Extending Objects to Support Multiple Interfaces and Access Control

Brent Hailpern, Senior Member, IEEE and Harold Ossher, Member, IEEE

Abstract—Object-oriented languages hide the details of objects from their users; all interaction with an object must be through the operations it supports. Objects must therefore support a collection of operations sufficient to satisfy all users. The requirements of different users can differ widely. It is therefore desirable to provide restricted subsets of the supported operations to specific users or kinds of users, rather than make all supported operations universally available.

This paper describes a mechanism called views that allows programmers to specify multiple interfaces for objects, and to control explicitly access to each interface. This mechanism provides a simple and flexible means of specifying enforceable access restrictions at many levels of granularity. It also results in system organization that supports browsing based on a number of different criteria.

The paper motivates and defines views, gives some examples of uses of views, discusses the impact of views on system organization, and outlines five approaches to implementing views.

Index Terms—Access control, browsing, capabilities, data abstraction, information hiding, inheritance, interfaces, object-oriented languages, system organization, views.

I. INTRODUCTION

A MAJOR goal of abstract data types in general and object-oriented systems in particular is information hiding [1]. Data in an object is accessed only through an interface consisting of operations or methods. The interface is usually defined in the abstract data type or class to which the object belongs. This encapsulation protects data from illegal manipulation. It also permits data representations to change over time, because clients only access the data by the public methods.

Most languages that support information hiding at all support a single interface to each abstract data type or class. There are a number of situations, however, in which multiple interfaces are useful. Multiple interfaces support additional information hiding over and above that of conventional abstract data types—not only is internal detail hidden, but exported detail can be made available on a need-to-know basis. This results in programs that are structured more clearly. The ability to maintain separate interfaces to an entity also supports easier maintenance: changing an interface affects just those clients that use that particular interface, rather than all clients of the entity.

One use of multiple interfaces within object-oriented languages has been proposed by Snyder [2], [3]. He points out that, while an object can have no direct access to the structure (instance variables) of another object, a subclass does usually have direct access to the structure of its superclasses, because the structure of a subclass is dictated by the structure of its superclasses. This reduces the freedom of class designers to change the representation of a class safely. Because a subclass can depend directly on the instance variables of its superclasses, the designer of the superclass cannot change the implementation of the superclass without adversely affecting the correctness of subclasses.

Snyder proposes a solution consisting of two separate interfaces for classes: one for public use and one for subclasses only. In general, the subclass interface provides greater access to the internal data of a class. In both cases, instance variables remain hidden unless deliberately exported by methods of one or both interfaces. Snyder discusses implementation issues, performance considerations, and the effect on code reuse and encapsulation.

Other examples of multiple interfaces abound. The Trellis/Owl language [4] provides four categories of visibility for operations and components: public, subtype-visible, private, and allocate-visible. This amounts to defining four interfaces for each type. Buffer objects might provide one interface to producers and another to consumers. System components that normally hide their internal data structures might make them available to a debugger, or to other, closely related components. System services might provide a safe, high-level interface to normal users, but a more dangerous and powerful interface to trusted system components.

In this paper, we propose a model in which objects can have multiple views. Each view represents a specified set of methods that can be called by a specified collection of clients. The model thus provides for both specification of interfaces and specification of access control. The model is presented in the context of object-oriented languages, but is equally applicable to data abstraction languages in general. Just as structured programs can be written in Fortran, objects and views can be used to good effect as a programming discipline in languages that do not support information hiding at all. The programmer must then enforce restrictions on information and maintenance of separate, well-documented interfaces and client sets.

Section II gives a detailed example that motivates our approach. Section III discusses related work. Section IV defines views more fully, and Section V discusses some specific uses of views. Section VI discusses the impact of views on system organization. Section VII describes some implementation techniques for views. Finally, Section VIII briefly describes our experience with using views.

II. MOTIVATING EXAMPLE

An important aspect of object-oriented languages is that they provide encapsulation. One object can interact with another only by sending it a message; it has no direct access to the details of the receiver. However, as pointed out by Snyder [2], [3], no such restrictions normally apply to subclasses. As a result, subclasses can depend on deep implementation details of superclasses, making such details difficult, dangerous, and

Manuscript received June 9, 1988; revised June 13, 1990. Recommended by Y. Matsumoto. The grid mechanism was conceived and designed by H. Ossher while at Stanford University. That work was supported in part by the Defense Advanced Research Projects Agency under Contract MDA 903-80-C-0432.

The authors are with the IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

IEEE Log Number 9038330.
expensive to change. This is a serious problem, since flexibility and ability to grow are valued properties of object-oriented systems.

This situation is illustrated by means of a simple example in Fig. 1. Class `PenPlot` provides a `draw` method for drawing thin lines on a plotter. Class `DoublePenPlot` is a subclass of `PenPlot` that provides a method for drawing thick lines. The implementation shown accomplishes this by drawing over the same line twice. It accesses the instance variables `x` and `y` of `PenPlot` directly, to determine the starting position of the pen and to reset the pen to that position for the second draw.

