A User Interface Management Approach for Object-Oriented Database Applications

Lin Padgham and Jonas Löwgren
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

Manuscript submitted for *Journal of Systems and Software*

Abstract

This paper discusses the development and management of object-oriented database applications using an approach where the user interface management is abstracted out from the underlying database, but nevertheless encapsulated in general purpose software — a user interface management module. We identify a list of requirements that this application domain introduces and then present the LINCKS object-oriented database system and the UIMS which was designed to meet these requirements. After introducing the architecture of the system, we illustrate by means of an example how an application is built and executed. We finally discuss our approach and identify several features stemming from the specialization of the UIMS to the area of object-oriented databases. These features include multiple views of application objects and encapsulated runtime behavior on the task level. Our conclusion is that the design of specialized UIMS for different application domains is a valuable alternative to the maximally general UIMS approach.

1 Introduction

When designing a User Interface Management System (UIMS) for a class of applications, it is important to understand the requirements that the class in question poses for the logical model of the user interface. We believe that the design and mapping of different logical models to different classes of applications and interfaces is a central task for UIMS researchers. Earlier examples of this kind of work include the Seeheim model for dialogue-oriented applications with relatively static data (Green, 1985), the Seattle model for direct manipulation interfaces with semantic feedback (Hudson, 1987) and its improvement for managing also overall control in the interface (Edmonds and Hagiwara, 1990). Along similar lines, Coutaz and Balbo (1991) present a general set of dimensions to characterize classes of applications and to inform the development of UIMS technology.

In this paper, we focus on user interface management for an object-oriented database system. We identify the requirements introduced by the application class and describe the architecture of our system, including an object oriented database (OODB) and a UIMS.

We illustrate by means of an example how this system supports the construction and execution of user interfaces to database applications. To conclude the paper, we discuss the logical model of the UIMS and compare it with some similar systems, highlighting the differences and special features of our approach.

The system which we describe is a system for supporting database applications where the information to be stored in the database is of a relatively complex nature and is more suited to an object-oriented representation[1] than to a relational database representation. It is expected that the objects in the database are highly overlapping, with many links to related information, causing problems in extracting the required information chunks for a particular composite object. Other characteristics of object-oriented databases with particular implications for user interface management include (Bancilhon and Kim, 1990) multimedia objects and the notion of active objects, as opposed to the static containers of a relational database.

These characteristics of our application domain imply certain requirements on the techniques used to manage the user interface, as follows.

- In order to manage the presentation of active objects, *asynchronous control* is necessary. Even though our current database system does not employ active objects, we consider this requirement important for the object-oriented database domain in general.

- The nature of the information stored in the database in generally complex. To achieve acceptable runtime performance in a direct-manipulation interface to such a database, it must be possible to represent some aspects of the *applications semantics*, i.e., the meanings of the interface components in application terms, in the user interface specification.

- Since the applications built with an object-oriented database must be capable of presenting intertwined and overlapping information, the management of *multiple views* in the user interface is necessary.

- Finally, since object-oriented database applications generally address information browsing and updating, there are good opportunities to *encapsulate interactive behavior* in the user interface manager.

We have now identified a list of requirements that object-oriented databases impose on user interface management. The rest of the paper describes a system, LINCKS, which includes an object-oriented database and a user interface manager which was built from these requirements.

2 Architecture

The architecture of the LINCKS system can be seen logically as consisting of three layers: the database, NODE; LOOCS (LOcal Output/input Communication Server) which inter-

[1] Object-oriented for us does not include the message passing paradigm or the notion of methods attached directly to types. The key aspect is the building of composite objects from parts.

prets information in the database and builds data structures which represent particular views of database information; and the user interface manager, UIM, which builds the display objects suitable for a particular display environment.

The UIM is of course dependent on the underlying environment (i.e. the window manager in a windowing environment, otherwise the operating system) and a new UIM module must be built for each separate environment. Our current UIM runs under X Windows and we are in the process of implementing a UIM for a text terminal milieu under Unix. The NODE and LOOCS modules are however relatively independent of the particular hardware or underlying operating system.[2] The current implementation runs under Unix on Sun Sparcstations. It is written in C and the executable code takes up 1.1 Mbytes of memory.

