QUERIES IN AN OBJECT-ORIENTED GRAPHICAL INTERFACE

S. Dar

AT&T Bell Labs
Murray Hill, New Jersey 07974
&
University of Wisconsin
Madison, WI 53706

N. H. Gehani

AT&T Bell Labs
Murray Hill, New Jersey 07974

H. V. Jagadish

AT&T Bell Labs
Murray Hill, New Jersey 07974

J. Srinivasan

DEC
Nashua, NH 03062

1. INTRODUCTION

Many database users prefer to access and manipulate information in databases visually, without using a textual interface, such as a traditional programming language or SQL. Consequently, there is much interest in the design of good graphical user interfaces (GUI) for database systems. There have been several research efforts to develop graphical query languages (*cf.* [12, 14]) as front-ends (user interfaces) to databases. However, the focus has been on graphical query languages for databases organized with traditional data models such as the entity-relationship model and the relational model. In these traditional data models, there is typically a small fixed set of operations that the user can perform on the data, and the data itself consists of values of predefined types. For example, in the relational model, the database supports only simple types such as integer and character and the operations of relational algebra [7] augmented by aggregate functions [13].

With the increasing popularity of object-oriented databases (OODBMS), there is a need for GUIs that provide friendly simplified interfaces to a database organized according to an object model. In OODBMSs, the classes (object types) defined can be arbitrarily complex. The operations that can be performed on objects can also be arbitrarily complex. Typically each class has different member functions (methods) for manipulating objects of that type. The GUI must provide users with the ability to execute type-specific operations. It must assist the user in determining which operations are applicable to objects of a specific class, taking proper account of the type hierarchy, and respecting the access control specifications protecting certain class members.

In an object-oriented database, objects may be related. In particular, objects can refer to other objects. For example, a `department` object may refer to a set of `employee` objects. These references should be displayed by the interface in a uniform and intuitive way. If `department` objects are being displayed in one window, and the related `employee` objects in another window, then sequencing through the `department` objects should also result in display of the corresponding `employee`s objects in the second window.

Because of the complexity of object types and the associated functions, and because objects can be related, it is hard to devise a standard display convention that will be appropriate for objects of all types. In this paper we address the problem of designing a simple intuitive GUI that can be used by database users

without having to resort to the textual database programming language of the OODBMS.

OdeView is a GUI for the Ode object-oriented database system [2] that allows users to perform complex queries (including nested selections and updates) against sets of objects in the database. OdeView provides facilities for examining the database schema, browsing the database, and performing arbitrary queries on the database.

OdeView displays sets of related objects in windows that reflects the relationship between the objects. OdeView supports the graphical specification of query[1] expressions involving values, operators, functions, and object attributes, including nested attributes. The the type of the partially constructed query expression is used to guide a user to construct a syntactically correct query expression. The query expression is eventually translated to an executable function, called the ‘‘selection’’ predicate, which is given to the object manager. The object manager evaluates this predicate against objects in the database, and retrieves the objects for which the predicate returns true. OdeView then displays the selected objects using a display function associated with their type.

OdeView is implemented using O++, the database programming language interface of Ode [2, 4], which is upward compatible with C++ [18]. OdeView and O++ are compatible in that each interface can be used to manipulate objects created or updated using the other.

In an earlier paper [3], OdeView's browsing facilities were described (reflecting the implementation at that time). Since then, OdeView has evolved from a browser to a sophisticated GUI giving the user considerable power to view and manipulate an object-oriented database. In this paper we focus on these new facilities: the display of objects and their relationships supporting sequencing operations and selections, the graphical construction of query predicates, the use of type information in queries and updates, and the implementation of these facilities. We hope that the design and implementation details provided in this paper will benefit those interested in building GUI's for database systems, particularly for object-oriented database systems.

The rest of the paper is organized as follows. We present the OdeView display model in Section 2, and describe the actual display in Section 3. We illustrate the OdeView query interface in Section 4, and then discuss in detail several features and design decisions. We concentrate on the use of type information to construct syntactically correct queries. Our implementation strategy is outlined in Section 5. We discuss related work in Section 6, and conclude with a summary and directions for future work in Section 7.

2. DISPLAY MODEL

We define the OdeView display model by specifying the relationship between objects, types, and windows, and the operations supported by OdeView.

There are two kinds of windows in OdeView:

1. *display* windows: used for displaying objects and for performing operations on these objects.
2. *specification* windows: used for specifying the arguments for an operation to be performed on an object or a set of objects. These windows are ephemeral and disappear once the specification has been completed.

Unless ambiguous, we use the unqualified term ‘‘window’’ to refer to a display window. We discuss display windows next, and defer the discussion of specification windows to section 3 and 4.

2.1 A Display Window

Each display window *W* displays objects of a single type, denoted as *type*(*W*). The set of objects shown in a window *W* is called the *binding* of *W*, denoted as *binding*(*W*). This set has a default value, denoted as *default-binding*(*W*). This default binding is the largest set of objects that can potentially be associated with

1. We use the term *query* to refer to both display and update queries.

the window, as discussed in the next sub-section. A selection predicate, *select*(*W*), may be used to restrict this default binding. The set *binding*(*W*) contains those objects in *default-binding*(*W*) that satisfy the predicate *select*(*W*).

Objects in *binding*(*W*) are displayed one at a time, and can be scanned using the sequencing operations *first*, *next*, and *prev*. The object displayed in *W* at a given moment is called the *current object*, denoted as *current*(*W*). Operations *first*, *next*, and *prev* modify *current*(*W*) to refer to the first, next, and previous object respectively, in some total ordering of the objects in *binding*(*W*). By default, this ordering is implementation dependent. However, an explicit ordering can be imposed by specifying an *order-by*(*W*) expression.

A display window can be in either *active* or *passive* mode. When a window *W* is in active mode, the *display* function associated with *type*(*W*) is invoked to display the current object. When a window is in passive mode no objects are displayed in the window, however the window still retains its binding and tracks its current object. The amount of information displayed in an active window can be controlled by a projection operation, which is used to specify the attributes of the object that should be displayed.

Object creation and display operations apply to one object at a time. However, other operations such as object deletion and update and the invocation of a member function can be applied to sets of objects. Each of these operations has an associated *range* that determines the objects on which the operation will be carried out. The range of an operation performed in a window *W* can be specified to be either the set *binding*(*W*) or the object *current*(*W*).