Thin arrows in the figure indicate message sends. Objects, called `clients`, that use `DoublePenPlot` objects can send them `draw` and `doubleDraw` messages, and `doubleDraw` itself can send `draw` messages. Thick arrows indicate the instance variable accesses described above. The dotted line bisecting the instance diagram separates it into an upper part derived from class `Penplot` and a lower part derived from class `DoublePenPlot`. The thick arrow crossing this separator indicates instance variable accesses across class boundaries.

The trouble with the instance variable accesses is that they make the code within `DoublePenPlot` sensitive to the implementation details of `PenPlot`. If the programmer responsible for `PenPlot` decides to change the representation of pen position, he must worry about whether the change will adversely affect `DoublePenPlot`. Alternatively, if `PenPlot` is part of a released system and `DoublePenPlot` is a customer extension, the extension might suddenly break when the next release is obtained, and the customer will have to reexamine the details of `PenPlot` to fix it.

A good solution in this case is to extend the interface of `PenPlot` to include a `move` method, and to implement `doubleDraw` in terms of `move` and `draw`, as shown in Fig. 2. There are now no instance variable accesses across classes, and code in `DoublePenPlot` is no longer sensitive to implementation details of `PenPlot`: it relies only on correct implementations of the two methods.

The trouble with this solution as it stands is that there is nothing to stop client objects from sending `move` messages directly. It can be argued that, in this particular example, such an ability is harmless or even desirable, but in general it is often important to prevent clients from using methods intended for use by subclasses.

Extending objects to have multiple interfaces allows these differences to be specified. Fig. 3 shows two interfaces to `DoublePenPlot` objects. The `subclass` interface provides access to operations inherited from `PenPlot`, both `draw` and `move`, that are for use by subclass code such as `doubleDraw`. The `public` interface provides access to those operations that are for use by clients in general; `move` is excluded. In addition to identifying the different interfaces and the methods they comprise, it is necessary to specify which objects are allowed to use which interfaces. This leads directly to the `view` model presented in Section IV.

### III. Related Work

A good deal of work has been done on providing access to objects through multiple interfaces, and some of this work is reviewed here. As the example in the previous section made clear, however, in addition to supporting multiple interfaces one needs the ability to restrict who can use them. As we shall see, this ability is usually not provided.
port multiplie inheritance using prototypes rather than classes, and allows the inheritance structure to be changed dynamically. This means that interfaces to objects can be changed at run time. Inheritance in its various forms thus provides multiple interfaces, but not any means of restricting access to them.

Emerald [6], [7] uses the elegant notion of abstract types to provide the benefits of static type checking while retaining the flexibility and extensibility of untyped object-oriented languages. An abstract type defines an object interface: a set of operations, their signatures, and, at least in principle, their semantics. Any type can be implemented by many different actual objects. Emerald defines a complete type-checking discipline, based on conformity of abstract types. It supports subtyping, but not code-sharing inheritance. It also has no mechanism for controlling who may use a particular abstract type.

PIE [8], [9] used a view mechanism called perspectives. A perspective is a special object that can be attached to another object, and serves to describe it from a particular point of view. Multiple perspectives can be attached to a single object; a client of the object specifies which perspective it is using, and messages are interpreted by the perspective rather than by the object itself.

Abdali [10] and Jenks [11] have each developed algebraic computation systems that contain a notion of views. In their systems, domains, which correspond to Smalltalk classes, can be tagged or viewed as specific instances of general algebraic structures. A category then specifies what operations a domain must support and certain properties that its operations must satisfy. A domain can be an instance of a category, without being a subclass of that category. One can then view a given domain as belonging to different categories at different times.

Garlan makes use of views for integrating tools in an environment [12]. Tools in an integrated environment share common structures. Garlan allows each tool to have its own view of these structures, so that the tools are not all forced to depend on a single, prearranged representation. This is akin to views in database systems [13]. Garlan goes further, however, and allows the various views of a structure to be developed independently before the structure itself; his system merges them automatically provided they are “compatible.” Garlan’s work, especially in the area of view merging, has since been extended by the Gandalf group at Carnegie-Mellon University [14].

As mentioned in Section I, Trellis/OWL supports four categories of visibility: public, private, subtype-visible, and allocate-visible. Each of these can be thought of as a separate interface. Public operations are universally available. A private operation on an object can be invoked only by code that implements another operation on the same object (the object denoted by me). Similarly, a subtype-visible operation is an operation inherited from a superclass that can be used only by code in subclasses implementing operations on the same object. Allocate-visible operations, usually “set” operations on selected instance variables, can be performed only during the process of object creation and initialization. Code built into the compiler enforces these visibility rules, so here we have a case of fully enforced access restrictions. Only the four standard interfaces are supported.

