Supporting Display Generation for Complex Database Objects*

Belinda B. Flynn and David Maier
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology†

1 Introduction and Related Work

Many database user interfaces are favoring a style in which the main mode of interaction is editing and browsing data objects–the interface produces the effect of directly affecting the data as if they were concrete objects. Several visual database interfaces have adopted this style to enable users to access databases more effectively without extensive training or knowledge of the database schema. Domain-specific database applications most likely would receive similar benefits by having such interfaces. The overall objective of this research is to develop support for creating such interfaces in applications that deal with complex database objects.

Presently, database applications that support visual interfaces on complex objects do so without much help from the DBMS. With record-oriented data models, the responsibility for displaying structured objects is typically associated with the application because the data is modeled using a "flat" generalized structure (e.g., a relation). Thus, the application must impose the connectivity or "object structure" on the data. Since the application realizes structured objects from data records, it is best suited to create and manage the object displays for the user interface. The application therefore performs two transformations: 1) between records and structured objects, and 2) between data objects and their external representation, i.e., the displays. In addition, the application most likely will be responsible for maintaining integrity constraints concerning object structure.

Object-oriented databases (OODBs) can directly model data as structured objects so that object construction need not be managed in applications and constraints on object structure can be associated with object classes. Since object structure is imposed by the database rather than the application, complex objects can be displayed directly, without any intervention by the application. Object displays may then have direct access to the database objects, and the application program only needs to specify how and when displays are created, not how displays operate. Thus, our approach is to move display management from the application to a display system that generates and manages the interactive displays.

This arrangement can improve productivity due to the benefits of modularity. Display definitions can be developed more independently of application processing, and once some basic requirements are established, development of the displays and the application program may be done in parallel. In addition, displays are reusable among different applications that operate on the same database. Without modularity, display implementation is tightly coupled with the rest of the application code, and it is difficult to know exactly what parts to extract for display execution and constraint checking among input values. Another benefit is that it is easier to experiment with alternative displays in an application. We have developed the Object Display Definition System (ODDS) to investigate our approach.

There are several existing approaches to providing support for displays of complex database objects. Many database systems include application development tools or 4th Generation Languages that support the creation of form-based user interfaces. Another approach is to integrate a user-interface toolkit or class library with the programming facilities of an object-oriented DBMS, to support a range of user interface styles beyond forms. A related area is display generation tools that produce browsers for object-oriented databases, generating default displays using the structure information kept in the object classes [Deux91]. Another area is work on User Interface Management Systems (UIMSs), whose goal is to support high-level specification of user interfaces, either through a language or a direct manipulation environment. In particular, *data-oriented UIMSs* are those targeted at applications dealing with data that is managed in a central repository, separate from the application. Examples include HIGGENS [Hudson88] and the Serpent UIMS [Bass90].

2 Design Issues

In designing ODDS, we have identified three critical design aspects: 1) the database model, 2) the amount of behavior a display definition captures, and 3) the responsiveness of display format to underlying object structure. This section discusses alternatives for each aspect and which alternative we feel is necessary to meet the system's goals.

*This material is based upon work supported by the National Science Foundation under Grant No. IRI-8805564.

†Address: 19600 NW Von Neumann Drive, Beaverton, OR 97006-1999. Email: (belinda, maier)@cse.ogi.edu

2.1 Database Model

Although all OODBs support complex objects, they differ in their support for defining object behavior. *Semantic database models* concentrate on structural abstractions, e.g., aggregation, grouping, and relationships. Other models can express some aspects of object behavior via rules that define *derived* or *active data*. These rules state how changing a particular data object causes values in other objects to change. However, such models often cannot express behavior that involves constraints among object relationships. *Behavioral models* use the concept of abstract data types to associate general behavior with complex object structure. An abstract data type includes an internal representation and a set of operations or *messages* that are understood by objects of that type. Examples of systems using behavioral models are GemStone [Butterworth91] and O2 [Deux91].

The choice of database model affects the capabilities of a display system because it determines what information the DBMS can provide on the objects being displayed. In particular, the display system should know about objects' structure, and needs to be informed of updates to displayed objects. The advantage of interfacing to an OODB with a behavioral model is that all this information can be managed within the database. With the non-behavioral models, this management must be split between the database and the applications that use the data. Thus, user-interface tools that are used with these models receive limited kinds of information from the database on the objects being displayed. Besides complicating the display system, such a partitioning admits the possibility that one application may violate constraints being maintained by another application.