2.2 The Display Forest

If an object displayed in window W_1 refers to an object or a set of objects displayed in window W_2, then we say that W_2 *depends* on W_1. The set of windows that depend on a window *W* is denoted by *depend*(*W*). The set of all display windows and the dependency relationship between the windows form a *display forest*. The display forest mirrors the relationships between the types of objects displayed in those windows. In particular, a tree in this forest corresponds to an aggregation graph.[2]

As an example, consider classes `department`, `employee` and `manager` (O++ definitions of these classes are given in the Appendix.) Each `department` object contains a reference to a `manager` object, and a set of references to `employee` objects. Figure 1 shows a display forest with `department`, `employee` and `manager` display windows. `department` is a root window and `employee` and

2. In the aggregation graph, each node represents a type, and there is a directed arc from type T1 to type T2 if T2 is used in the definition of T1. For example, a member of T1 may be a set of, an array of, or a pointer to T2. This is a generalization of the notion of an aggregation hierarchy [17].

3. Since the publication of [3], the user interface has been redesigned. In particular, audio and text-to-speech capabilities have been incorporated into the display.

`manager` depend upon it.[3]

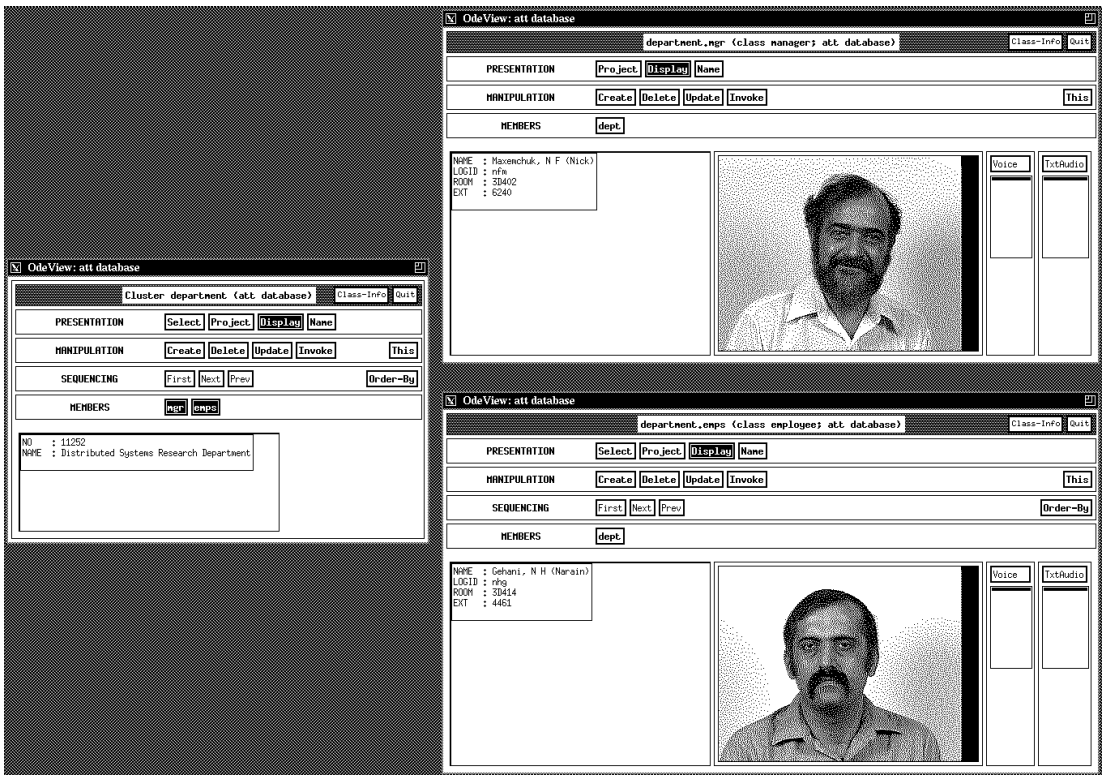

Figure 1: A display forest: `department` is the root window

The dependency relationship determines the default bindings of each window *W* in the display forest.

1. If *W* is a root window then *default-binding*(*W*) is equal to *extent*(*type*(*W*)), where *extent*(*type*(*W*)) is the set of objects of type *type*(*W*) in the database.
2. If *W* depends upon another window W_1, then *default-binding*(*W*) is equal to the object or set of objects referenced by (an attribute of) *current*(W_1) along this dependency.

The following relationships hold for each display window *W*:

$$current(W) \in binding(W) \subseteq default\text{-}binding(W) \subseteq extent(type(W))$$

When the object being displayed in a window *W*, i.e., *current*(*W*), is changed, then the binding of every window W_i that depends upon *W* is changed accordingly. This may cause the object being displayed in the dependent window W_i to change, which in turn may cause the bindings of windows that depend upon W_i to change, and so on. This recursive propagation of bindings is called *synchronized browsing*.

A window *W* may have no objects associated with it, i.e., *binding*(*W*) may be empty. This is the case, for example, when a specified selection predicate is not satisfied by any of the objects in *default-binding*(*W*). In this case, *current*(*W*) has the value *null* and no object is displayed in the window. By definition, the (default) bindings of windows W_i that depend upon *W* are empty and *current*(W_i) has the value *null*. Thus, an empty binding recursively propagates down the tree of display windows.

3. LOOK AND FEEL OF THE ODEVIEW DISPLAY FACILITIES

By and large, the actual display of objects in OdeView directly reflects the model described above. For example, in OdeView scans are performed using the `First`, `Next`, and `Prev` sequencing buttons located on each display window. The order in which objects in *binding*(*W*) are displayed can be specified using the `Order-By` button. A selection predicate may be entered using the `Select` button, as will be described in

detail in Section 4. An *All* or *This* toggle is supplied for specifying the range of operations (deletion, update, or member function invocation) as *binding*(*W*) or *current*(*W*) respectively. The default state of the range toggle is *This*.

Every display window has a *Display* toggle on it. The window is active if the *Display* toggle is on and passive otherwise. The user may control the amount of information displayed in an active window through the `Project` button. Clicking this button brings up a specification window that allows the user to turn on or off the display of individual attributes. For example, the projection specification window of class `employee` is shown in Figure 2.