The issue of access control is the focus of Minsky and Rozenshtein’s work on law-based systems [15]. They allow the programmer to specify explicitly a discipline that governs the exchange of messages among objects; the specification is called the law of the system. Every message exchange is verified against the law, which can specify that it be permitted unchanged, that it be disallowed, or even that it be rerouted or otherwise altered. The law can thus be thought of as a filter through which every message must pass. In their experimental environment, Darwin, the law is written in a subset of Prolog that is powerful enough to express a wide variety of disciplines, including disciplines involving controlled use of multiple interfaces. Their work and ours are at different points on a “generality spectrum.” They have proposed an extremely general approach to specifying mechanisms that control interaction among objects. We are proposing a specific mechanism that is nonetheless flexible and general enough that it subsumes many existing, more specific ones.

IV. Views

In this section we present views as a relational model whose objective is to permit detailed, fine-grained specification of access restrictions. The discussion is abstract and general, and not tied to any particular implementation approach. As the paper proceeds, we will introduce more concrete uses and details, effectively giving different specializations of the basic view model.

Consider an object-oriented system made up of a set of objects, each of which supports some operations. During execution of the system, operation invocations take place, usually by the mechanism of message passing. An operation invocation is characterized by an invocation triple consisting of the object requesting performance of the operation, called the client, the object performing the operation, called the server, and the operation itself. The intent of our view model is to provide a general and flexible means of specifying which operation invocations are permissible. To this end, we define an access discipline as a ternary relation between server objects, client objects, and operations, and views as a convenient means of specifying access disciplines.

We do not deal with access to instance variables separately. Instead, we take the approach that all access to instance variables, even within a single object, is via primitive “get” and “set” operations, and hence can be controlled by controlling access
to these operations. These operations need not be implemented as general methods; they can be compiled into efficient storage accesses. This approach therefore provides conceptual simplicity with no necessary loss of efficiency.

We now proceed to more formal definitions of access disciplines and views. Consider a system $\Sigma$ consisting of objects $B$ and operations $O$. In a strongly typed language, "operation" includes signature: the name, number of parameters, type of each parameter, number of results, and type of each result. In an untyped language, only the name is needed.

An invocation triple is a triple $(s, c, o)$, where $s \in B$ is called the server object, $c \in B$ is called the client object, and $o \in O$ is an operation.

An operation invocation characterized by $(s, c, o)$ occurs during system execution if object $c$ sends an $o$ message to object $s$. An access discipline $D$ on $\Sigma$ is a set of invocation triples. System $\Sigma$ conforms to access discipline $D$ if and only if the invocation triple characterizing every operation invocation that occurs during the execution of $\Sigma$ is a member of $D$.

This model is useful as a starting point, but is too general to be of much direct practical use. The two primary questions that arise are: how is the access discipline to be specified, and how is it to be enforced? Working directly at the granularity of the tuples above would be onerous and confusing for system designers. Instead, we need to find useful abstractions that permit simpler specification of important restrictions.

In their work on law-based systems, discussed in Section III, Minsky and Rozenshtein describe a general approach to specifying any desired access discipline. In this paper we take a different approach and present a particular abstraction, called views, that is simple and intuitive yet powerful and quite general, and has important implications for system structure.

A view is a cluster of invocation triples grouped simultaneously by server, client, and operation. Formally, a view is a triple $(S, C, I)$ where

- $S \subset B$ is called the server set,
- $C \subset B$ is called the client set, and
- $I \subset O$ is called the interface.

It defines the set of invocation triples

$$\{(s, c, o) \mid s \in S \land c \in C \land o \in I\}.$$  (1)

It thus specifies concisely that a set of servers each provides a specific interface (set of operations) to a specific set of clients.

Fig. 4 illustrates two views of a single server $S$. Written as triples, these views are

$$V_1 = \{(S), \{a, b, c, d\}, \{op1, op3, opm\}\}$$  (2)

and

$$V_2 = \{(S), \{d, e, f, g\}, \{op5, opm\}\}.$$  (3)

The sets of invocation triples defined by these views are

$$D_1 = \{S\} \times \{a, b, c, d\} \times \{op1, op3, opm\}$$

$$= \{(S, a, op1), (S, a, op3), (S, a, opm), (S, b, op1), \cdots\}$$  (4)

and

$$D_2 = \{S\} \times \{d, e, f, g\} \times \{op5, opm\}.$$  (5)

respectively. Some more concrete examples of views will be given in Section V.

A view discipline is a set of views. Every view discipline defines a corresponding access discipline: the union of the sets of invocation triples defined by the individual views. A system is said to conform to a view discipline if and only if it conforms to the corresponding access discipline. For example, the two views in Fig. 4 constitute a simple view discipline that defines the access discipline $D_1 \cup D_2$.