All user input is via the UIM which then sends commands to LOOCS. LOOCS in its turn operates on NODE. Following NODE operations LOOCS updates all of its data structures which are affected by the NODE updates and notifies the UIM to regenerate the necessary display structures. Thus when a change is made to a particular information chunk then the change is immediately propagated to all visible objects containing that chunk. It is intended that applications will also be able to affect the database and thus the user interface via calls to LOOCS.

We explore in a little more detail the different modules.

2.1 NODE

NODE is an object oriented (or object centred) database designed and built for the purpose of supporting applications which require representation of more complex objects than those found in relational database applications. It is also designed to support applications requiring AI reasoning capabilities, and for this reason manages various kinds of temporal information and versioning capabilities.

The historical information maintained in NODE includes the development histories found in systems such as ObjectStore (Lamb et al., 1991), Iris (Beech and Mahbod, 1988), Versant, Objectivity and Ontos (Cattell, 1991). In addition, NODE contains two somewhat different kinds of temporal information which we call temporal history and command history. Temporal history maintains approximately the information which can be found in time stamps in some other systems, e.g., Orion (Kim et al., 1989), but in a way which is logically more correct, while the command history maintains a history (partial ordering) of actions and the objects affected by those actions. Further information on the temporal aspects of NODE can be found in (Padgham, 1988) and (Hall, 1992).

The basic data structure for representing an information chunk at the database level consists of three sections — attributes, links and image. *Attributes* consist of tags and a value which is stored along with its size. The default interpretation of attribute values is that they are strings but an application can also store other data structures as attribute values. Each *link* field consists of a tag and an ordered list of pointers to other information chunks. Pointers to other information chunks may be either *bound* or *unbound*. An unbound link refers to an information chunk in its most recent stage of development,

[2] Change to a different operating system may require minor changes to communication protocols used.

whereas a bound link refers to an information chunk at a particular time. The *image* is unstructured information intended primarily for direct use by a user. As with attribute values the default interpretation of an image is as a string, but it may also be interpreted as graphics or sound. An application can easily specify other types of image data.

An information chunk (together with its development over time) is referred to as an atomic object. Composite objects are built up by linking together a structure of atomic objects (e.g. a document is a structure of sections, subsections and paragraphs). Atomic objects often participate in many different composite objects leading to a database which is highly interconnected with many overlapping substructures.

NODE consists of three communicating processes which may reside on a remote machine, plus a library of functions which establish communication with the remote machine and retrieve and store information chunks. All processing of information is done by the library functions on the local machine. Information chunks once retrieved are kept locally as long as space allows, allowing minimization of communication costs. The NODE function library is compiled together with the LOOCS program.

We refer the reader who is interested in further details regarding NODE to (Padgham, 1988) and (LINCKS Manual, 1988).

2.2 LOOCS

One of the key functions of our database interface management system is to maintain the display of the interface objects dynamically as they change. Because the objects to be presented are data, rather than simply containers, it is not possible to make a complete separation between presentation information and database structure. The presentation information does of necessity contain information regarding the structure of the data in the database as well as information regarding the presentation of the information at the user interface. This reduced separation between user interface and application objects in the database appears necessary in order to achieve the desired flexibility and power.

The task of LOOCS is to mediate between the database module and the display dependent UIM module. LOOCS interprets templates known as *presentation descriptors*, resolving access patterns to obtain object parts relative to the given root node. LOOCS then builds a structure which we call the *reference structure* which represents an intermediate step between the database structures and the display structures, and accepts editing commands from the display which are transmitted to the database.

LOOCS has 3 main tasks:

- interpretation of presentation descriptors in order to retrieve information from the database and build a data structure suitable for use in generating the display structure. This data structure must also contain sufficient information to map user interface actions to the appropriate NODE actions;

- interfacing between UIM and NODE — user interface actions are passed to LOOCS and translated into NODE actions. It is always via the UIM-LOOCS chain that a user operates on the database;

- propagation of changes: an item changed at the user interface may represent a database information chunk which has several instantiations on the screen (in different composite objects, or in multiple occurrences of the same object). LOOCS notices which of its data structures are affected by database changes, updates its own data structures and requests the UIM to rebuild display structures where necessary.