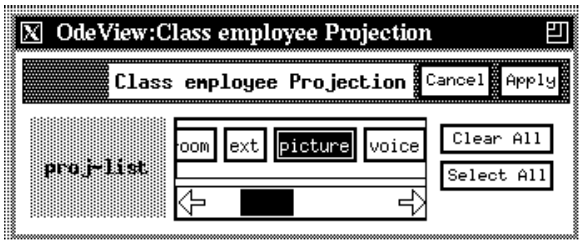

Figure 2: Projection window for `employee`

Figure 3 shows the same objects as Figure 1, but the `manager` window has been deactivated, and projection has been applied to the `employee` window so that only the picture of an `employee` is displayed.

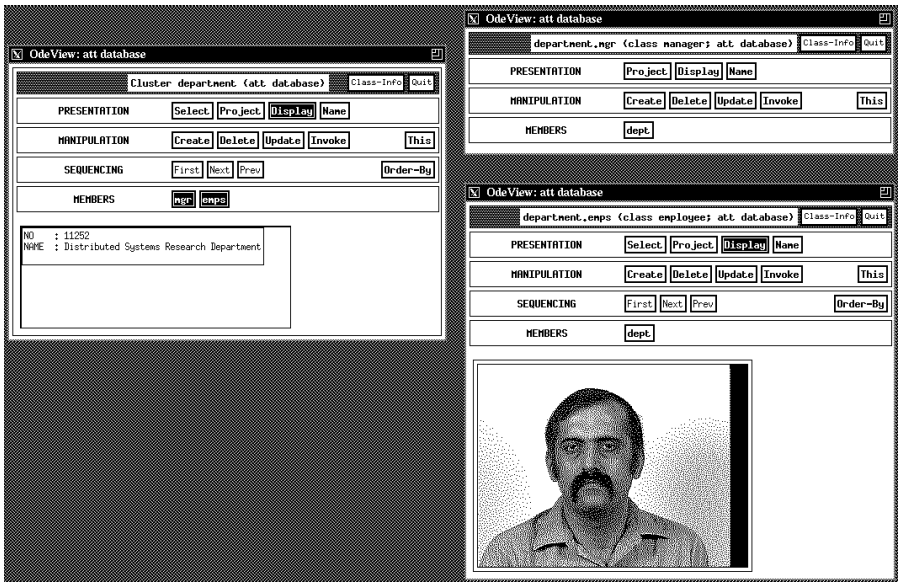

Figure 3: A display forest: `manager` deactivated, `employee` projected on picture

The title of each display window reflects its position in a dependency tree, by means of a path from the root of the tree to this window. For example, the title of the `manager` window in Figure 1 is ‘‘`department.mgr`’’. In addition, the title specifies the type of objects displayed in the window and the database the objects reside in — ‘‘`(class manager; att database)`’’.

In some cases, the actual OdeView display differs from the model for ergonomic and performance considerations. For example, OdeView differentiates between a *set* window, which is bound to a set of objects, and a *singleton* window, which is bound to (at most) one object. Because the default binding of a singleton window contains exactly one object, namely the current object, such windows have no selection

capability and no sequencing operations. In addition, the *All* or *This* range toggle is not provided on singleton windows. For example, in Figure 1 `department` and `employee` are set windows and `manager` is a singleton window.

OdeView maintains binding and current object information for each window on the screen, regardless of whether it is active or passive. However, when *current*(*W*) is modified (and is not *null*), OdeView will actually fetch that object only if it is needed in order to refresh the display in *W* or in other windows, which is the case exactly when *W* is active or has dependent windows. For example, the user may deactivate the manager window, scan through the department objects and then reactivate the manager window. At that point the current manager object is fetched, and the display function for `manager` is invoked, with this object and the current projection list for the `manager` window as arguments.

If the current object of an active window becomes *null*, an implementation may present some ‘‘empty’’ display representing the null object. In our implementation, the effect of the empty display is the same as deactivating that window, i.e., the portion of the display window used for displaying the object is shrunk (as is the `manager` window in Figure 3). If later on the current object becomes non-null, the display function is again invoked, and the bindings and current objects of dependent windows are updated recursively.

In summary, our model provides a framework and a uniform semantics for display of related objects in a type specific fashion. Synchronized browsing, the recursive propagation of values through the display forest, is based on a paradigm popularized by spreadsheet applications, such as LOTUS 1-2-3 [1]. We have combined this paradigm with the notions of binding-set, current object, and dependency between windows to obtain a model describing the semantics of sequencing and selection operations in the display forest.

4. QUERY INTERFACE

In this section we describe the interface provided to the user for specifying selection predicates, and ‘‘walk through’’ the specification of some queries. We first give a high-level overview of the OdeView query interface, and then discuss in detail several features, design decisions and alternatives. We use a selection query on `employee` as a running example.

4.1 Building a Query

An OdeView user may specify a query on any display window. The user starts the query by clicking on the `Select` button in a window. This brings up a selection specification window. For example, the selection specification window of class `employee` is shown in Figure 4.

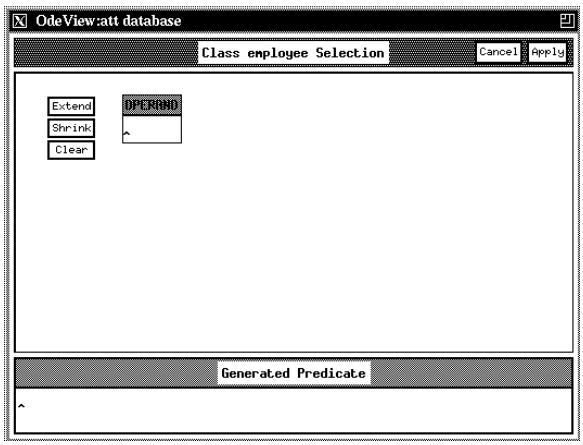

Figure 4: Empty selection specification window for `employee`

The user can then specify a selection predicate. Operands and operators in this predicate are each either chosen from context-sensitive menus or entered directly, as follows:

- The user specifies the first operand by clicking on the box marked `operand`. This brings up a menu listing the object attributes that can be used as an operand. The user can select an operand from the menu. Alternatively, the user can enter text directly in the box below `operand`.
- The user may extend the current expression by clicking on the `extend` button. In that case an `operator` box is opened and the user may choose an operator from the associated, or type its name in the box. The expression that was specified thus far serves as an implicit first operand to the operator. If the specified operator is binary, an additional operand box is opened on the right of the operator.
- The user may delete the last specified operator by clicking on the `Shrink` button. If the deleted operator was binary, its right-hand-side operand is deleted as well.