To review, an invocation triple $(s, c, o)$ represents a client $c$ invoking operation $o$ of server $s$. An access discipline is a set of invocation triples. A view denotes an access discipline defined as a cross product, $S \times C \times I$, where $S$ is a set of servers, $C$ is a set of clients, and $I$, an interface, is a subset of the operations provided by the members of $S$. A view discipline is an access discipline defined as a set of views (that is, as a union of cross products).

At this stage of the exposition, the notion of view is deliberately left abstract, with no concern for implementation issues such as whether view triples are entities in the system or whether server and client sets are represented explicitly. It also does not matter here whether view components are static or dynamic. Conventional object-oriented systems usually have static interfaces (visible methods), but dynamic client sets (due to object creation). Many object-oriented language compilers exploit this to avoid the expense of run-time inheritance. Highly flexible systems, such as Self [5], require both dynamic interfaces and client sets. The abstract view model described supports both equally well. In subsequent sections we will examine some specific uses of views, and the demands they place on implementations. We will then discuss some implementations of views, and the extent to which each satisfies the identified demands.

A view consisting of three singleton sets is equivalent to an individual invocation triple, so specifying access disciplines in terms of views rather than in terms of invocation triples directly need result in no loss of granularity or accuracy. However, the larger the sets used in views, the more concise the specification.
and the more "uniform" the access discipline. Since such conciseness and uniformity are significant aids to understanding, designers of systems using views should strive to achieve them to the greatest extent possible.

A particularly useful and uniform way of specifying server and client sets is in terms of classes. We use the following simple notation: \( \text{instances}(K) \) denotes the set of all instances of class \( K \), and \( \text{members}(K) \) denotes the set of all instances of class \( K \) or its subclasses, closed under transitivity. Specifying views in terms of classes follows the Smalltalk [16] philosophy of defining the structure of an object in the class of the object, rather than in the object itself. It leads to clean and simple organization, but does not permit individual variation among objects: all instances of the class require and provide the same access rights. The object-by-object approach, in which individual objects are named in server and client sets, provides greater flexibility in organizing and restricting services, at the expense of some loss in uniformity. Which approach is most appropriate depends primarily on the nature of the host language and the degree of flexibility required. The approaches can be mixed within a single system by specifying views in terms of classes whenever possible, but naming distinguished individual objects when necessary.

Section V gives some concrete examples of using views for specifying access disciplines. Views also have important implications for system organization. They provide a means of referring to a number of important collections of code and objects, and these collections can be used as a basis for system browsing or application of operations. Organizational issues are discussed in Section VI.

V. ACCESS CONTROL USING VIEWS

In this section we give examples of uses of views for imposing access restrictions in a more detailed manner than is available in conventional object-oriented systems. We show how views can allow special interfaces for debugging, security, and priority, and how they can be used to address the problem of encapsulation of inheritance introduced in Section II. Abstract types, described in Section III, can also be used to specify such interfaces, but cannot restrict the visibility of the interfaces once specified. Views provide a means of making the interfaces available to selected clients only.

In the course of discussing each use, we discuss what requirements that use places on implementation approaches.

A. Debugging

A common objection to strict encapsulation and access control is the seeming inability to provide low-level debugging and display information when all instance variables and private methods are hidden. Our view proposal allows objects to export low-level "get" and "set" methods on instance variables to clients of a debugging view, as illustrated in Fig. 5. The debugging views also contain "list-ivs" methods to enable a general debugger to obtain the list of instance variable names for any object. Similarly, information necessary to display an object can be exported through a visualizing view, as illustrated in Fig. 6. Normal objects and classes are prohibited from being clients of these views. System or other trusted objects, however, can be given all the access they need. During the development cycle, a debugging view can thus provide complete access to the internal structure of each object. When development is complete, this view can be eliminated (or further restricted) without affecting objects or classes that are not clients of the debugging view.

This use of views, or any other that simply defines an interface and makes it available to a specified set of objects, requires of a view implementation only that it be able to verify accesses. This is the fundamental requirement, and all implementations described in Section VII support it.
B. Security and Priority

As an extension to the approach used for debugging views, different levels of trusted objects can be defined in a system. Objects can provide different views to other objects according to their levels of trust. This technique can be used to provide a greater level of security than is available in conventional object-oriented languages.

For example, user objects can be given only statistical access to sensitive data (for example, mean and standard deviation). Trusted subsystems can have direct read access to this data. Owning or management objects can retain read-write control.

Similarly, objects can be divided into priority classifications. Each group can be afforded a different level of service. Initially, an object joins the lowest priority view for service. Based on time or identity, the server can promote clients from one priority view to another.

In terms of implementation, the promotion of clients from one priority view to another clearly requires the ability to add objects to and remove them from client sets at run time.