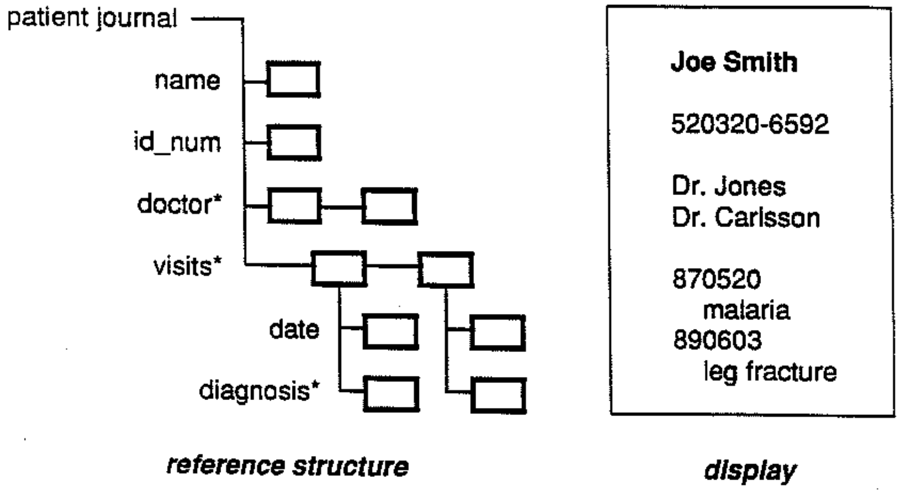

Figure 1: An example reference structure and corresponding display.

Reference Structures. It is the interpretation of presentation descriptors to build the LOOCS data structure that is the core of LOOCS. The primary data structure which is built is called a reference structure and represents the complex object, built from database parts, which is to be presented at the UIM level. The reference structure is an ordered tree with named branches, as shown in Figure 2.2. This structure reflects the internal structure of a complex object. The leaves are used to build the display structure for the UIM.

LOOCS also generates a reference structure template which gives information as to how the reference structure (and also in turn the database), should be built up as new information chunks are added via the UIM.

Presentation Descriptors. A presentation descriptor contains two key groups of information — Structure and Access. Structure is a description of the conceptual structure of the type of object to be viewed, and maps directly to the reference structure template. Access is a pattern which indicates where in the data base, relative to a root node, the various information pieces are to be found. Each element in a structure definition must have access information defined. A presentation descriptor can also contain information such as formatting attributes, target device (for multimedia applications which may, for example, send data to a speech generator), and which presentation descriptors to use in expansion of object parts.

Presentation descriptors can be linked together so that partial definitions can be reused. This is accomplished by using a link to a new presentation descriptor as the definition of a part of the structure.

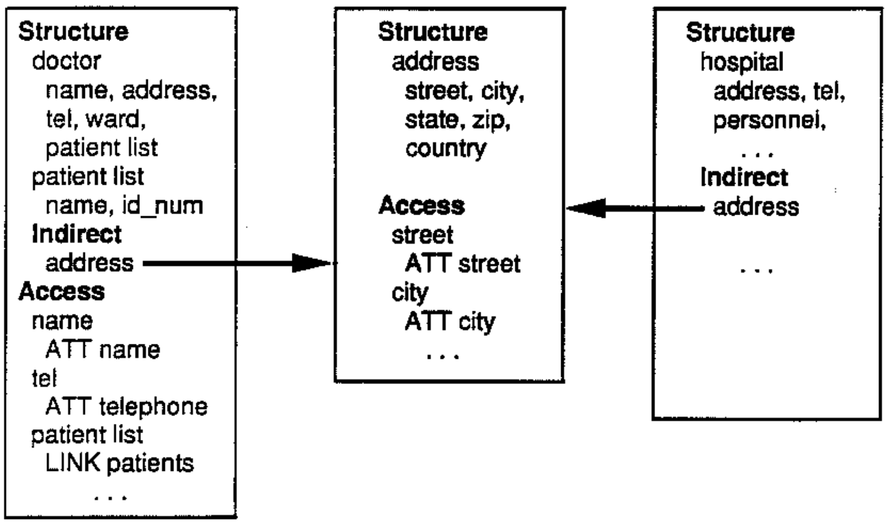

Figure 2: Combination of presentation descriptors. ATT = attribute, LINK = link to other chunks.