2.2 Scope of Display Descriptions

The second aspect is the extent to which display definitions capture the activities in the display, which affects how well display management can be separated logically from an application. Different tools for building user interfaces capture various levels of display activity. At the lowest level, toolkits provide predefined interaction techniques (e.g., buttons, menus, gauges) and require the designer to combine them, usually using a procedural programming language. Toolkits give no support for logical separation since code that manages displays is interspersed with application code.

UIMSs capture more activity, since they allow the designer to specify compositions of interaction techniques outside of the application code; i.e., via a special-purpose language or environment. Many UIMSs are based on the Seeheim architecture [Green85], which separates the user interface and application such that the interface is concerned only with processing user input and sending an appropriate request to the application. In this architecture, there is no explicit means, such as a data management component, to let the user interface directly access information about the structure of the application's objects. These UIMSs are not adequate for displaying feedback on application objects with complex structure, since their specifications cannot express the mapping of display subcomponents onto object subcomponents. Therefore, the application, rather than the UIMS, typically manages the semantic feedback in the displays.

A third level of support would include the description of display behavior that is based on changes in the underlying objects. Basic display responses include feedback on scalar data values such as strings, numbers, and enumerations. More complex responses involve some interpretation or intermediate processing of the updated values in the underlying objects. For example, a list may be displayed differently based on whether it is empty or non-empty. Another example is bringing up a notifier that signals an exception, such as when a course's enrollment exceeds its classroom's size. Data-oriented UIMSs have addressed this level of support by introducing a data model in which the objects' structure can be defined. Naturally, ODDS should try to achieve this level of support since its objective is to manage displays of complex database objects.

2.3 Support for Display Responsiveness

The third aspect is the system's ability to specify and produce displays whose structure is responsive to changes in underlying objects or to other relevant conditions. Some systems support *static* representation, where displays use a fixed template for all instances of a class, and only the basic data values may change. Some systems allow *variable* representation, where an object can be displayed in various ways based **only** on conditions evaluated when the display is created. *Dynamic* representation calls for displays whose format may change during their lifetime, either to reflect the state of the underlying object, to respond to user requests, or to meet space requirements. A *format change* refers to altering the number or arrangement of display subcomponents, the graphics that connect subcomponents, or the format of any subcomponent.

Dynamic representation is necessary for displaying complex objects because an object's structure is changeable, just as basic values within the object are. An object's structure can be viewed as a set of connections or references to other complex objects, where each is representing a semantic relationship; e.g., a Course object may reference a Classroom object, to capture the "taught in" relationship. Structural changes that create or delete an object's connections to other objects could result respectively in adding or deleting the subcomponents of a display. In addition, replacing one of the object's references can result in changing the format of a display subcomponent. Such a change occurs if the newly referenced object has a different type. For example, the location of a course may be a Classroom object or a Lab object, and a Lab display could include additional information not applicable to a

Classroom.

The description of dynamic representation has rarely been addressed within existing user-interface tools. This aspect of display behavior generally must be specified procedurally, as part of the application program. Data-oriented UIMSs are mostly concerned with how display attributes are dependent on basic values, not how display structure depends on object structure.

2.4 Desired Features

To summarize the previous discussion, the desired features for supporting interactive displays are: a behavioral data model for defining complex objects, support for capturing display behavior that includes complex responses to database changes, and support for dynamic representation. These three features are needed within a display development tool to create displays that produce a sense of directness in manipulating objects. In particular, the features contribute to *reduction of distance* and *direct engagement*. These are two factors identified as being crucial to obtaining a feeling of directness in user interfaces [Hutchins86].

Distance refers to how closely the mechanisms in a user interface accommodate the user's task. *Semantic distance* is reduced through commands that allow the user to accomplish the task in a concise manner, and by providing feedback that lets the user evaluate readily whether the desired goal has been achieved. *Articulatory distance* is bridged by choosing appropriate input techniques and forms of output for representing the concepts and objects of the application domain. Direct engagement is the sense that one is controlling the displayed objects through one's physical actions, and is fostered by displays that exhibit some behavior instead of merely being static output expressions.

Display behaviors that result in direct engagement likely require complex display responses (such as described in Section 2.2), especially if their intent is to reduce semantic distance by making the displays appear to behave in a manner similar to the underlying objects. Thus the ability to define such display responses supports both of the factors. A behavioral model also contributes to direct engagement because it can supply a more complete account of the objects' behavior, including information on changes to an object's structure or relationships. Support for dynamic representation helps towards direct engagement because it implements a specific form of display behavior. It helps reduce articulatory distance when the intent of the output is to reveal object structure, because a display can adjust its format to match changes in an object's connections to other objects.

