make the object a transient member of the virtual classes on access. This is no longer sufficient for capacity-augmenting views.

[6]Most OODBs typically represent an object as a chunk of contiguous storage determined at object creation time. They thus adhere to the invariant that *an object belongs to exactly one class only* — and indirectly also to all the class's superclasses.

We formalize the object-slicing paradigm as follows. Let $O_i \in O$ be a user-defined database object. In the object-slicing model, O_i is represented using two kinds of objects: a single conceptual object, $O_{i_{conc}}$ and one implementation object for each class $C_j \in C$ to which the object belongs, $O_{i_{impl_{C_j}}}$.

Definition 10 (conceptual object, implementation object) *A* ***conceptual object*** *consists of a tuple, <OID, implementationObjects>, where OID is the unique, system-generated object-identifier of the conceptual object, and implementationObjects is the set of* ***implementation objects*** *that are linked to the conceptual object. An* ***implementation object*** *is a 4-tuple* ***<OID,oid,class,state>*** *where OID is the object-identifier of the linked conceptual object, oid is the object-identifier of the implementation object itself*[7]*, class is the class of which the implementation object is a direct instance, and state corresponds to the values of the local instance variables stored for the given object.*

Each implementation object $O_{i_{impl_{C_j}}}$ is an object instance of the database class $C_j \in C$ it represents.

Conceptual objects are object instances of a special system class, *ConceptualObject*, rather than of a user-defined class.

Because a single real-world object is now represented using multiple database objects, we define a number of functions to operate on objects in the model, including object creation, equality comparison, etc.

5.1 Implementation of Required Properties Using Object-Slicing

The object-slicing architecture offers the advantages of a flexible and powerful implementation base that can readily be configured to support all of the properties we require for the support of the *MultiView* model.

Multiple Classification. The object-slicing approach implements multiple classification by using implementation objects to represent an object's membership in multiple classes. This technique of representing class membership via implementation objects is quite flexible. For example, an object-slicing system can dynamically reclassify an object from being the instance of one class ($C1$) to becoming that of another class ($C2$), by linking the object instance to an implementation object of the class $C2$ and discarding that of the class $C1$. The effect of reclassification under object-slicing is minimal. No new classes are created using this approach, nor is the type of any existing class affected by the reclassification. Furthermore, because objects are compared using the oid of their conceptual objects, the multiply-classified object's identity is not compromised.

Customizing Inheritance Using Object-Slicing. If an object O_i possesses an implementation object $\mathrm{O}_{i_{impl_{C_j}}}$ for some $C_j \in C$, then O_i must also possess implementation objects for all classes C_k s.t. C_j *is-a* C_k. Thus the implementation objects associated with a given conceptual object will mirror the structure of the class hierarchy. Object-slicing intrinsically includes its own inheritance mechanism. Let there be an implementation object $\mathrm{O}_{i_{impl_{C_i}}}$, $O_i \in O, C_i \in C$. Suppose the method $m_i \in M$ were to be invoked upon object $\mathrm{O}_{i_{impl_{C_i}}}$, and m_i is not defined locally in **type**(C_i). In this case, $\mathrm{O}_{i_{impl_{C_i}}}$ will delegate the method m_i to $O_{i_{conc}}$, which will then conduct a search "upwards" through **SuperclassHierarchy**(C_i). If method m_i is not found in the type of some $C_k \in$ **SuperclassHierarchy**(C_i), then an error is returned. Otherwise, the method is invoked upon object $O_{i_{impl_{C_k}}}$, which can be located using the function **ImplLink**($O_{i_{conc}}$, C_k) [8]. Objects of user-defined classes in an object-slicing representation use this object-slicing inheritance mechanisms.

Figure 15 shows the process that takes place if one of the implementation objects receives a message for a method that is not locally defined in the implementation object's class. The left side of the top half

[7] Each implementation object by default possesses its own object identifier. However, because the implementation object serves as an interface for a specific conceptual object, the object-identifier of the conceptual object supersedes that of the implementation object for most practical purposes, such as determining object-equality.

[8] If a method with selector m_i is found in more than one class in **SuperclassHierarchy**(C_i), then the user is prompted to **cast** O_i into a non-ambiguous implementation object.

of Figure 15 shows a class hierarchy composed of five classes. The right side of the top half of Figure 15 represents a conceptual object that belongs to all of the classes in the hierarchy. The smaller circles within the object each represent an implementation object for a specific class whose type the object possesses. Assume that an implementation object for the *PedigreedCat* class receives a message for method "color," which is an access method for instance variable "color" defined in *Cat*, Figure 15(c). Because the receiving implementation object does not understand the method "color," the request is delegated upwards until it reaches the *Car* implementation object. Because both the instance variable and the accessing method are locally defined at *Cat*, the *Cat* implementation object both contains the actual instance variable "color" and can respond to message "color." It thus recognizes the message and performs the appropriate operation, returning the value of "color" to the user (Figure 15(c)).

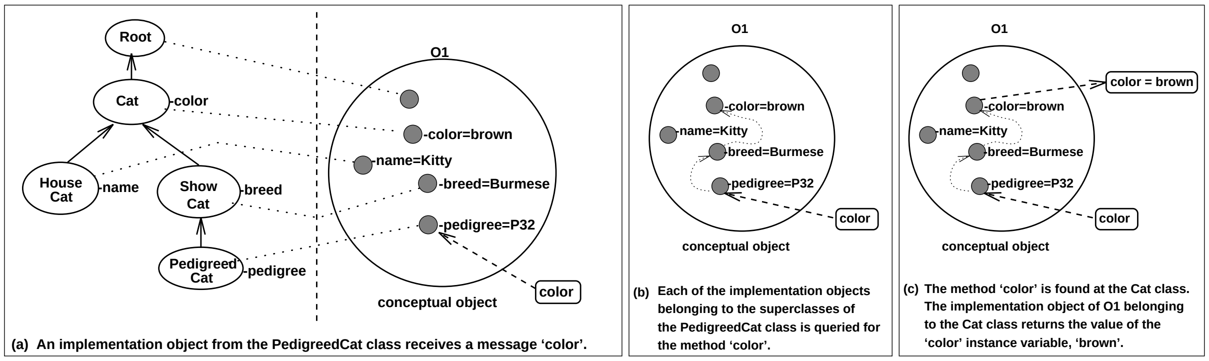

Root
Cat
-color
House Cat
-name
Show Cat
-breed
Pedigreed Cat
-pedigree
O1
-color=brown
-name=Kitty
-breed=Burmese
-pedigree=P32
color
conceptual object
(a) An implementation object from the PedigreedCat class receives a message 'color'.
O1
-color=brown
-name=Kitty
-breed=Burmese
-pedigree=P32
color
conceptual object
(b) Each of the implementation objects belonging to the superclasses of the PedigreedCat class is queried for the method 'color'.
O1
color = brown
-color=brown
-name=Kitty
-breed=Burmese
-pedigree=P32
color
conceptual object
(c) The method 'color' is found at the Cat class. The implementation object of O1 belonging to the Cat class returns the value of the 'color' instance variable, 'brown'.