After specifying the predicate, the user can execute the query by clicking on the `Apply` button. OdeView translates the query into a function and compiles it. This function is evaluated for every object (in the default binding set of the window) returned to OdeView by the object manager, e.g., in response to a sequencing operation. Effectively, the specification of a predicate modifies the binding of this window and its dependent windows; the new bindings become visible immediately in the active windows.

OdeView also keeps a textual representation of the query, presented on the bottom of the selection specification window, in the ‘‘Generated Predicate’’ subwindow. The textual representation is updated after each stage in the query construction, such as the specification or modification of an operand or operator.

As an example, Figure 5 shows the selection specification window of class `employee` after the following query has been specified:

Retrieve all employees in department 11252 whose age is between 50 and 60.

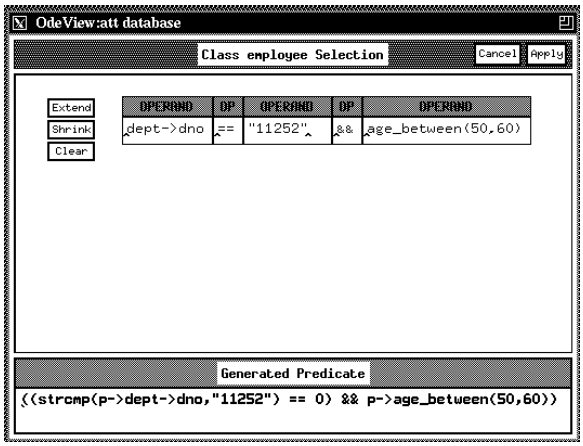

Figure 5: Selection predicate for `employee`

4.1.1 Operands

OdeView uses type information to provide users with information about possible operands and operators. Initially, a menu of all the members of the current object type is provided. If the user selects as operand a member that refers to another object, then OdeView automatically makes available a ‘‘sliding’’ menu with elements that are members of the referenced object. This is indicated to the user by an arrow on the right hand side of the member name. This mechanism allows the user to specify a path expression [16] (a.k.a ‘‘implicit join’’) of any length. Figure 6 shows the operand menu for `employee` with sliding menus for `department` and `manager`.

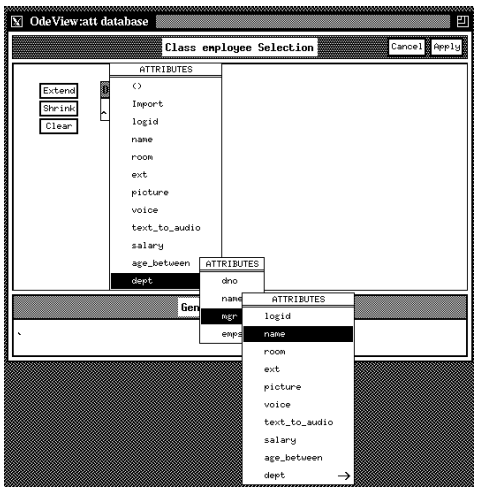

Figure 6: Sliding operand menus

4.1.2 Operators

The `Extend` button is used for extending the specified query expression by applying some operator to it. Clicking on the `Extend` button opens an operator box. OdeView determines the type of the expression specified as operand and uses this information to show the appropriate operators in a menu. For a built in type OdeView brings up a menu containing predefined operators for that type. In Figure 7, the user has specified member `logid` of `employee` as operand. The user then clicked on the `Extend` button in order to specify an operator to apply to `logid`. The type of `logid` is a character string. The operator menu brought up by OdeView includes operators for string manipulation, such as comparison (menu choices >= and <=) and containment (menu choices >> and <<).

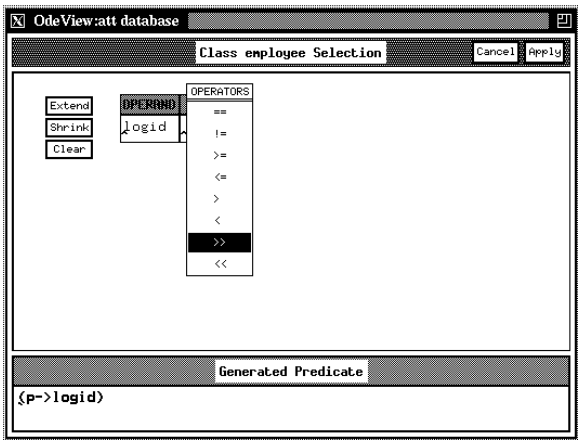

Figure 7: Operator menu for strings

A query predicate can be built in steps using the menu for Boolean types, which includes logical conjunction (&&), disjunction (||) and negation (!). For example, conjunction was employed in the query displayed in Figure 5.

When the type of the operand is a set of values, the menu shown by OdeView includes set operators for a set of values of the appropriate type. As with single values, the operators in the menu can be unary or binary. The former include aggregate functions (e.g., `COUNT`), which reduce a set of values to a single value. The latter include generic set operations, which take two sets as arguments and produce a single set

(e.g. $\cup$) or a single value (e.g. $\subseteq$). As an example, the menu for a set of numerical values includes the aggregate functions `SUM`, `AVG`, `MIN`, `MAX`, and `COUNT`, as well as the generic set operations `==`, `!=`, $\subseteq$ and $\supseteq$. If a binary set operator is selected, an additional operand box is opened on the right of the operator.

As an example, consider a query to find all employees earning more than the average for their department. The user may start by designating the set of salary values for the employees in a department, by using the sliding operand menus for class `employee` to designate the set `dept->employees->salary()`. Next he can apply the aggregation operator `AVG` to that set-valued operand, resulting in a single numerical value. The user may now extend the predicate and choose the `<=` operator. OdeView will open a second operand box for the operator, in which the user can specify the `salary()` attribute of `employee`.[4] Thus the predicate can be specified with three steps, each involving a single menu choice.

4.1.3 Functions

When a function requiring arguments is chosen as an operand by the user, OdeView assists the user in specifying the arguments graphically by moving down one ‘‘level’’ in the specification window. The function arguments are then entered in the same manner as top-level operands. After the arguments have been specified, the user clicks on the `Done` button for this level. OdeView then removes the argument specification boxes for the function, while instantiating the arguments in the function invocation one level up.

There could be additional function invocations in the arguments to a function, in an arbitrarily deep nesting. Conceptually, we have a tree of invocations. However only one path down the tree from root to leaf may be active at any one time, and this is the only path that is displayed. The user must complete lower levels and close the corresponding nodes before higher (parents in the tree) levels can be completed.