3 The Display Facility and System Architecture

We discuss some design decisions concerning the form of the display descriptions. First, we choose to have the displays described by declarative specifications, because they provide abstractions that match the design task more closely than procedural specifications, which often require attention to programming details. A possible alternative is to create displays within a direct manipulation environment such as HyperCard. A problem with this approach is that temporal aspects of the display are not easily expressed, making it difficult to describe dynamic representation. Also, the resulting displays are not sharable across different invocations of the environment. Due to our choice of using declarative specification, designing ODDS involves two major parts: 1) the specification framework for describing the desired end-product, i.e., a display, and 2) a runtime system that generates a display from a specification.

Second, we choose to store display specifications in the database along with the objects being displayed, mainly because they will be accessible to, and sharable by, all database applications as well as to ODDS's runtime system. The runtime system requires this access during display execution in order to handle the displays' format changes incrementally, as opposed to generating all possible display formats initially. An alternative is to store the specifications in files; however, that approach duplicates the data management capabilities already present in a DBMS. An added advantage to placing the specifications in the database is that they represent yet another aspect of the data, and they may be examined, modified, and displayed just as other objects are [Anderson86].

In ODDS, a specification object is used to generate displays for objects of a particular class, called its *source class*.[1] We call our specifications *Outlines* because they do not completely describe displays of individual objects. Rather, they are templates or partial descriptions that lack the actual data values that would come from the object being displayed. An Outline basically consists of two kinds of information: 1) a description of the display's graphical image and 2) a behavioral description; i.e., the display's responses to user-input events, including visual feedback and message passing to underlying database objects. Thus, there are two hierarchies of specification classes, the *Layout* and *Interaction* classes, whose instances are used within an Outline and are called Layout and Interaction *specs*. The remainder of this section sketches the runtime system and its relationship to the application program, then briefly explains Outline objects and presents a small example.

3.1 System Architecture

Figure 1 shows the architecture of ODDS's runtime system in relation to the database and the application program. The runtime system is made up of the components shown in ovals. Both the runtime system and the application have read and write access to the database ob-

[1] The specification can also be used to display instances of the subclasses of its source class.

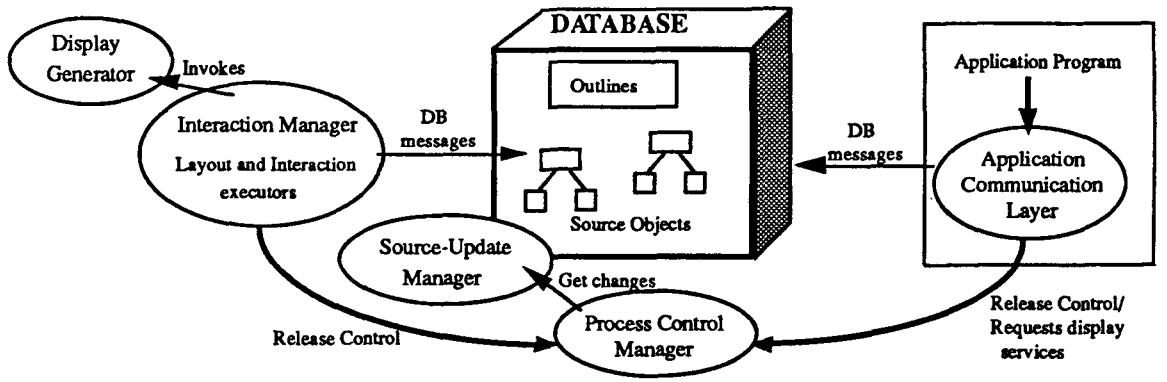

Figure 1: Components in System Architecture

jects; they communicate indirectly through modifications to those objects. With respect to the flow of control, the runtime system and the application program cooperate as coroutines; i.e., each will voluntarily give up control to the other at certain points in its execution. The Process Control Manager handles this exchange of control, hiding the details of process communication from the other components. Thus, the rest of the runtime system is not concerned with whether the application is a local process or on a remote machine, nor with the choice of the application's implementation language.