Figure 15: Inheritance Under the Object-Slicing Model

Dynamic Restructuring of Objects and Classes. Recall that our data model supports a *single point of inheritance* characteristic for database properties, meaning that if two classes C_1 and C_2 share some common property then they both must have inherited it from some lowest common superclass (LCS). Not only does this approach support true upwards inheritance of method code for both base and virtual classes, but it also avoids the possible maintenance and storage problems that could be caused by the duplication of code. This approach is compatible with object-slicing, because the point of inheritance corresponds to the class that maintains all implementation objects that have values for the particular instance variable.

In order to preserve uniform upwards inheritance, it may be necessary to *promote* methods and/or instance variables from a subclass to a new superclass during the classification of a new virtual class into the global schema [9]. We call this promotion *property migration*. This will ensure that the property will be located above all the classes that inherit that property. When methods or instance variables are moved from an existing class to a new superclass, the system performs the following tasks: First the new superclass is extended to include the migrating (to be locally defined) methods and/or instance variables. Next, the migrating methods and instance variables are removed from their original class. Finally, each implementation object instance of the original class is *split* into an implementation object of the modified original class and an implementation object of the new superclass [10]. These new implementation objects are linked together by their corresponding conceptual objects. Below we present a result that limits the effort required for property relocation.

Axiom 1 *If two classes C_i, $C_j \in C$ share some common property p_l, then they must ultimately have inherited it from the same superclass; i.e., there must exist a lowest common superclass in the class lattice $C_k \in C$ s.t. C_i* is-a *C_k, C_j* is-a *C_k, and $p_l \in$ **properties**(C_k).*

This axiom follows from our classification algorithm, which places each newly-created virtual class into the global class hierarchy in a consistent and correct way [32].

Proposition 1 *Only two classes will be involved in any property migration caused by virtual class integration.*

[9]The methods used to set or retrieve an instance variable's value are called that instance variable's *accessing methods*. Accessing methods are always located at the same class as the instance variables for which they are defined, and thus when an instance variable migrates from one class to another, that instance variable's accessing methods must make the same migration.

[10]For example, we could swap the identities of two implementation objects to give the implementation object of the modified existing class the same object identifier as the original implementation object.

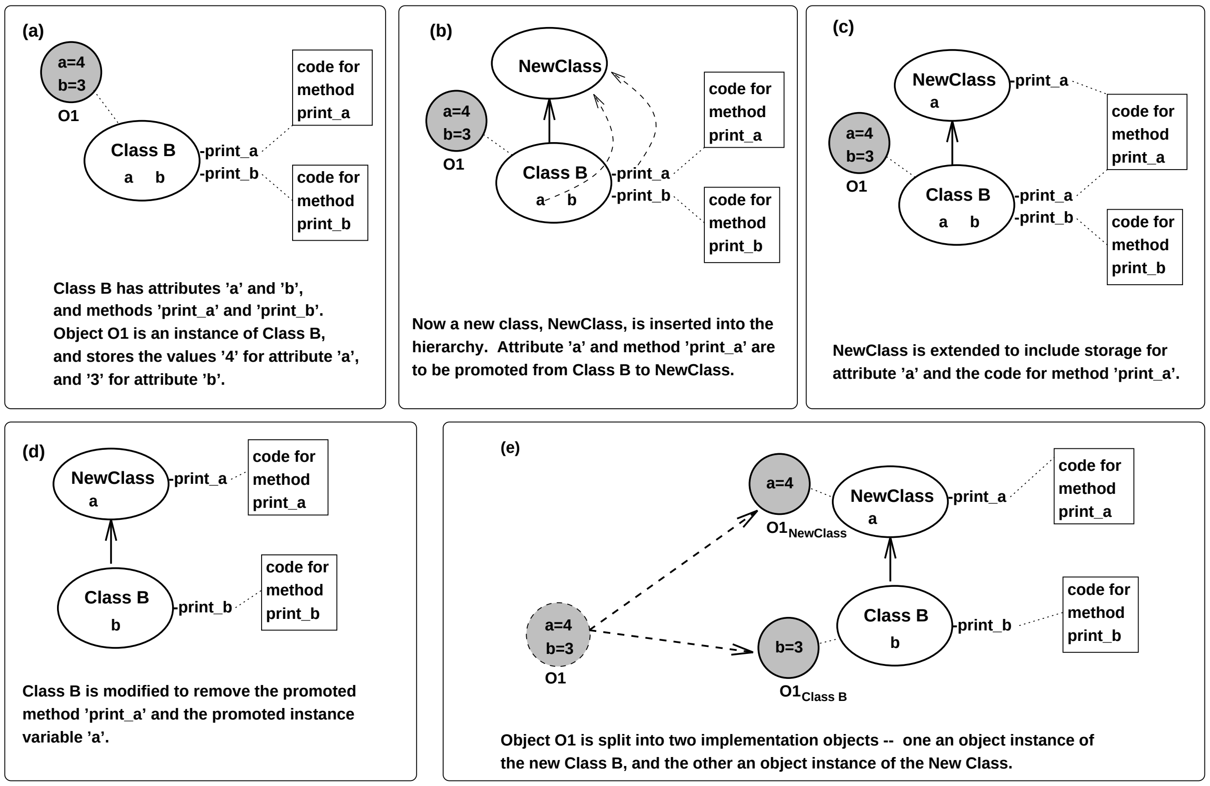

(a)
a=4
b=3
O1
Class B
a b
-print_a
-print_b
code for method print_a
code for method print_b
Class B has attributes 'a' and 'b', and methods 'print_a' and 'print_b'. Object O1 is an instance of Class B, and stores the values '4' for attribute 'a', and '3' for attribute 'b'.
(b)
NewClass
Now a new class, NewClass, is inserted into the hierarchy. Attribute 'a' and method 'print_a' are to be promoted from Class B to NewClass.
(c)
NewClass is extended to include storage for attribute 'a' and the code for method 'print_a'.
(d)
Class B is modified to remove the promoted method 'print_a' and the promoted instance variable 'a'.
(e)
O1NewClass
O1Class B
Object O1 is split into two implementation objects -- one an object instance of the new Class B, and the other an object instance of the New Class.

Figure 16: Attribute `a` and method `print_a` are moved from class B to a new superclass.

Proof. Suppose that there two distinct source classes, *Class A* and *Class B*, and that there were two methods, "a" and "b", respectively located in *Class A* and *Class B*, as shown in Figure 17. Because we do not allow cycles, the relationship between *Class A* and *Class B* cannot be ambiguous – class *Class A* can be a superclass of class *Class B* (in which case *Class B* would inherit method "a" from *Class A*), as shown in Figure 17(a), or class *Class B* can be a superclass of class *Class A* (in which case *Class A* would inherit method "b" from *Class B*), as shown in Figure 17(b), but one class cannot be both superclass and subclass of the other, Figure 17(c). Therefore, by Axiom 1, at least one class of *Class A* and *Class B* must not have inherited any methods from the other class. Because the only way that a class could include method "a" or "b" in its type would be to have inherited them from class *Class A* or *Class B*, and because *Class A* and *Class B* cannot both inherit from each other, at most one of classes *Class A* and *Class B* could include both methods "a" and "b".