C. Encapsulation of Inheritance

With respect to visibility of operations, especially of operations whose purpose is to provide access to instance variables, Snyder concludes:

Language support is needed to permit classes to directly invoke parent operations (on self) and to permit such operations to be made available in this manner but not via ordinary operation invocation. [2]

Snyder's proposal for encapsulating inheritance also calls for partitioning the instance variables of an object according to the class in which they are defined: classes cannot directly access instance variables defined in superclasses, and an instance variable of a class bears no relationship to an identically-named instance variable of a superclass. These restrictions reduce the impact that changing a class can have on the classes below it in the class hierarchy. The issue of naming of instance variables is outside the scope of our view model. Our model can, however, express the access restrictions both to operations and to instance variables, as described in the rest of this section.

We define two interfaces for each class k: a public interface, $i_{public}$, consisting of operations on instances of k that are available for general use, and a subclass interface, $i_{subclass}$, consisting of those operations available only to instances of immediate subclasses via self calls. The subclass interface should not include primitive "get" and "set" operations providing direct access to instance variables. It can, if necessary, include nonprimitive operations that effectively provide such access, but they should be in truth be abstract operations that can readily be supported even if the corresponding instance variables are changed or removed in future.

The public view

\[
\text{(instances(k), all objects, } i_{public})
\]

makes the operations on instances of k that are in the public interface available without restriction. Only instances of k export these operations through this view, not instances of subclasses of k. If subclasses do wish to export these operations also, they can inherit them through their subclass views and then export them through their own public views.

Let $k$ be the set of all immediate subclasses of $k$. Then for each instance, x, of a class in $k$ we define a subclass view

\[
(x, x, i_{subclass})
\]

that makes the operations in the subclass interface of $k$ available only to $x$ itself. Note that the server of each subclass view is an instance of a subclass of $k$, but the interface lists operations inherited from $k$ itself. This formalizes the fact that the subclass interface contains inherited operations for use by self calls in subclass code. The four categories of visibility supported by Trellis/Owl [4] and described in Section III can be provided in similar fashion using four views.

The motivating example of Section II showed a public interface to DoublePenPlot objects:

\[
i_{public} = \{\text{draw, doubleDraw}\}
\]

and a subclass interface to PenPlot operations for use by subclasses such as DoublePenPlot:

\[
i_{subclass} = \{\text{move}\}
\]

The public view of DoublePenPlot

\[
\text{(instances(DoublePenPlot), all objects, } \{\text{draw, doubleDraw}\})
\]

makes the public interface universally available. For each instance $d$ of DoublePenPlot, the subclass view

\[
(d, d, \{\text{draw, move}\})
\]

makes the subclass interface of PenPlot available to $d$ itself.

The fact that the subclass view can be used only when the client and server objects are in fact the same object is expressed above by using a separate view for each individual instance. This is a rather clumsy approach. A simple extension to the notation for specifying views removes this clumsiness, with no change in semantics: the token server can be used as the client set component of a view triple, defined as follows for any server set $S$ and interface $I$:

\[\text{(S, server, I) = \{(s, s, I) | s \in S\}}\].

Using this notation, the subclass view in the DoublePenPlot example is simply

\[
\text{(instances(DoublePenPlot), server, } \{\text{draw, move}\}).
\]

Some interesting semantic variations can be obtained by changing the server and client sets of subclass views. Three possibilities are discussed here, providing successively weaker encapsulation.

First, a single subclass view can be defined per subclass, instead of one per instance. For each subclass $u \in k$ define the subclass view

\[
\text{(instances}(u)\text{), instances}(u), i_{subclass}).
\]

By this definition, one instance of $u$ can use the inherited operations of another. Access to the subclass interface is still restricted to be within a single subclass of $k$, but is no longer restricted to self calls. This follows the C++ approach in which the unit of encapsulation is a class rather than an individual...
A. Relational Queries

The server and client sets now contain all instances of all immediate subclasses of \( k \), so an instance of one subclass of \( k \) can call the inherited operations of instances of other subclasses of \( k \). Going even further, we can define the single view

\[
\text{members}(k), \text{members}(k), i_{\text{subclass}}.
\]

Now the subclass interface is restricted to the subclass hierarchy rooted at \( k \), but any member of \( k \) can use any of its operations on any other member of \( k \).

The ability to express formally and concisely such subtle semantic variations in encapsulation mechanisms is an important strength of our view model. A general implementation of the model also permits selection and/or tailoring of the encapsulation mechanism to be used.

VI. ORGANIZING OBJECT-ORIENTED SYSTEMS WITH VIEWS

Object-oriented systems are generally organized according to their inheritance lattices. System browsers allow one to traverse the lattice easily, upwards or downwards. For each class or object in the lattice one can examine its instance variables and methods. This is just one, particularly useful way of organizing object-oriented systems. Views permit other useful ways, and allow the multiple organizations to coexist. These are discussed in the remainder of this section.

A. Relational Queries

The access discipline corresponding to a view discipline is a ternary relation, a set of \((s, c, o)\) triples. Relational queries can be applied to this relation, permitting indexing of objects or classes according to the three criteria:

- **By server.** One can look at objects or classes in their roles as servers, for each one seeing what interfaces it provides and to what clients.
- **By client.** One can look at objects or classes in their roles as clients, for each one seeing what interfaces it uses and which servers make those interfaces available to it.
- **By interface.** One can look at the system based on interfaces, for each one seeing what servers provide that interface and what clients use it.