4.1.4 Parenthetical Expressions

By default, the evaluation of a selection predicate proceeds in left-to-right fashion. The user may override this default by using a parenthesized (sub)expression. This is done by choosing the `()` option from the corresponding operand menu. Parenthetical expressions are treated just like functions — a new level of specification is opened below the current one. After the specification is complete, the new level is closed and the subexpression is integrated into the higher level.

Figure 8 shows a selection window with multiple levels of specification. The query retrieves all employees in department 11252 who either earn more than 50,000 dollars, or whose age is between 50 and 60. At the top level the user has specified an expression involving a conjunction. To override default left-to-right evaluation, the right conjunct is encapsulated in a parenthetical expression. OdeView has opened a second level of specification to allow the user to enter that expression. The nested expression involves a disjunction, with the right disjunct being an invocation of a function, `age_between`. The function takes two integer arguments and returns a boolean value indicating whether the employee’s age is between the two numbers. A third level of specification was opened to allow the user to enter the function arguments. The user has then entered both arguments. Figure 8 shows the state of the selection window at this point. The user can now incorporate the function invocation into the second level by clicking `Done` at the third level, and then incorporate the nested expression into the top level by clicking `Done` at the second level. The user may now execute the query by clicking on the `Apply` button.

---

4. We have specified the predicate in the above order to match the default left-to-right evaluation order. A different order could be obtained using parenthetical expressions, as described below.

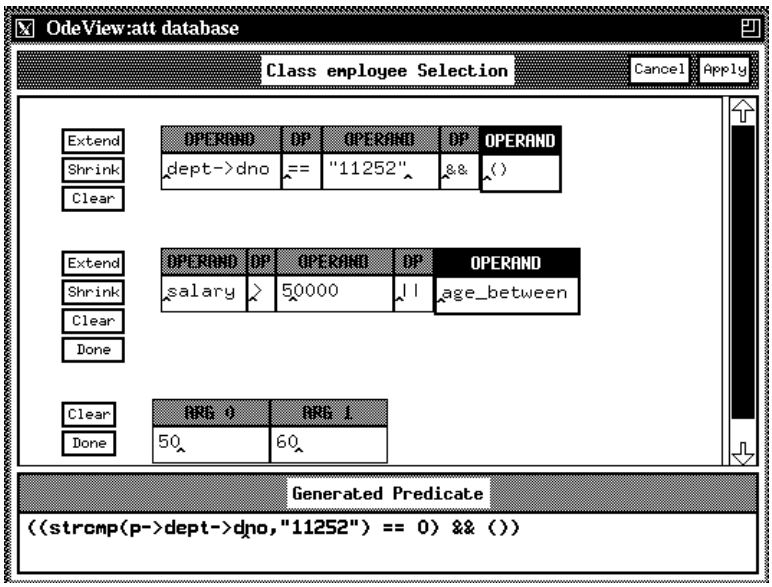

Figure 8: A selection window with multiple levels of specification

4.2 Query Modification

Odeview allows the user to modify previously submitted queries. For each display window, OdeView remembers the last query specified by the user in that window. This query may be recalled and modified either by editing the query text directly, or by using the graphical editing feature of OdeView that lets one modify parts of a query on the screen. For example, suppose the user had specified a selection predicate `dept->dno == "11252"` for the `employee` extent, which would find all employees in department 11252. Now the user wants to apply the predicate `islongname` on the resulting set. The user can extend the selection predicate by using the && operator and specify the next part of the selection predicate. If the user modifies an operator box, the part of the selection predicate to the right of the operator box is destroyed if the modified operator does not have an appropriate return type, or the right number of arguments. However, if the modified operator ‘‘matches’’ the original operator, then part of the selection predicate to the right of the operator box is not affected. Changing an operator box has no effect on the part of the selection predicate to the left of it because OdeView uses the type information in a strict left-to-right fashion. The effect of changing an operand box is similar.

4.3 Object and Set Identification

An object or a set of objects can be named, for use in updating objects and in complex queries (illustrated later). Objects are named by clicking on the `Name` button. OdeView then prompts the user for an identifying string. If the range toggle is in the `This` position, the string is associated with the current object. Otherwise, the string is associated with the binding set of the window. If an identical name tag is currently associated with a different object or set of objects, the requested association is refused, and an error message is displayed.

The name association with an object set is static — even though updates to the database may modify the binding set of a window, the set of objects associated with the name do not change. However, any updates to objects in the set will be visible whenever the objects are displayed.

4.4 Invoking Functions

Each display window has an `Invoke` button that allows the user to call a member function of the corresponding class. If the user clicks this button, a new window pops up in which the user can specify the function to be invoked, in a similar manner to the specification of a selection predicate. When the `Apply` button is clicked, the specified function is applied to all objects in the binding set of the window if the range toggle is in the `All` state, or only to the current object, if the range toggle is in the `This` state.

4.5 Updating the Database

OdeView provides facilities for creating, deleting, and updating objects.

4.5.1 Object Creation

When the `Create` button is clicked, a creation specification window pops up, with attributes listed for which the user must supply values. The values are supplied through the same mechanism as when operands are supplied for predicates and functions: from the keyboard or by pointing and clicking, and as constants, as the result of function evaluation, or through arbitrarily long chains of pointer dereferences. In case a requested value is a previously named object or set of objects (or can be derived from one through pointer dereferences), the user may specify it using the name tag defined earlier. For example, suppose that a new employee joins department 11252. The user may first name the department 11252 object, by making it the current object in a department display window, (through sequencing operations and/or selection), and then using the `Name` button with the range toggle in the `This` state, as illustrated in Section 4.3. Let this name be ‘‘d11252’’. The user may then create a new employee object, and enter ‘‘d11252’’ as the value of the `dept` data member.

A new object created in a window is placed in the extent corresponding to the type of that window. It is also implicitly included in the default-binding of the window. The new object may or may not be included in the binding of the window, depending upon whether or not it satisfies the window's selection predicate (if any).

4.5.2 Object Deletion

When the `Delete` button is clicked, if the range toggle is in the *This* position, then the current object is deleted, and the next object becomes the current object. If there are no more objects, the deletion succeeds but a display error occurs, just as if the `Next` button is clicked on the last object, or the `Prev` button is clicked on the first object. If the invocation is of a `Delete All`, a window pops up, requesting the user to confirm (or cancel) the operation. Upon confirmation, the objects in the binding set of the window are deleted, and the current object becomes *null*.