When such a link occurs in the structure definition the second presentation descriptor is used for resolving access and all other information, as well as for developing the structure description. A link to another presentation descriptor can also occur within the Access information, but in this case only the Access is resolved relative to the second presentation descriptor. All other information is resolved relative to the initial descriptor.

Figure 2.2 shows two presentation descriptors each of which refer to a third descriptor for a common part. Instead of address being broken down into its parts under the attribute section of Structure, there is a link (under the link group Indirect with tag address) which points to a presentation descriptor where the appropriate breakdown is made. This system of indirection facilitates modularity and reuse.

The Access section in the presentation descriptors in figure 2.2 shows that certain parts (here name, tel, street and city) can be found as attributes (indicated by the reserved word ATT) under the attribute tag following the word ATT (name, telephone, street, city). Other parts (patient list in this example) can be found by following a link — indicated by the reserved word LINK — with the indicated name (in this case patients).

Usually a presentation descriptor is specified in order to apply to a whole class of objects. For instance one may have a patient descriptor, an index descriptor, a journal descriptor, etc. Formatting information which applies just to an individual document rather than to the whole class is given in a *specific presentation descriptor* attached to the root node of the particular database object.

A presentation descriptor is an object in the database and can itself be viewed (and edited) using a presentation descriptor appropriate for presentation descriptors.

Choosing a Presentation Descriptor. When LOOCS is asked — by the UIM, NODE, or an application — to display a particular object, then a presentation descriptor is chosen for the object and is interpreted by LOOCS in order to request the necessary information chunks from NODE.

The name of the presentation descriptor to be used can be found in three different places, as described below, allowing for both specific and default declarations. This name is then used to search for the actual descriptor in a dynamically maintained list of presentation descriptors. The most specific, or explicit place to look for the name of the presentation descriptor is in an attribute of the object itself. The most general possibility is to use the type name of the desired object as the name of the presentation descriptor to be used. Most often however a context dependent default is used. The name of the presentation descriptor is given as a default for the object type within the context from which the desired object is accessed. For instance if a file is accessed from an index, then the presentation descriptor for that index may contain the information "expand file as filecontents", where file is a name of part of the structure of the index, and filecontents is the name of a presentation descriptor which can be used for expanding, or viewing separately, this part. These mechanisms allow for flexible management of presentation of objects. For instance a particular sort of item such as a journal page, within a composite object such as a patient journal, can be declared as viewable via a particular sort of descriptor, let us say, text. However if one of the items, perhaps a result of some lab-test, is better displayed with another descriptor, say graph, then that can be declared as an attribute within the deviating object.

Dynamic context (i.e. what the user has been doing) can also affect the choice of descriptor as the list of presentation descriptors in which the system searches is built dynamically, and can differ depending on such things as context and user. The system has a base list of presentation descriptors which are always available. User or context specific descriptor lists are then added to the head of the base list, allowing them to be searched first. Thus a user defined presentation descriptor will always override a system defined descriptor of the same name.

2.3 UIM

The UIM module manages all input and output to and from the screen, i.e., the interaction with the user. An important part of the UIM is to take a LOOCS structure and build a two dimensional display structure. Editing operations on this structure are then recognized by the UIM and where necessary result in calls to LOOCS.

The particular display structures and display objects built of course depend on the underlying hardware and software being used. Our current version of the UIM is based on the X Window system, but we are also working on a UIM based on Unix on a text terminal. An important task of the UIM is to manage cursor movement and editing operations on the displayed objects. In order to accomplish this the UIM must maintain sufficient information in the display structure to be able to map back to the items of LOOCS, which must in their turn be mapped back to database entries. At the same time we want to avoid imposing on the user the need to deal explicitly with the structure of composite objects at all times. For instance a patient journal is displayed and behaves as a single object just as in a traditional screen oriented editor, even though the underlying structure is made up of name, address, treating doctor, date, symptoms, diagnosis, comments, etc. This requires that the display structure managed by the UIM be more complex than that available in most text editors, as the system must recognize individual parts of the structure without being told explicitly by the user.

The editor which is integrated with the UIM provides a sufficiently powerful interactive base for many applications. If an application wants more specialized interaction than is available through the system display objects, then the application programmer can add specializations of the basic display objects to the existing UIM in order to obtain increased interactive functionality.