One can, of course, formulate queries involving combinations of these criteria, such as looking at all clients of a particular server that support a particular interface. This flexibility is particularly important in software engineering and computer-aided software design. For example, impact analysis determines what clients use it. However, it is possible for some methods in an interface to use them in contexts manipulated by that code provided they satisfy the appropriate abstract types.

B. Viewpoints

Objects that are significantly different often have views that serve similar purposes. For example, every object has both a public view and a subclass view in our approach to encapsulation of inheritance. Every object that is to be involved in debugging has a debugging view and every object that is to be displayed has a visualizing view, as in the examples discussed in Section V-A. A set of views serving a common purpose is called a viewpoint.

Viewpoints provide a means to examine or interact with many different objects from a single, consistent point of view. This relates to one of the important advantages of abstract types \([6, 7]\), discussed in Section III: that clients can operate in terms of abstract types rather than concrete objects. This allows a variety of objects to be used in a given context, with concern only for whether those objects support the abstract type expected in that context. New types of objects can be created, never imagined by the designer of some piece of code, yet it will be possible to use them in contexts manipulated by that code provided they satisfy the appropriate abstract types. This is similar to the case of viewpoints:

- **By server.** One can look at objects or classes in their roles as servers, for each one seeing what interfaces it provides and to what clients.
- **By client.** One can look at objects or classes in their roles as clients, for each one seeing what interfaces it uses and which servers make those interfaces available to it.
- **By interface.** One can look at the system based on interfaces, for each one seeing what servers provide that interface and what clients use it.

Abstract types, once defined, can be used anywhere; the abstract type mechanism contains no way of restricting access to some views provided in the abstract type. In the case of viewpoints, each view within a viewpoint can control access to its servers as it sees fit. An important case is that of all views in a viewpoint having identical interface. For example, all views in the debugging viewpoint might be made visible only to the system debugger. In this case, the access restrictions are effectively not on individual views but on the viewpoint as a whole. In abstract type terms, they control which clients may use an abstract type, without restricting the concrete types that may be used to support that abstract type.

Viewpoints, then, collect together sets of views that serve common purposes. They are important because they allow such purposes to be identified and made explicit, they allow views to
be organized according to such purposes, and they allow for simple specification of access restrictions based on these purposes. Viewpoints will be illustrated and considered further when the grid mechanism is presented as a means of implementing views, in Section VII-E.

C. Operations over Collections of Objects

Organization effectively consists of identifying and making visible sets of entities that are related in important ways. We have seen how views allow for the identification of a number of such sets: the server and client sets of particular views, the sets that result from relational queries, and the sets identified by viewpoints that serve important purposes in the system.

Thus far we have considered using these sets as a basis for browsing. It also makes sense to allow access to them by the system itself at run time: iterators and mapping functions can be defined to operate over them.

In a hierarchical organization, for example, it is often necessary to perform some operation on all children of a node. In object-oriented systems, a common case is broadcasting a message to all instances of a class. This can be accomplished by defining an instance view for class \( k \) as

\[
(S, \text{instances}(k), I)
\]

(17)

which makes operations in \( I \) provided by servers in \( S \) available to all instances of \( k \). In addition, the client set itself can be made available to objects that wish to make use of it. A similar approach is possible for server sets.

Use of views in this way has profound implications for their implementation. It must be possible to refer to a view within the system governed by that view, so as to identify the desired set. It must also be possible for that set to be synthesized with acceptable efficiency. The ability to refer to a view is easily provided, either by materializing views as objects in the system or by associating with them identifiers that are objects or symbols in the system and that serve to identify them uniquely in whatever space they occupy. Ensuring that the elements can be found efficiently is, however, a significant constraint on implementation, discussed further in Section VII.

In summary, there is still work involved in maintaining sets of objects, but views provide a convenient, uniform framework within which to do it. Having it done by the view mechanism relieves the programmer of the burden of doing it him/herself. Iterators over such sets thus represent a significant new capability for conventional object-oriented systems.

VII. IMPLEMENTATIONS OF VIEWS

In their access control role, views are similar in many ways to the operating system concept of capabilities. A capability represents a permission to use a shared resource. Capabilities are usually implemented in one of two ways: membership sets or unforgeable keys. Like capabilities, views can be implemented in different ways, including these two.

In addition to the fundamental issue of access control, we have identified a number of ways of using views that have implications for implementation approaches:

- Performing relational queries on the access discipline specified by a view discipline. This might be done either at browse time by the user or programming environment, or at run time by the running system itself.
- Allowing access to the server and client sets of views. These are separate and important special cases of performing relational queries.