4.5.3 Object Update

When the `Update` button is clicked, an update specification window pops up listing all public data members of the class along with their values for the current object. The user can change these values and then click on the `Apply` button to record the new values in the database. As with deletion, an `Update All` operation requires the user's confirmation.

The user may supply the values graphically or by typing. In particular, clicking on an attribute whose type is a set of objects brings up a menu with entries `Assign`, `Add`, and `Remove`. Continuing the example of Section 4.5.1, the user may now want to insert the new employee into the set of employees of department 11252. The user can do so by assigning a name tag to that employee, clicking `Update This` on the `department` display window, with department 11252 being the current object, bringing up a menu for the `emps` attribute, choosing the `Add` option, and finally designating the new employee to be added.

4.6 Complex Queries

The process of executing a selection operation involves iterating over the default binding of the window and evaluating the selection expression for every object iterated on. When an operand value is specified in terms of the attributes of the window type, it is assumed that the intention is to specify this value relatively. That is, a different absolute value of the operand is computed by evaluating this expression on each object being iterated on. For instance, in Figure 5 the specified predicate contains the expression `dept->dno`, which is computed relative to each object in the default binding of `employee`.

Sometimes, it is useful to be able to specify a value absolutely, so that it is constant throughout the iteration. For example, the value `"11252"` in Figure 5 is absolute. If the value is of a simple type, it may be typed in directly. But a different mechanism is required if the value is an object or set of objects. For instance, we may wish to select all employees who work in the same department as Gehani. Here Gehani is a specific object, and his department is also a specific object, which must therefore be specified in an *absolute* way.

To permit absolute value specification, OdeView provides an `import` option in each operand menu. When the user selects `import`, an import specification window pops up. The user may specify an object or set of objects to import in one of two ways. The user may key in the name of a previously named object, set, or type extent. Alternatively, the user may perform the import by pointing and clicking in any display window in the display forest. Depending on the setting of the All/This toggle, either the current object in the window or the binding set of the window is imported into the predicate specification.

Once the user has import specification is complete, the specification window disappears, and the mouse is warped back to the menu, where the `import` option is replaced by a string identifying the imported value, and shaded to indicate selection. If the import was done by pointing, the menu option shows `specified` <*type*>, where <*type*> is the type of objects in the pointed window. Otherwise, the menu option shows the name of the selected object or set of objects. An arrow to the right of the imported name indicates that OdeView has made available to the user a ‘‘sliding’’ menu, allowing the user to specify attributes of the imported object or set of objects. Thus, the use of imported objects is consistent with the normal use of sliding menus to perform member selection (i.e., specify a path expression), described in Section 4.1.1

With the `import` mechanism, users can specify complex queries including value-based joins and nested queries. As an example, consider again the query to find all employees in the same department as Gehani. To illustrate the use of the import mechanism, we give four different ways in which this predicate could be specified. Perhaps the most intuitive way is for the user to open an `employee` window (as a root window in a display forest) and make Gehani the current object. Then the user may click on the selection button, and specify the partial selection predicate (operand and operator) `‘‘dept == ’’`. The second operand may be obtained by importing the current object from the same window. To do so, the user sets the range toggle to the *This* position, chooses *import* from the operand menu, and then clicks on the `employee` display window. The user can then follow the sliding menu to the right and choose `dept` attribute of the imported object.

Alternatively, the user could open a `dept` window dependent on the root `employee` window, and directly import Gehani’s department without having to use the sliding menu. Finally, the user could name either the Gehani employee object, or his department, and import these by name.

5. IMPLEMENTATION

OdeView’s design is based on the ‘‘principle of separation’’:

The class designer should not have to know the specifics of object display (windowing software) and the display software should not have to know about the object types.

OdeView has no *a priori* knowledge of the database schema or specific class definitions. OdeView by itself cannot determine how an object of a certain type is to be displayed, or what attributes of the object can be used in a selection or projection operation. This information must therefore be provided by the class interface, i.e. using member functions of the class.

We now describe how OdeView obtains specific class information, and how it compiles and executes queries. We assume that the reader is familiar with C++ [18].

5.1 Class Interface

The class designer writes class definitions in O++. OdeView expects the class designer to provide two member functions, `display` and `member_list`, so that objects of the class can be viewed and manipulated. The `display` function returns information specifying the resources needed to display an object (e.g., the window type) and the contents to be displayed.

The `member_list` function returns a list of class members that can be manipulated by the user, i.e., projected on, selected on, or, in the case of data members, directly updated. The member list is used for building the class attributes menu (including the extended menus for nested attributes). The following information is supplied for each member:

```
enum member_kind { DATAMEMBER, FUNCMMEBER, OPERMEMBER };

typedef struct {
  char name[MAXNAME];   // name of member
  char type[MAXNAME];   // for function use return type
  member_kind kind;     // data member, member function or operator
  int  isobj;           // is this an OID?
  int  isset;           // is this a set?
  int  nargs;           // only used if it is a function or operator
} member_info;
```

The information stored by the members of `member_info` is as follows:

1. `name`: member name.
2. `type`: the member type; If the member is a member function or a member operator, then the result type is used. We eventually plan to record the ‘‘signature’’ of the member function or operator (the signature will contain the function argument types along with the result type).
3. `kind`: specifies whether the member is a data member, member function or operator. We distinguish between the latter two because they are treated differently in the interface: a second argument for a binary operator is entered on its right, while function arguments are entered on a separate level.
4. `isobj`: specifies whether the member refers to another object (or set of objects), i.e., it is an OID (or set of OID’s).
5. `isset`: specifies whether the member is a set of values.
6. `nargs`: the number of arguments expected by a member function or operator (applies only to members that are member functions or operators).

The information in a `member_info` is somewhat redundant; it is possible for example to deduce whether a member is an OID or not (value of `isobj` field), from its type declaration (value of `type` field). In fact, had the `type` field contained a complete type declaration, only the first two fields would be required. But such deduction would require OdeView to include an O++ parser. We decided it was cleaner to have the O++ compiler analyze the class definition and create a list of `member_info` descriptors. Currently, the `member_list` function cannot be synthesized automatically, since C++ does not provide persistent type information about objects. In the future, we plan to provide this functionality using a persistent type catalog for the Ode database. The type descriptors in the catalog will be filled in by the O++ compiler [4, 9], and made available to the O++ programmer and to other interfaces to Ode, including CQL++ [8], and OdeView. We are already using the persistent type catalog to give OdeView the class-subclass information it requires to graphically display the class hierarchy [3].