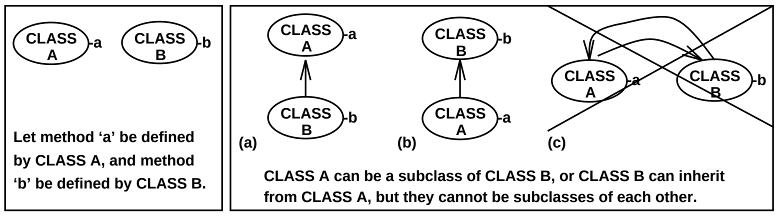

CLASS A
-a
CLASS B
-b
Let method 'a' be defined by CLASS A, and method 'b' be defined by CLASS B.
(a)
(b)
(c)
CLASS A can be a subclass of CLASS B, or CLASS B can inherit from CLASS A, but they cannot be subclasses of each other.

Figure 17: Cycles cannot exist in the class hierarchy.

Now suppose, contrary to fact, that there were a new class such that two methods, "a" and "b", from different source classes, *Class A* and *Class B*, were to be relocated in the new class. Axiom 1 states that every method is located in exactly one place; thus both of the source classes would now be subclasses of the new class.

If both source classes are subclasses of the new class, then each of them must inherit both "a" and "b" from the new class. However, we already established that it is not possible that both *Class A* and *Class B* include both methods "a" and "b". Therefore, there cannot be a new class such that two methods from different source classes are relocated in the new class.□

Object-Slicing and View Materialization. The object-slicing technique, combined with the characteristic of *single point of inheritance* and *classification*, lends itself well to materialization, because (1) it elegantly avoids the need to duplicate data for materialized classes and (2) any update to an object will take place at a location determined by the property involved regardless of the source of the update request. Since object-slicing requires an object to possess implementation objects for any class to which it belongs, an object-slicing view implementation is effectively materialized in that the implementation objects representing an object's membership in various virtual classes are actual object-instances of those virtual classes. The object-slicing technique is thus subject to the intrinsic problem of view materialization – how to update materialized views so as to keep their extents consistent with the rest of the database. This problem and its solutions are addressed by the *MultiView* system's update model, which is described in Section 6.

5.2 Evaluation of Object-Slicing's Impact on Performance

Extending an existing DBMS with object-slicing techniques necessarily involves the overhead of additional data structures, maintenance costs, and processing time. Because although object-slicing is a known technique that is being utilized for view systems [24], schema evolution [30], and role systems [12], to our knowledge no other work has been done evaluating the costs of object-slicing, we were motivated to perform some initial experiments evaluating the relative costs and benefits of adopting the object-slicing techniques. The results of these experiments are presented in [21]. In the remainder of this section, we summarize the conclusions of our evaluation.

In evaluating the performance of databases, I/O operation time typically dominates CPU operation time. Consequently, an evaluation of object-slicing must consider the effect of object-slicing on I/O time. One major variable for calculating I/O time is the number of objects that can be stored in a disk block, known as the *blocking factor* (bf), namely: $\lceil$ disk block size / object size $\rceil$.

In traditional (non-object-slicing) architectures, the size of an object is calculated to be the total amount of storage needed to store the state of the object (data size), oid size, and pointer size (to reference the object's class), and some fields for the system use. Because the object-slicing model represents any given object using a conceptual object and some number of implementation objects, objects in the object-slicing architecture inherently require more storage space than their counterparts in traditional architectures. Like a traditional object, an object under the object-slicing architecture contains data, an oid, and a pointer to its class. In addition, an object-slicing object that belongs to l classes also requires storage for (ignoring system fields) l implementation objects (each with a reference to its class and to the object's conceptual object) and the conceptual object (which has a dictionary of references to its implementation objects, respectively indexed by the class the implementation object belongs to) [11]. That is, while in a *conventional architecture* we would have:

$$obj\ size = data\ size + oid\ size + pointer\ size;$$

in the *object-slicing architecture* we now have:

$$obj\ size = data\ size + (l+1) \cdot oid\ size + (4l+1) \cdot pointer\ size.$$

To simplify, we assume that oid size is equal to pointer size. The ratio of the sizes is:

$$SizeRatio(SR) = \frac{DS + (5l+2) \cdot pointersize}{DS + 2 \cdot pointersize},$$

where SR is the size ratio and DS is data size. In general, the ratio increases as the value of l increases and decreases as the data size increases. This means that the disadvantage of the object-slicing architecture's storage overhead is ameliorated by an increased object size/depth of schema ratio.

[11] While this storage of references linking conceptual to implementation objects and back could be reduced, we've chosen this representation for reason of efficiency.

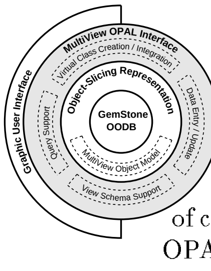
In order to determine the base cost of implementing the object-slicing representation paradigm, we have run test queries from the University of Wisconsin's OO7 benchmark suite [9] with the intention of comparing GemStone's native implementation versus our object-slicing extension to GemStone. For this study, we used *MultiView* to create and populate the OO7 benchmark's parts-assembly database with 10,000 Atomic Parts.

Our experiments confirm that in the case of queries upon local properties object-slicing can actually improve query performance. The improved performance can be explained as follows. First, the navigation was limited to access of local instance variables (rather than inherited ones). Thus there is no overhead of finding appropriate implementation objects for *MultiView*. Consequently, only one implementation object must be retrieved per queried object. The query avoids having to perform random accesses to retrieve additional implementation objects. Not only was retrieval of the single implementation objects sequential, but also these navigated implementation objects are much smaller in size (containing only local instance variables) compared to GemStone's native objects (containing both local and inherited instance variables in one contiguous allocation) and thus offered a higher blocking factor.

However, although the object-slicing paradigm improves the flexibility of our object model, retrieval of inherited attributes in object-slicing can involve significant overhead. First, the cost of storing the conceptual object, the oids of the implementation objects, and the extra links connecting conceptual and implementation objects may require a larger number of blocks and result in more page faults. Second, because the implementation objects belonging to the same class are by default clustered together, a traversal to an implementation object for getting an inherited value may require a random block access, which can result in a page fault.