The runtime system cedes control as directed by a display's specification. The specification framework provides a special object to be used in Outlines for indicating that control should return to the application. The application grants control to the runtime system for several purposes: to create or close displays, to hide or unhide displays, or to resume operation of the displays, thus allowing the user to manipulate database objects or provide input values to the application. The application makes these requests at the Application Communication Layer, which forwards the requests to the Process Control Manager.

When the application requests that a display be created, it supplies the name of an Outline and the database object to be displayed, which we call the *source object.* This information is passed to the Display Generator, which generates a set of objects called *executors* that draw the screen images and carry out the behaviors as defined in the Outline. To construct executors, data from the source object is merged into the Outline's partial descriptions, producing a complete description of the source object's display. The structure and content of the generated executors essentially parallels that of the specs in the Outline. Thus, the classes of executors, called *LayoutExec* and *InteractionExec* classes, have an exact correspondence with the Layout and Interaction specification classes. A key difference between the executors and the specs is that the executors are non-persistent objects known only to the runtime system.

The Interaction Manager is responsible for the overall execution of the displays. Most of its work is performed by the executors. Work that is performed outside of the executors includes processing the events originating from input devices and from database changes, and routing them to the appropriate Interaction executor. During display execution, the Interaction Manager will re-activate the Display Generator when the display format needs to be changed.

The Source-Update Manager monitors the source objects currently being displayed, and produces a list of updated source objects, the particular instance variables that were affected, and the object identifiers for the new values. The Process Control Manager requests this information whenever there is an exchange of control between the runtime system and the application, and passes the update information to the one that has just regained control, so that it may react to the changes.

3.2 Specification Capabilities

An object class may be the source class for several Outlines that display its instances differently. A class may also indicate a default Outline to be used when no specific Outline is requested for displaying an instance. Since an Outline does not contain the actual data values from a source object, it instead contains special "placeholder" objects called *FieldPaths*, which indicate a message or message sequence that will return a particular part of a source object. An Outline may also contain objects called *Deferments* to specify that another Outline will be used to generate a subpart of the display. Rather than having an Outline directly refer to another Outline, Deferments provide more flexibility in indicating which Outline is to be used. A Deferment can either indicate a particular Outline by name, provide a test to choose among several Outlines, or just use the default Outline.

The Layout specs within an Outline specify the display's visual image. Each Layout spec holds a set of attributes that determine graphical details such as foreground and background colors, text fonts, and spacing. At present, the primitive components available for display images are

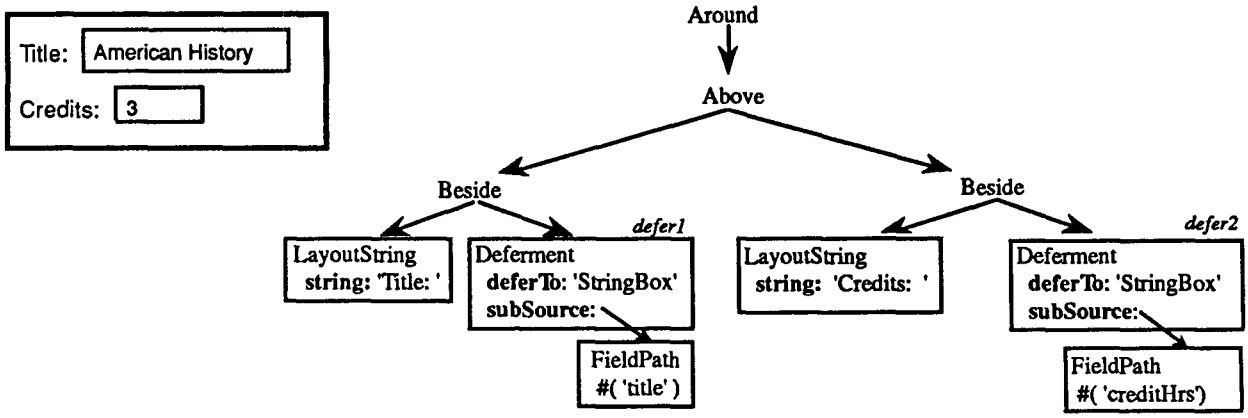

Figure 2: Sample Display and Layout Specification

text strings, bitmaps, lines, and rectangles; different types of Layout specs exist for each of these primitives. A *ComposerLayout* is a Layout spec that indicates a composition of other Layout specs. Specific kinds of ComposerLayouts, such as *Above*, *Beside*, and *Around* specs specify how the display image is spatially composed from its subparts. Finally, a *Correspondence* spec is a Layout spec that describes a visual representation for object relationships. We say that a *domain* object `x` is related to a *range* object `y` if there is some message in `x`'s protocol that when sent to `x` returns `y`. A Correspondence is used to present a mapping from a collection of domain objects to another collection holding potential range objects with respect to a particular message. An example is a mapping from a set of Course objects to a set of Classroom objects, where the domain and range objects are related by the `taught_in` message. Correspondence specs provide an alternative to the customary way of expressing relationships, which is to nest the display of a range object within the display of the domain object.