2.4 UIM/LOOCS interface

There is a well defined interface through which the UIM and LOOCS communicate, allowing the user to modify the database, and allowing changes to the database to be propagated back up to the user interface. This interface is also used by applications, both in order to request modifications to the database and to receive input from the user interface.

The calls to the UIM enable creation, deletion and change of display objects. Calls to LOOCS allow changes to the contents of database objects, linking between objects, removing of objects, creation and linking in of new objects in the database, and access of database objects.

3 Developing a User Interface

We now look at how this system can be used by an application designer of a particular application. The two most important tasks of the application designer are:

- to identify what sort of objects are to be available in the given application including how they appear to the user; and
- to design the representation of the data in the database, with all relevant interconnecting links.

When these design tasks are done the implementation is simply a matter of writing the presentation descriptors for the desired object types, using a simple special purpose specification language. This is done via the standard system interface of the database editor. For each object type the structure definition is given first, reflecting the design of the user-visible object. The access information for each element in the structure is then also entered, reflecting the design regarding the arrangement of the data in the database. Here the reserved words LINK and ATT are used to indicate that the object part can be found by following a link, or looking in an attribute, respectively. Formatting information and information regarding how to expand parts of the object can also be given if desired.

The actual data objects of the application can then be built up interactively by instantiating various types of objects. Alternatively, if there is a large amount of data with a regular structure which already exists as text files then it is simpler to write a small program which parses this data and sets up the appropriate database objects. We have done this successfully for a number of fairly large databases.

A powerful aspect of the system is the dynamic quality of views. The decision as to which view is to be used is made at run-time based on the use of a symbolic name which is searched for amongst the list of currently available presentation descriptors. The designer can make different presentation descriptors for different roles and contexts within the entire application. In this way the same underlying data objects are viewed differently depending on the role of the user and the context he/she has entered.

For instance, an object representing an overview of the system may well be seen differently by a hospital administrator and a doctor. The doctor, for example, probably wants to see the access to the medical information database, while an administrator is probably not interested in this but wants easy access to other hospitals. An individual doctor (or hospital administrator) can modify their own personal system overview if they desire something other than that developed by the application designer.

The system contains a number of different presentation descriptors which can be used by the application designer either as templates which are then modified, or as ready-made descriptors. A basic system-defined presentation descriptor can remain in the basic system context but also be linked to from the application using a different, more application appropriate name.

The base system provides functionality for editing and browsing in the database in an advanced manner. If however the application requires specific functionality outside of that already provided, then this requires some additional work. Extra functionality which does not require additional kinds of interaction than that already provided is relatively easily added. Menu choices indicating the additional functionality are added to an existing menu (or displayed on a separate menu) allowing entrance to the new functionality. Following the computations of the new function, the results are displayed by having the application create or modify (via LOOCS) a database object and then send a request to the LOOCS module to display this object.

An example of such an added function would be to calculate the number of patients currently admitted in a particular ward. The menu choice would be added allowing the user to select this option and pass the required parameter (the ward object). The application routine would perform the necessary calculations and request LOOCS to store the result in a new or existing DB object. The function may then request LOOCS to display the resulting DB object if it is not already visible.