Using Clustering to Ameliorate Object-Slicing Costs. A *MultiView* object's state is distributed among multiple object-slicing implementation objects, which lends itself to clustering strategies that resemble the vertical partitioning of the relational model. In order to determine under which circumstances it is preferable to cluster the implementation objects by class (which we call *class clustering*), and under which circumstances it is better to cluster all the components of a *MultiView* object together (which we term *object clustering*), we designed and carried out an extensive experimental study evaluating both clustering techniques [21]. Based on these results, we conclude that as in a conventional architecture, various types of access patterns can best be optimized by providing distinct types of clustering techniques. We found that class-clustering offers superior blocking factors for scenarios where only locally-defined attributes are accessed, and that an object-clustering strategy can ameliorate the cost of retrieving inherited attributes.

6 *MultiView* View System



MultiView supports customized virtual classes and customized virtual schemata in such a way that the system's class-restructuring capabilities are powerful and flexible, and that the virtual classes and virtual schemata look and feel like the actual database. The *MultiView* system automates as many of the database maintenance tasks as possible, including the integration of new classes into the global schema, the construction of the class relationships in view schemata, and the maintenance of class extents. The *MultiView* OPAL interface preserves all the functionalities of the original GemStone OPAL system, while providing new view capabilities. The subsections of the third circle of Figure 1 illustrate the four major functions performed by the *MultiView* view system:

Virtual class creation / integration. *MultiView* users can customize existing type structures and object sets (class extents) by deriving virtual classes via object-oriented queries. *MultiView* will then automatically integrate the new virtual class with existing classes into a single consistent global schema graph, maintaining relationships between base and virtual classes.

View schema support. *MultiView* users can specify (at any time) view classes (both base and virtual) from the augmented global schema for membership to a particular view. *MultiView* will then construct arbitrarily complex view schemata composed of these view classes.

Queries. Because *MultiView* is completely compatible with the underlying GemStone system, any set-based query that can be performed using GemStone's OPAL interface can also be performed by *MultiView*. In addition, *MultiView* also supports object algebra queries over class extents and query-by-form (via the graphic interface).

Data manipulation. *MultiView* supports updates on both base and virtual classes. Because *MultiView* supports materialized views, the system incrementally maintains materialized virtual class extents in the face of the entry and update of data objects (using efficient update propagation algorithms).

6.1 Virtual Class Creation and Integration

In this subsection, we focus on the implementation of the first task—the specification and classification of virtual classes in *MultiView*.

Virtual class creation. *MultiView* provides a number of class methods that implement the query operators shown in Figure 8. These class methods automatically create the new virtual class, construct its type (migrating all appropriate properties), construct its extent (preserving all property values), and integrate the new class into the global class hierarchy.

Virtual class classification. In *MultiView*, all virtual classes are automatically integrated into one global generalization hierarchy. Such complete classification offers a number of advantages, including (1) the relationships between base and derived classes are made explicit (which aids query processing and view materialization strategies); (2) because virtual classes fully participate in the inheritance schema, property functions and and object interfaces can be shared; and (3) it facilitates the formation of arbitrarily complex view schema graphs. Both the classification algorithm and its proof of correctness are presented in [33]. Classification in *MultiView* is automatic and compulsory. That is to say, every time a class is added to a *MultiView* database, the system automatically integrates it into the inheritance hierarchy. *MultiView* makes classification the responsibility of the system rather than requiring users to create their own *is-a* arcs manually because we want the global class hierarchy to be unique, correct, and consistent.

There are two steps to the classification algorithm: first the class hierarchy is prepared by the addition of any necessary intermediate classes, then the virtual class is placed into the global schema. The *MultiView* view creation process includes the invocation of methods which use deterministic algorithms to perform the automatic integration of virtual classes into the global schema.

In [32] we described the problem of how sometimes there may be no correct location for the placement of a new virtual class in an existent global schema graph – in order to allow for both the full inheritance invariant as well as subsumption. The current implementation of the *MultiView* system includes this classifier, further described in [32], which generates one or more intermediate classes in order to guarantee these properties. This solution is both necessary and sufficient to guarantee the closure of the resulting class hierarchy. That is to say, integrating the newly generated classes into the global schema will never cause the generation of additional new classes.

Once the class hierarchy has been prepared by the insertion of any necessary intermediate classes, the new virtual class can be inserted into the global schema by identifying all direct *is-a* relationships between the new class and the other classes using a single depth-first downwards traversal algorithm. We have shown elsewhere that the resulting global schema incorporates the virtual class in a consistent and most efficient manner [33].

6.2 View Schemata

As proven in [32], once the global schema integration has been achieved, the tasks of specifying and constructing view schemata can be reduced to simple graph algorithms. The *MultiView* provides a simple graphical interface for the definition of view schemata. The user can create any number of view schemata, and can add both virtual and base classes to a view schema as *view classes* (see Figures 11 and 12). The *MultiView* view management module automatically computes the proper hierarchical relationships for classes in a view schema. View classes can have different names in different view schemata. Each view schema is associated both with an access control list of GemStone users who are allowed to access the view, and with a password that controls access to this access control list.

6.3 Query Capabilities

The *MultiView* OPAL interface is fully OPAL compatible, and thus users can employ all queries supported by the native GemStone system. In addition, because the *MultiView* model associates an extent with each class, users can explicitly perform queries upon class extents. *MultiView* class extents are implemented as a subclass of the OPAL *Set* class, and thus all *Set* methods can be invoked upon class extents, including iteration, collection, and selection. Note that because inserting or deleting an object from a class's extent can have an impact upon the extent content of other classes, we overrode the adding and removing methods to perform the necessary update propagation. *MultiView*'s query capabilities exceed GemStone's, as the *MultiView* object algebra supports declarative queries across both base and virtual classes. Section 7 presents examples of *MultiView*'s query functionalities as captured by the *MultiView* graphic interface, called the *MultiViewer*.

6.4 Data Manipulation

MultiView supports a suite of update operations, as well as the underlying view update propagation strategies necessary to keep materialized classes consistent. The update operators *MultiView* supports include creation, deletion, addition of a type, removal of a type, and modification of an instance variable.

Since *MultiView* maintains materialized views, the effects of updates must be propagated to all classes affected by it. For example, if an object gains the type of a class that has a union class derived from it, then one direct effect of that update would be that the object gains the type of the union class (i.e., a new implementation object of the union class is created and linked to the updated object). *MultiView* minimizes the number of times an updated object is evaluated to determine if the object should be a member of a virtual class.

We identify specific techniques by which we can exploit unique features of the object-oriented paradigm to optimize updatable materialized views. Our update algorithms are incremental, and perform selective notification to the set of classes determined to be **directly-affected** by a given update. We use inherent object-oriented features, such as the integration of classes into a generalization hierarchy, encapsulation, and membership materialization, both to facilitate the identification of the classes that are **directly-affected** by an update and also to optimize the propagation to the classes derived from classes **directly-affected** by the update. Our update optimization process performs two tasks:

1. First, we identify the set of classes that are **directly-affected** by the update, i.e., the set of classes whose set-membership changes as a direct result of the update.