Some implementation approaches favor some of these uses over others, and some make certain uses effectively impossible. An implementor must decide his/her priorities in terms of these requirements, and must choose his/her implementation strategy accordingly.

The rest of this section presents five approaches to implementing views: a relational database, membership sets, views as objects, keys, and the grid mechanism.

A. Relational Database

Since a view discipline defines an access discipline that is just a ternary relation, an obvious approach to implementing views is by means of a relational database containing that access discipline. A check for the validity of an operation invocation then becomes a check that the corresponding invocation triple exists in the database.

Representing the access discipline alone is sufficient for enforcing access control, but the database will have no record of the views used to specify it. The views can be represented as additional relations, as can viewpoints. If this is done, browsing and access using relational queries and viewpoints, both at browse time and at run time, are supported directly. Update transactions to the database can be used at any time to change the view discipline.

There are two primary problems with this implementation approach. The first is that a view discipline must be explicitly converted to the corresponding access discipline. How difficult this is depends on the manner in which server and client sets are specified. If they are specified as sets of individual objects listed explicitly, there is no problem. If they are expressed using the instances or members functions, however, there is a serious problem due not only to size explosion but also to the changing nature of these sets, which are often not maintained explicitly in an object-oriented system. This can be dealt with by allowing a class identifier \( k \) to be used as a server or client in an invocation triple to stand for instances(\( k \)).

Uses of member in views are converted to identifiers of all subclasses; this conversion depends just on the class hierarchy and not on the instances in existence, and so is less prone to frequent change. A check that the operation invocation characterized by \( (s, c, o) \) is permissible now becomes a check that any of the following four tuples is in the database: \( (s, c, o) \), \( (class(s), c, o) \), \( (s, class(c), o) \), \( (class(s), class(c), o) \). If all views are specified exclusively in terms of classes, then only the last tuple need be checked.

The second problem with the database approach is efficiency. Operation invocation occurs with such frequency in object-oriented systems that it must be highly optimized, and a good deal of work has been done on optimizing method call. Full database accesses on each invocation would thus be intolerable. Caching techniques have been employed successfully in optimizing method call, however, and should also be applicable to access control checking.

Another approach to dealing with the efficiency problem is to perform access checking statically. This obviously imposes...
serious restrictions: the view discipline cannot change at run time, and there must be enough information manifest at compile time relative to the granularity at which the view discipline is specified to make accurate checking possible. If such information is available in many cases but not all, a compiler could perform static checks when possible but insert code for dynamic checks when needed.

B. Membership Sets

The simplest implementation of views in a conventional object-oriented system is a special instance variable of the servers, called the view table. A view with multiple servers (i.e., whose server set is not a singleton) is split into multiple views, each with a single server. Each of these views is then defined as an entry in the view table for the server. An entry consists of a view (or viewpoint) identifier, a client set and an operation set. View tables can be shared among objects, such as among all instances of a class.

The view tables can be created in many ways, such as at system initialization time for all classes, at object creation time, or by action of privileged objects in the system. An especially natural approach in this context is registration. A client registers with a server by means of a request method to the server. This request method can be part of the public view of the server, so that any object can attempt to register. The request method can use any criteria that are appropriate to determine whether or not to grant the request. This includes the name of the object requesting registration and the number of existing clients. Similarly, a client leaves a view by means of a withdraw method, which is only available to members of that view.

To check that operation implementation \((s, c, o)\) is valid, the underlying system must have access to the view table of \(s\), and the operation set of \(c\) and \(o\). This is potentially an unacceptably time-consuming operation, but caching can reduce the cost significantly.

In early-binding implementations, we assume that any object that can access a view knows all operations made available by that view. Late-binding systems can exploit the request message for entering a client set as a means to dynamically determine the methods associated with a view. A positive response to a request can acknowledge view membership, and can also return the set of available view methods. Changing the set of view methods can be handled through a selective broadcast.

This implementation technique does not provide explicit server sets. It maintains client sets explicitly for each server, that can be used for browsing or iteration, but it cannot provide access to the server or operation. It does facilitate all the other uses of views identified above.

C. Views as Objects

An approach related to membership sets that eliminates some of their disadvantages is to materialize views as objects in the system. Each view holds a representation of the server, client, and operation sets. These representations need not be as sets of individual objects; their representation can be chosen based on the designer’s priorities regarding the operations to be performed on them.

Checking of operation invocations can be performed in one of two ways. The first approach fits most naturally into standard object-oriented systems. In addition to maintaining the views as objects, include a view table in each server as described in the previous section, except that entries are now actual views. Checking then proceeds much as described for membership sets, with caching for efficiency.

The second approach requires a fundamental change to the mechanism of operation invocation in the underlying language: all operation invocations must be addressed to a particular view, with the server, client and operation selector as arguments. The client argument (the sender or caller) should be supplied implicitly by the underlying system. The view object can then perform authentication, and if it succeeds, invoke the server on behalf of the client by means of a primitive form of invocation not available to other objects.