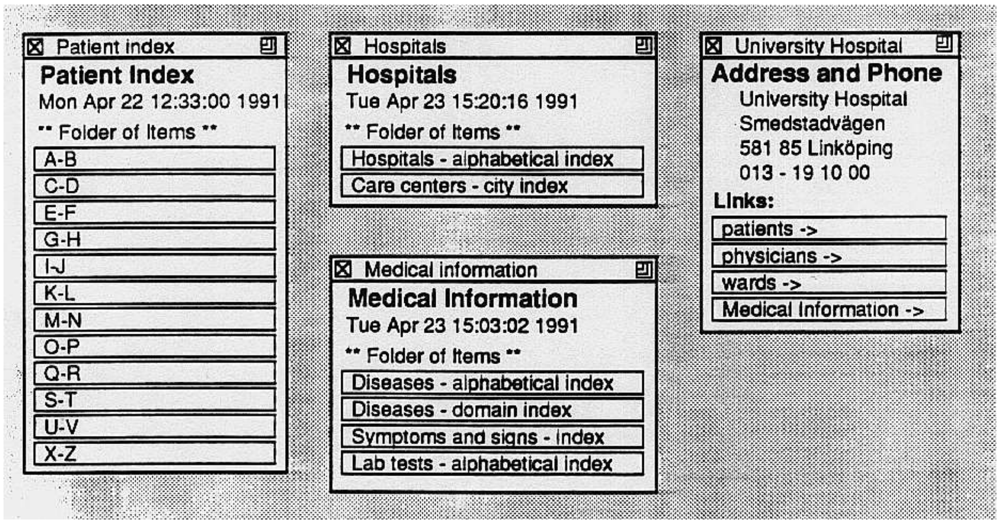

Figure 3: The initial view of our prototype application displays different indexes to the information in the database.

If the added functionality requires a new kind of interaction, for example specialized graphics, then the addition is more complex. A new kind of display widget must be built (using in our case X Windows) supporting the desired functionality. A module must also be added to the UIM which builds the new type of widget from the reference structure. The device field of the presentation descriptor is used to indicate which particular type of widget (or more generally, display object) is to be used in building the display.

4 Executing the User Interface

The predominant paradigm for the user interface of an application produced with our system is the editor paradigm. Everything that can be seen on the screen can be edited directly (unless it has write protections), using standard screen oriented editing techniques. Objects are seen in an integrated manner at the same time as the system maintains the underlying structure of composite objects.

The insertion of things such as a new paragraph in a string of paragraphs in a patient journal is done by a simple command (e.g. double newline). The generation of the appropriate database objects for storing this information is done by the system and is invisible to the user.

New objects can also be instantiated by inserting a link to the new object within some other object, or by expanding (double click) on a place marker for the object. The instantiation and linking in of the relevant database node(s) is again invisible to the user.

The hypertext paradigm is also an important part of the user interface. Navigation in the system is primarily via following links or expanding objects. Depending on the types of objects and the design of the application it may be preferable to think of link following or expansion. If we go from an overview of a patient journal to a full text view of a particular visit, it may well be intuitive to think of expanding the denotation of the visit within the overview of the journal. If we go from a patient name in a register it may be easier to think of following a link. The functionality is however the same — a new object is displayed using (possibly) a new presentation descriptor.

In order to more concretely describe and give the feel for the execution of a user interface built with our system, we describe some typical tasks in a prototype application which we have built using the system. All functionality in this prototype comes from the base system — that is, no extra functionality has been added for internal processing or for interface.

The prototype application chosen is a county medical system.[3] The initial view of the application (see Figure 3) is a display of various indexes over the different entities. There are, for example, indexes over hospitals, community health centers, personnel, patients, medical texts and lab facilities.

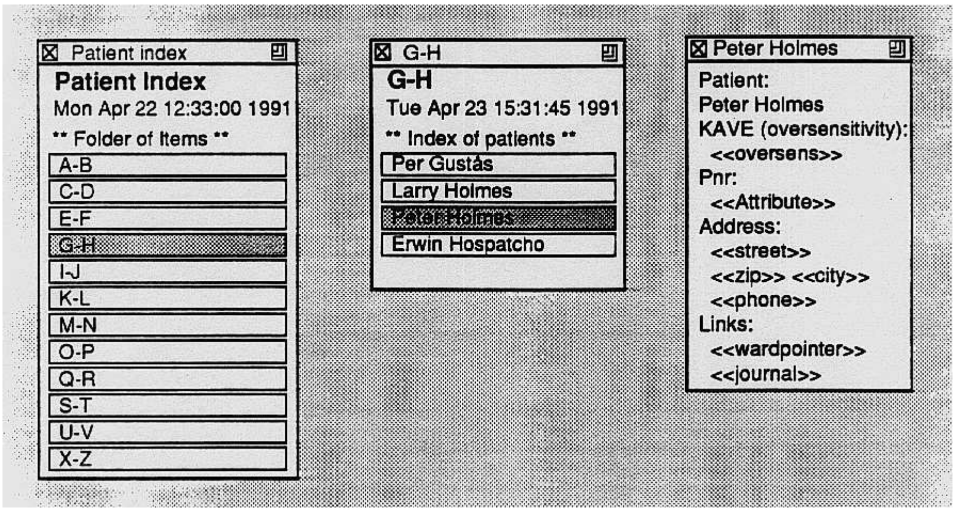

Figure 4: A new patient, Peter Holmes, has been created and the expanded view of the information associated with him opened. No specific information has as yet been entered in the expanded view.