2. We propagate to the classes derived from the **directly-affected** classes in an optimized manner, so as to avoid both propagating to classes that are not **indirectly-affected** by the update and self-cancelling update propagation.

Identifying directly-affected classes. In order to efficiently be able to identify the set of classes directly-affected by modification updates, we introduce a "registration" service by which virtual classes can register their interest in specific properties and be notified upon modification of those properties. We are aided to this end by our data model's injunction that each property must reside at a single class that defines the property and acts as its **point of inheritance**. All classes whose membership may be affected by modifications to an attribute are thus registered with the attribute's **point of inheritance**. We associate *triggers* with the accessing methods to propagate updates to the registered classes if and only if the relevant property's value is modified. The principles of our registration strategy are:

- When a virtual class is created, its predicate is parsed and the class is *registered* with each property involved in the query specification. Each registration is sent to the class that serves as the point of inheritance for the property involved. Because all classes in our model are arranged in a generalization hierarchy, we optimize our registration process to take advantage of subsumption relationships. The *registration* structures can thus be distributed along the generalization hierarchy, as illustrated in Figure 18. Distributing the *registration* structures allows us to avoid evaluating the update in terms of classes that are obviously unaffected by the update (i.e., classes derived from classes to which the object did not belong either before or after the update).
- When an object is updated, the update method [12] triggers a *notification* function that propagates the update to all virtual classes that have *registered* an interest in the value of the updated property.
- When a virtual class has been informed, via the *notification* class method, that an object in which it has a potential interest has been modified, it then evaluates the update. Should the update warrant that the object gain, keep, or lose the type of the class, then the class will send a message notifying those dependent upon it of the change. Otherwise it does not propagate the change across its derivation hierarchy and the update propagation terminates.

[12]For the sake of simplicity, we currently make the assumption that instance variables reside at the same location as their update methods.

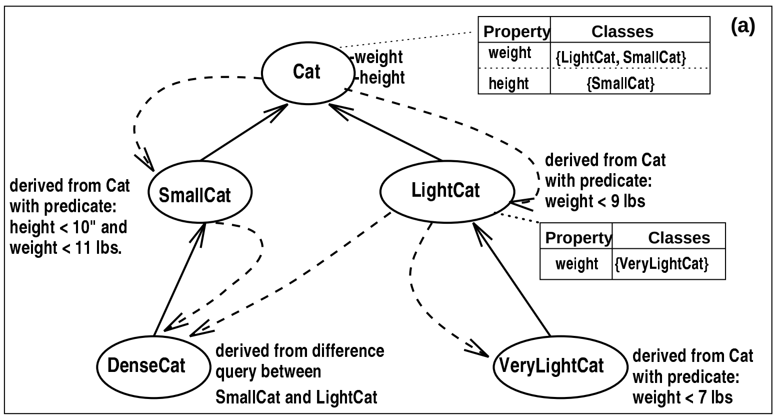
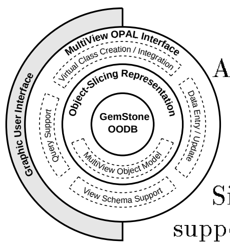

(a)
Cat
weight
height
Property | Classes
weight | {LightCat, SmallCat}
height | {SmallCat}
SmallCat
LightCat
derived from Cat with predicate: height < 10" and weight < 11 lbs.
derived from Cat with predicate: weight < 9 lbs
Property | Classes
weight | {VeryLightCat}
DenseCat
derived from difference query between SmallCat and LightCat
VeryLightCat
derived from Cat with predicate: weight < 7 lbs

Figure 18: Registrations use both point of inheritance and subsumption.

Eliminating Self-Cancelling Update Propagation The propagation process could involve updates that cancel out each other's effects. We call this the *self-cancelling update propagation* problem because it could lead to repetitive evaluations of the object in the context of a single class, sometimes with self-cancelling results. In order to overcome the problem of self-cancelling propagation, *MultiView* incorporates the following strategy. First, we assign a unique **derivation number** to each class in the database, indicating its global derivation depth in the schema. Using this derivation number framework, we propagate changes to affected classes using a breadth-first traversal by order in the derivation hierarchy. Once the traversal reaches a class to which the updated object does not belong either before or after the update, it **terminates update propagation** to that class's branch. Because a derived class will only be processed after all of its source classes have been processed, we can guarantee that no class will be processed more than once. This solution avoids the problem of self-cancelling update propagations by ensuring that if $\text{DerNum}(C_k) < \text{DerNum}(C_l)$, then a modified object's membership in C_k can be evaluated only before the object's membership in C_l has been evaluated.

Identifying Branch Termination Conditions Under certain conditions, propagation to branches of the derivation or generalization hierarchy can be terminated. For example, propagation to a branch of the derivation hierarchy can be terminated if the object does not qualify to belong to any of the entrance points into the branch either before or after the update. *MultiView* incorporates strategies that achieve early termination whenever possible.

7 *MultiView* Graphic User Interface



Although the GemStone OODB provides its own native graphic programming tool, named GeODE, GeODE is not an appropriate tool for *MultiView* users for a variety of reasons. First of all, GeODE does not reflect the *MultiView* object model. For example, GemStone's object model is single-inheritance, and thus does not recognize the multiple inheritance schemata of *MultiView*. Similarly, because the GemStone model does not associate extents with classes, GeODE does not support the concept of class extents and as a consequence does not support queries upon class extents. More importantly, GeODE does not provide support for the *MultiView* virtual class operators and view schema capabilities. We therefore built a custom graphic user interface for the *MultiView* system.

For our interface, we created a customized implementation of the Tcl Windowing Shell ("wish") with extensions to interface with both *MultiView* and the underlying GemStone database system. We call our graphic interface *MultiViewer*. The *MultiViewer* extensions allow the manipulation of database objects using Tcl language constructs. The *MultiViewer* provides convenient access to basic GemStone functions such as logging into and out of the GemStone database, sending OPAL code to the database and accessing the objects returned, and committing changes to the database structures, In addition, *MultiViewer* also

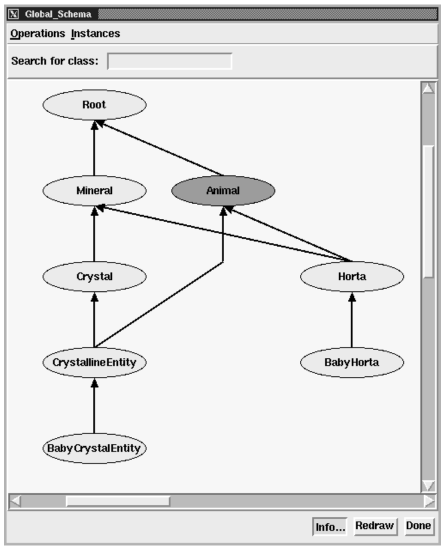
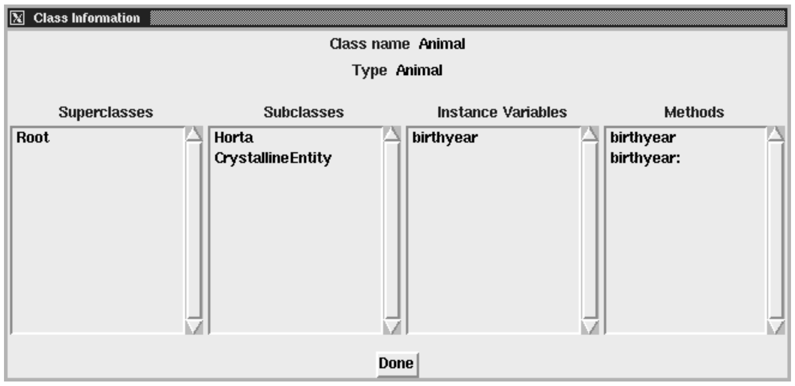