Here we introduce a small example to provide a more concrete picture of the specs in an Outline. This Outline defines a display for a Course object, showing its title and number of credit hours. (See Figure 2.) Each boxed area containing a data value is to be highlighted by switching the foreground and background colors whenever the cursor enters it. Another behavior is that a mouse click within the Credits subdisplay causes the value to be incremented, modulo some maximum allowable number of hours.

Figure 2 shows the basic structure for the Outline's Layout spec. (Layout attributes have been omitted for simplicity.) The arrows in the diagram indicate the subcomponents for a ComposerLayout. The two subdisplays holding the data values are generated from a separate Outline (named StringBox), as specified by the Deferments. Deferments are defined with two parameters: `deferTo` holds the name of the Outline used to generate the subdisplay and `subSource` holds a FieldPath indicating the object subpart to be displayed.

Next we discuss the types of Interaction specs. An Interaction spec can reference Layout specs and specifies what changes will take place in the executor counterparts for those Layout specs, thus defining how the display images change. In general, display behavior is expressed in terms of *events* and *event responses*. Examples of events are the arrival of data through input devices, the occurrence of a certain condition or state in an Interaction executor, or an update to some source object. An event response describes the action(s) that will take place whenever a certain event occurs. Types of actions described in an Interaction spec include:[2] 1) an update to a Layout executor, 2) an update to its Interaction executor counterpart, which affects the future behavior of the display, or 3) the generation of an *internal event* to be forwarded to another Interaction executor.

The Interaction subclasses provide a framework in which to structure or organize the collection of events and event responses that make up a display's behavior. Each Interaction spec holds a mapping from expected event types to the responses for events of a given type. There are three main Interaction subclasses:

- *ImageOp* specs group together the events and responses of interaction techniques such as menu buttons or gauges. Their responses mainly describe updates to Layout executors.
- *DBConnect* specs cover behavior that reflects the activity in the database source objects, as well as behavior where the display initiates operations on the database objects. A DBConnect holds a FieldPath that defines the part of the source object it is responsible for. A response action in a DBConnect may be a *MessageAction* spec, which holds the name of database message to send, plus the location of any arguments required.
- *Coordinator* specs group the events that define the communication and data transfer among several Im-

[2] We do not describe specification objects for event responses in detail due to space limitations. This information is available in an extended version of this paper [Flynn92].

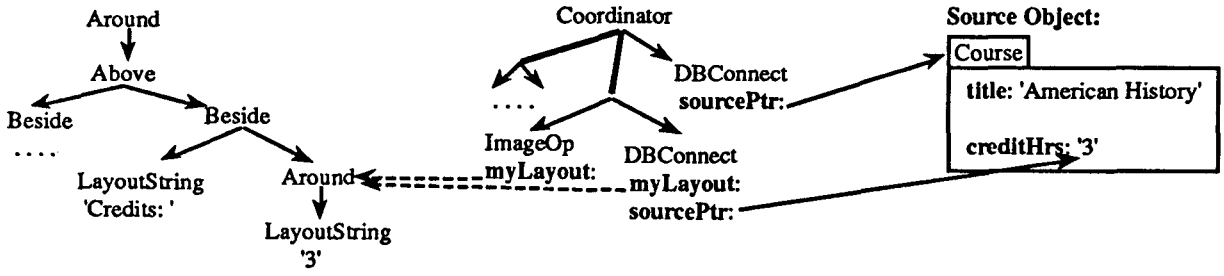

Figure 3: Source Object and Runtime Objects for Course Display

ageOps or DBConnects. Thus, Coordinators can describe the behavior of a complex interaction technique as the composition of more basic techniques; e.g., a panel of radio buttons. A Coordinator holds a collection of *sub-Interaction* specs whose executors will be communicating at runtime. The event mapping of a Coordinator defines which sub-Interaction executor will receive internal events of a particular type.