If I now have a new patient I can instantiate him/her within the system simply by clicking on an existing patient, giving an insert command and then typing in the patient name. If there is no patient already in the index then a placeholder is displayed as <patient> and I can click on it and insert.

If I want to add further information about the patient I simply double-click on the

[3]Figures 3 through 4 were redrawn to improve legibility. They are, however, conceptually identical to the screen snapshots taken from the prototype application.

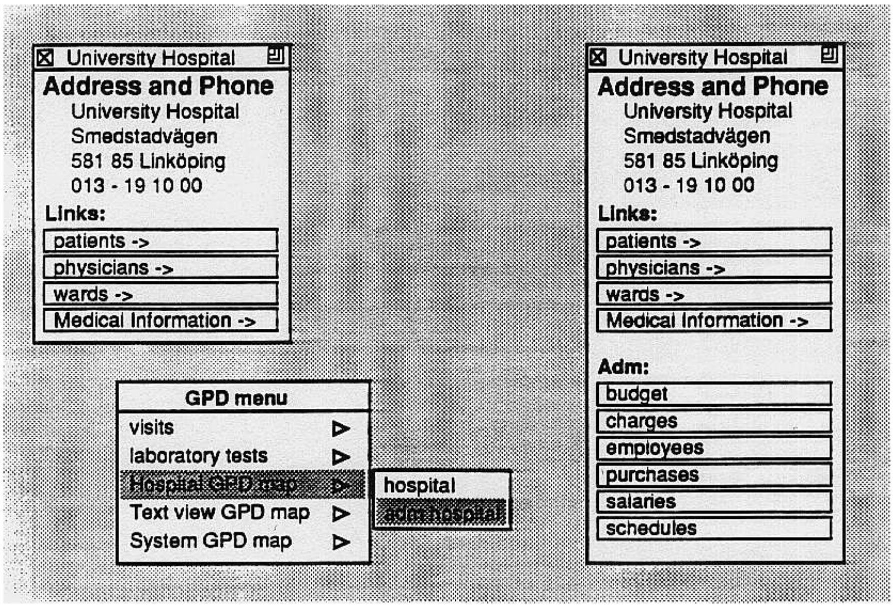

Figure 5: Using the menu of presentation descriptors, the user can have the hospital object presented from an administrative point of view (right) instead of the normal presentation (left).

name to obtain an expanded view of a patient. This contains such information as address, telephone number, id-number, link to patient journal, etc. I can then fill in desired fields and instantiate objects linked to, such as the patient journal. Figure 4 illustrates the creation of a new patient and the expanded view prior to any specific information being entered.

The patient name is one piece of information here, the piece that is used in the condensed view of the patient within the index. If desired I can wait until expanding the patient view before inserting the name. In this case it will appear in the index simply as "patient" until I have added it in the expanded view. As soon as it is added in the expanded view the information is propagated to the index and "patient" is replaced by the actual name. All information which is shared by several objects (or even two copies of the same object) is automatically propagated to all occurrences after editing. Thus all information on the screen is always current.

When I look at the view of the patient it is possible that I decide that some other information field (e.g., id number) is more suitable for display in the index. This is relatively easy for the user to change. The user simply clicks on the index object and chooses the option "show presentation descriptor" from a menu. The presentation descriptor is then displayed and can be edited. Presumably the structure of the index is defined as a list of patients. The access for patient is then given as ATT name. Editing this access information to be ATT idnum changes the presentation for the index. Next time we display the index (which can be done by a menu choice) we see a display of patient numbers instead of patient names.

In order to successfully accomplish this editing we must know the database name of the attribute for id-number. In that it is displayed within the full patient view this is easily accomplished by looking at the presentation descriptor for the patient. Alternatively if a *node presentation descriptor*[4] is included in the application, then the user can use it to look directly at the database representation of the patient node.

In a multi-user system such as our medical information system it is likely that there is a standard interface which may not be changed except by the system administrator. In this case the individual user may still obtain a customized view of the index file as described above. The only difference is that he/she must first insert a copy of the system presentation descriptor into a list of personal presentation descriptors (using the same name). The changes are then made on this personal copy. In that the personal list of presentation descriptors is always used first, the individual change overrides the system default.