Figure 19: Schema View



Class name Animal
Type Animal

Superclasses	Subclasses	Instance Variables	Methods
Root	Horta	birthyear	birthyear
	CrystallineEntity		birthyear:

Figure 20: Class Information Window.

performs a number of *MultiView*-specific functions. In the remainder of this section, we describe the main features.

7.1 Schema Views

The *MultiViewer* displays a class hierarchy using a directed graph representing inheritance relationships. The graph layout is determined by an algorithm designed to minimize the number of edge crossings [40]. This results in a clean-looking directed graph that simplifies visualization of the generalization relationships by grouping similar classes together on the display. A sample schema graph is shown in Figure 19.

The schema graph itself can be of arbitrary size; schema graphs too large to fit on the physical display can be browsed using the scroll bars below and to the left of the graph display. Each schema graph includes a text widget that can be used to bring a specified class to the center of the display.

7.2 Virtual Classes

Pressing mouse button 1 above one of the schema graph's vertices (i.e., a class) *selects* that class for *MultiView* operations. For example, in Figure 19 the *Animal* class has been selected. The **Info** button in the lower right will present you with information about the selected class, as shown in Figure 20.

Once a class has been selected, it can be used, for example as a source for virtual class creation. If the virtual class is created using a query that takes two source classes (e.g., a union query), then a dialog box will prompt for the right-hand class. A new class will immediately appear on the global schema graph, allowing the user to visually confirm that the operation had the expected result. For example, in Figures 21 and 22, we create a new class, *BabyHybrids*, that is defined as the intersection of the *BabyHorta* and *BabyCrystalEntity* classes from Figure 19. *MultiView* automatically integrates the new virtual class into the global schema, which appears in Figure 23.

7.3 View Schemata

As is apparent from Figure 23, schemata can become quite complex as they are augmented and grow over time. *MultiView* therefore supports the creation of view schemata. Users can use the *MultiViewer* to graphically create a new view schema (as shown in Figure 24), then populate the schema with selected classes (illustrated by Figure 25). The latter task can be accomplished by either graphically selecting classes from the global schema (by clicking on them) or by incorporating one view schema into another. Access to classes and objects can now be performed through any of these customized view schemata, which provide a personalized view of the database to the user.

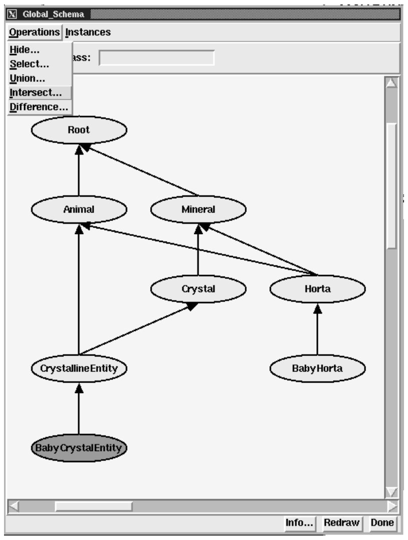


Figure 21: Creating a new virtual class using the intersection operator.

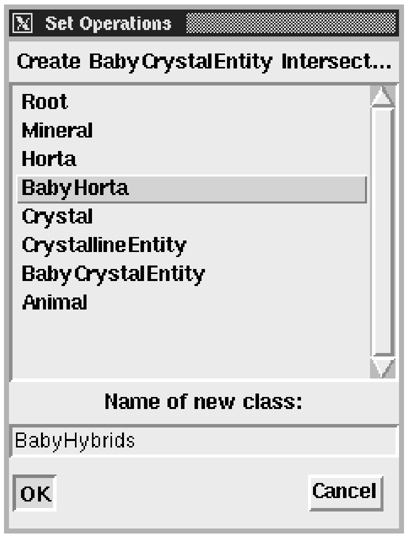


Figure 22: The *BabyHybrids* class is formed by intersecting *BabyCrystalEntity* with *BabyHorta*.

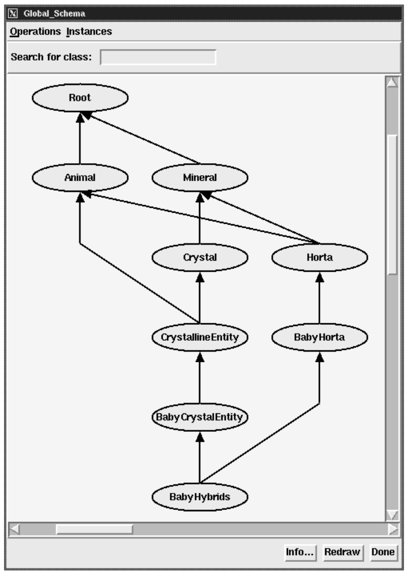


Figure 23: The augmented global schema (containing both base and virtual classes).

7.4 Queries and Data Entry

The *MultiViewer* allows a user to interact with object instances without using native database syntax. This is especially useful when the person responsible for entering or updating records is not a computer specialist. The *MultiViewer* generates a user interface, called a form, for any object based on constraints established by the class's creator. The exact way that a class or object presents itself is completely configurable.

7.4.1 Object Forms for Data Display, Browsing, and Editing

The basic interface to an object is a collection of graphical entries, one for each attribute defined by the object's class. The values displayed in the fields will update dynamically as changes are made in the database. The user can change any field to modify the associated object. Figure 26 shows a basic object form for a *Person* class object. *MultiViewer* builds the interface to an object dynamically by querying the object's class about which of its properties are externally modifiable. The class responds with a set of ordered tuples specifying, for each value, the name to be presented to the user, the domain class, and the accessing methods that can retrieve and set the value.

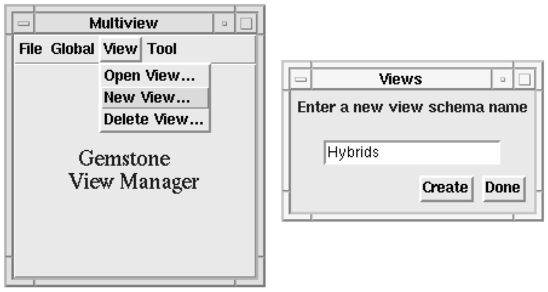


Figure 24: Creating a new view schema.