5.2 Query Translation

As the user constructs a selection predicate by building a tree of operands and operators, OdeView maintains an equivalent textual representation of the predicate as an O++ expression. OdeView displays this expression at the bottom of the selection specification window. When the user clicks on the `apply` button, indicating that the selection predicate has been fully specified, OdeView wraps the expression specified by the user into an O++ function, called `sel_func`. The predicate function generated by OdeView takes as argument a pointer to an object of the corresponding class. It returns either true or false indicating whether or not the object satisfies the query predicate.

For example, consider the query to retrieve all employees in department 11252 whose age is between 50 and 60. The graphical representation of this query is shown in Figure 5. The generated query expression can be seen on the bottom of the selection window. The predicate function synthesized by OdeView is as follows:

```
typedef unsigned int boolean;

boolean
sel_func(employee *p)
{
    return
    ((strcmp(p->dept->dno,"11252") == 0) && p->age_between(50, 60));
}
```

OdeView submits the predicate function to the O++ compiler, which translates it into executable code. OdeView then dynamically loads the resulting object file and gives the object manager a pointer to the function. As described in the next subsection, the object manager then applies the predicate function to each object in the corresponding set of objects, and returns the objects satisfying the predicate.

5.3 Object Manager Interface

OdeView calls the Ode object manager to get objects from the database. When a display window is opened, the object manager creates an iterator[5] on the associated set of objects, i.e., the window's default binding. The iterator is provided with a pointer to a selection function, of type `sel_func_ptr`, defined as follows:

```
typedef boolean (*sel_func_ptr)(void *);
```

If the selection function pointer is not *null*, then the sequencing operations provided by the iterator only return objects for which the application of the function returns TRUE.

We give below a simplified definition of the iterator class provided by the object manager.

```
class Iter {
    OID current;  // current object
    sel_func_ptr sfp;   // selection function pointer
    ...
public:
    void reset(sel_func_ptr = 0); // reset the iteration w/ or w/o selection
    OID getNext();      // get next object (satisfying selection)
    OID getPrev();      // get previous object (satisfying selection)
}
```

The `reset` function is used to reset the iteration with or without a selection function specified. The last two functions, `getNext`, and `getPrev` are used for sequentially getting objects from the iterator. Currently, the implementation does not support an `order-by` specification.

In addition, function `fetchObj` is called to dereference an object id:

```
void *fetchObj(OID);
```

For example, when a window is active and the current object is changed, `fetchObj` is called to fetch the current object so it can be displayed.

6. RELATED WORK

Formal models for GUIs are usually built around some well understood formal concept such as state transition networks (several examples can be found in [6]). These models describe the different states of the system, the inputs to the system, and the resulting state transfers and outputs. Our model is more specific: we assume a fixed structure, a forest of directed acyclic graphs, obtained by ‘‘unrolling’’ the aggregation (sub)graph corresponding to a portion of the schema of an object-oriented database, such as

5. An *iterator* is a control abstraction used for the production of a sequence of values [15].

Ode. Our concern is the flow of values, representing object ids, through this structure.

Other work in GUI design theory studies the interaction with a user, using some *dialog modes*. Where possible, OdeView supports both the menu-type (MM) and command-language-type (CLM) dialog modes [11]. In particular, while constructing a query predicate, both representations are kept synchronized, and both are visible to the user and available for editing.

Most of the reported work on GUI design for database concentrates on providing useful functionality. Several database GUIs that support more than relational functionality have been developed in recent years. For instance, Picasso [12] is a graphics based query language designed for use with a universal relational database system. It exploits the hypergraph semantics of the universal relation to help the user form queries.

The Pasta-3 graphical query language [14] interfaces a knowledge based system KB2, which uses an entity-relational model, extended with inheritance and deduction rules. Pasta's querying capability supports all queries that are expressible in KB2's linear language, which include – complex logical queries, quantification, limited recursion, and mathematical formulas expressed as predicate calculus formulas. Pasta-3 and OdeView are similar in that both provide a graphical interface (environment) rather than a ''naked'' language [5]. However, OdeView supports an object-oriented data model, allowing a selection predicate to be built up of type-specific operators, function and data members (including nested members), and values. OdeView is able to use type information to assist the user in specifying a selection predicate, whereas Pasta-3 does not appear to use any type information. Another difference is that in Pasta-3 logical operators are treated specially, whereas in OdeView all operators (including logical operators) are treated uniformly.

Recently, some simple GUIs for object-oriented database management systems have been designed. For instance, The O_2 user interface generator, O_2Look, supports the display and manipulation of complex objects of user-defined types [10]. The user can customize the display through the use of *masks*, which perform projection, and *resources* which control properties of the display such as font, color, and layout. However, relationships between objects are not shown, and a query facility is not provided.

The SMARTIE system is a forms-based GUI for the ITASCA distributed ODBMS [19]. It automatically generates visual presentations for class objects based on domain information of the class attributes. Such an approach could be used in OdeView to generate a *default* `display` function for a class, when the class designer does not provide one. SMARTIE supports dynamic schema evolution, by storing presentation functions in the database. However, it only provides a simple textual query capability rather than a full-fledged graphical query interface.

7. CONCLUSIONS

We have described the design and implementation of a GUI for the Ode object-oriented database system. The interface uses display windows that show objects and their relationships, and specification windows, used to apply operations on the objects. We first presented a formal model that defines the semantics of operations on display windows, namely *what* these operations do. We then described the engineering of the specification windows, illustrating *how* the operations are specified. In particular, we illustrated the specification of selection predicates. Type-specific menus are used to guide the user in graphically constructing predicate expressions, which may involve values, operators, functions, and object attributes, including nested attributes. We also described how objects can be created and updated by filling information in templates specifying the object's attributes. We have attempted to keep the interface simple and intuitive, and hide the complexity of the underlying database model from the user.

We have completed the implementation of most query facilities of OdeView. The facilities for complex queries and for updates have not been implemented yet. In addition to continuing the implementation, we also need to make OdeView more easily extendible, so it can be applied in a straightforward manner to new types and objects added to the Ode ODBMS. An important issue that we plan to address is the interaction of OdeView with the Ode type system, which is based on C++ classes. As suggested in Section 5.1, we are investigating the use of a persistent type catalog to store information about types of objects in Ode. OdeView should probe this catalog for such information as type derivation (needed for examining the database schema [3]), and pointers to the `display` and `member_list` functions (needed for displaying

objects and type-specific menus). The design of such a catalog is an interesting open research problem.