In the example, the behavioral specification for the StringBox Outline consists of two parts:

- An ImageOp specifies the highlighting behavior and the response for a mouse click through three associations in the event mapping:

  EventType('enterLayout') – reverseColors
  EventType('leaveLayout') – reverseColors
  EventType('buttonClick') –
  create InternalEvent('selected')

- A DBConnect defines the connection between the LayoutString spec and the source object being displayed. Its event mapping specifies that when an arriving event indicates a change in the source object, the new value (carried in the event) is placed into the `string` field of the LayoutString executor.

The behavior portion for the Course Outline consists of a Coordinator whose sub-Interaction specs are *defer1*, *defer2* (in Figure 2), and a DBConnect that holds a reference to the Course object. The behavior to specify is that a mouse click in the Credits subdisplay increments the value shown there. The Coordinator's event mapping states that at runtime, the 'selected' internal event from that subdisplay will be forwarded to the DBConnect executor. The DBConnect spec specifies that the response to the internal event will be to send a database message that updates the number of credit hours in the Course object.

To support dynamic representation, *FormatOption* specs describe how a display's subcomponents may be rearranged during display execution. A FormatOption may have only one arrangement of Layout objects, or it may hold several possibilities and define a selection condition to choose among them. If a FormatOption contains different format descriptions that share subcomponents, the runtime system will reuse executor counterparts for the shared subcomponents, instead of creating new executors whenever a format change occurs. FormatOptions are used as response actions within an ImageOp or DBConnect object. A FormatOption may also be placed at the top level of an Outline, to specify the display's initial format. The concept of interpreting specifications to execute format changes is based on an earlier system, the Smalltalk Interaction Generator [Maier86].

Figure 3 is a sketch of the executors generated from the Course outline, which execute the display in Figure 2. In the generation process, the Deferments in the Course Outline are replaced with executors generated from the StringBox Outline. FieldPaths in the DBConnect specs are replaced with references to the source objects, which are needed to send database messages to those source objects.

4 Summary and Current Implementation

To summarize, we have designed the specification framework and runtime architecture of ODDS to meet the desired features set forth in Section 2. ODDS provides mechanisms for interacting with a behavioral database model to manipulate source objects and to obtain information for semantic feedback. The FieldPath, DBConnect, Coordinator, and Correspondence specs are particularly useful in specifying complex display responses to database changes. Dynamic representation is supported through the FormatOption specs and the execution of format changes by the Display Generator.

Our prototype of ODDS uses the GemStone DBMS, which has a behavioral database model similar to the object model in Smalltalk. The runtime system is implemented using Smalltalk, with the exception of the Source-Update Manager, which is implemented as a GemStone class. The work needed to complete the implementation mostly involves the execution of format changes. We have devised a sample set of displays which will be specified and generated through ODDS, so that we may evaluate the expressiveness of the specification classes as well as the performance and resource usage of the runtime system. We also plan to examine the extent to which Outlines are reused through Deferments or by borrowing Layout and Interaction specs from existing Outlines.

References

[Anderson86] T.L. Anderson, E.F. Ecklund, Jr., and D. Maier, "PROTEUS: Objectifying the DBMS User Interface", *Proceedings of the International Workshop on Object-Oriented Database Systems*, ed. D. Dittrich and U. Dayal, Pacific Grove, California, September, 1986.

[Bass90] L. Bass, E. Hardy, R. Little, and R. Seacord, "Incremental Development of User Interfaces", *Engineering the Human-Computer Interface*, A. Cockton, ed., North Holland, 1990.

[Butterworth91] P. Butterworth, A. Otis, and J. Stein, "The GemStone Object Database Management System", *Communications of the ACM*, October 1991.

[Deux91] O. Deux et. al., "The O_2 System", *Communications of the ACM*, October 1991.

[Flynn92] B. Flynn and D. Maier, "Specification and Generation of Displays for Complex Database Objects", OGI Technical Report, January 1992.

[Green85] M. Green, "Design Notations and User Interface Management Systems", *User Interface Management Systems*, Eurographics, 1985.

[Hutchins86] E. Hutchins, J. Hollan, D. Norman, "Direct Manipulation Interfaces", *User Centered System Design*, ed. D. Norman and S. Draper, Lawrence Erlbaum Associates, Inc., 1986.

[Hudson88] S. Hudson and R. King, "Semantic Feedback in the Higgens UIMS", *IEEE Transactions of Software Engineering*, August 1988.

[Maier86] D. Maier, P. Nordquist and M. Grossman, "Displaying Database Objects", *First International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.