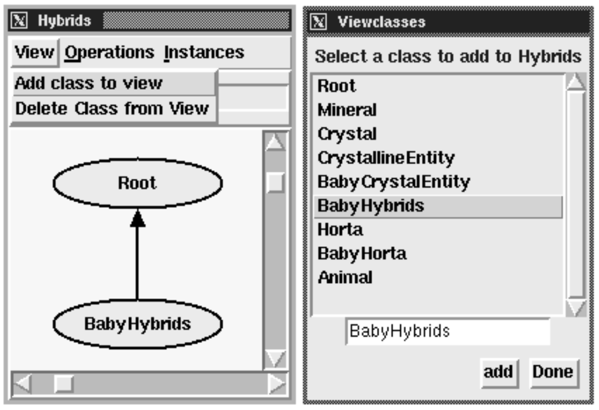
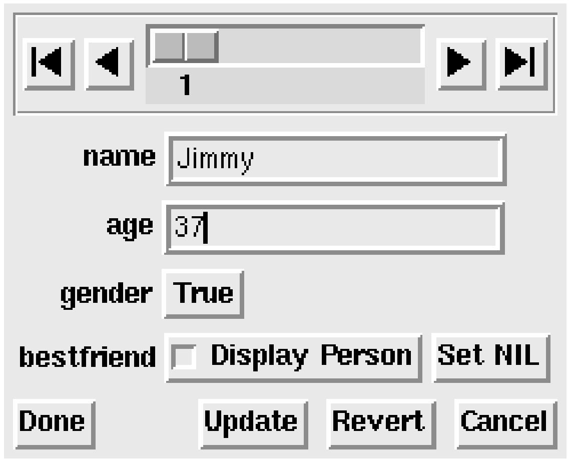
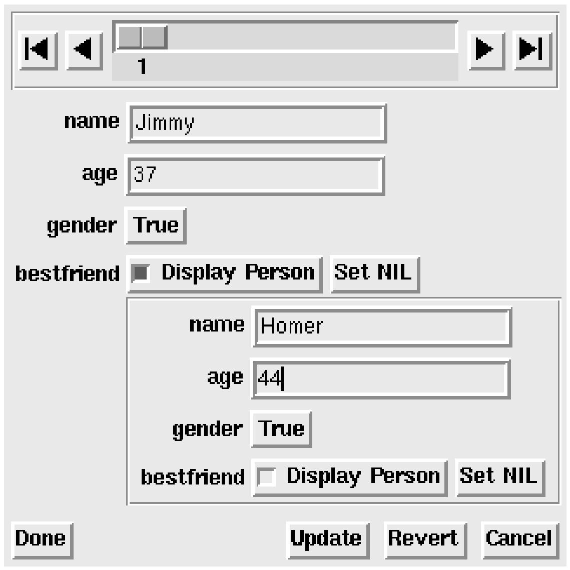


Figure 25: Adding classes to a view schema.



name Jimmy
age 37
gender True
bestfriend

Figure 26: Object Form for Person class objects.



name Jimmy
age 37
gender True
bestfriend
name Homer
age 44
gender True
bestfriend

Figure 27: Object Form with expanded subform.

MultiView provides appropriate interface forms for many internal built-in GemStone classes, such as Integer, Strings, Set, Boolean, etc. For example, Booleans are represented in the *MultiViewer* by toggle switches representing the value's current state. The system is extensible in that additional primitive representation forms can easily be defined as needed.

For more complex classes, the *MultiViewer* will query the class about its interface form by interacting with the *MultiView* system and dynamically constructing an interface within the original form. For example, suppose a class called *Person* has a field *BestFriend*, which is constrained to be another *Person*. When editing a *Person*, the *MultiViewer* will build a form consisting of various *tools*, say, *Name*, *Age* and *Gender*. It will also display the person's *BestFriend* as a subform, containing its own fields *Name*, *Age*, *Gender* and *BestFriend*. To prevent infinite recursion, each subform is not actually built and displayed until activated by the user. Figure 27 shows a form that displays the instance indirectly referenced by an object, namely the *Person* "Jimmy" has a best friend named "Homer."

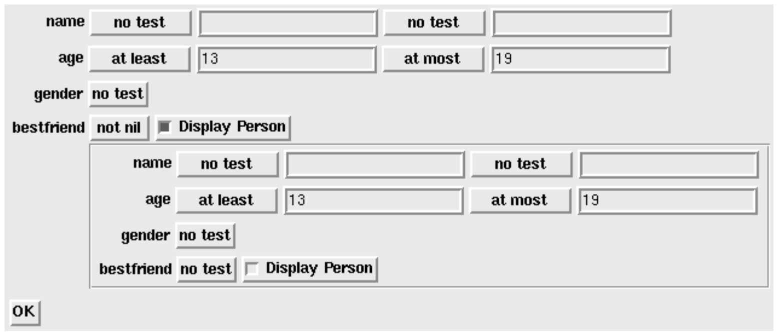
7.4.2 Selection Forms for Querying

In addition to the ability to browse and edit object instances, forms can be used to formulate queries. The *MultiViewer* particularly facilitates the specification of selection predicates. These can be used to filter a large database for objects with interesting properties, or in the creation of virtual classes. A selection form is created in much the same fashion as an object form, but the fields themselves are different. For each field, several tests customized to the data type of the attribute can be performed.

These query forms support the specification of nested queries involving implicit joins. Namely, the user can base the selection predicate on attributes of other objects. By displaying the *BestFriend* subform and interacting with its fields, it is simple to select, for example, the set of all *Persons* whose *BestFriend* is a teenager. For example, the query shown in Figure 28 will return all teenagers whose *BestFriend* is also a teenager.

7.4.3 Configurability

If *MultiViewer*'s form generation mechanism is not sufficiently flexible for an application's purposes, the creator of an object can design a custom interface to its externally visible attributes (e.g., for multi-media display). Because *MultiViewer* is written in Tcl, an interpreted language, it can easily be extended with dynamic user-interface code. This would allow a *MultiViewer* user to view pictures or use tools custom-built for another user's class without having to build or install any software packages.



name	no test		no test	
age	at least	13	at most	19
gender	no test			
bestfriend	not nil	▣ Display Person		

name	no test		no test	
age	at least	13	at most	19
gender	no test			
bestfriend	no test	☐ Display Person		

OK

Figure 28: Selection Form.