In addition to supporting access control, views can provide a variety of other operations, such as to iterate over or modify their server, client, and operation sets, or to permit navigation of views by viewpoint or other criteria. The set of operations provided clearly affects the uses to which the views can be put and will dictate the nature of their representation.

D. Keys

Unforgeable keys represent a major departure from conventional object-oriented systems. Keys require a reliable underlying implementation that prohibits or detects tampering with this particular datatype. However, like capabilities, views implemented with keys provide greater flexibility than views implemented with membership sets when it comes to dynamic changes to client sets.

Keys allow view membership to be controlled both by the server and by the clients. The server issues keys upon request, after validation based on suitable criteria. A client, however, can pass a key to another object: this results in the first client leaving the view and the second joining the view. A key that can be duplicated allows not only the identities of the clients to change, but their number as well.

Key-based validity checking of operation invocations is much more efficient than any of the other means described, especially with the special hardware normally available for capability checking. Key-based implementations do not, however, support browsing or iterating over the various sets of objects described earlier.

E. The Grid Mechanism

The grid mechanism [18–21] is a structuring mechanism designed to specify and enforce the structure of large, layered systems. It supports system organization based on a two-dimensional matrix and two directories. As such it serves as a good vehicle for implementing and managing systems that are organized in two different ways simultaneously. This ability can be exploited to advantage in systems with viewpoints.

Fig. 7 shows a sample grid based on earlier examples. The first two columns represent classes from the example of Section II, and the third represents a distinguished object, the system debugger. The column headers are arranged in the familiar class hierarchy. Each row is a viewpoint; the figure shows the four viewpoints discussed in earlier sections. Each matrix entry is an individual view in this example, a set of views in general. A view is placed in the matrix in the column determined by its server set and in the row determined by the viewpoint to which it belongs. Views with large server sets spanning multiple columns can be split into separate views that fit in a single column each. Similarly, views belonging to more than one viewpoint can be duplicated in the appropriate rows.

The advantages of this organization are that viewpoints, as well as classes (or objects), are explicitly identified, and views
Views are a natural technique for collecting and organizing these caller-method pairs.

The prototype implementation of the grid mechanism was written using multiple views. The result was a clean structure, with a number of desirable properties [20]. The structure of the Scribe document processing system [24] was specified using the grid. Although Scribe was not originally written with multiple views in mind, views imposed after the fact helped to specify important structural features of the system [21].

Our current work on views is within the context of RPDE. [25], an open-ended, structural framework for integrating tools that manipulate objects. Each tool in an RPDE environment expects the objects it manipulates to implement a particular collection of methods, and any object that does so can be manipulated successfully. Such collections of methods are called roles, and they are used as the basis for type checking [26].

Roles, like abstract types in Emerald, express the structural property that an object or tool can depend on the operations supported by an object in a particular context, but should not depend on the actual class to which that object belongs. Our experience with roles has shown this property to be important. It permits new kinds of objects to be used in unanticipated contexts. We find regularly that new kinds of objects created with a specific tool in mind can be manipulated successfully by other tools as well, and that new tools can manipulate a wide variety of existing kinds of objects without needing to change them.

Our experience has also shown that many roles are specifically intended for use by specific clients or kinds of clients, or for use by implementations of operations in other specific roles. Such intentions are important from the point of view of system understanding; they are, in fact, quite fundamental structural properties of the system. To express them requires moving from roles alone, which correspond to view interfaces in our model, to full-fledged views including client sets. The issue of what restrictions are appropriate and how best to characterize the client sets is a current area of research.

IX. CONCLUSION

Allowing objects to have multiple interfaces provides finer-grained information hiding than is usually available. A client can use an interface that contains just the operations it actually needs, and so can avoid becoming dependent on other details of servers that it has no need to know. A number of languages and systems have provided multiple interfaces, but usually with no control over what clients can use what interfaces.

In this paper we proposed a model, called views, in which objects can have multiple interfaces, and the clients permitted to use each of the interfaces can be controlled by means of client sets. Views thus provide secure, fine-grained access control not normally available in object-oriented systems. They also provide a clean, uniform technology for structuring and browsing complex systems. We have outlined five approaches to implementing views: a relational database, membership lists, views as objects, unforgeable keys, and the grid mechanism.

Our current research on views is proceeding in the context of RPDE. It centers on the issue of how to specify, present, and enforce the important relationships among tools, roles, and object types, and on the use and implementation of multiple views in object databases [27]. The RPDE environment will enable us to investigate issues associated with multiple views and to gain further experience with their use in practice.
ACKNOWLEDGMENT
We thank A. Snyder for helping us realize that our work had a common focus. We also want to thank A. Black, J. Padgett, J. Shilling, P. Sweeney, and D. Yellin for their comments on earlier drafts of this paper and G. Kuper for valuable assistance with LaTeX. Finally, we thank the referees for many helpful suggestions.

REFERENCES