8. ACKNOWLEDGMENTS

We are grateful to Alex Biliris, Bala Krishnamurthy, and Oded Shmueli for their comments and suggestions. Jia-Lin Chen and Ted Roycraft contributed to the implementation of OdeView. Ted Kowalski provided us with a dynamic linker for C++. We also appreciate the help of Doug Blewett who helped us understand X-Windows and HP Xwidgets and gave us a utility he had written to print X-windows.

QUERIES IN AN OBJECT-ORIENTED GRAPHICAL INTERFACE

S. Dar

AT&T Bell Labs
Murray Hill, New Jersey 07974
&
University of Wisconsin
Madison, WI 53706

N. H. Gehani

AT&T Bell Labs
Murray Hill, New Jersey 07974

H. V. Jagadish

AT&T Bell Labs
Murray Hill, New Jersey 07974

J. Srinivasan

DEC
Nashua, NH 03062

ABSTRACT

The graphical user interface OdeView for the Ode object-oriented database system allows users to perform complex operations against sets of objects. These include selection, projection, display, creation, deletion and update of objects, and the invocation of member functions (methods). OdeView utilizes type information for displaying objects and the relationships between objects, and for assisting the user to construct syntactically correct query predicates.

In this paper, we present a formal model that underlies the OdeView display interface, illustrate the query facilities through examples, discuss our design decisions, and describe our implementation. As object database systems are becoming increasingly popular, we hope that the design and implementation details provided in this paper will benefit those interested in building graphical interfaces to database systems, particularly object-oriented database systems.

REFERENCES

[1] *Lotus 1-2-3 User Manual*, Lotus Corporation, 1986.

[2] R. Agrawal and N. H. Gehani, ‘‘Ode (Object Database and Environment): The Language and the Data Model’’, *Proc. ACM-SIGMOD 1989 Int’l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[3] R. Agrawal, N. H. Gehani and J. Srinivasan, ‘‘OdeView: The Graphical Interface to Ode’’, *Proc. ACM-SIGMOD 1990 Int’l Conf. on Management of Data*, 1990, 34-43.

[4] A. Biliris, S. Dar and N. Gehani, *Ode 1.1 User’s Manual*, AT&T Bell Laboratories, 1991.

[5] D. M. Campbell, D. W. Embley and B. Czejdo, ‘‘Graphical Query Formulation for an Entity-Relationship Model’’, *Data and Knowledge Eng. 2*, (1986), 89-121.

[6] G. Cockton, in *Engineering for Human-Computer Interaction — Proceedings of the IFIP TC 2/WG 2.7 Working Conference on Engineering for Human-Computer Interaction*, Elsevier Science Publishers B.V. (North Holland), Napa Valley, California, Aug. 1989, 63-86.

[7] E. F. Codd, ‘‘A Relational Model of Data for Large Shared Data Banks’’, *Commun. ACM 13*, 6 (June 1970), 377-387.

[8] S. Dar, N. H. Gehani and H. V. Jagadish, ‘‘CQL++: An SQL for a C++ Based Object-Oriented DBMS’’, *Proc. of Int’l Conf. on Extending Database Technology*, Vienna, Austria, Mar. 1992.

[9] S. Dar, R. Agrawal and N. H. Gehani, ‘‘The O++ Database Programming Language: Implementation and Experience’’, *Proc. IEEE 9th Int’l Conf. Data Engineering*, Vienna, Austria, 1993.

[10] O. Deux et al., ‘‘The O_2 System’’, *Comm. ACM 34*, 10 (October 1991), 34-49.

[11] E. Kantorowitz and O. Sudarsky, ‘‘The Adaptable User Interface’’, *Communications of the ACM 23*, 11 (Nov. 1989), 1352-1358.

[12] H. J. Kim, H. F. Korth and A. Silberschatz, ‘‘PICASSO: A Graphical Query Language’’, *Software Practice and Experience 18*, 3 (March 1988), 169-203.

[13] A. Klug, ‘‘Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions’’, *J. ACM 29*, 3 (July 1982), 699-717.

[14] M. Kuntz and R. Melchert, ‘‘Pasta-3’s Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power’’, *Proc. 15th Int’l Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989, 97-105.

[15] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, ‘‘Abstraction Mechanisms in CLU’’, *Commun. ACM 20*, 8 (Aug. 1977), .

[16] D. Maier and J. Stein, ‘‘Indexing in an Object-Oriented DBMS’’, *Proc. Int’l Workshop Object-Oriented Database Systems*, Asilomar, California, Sept. 1986.

[17] J. M. Smith and D. C. P. Smith, ‘‘Database Abstractions: Aggregation and Generalisation’’, *ACM Trans. Database Syst. 2*, 2 (June 1977), 105-133.

[18] B. Stroustrup, *The C++ Programming Language (2nd Ed.)*, Addison-Wesley, 1991.

[19] R. V. Zoeller and D. K. Barry, ‘‘Dynamic Self-Configuring Methods for Graphical Presentation of ODBMS Objects’’, *Proceedings Eighth International Conference on Data Engineering*, Phoenix, Arizona, February 1992, 136-143.

APPENDIX—

We give below the definitions of classes `employee`, `department` and `manager`. The definitions are written in the O++ language [2, 4]. Some details are omitted from the code. Comments were added to explain O++ extensions to C++, namely the use of persistent pointers and the set type constructor.

```
class department;

class employee {
    …
    // private members such as birth date and salary
public:
    char logid[25];
    char name[80];
    char nickname[20];
    persistent department *dept; // persistent ptr. to department
    char room[16];
    char ext[10];
    int salary();
    int age_between(int, int);
    int operator==(employee *);
    int operator!=(employee *);
    void *member_list();
    void *display(char **list);
};
```

```
class employee;
class manager;

class department {
public:
    char dno[20];
    char name[250];
    int nemps;
    persistent manager *mgr;       // persistent ptr. to manager
    persistent employee *emps[[]]; // set of pers ptrs. to employees
    int operator==(department *);
    int operator!=(department *);
    void *member_list();
    void *display(char **list);
};
```

```
class department;

class manager {
    …
    // private members such as birth date and salary
public:
    char logid[25];
    char name[80];
    char nickname[20];
    persistent department  *dept; // persistent ptr. to department
    char room[16];
    char ext[10];
    int salary();
    int age_between(int, int);
    int operator==(manager *);
    int operator!=(manager *);
    void *member_list();
    void *display(char **list);
};
```