8 Conclusions

In this paper, we have provided an overview of the *MultiView* system – one of the first implemented OODB systems to support dynamic and updatable materialized object-oriented database views. In particular, we have presented the motivation and execution of our design decisions, system architecture, and the *MultiView* system's capabilities. The implementation solutions we describe are general – and thus should be applicable to other object-oriented systems. To summarize, the *MultiView* system is distinguished by a number of unique features, which include:

- *MultiView* treats virtual classes as first-class database citizens, integrating virtual and base classes into a unified global schema.
- Virtual classes in *MultiView* participate in the actual inheritance hierarchy and thus behave just like base classes.
- *MultiView* supports capacity-augmenting views (virtual classes can independently define additional attributes and methods).
- *MultiView* automatically generates view schemata composed of user-selected base and virtual classes.
- *MultiView* includes optimized incremental view maintenance algorithms for materialized views that exploit the integrated class hierarchy structure.

During our initial design stage, we identified a set of key object-model properties that are critical for the realization of an object-oriented view system. To the best of our knowledge no existing commercial OODB supports all of these properties. Our implementation of the *MultiView* system uses the object-slicing representational technique to provide these necessary object model properties, which include multiple classification, object migration, and dynamic classification.

The current version of *MultiView* uses the 4.0.1 version of GemStone and was developed using the OPAL interface. The database is built on a Sun 4m running SunOS 4.1.3 with 32 megabytes of memory. *MultiView*'s implementation consists of approximately 48 classes and approximately 230 class and instance methods, all of which are defined using less than 10,000 lines of code (over 50% of which are comments). The code used to implement object-slicing makes up about 5% of the total code. The graphic interface was implemented separately, using Tcl.

We have performed several studies to evaluate the *MultiView* system and its embedded algorithms. Our experiments demonstrate the benefit of techniques used by our update propagation algorithms [23]. Namely, we show that the potential problems of self-cancelling propagation and of early branch condition termination are indeed handled by our solution and result in significant performance gains. In a related paper, we explore the use of clustering to ameliorate the overhead of the object-slicing technique employed by our implementation [21].

Because it is one of the only implemented view systems, *MultiView* is currently being used in a number of projects. For example, the *Transparent Schema Evolution* project, funded by an NSF Young Investigator Award, uses *MultiView* as an implementation base to develop a suite of schema evolution tools that use the view system to preserve existing database interfaces through schema change. We are also utilizing *MultiView* in a NASA-funded study for the storage, retrieval, and interpretation of scientific data gathered by space physics researchers.

In the future, we plan to extend our current work to examine the area of distributed views. We also want to study issues related to the support of deferred updates and multiple (batched) updates for materialized views.

9 Acknowledgements and Thanks

Thank you and appreciation to Daryl Scott Cantrell and Victor Ng, who helped design and implement *MultiView* 's graphic interface. A thank you also goes to Young-Gook Ra, who helped design and implement the *MultiView* object-slicing architecture. We also would like to thank the other members of our research group for valuable discussions and suggestions throughout the development of the MultiView system.

References

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

[2] R. Agrawal and H. V. Jagadish. Materialization and incremental update of path information. In *IEEE International Conference on Data Engineering*, pages 374–383, 1989.

[3] 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.

[4] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. *SIGMOD*, pages 248–257, 1991.

[5] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. *International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, April 1991.

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

[7] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. *SIGMOD*, pages 61–71, 1986.

[8] J. A. Blakeley, N. Coburn, and P-A Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[9] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD*, 1993.

[10] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *International Conference on Very Large Data Bases*, pages 577–589, September 1991.

[11] D.H. Fishman. Iris: An object oriented database management system. In *ACM Transactions on Office Information Systems*, volume 5, pages 48–69, January 1987.

[12] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. manuscript, 1993.

[13] E. N. Hanson. A performance analysis of view materialization strategies. *SIGMOD*, pages 440–453, 1987.

[14] M. Hardwick and B. R. Downie. On object-oriented databases, materialized views, and concurrent engineering. In *Proceedings of the 1991 ASME International Computers for Engineering Conference and Exposition*. Engineering Databases: An Engineering Resource, 1991.

[15] S. Heiler and S. B. Zdonik. Object views: Extending the vision. In *IEEE International Conference on Data Engineering*, pages 86–93, 1990.

[16] H. Jakobsson. On materializing views and on-line queries. In *ICDT*, pages 407–420, October 1992.

[17] M. Kaul, K. Drosten, and E. J. Neuhold. Viewsystem: Integrating heterogeneous information bases by object-oriented views. In *IEEE International Conference on Data Engineering*, pages 2–10, February 1990.

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

[19] 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.

[20] S. Konomi, T. Furukawa, and Y. Kambayashi. Super-key classes for updating materialized derived classes in object bases. In *International Conference on Deductive and Object-Oriented Databases*, July 1993.

[21] H. A. Kuno, Y. G. Ra, and E. A. Rundensteiner. The object-slicing technique: A flexible object representation and its evaluation. Technical Report CSE-TR-241-95, University of Michigan, 1995.

[22] H. A. Kuno and E. A. Rundensteiner. Implementation experience with building an object-oriented view management system. Technical Report CSE-TR-191-93, Univ. of Michigan, Ann Arbor, 1993.

[23] H. A. Kuno and E. A. Rundensteiner. Incremental update propagation algorithms for materialized object-oriented views in *MultiView.* Technical report, Univ. of Michigan, Ann Arbor, June 1995.

[24] H. A. Kuno and E. A. Rundensteiner. Materialized object-oriented views in *MultiView.* In *ACM Research Issues in Data Engineering Workshop*, pages 78–85, March 1995.

[25] J. Martin and J. Odell. *Object-Oriented Analysis and Design.* Prentice-Hall, Inc., 1992.

[26] O2 Technology. *O2 Views User Manual*, version 1 edition, December 1993.

[27] M. P. Papazoglou. Roles: A methodology for representing multifaceted objects. In *International Conference on Database and Expert Systems Applications*, pages 7–12. Springer-Verlag, 1991.

[28] Z. Peng and Y. Kambayashi. Deputy mechanisms for object-oriented databases. In *IEEE International Conference on Data Engineering*, pages 333–340, March 1995.

[29] Y. G. Ra, H. A. Kuno, and E. A. Rundensteiner. A flexible object-oriented database model and implementation for capacity-augmenting views. Technical Report CSE-TR-215-94, University of Michigan, 1994.

[30] Y. G. Ra and E. A. Rundensteiner. A transparent object-oriented schema change approach using view schema evolution. In *IEEE International Conference on Data Engineering*, March 1995.

[31] E. A. Rundensteiner. *MultiView*: A methodology for supporting multiple views in object-oriented databases. In *18th VLDB Conference*, pages 187–198, 1992.

[32] E. A. Rundensteiner. Tools for view generation in OODBs. In *CIKM*, pages 635–644, November 1993.

[33] E. A. Rundensteiner. A classification algorithm for supporting object-oriented views. In *CIKM*, pages 18–25, November 1994.

[34] E. A. Rundensteiner and L. Bic. Set operations in new generation data models. *IEEE Transactions on Knowledge and Data Engineering*, 4:382–398, August 1992.

[35] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. International Conference on Extending Database Technology (EDBT), 1994.

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

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

[38] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, pages 103–122, April 1989.

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

[40] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 109–125, February 1981.

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