In a similar way it is possible to modify such things as the initial display, so that for example a nurse in a particular ward sees initially an expanded view of his/her own ward, and obtains the other information only on demand.

Sometimes a user may want to temporarily use a different presentation descriptor for viewing an object. This is achieved by using a menu of available presentation descriptors and showing the object according to the descriptor chosen (see Figure 4). If the user wishes to always have the object shown with the alternative descriptor then a change must be made either to the object or to the presentation descriptor from which it is expanded. If an "expand newdescriptor" attribute is added to the object (or the expand attribute is modified) then this particular object will in future be displayed with the newdescriptor presentation descriptor. If this expand attribute is instead added in the presentation descriptor for the parent object[5] then all objects of this type will be shown using newdescriptor.

5 Discussion

We have now seen how a UIMS for object-oriented databases can be designed and used to support construction and execution of user interfaces. This concluding section discusses our system in terms of the logical model employed and indicates some important features. We compare our approach with some other systems with similar aims.

5.1 Logical model

The logical model of our approach is shown in Figure 5. It can be described as a specific application of the general Seattle model (Hudson, 1987) for the management of direct manipulation user interfaces, where UIM and LOOCS correspond to the presentation and

[4]A node presentation descriptor shows the current version of an atomic object, exactly as it exists in the database.

[5]The parent object is the object from which the current object is expanded or followed.

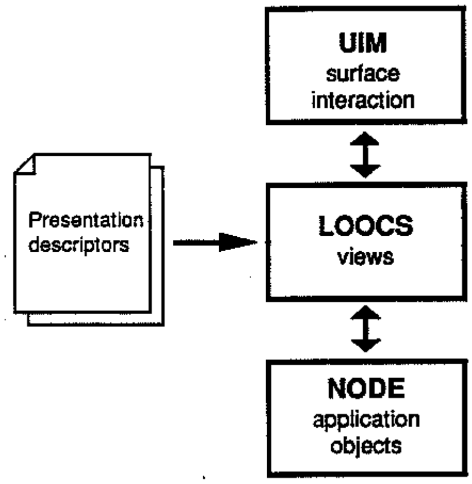

Figure 6: The logical model.

semantic support modules, respectively. We can also note that the separation between user interface and application considerations is not complete: the presentation descriptors in the LOOCS module contain information regarding the structure of the application objects. This is a result of addressing the domain of object-oriented databases, where the object structures may be complex, overlapping and highly interconnected. We regard this inclusion of application semantics in the user interface specification not as a failure of the UIMS approach but rather as a constructive contribution to the understanding of the requirements posed by our domain of application.

The Humanoid system (Szekely, 1990), which manages user interfaces to knowledge bases, uses the same type of logical model. A difference is that the presentation level is considered solely responsible for managing the runtime behavior (see section 5.3 below). Moreover, the Humanoid system cannot deal with arbitrary overlapping application objects.

5.2 Multiple presentation descriptors

Figure 5 also shows that the LOOCS module may be given a set of presentation descriptors. These are all available when the application is executed and the choice of which one to use for the presentation of a particular object is made when the object is to be presented. Consequently, the views onto the database can be changed dynamically by requesting a different presentation descriptor. This is an important contribution to the flexibility of the runtime user interface management.

5.3 Encapsulated behavior

In existing UIMS research, the aim is often to provide encapsulated behavior on the surface interaction level. A prominent example is the Garnet system (Myers, 1990) where six different *interactors*, or abstracted interactive behavior descriptions, are claimed sufficient to handle all interaction in the class of user interfaces addressed by the system. These interactors provide behavior strictly on the level of presentation and surface interaction, e.g., selection and positioning. In our case, the decision to build a specialized UIMS for the management of user interfaces to object-oriented database applications has enabled us to provide encapsulated behavior also on the task level. This is manifested in the automatic support for end-user editing and hypertext-style browsing of the database objects in the applications. These tasks are fundamental in object-oriented database applications, and we consider the encapsulated support for them most useful. There is ample support for this so-called *editing* UIMS approach in other domains of user interface management. For example, Olsen and colleagues (Olsen, 1992; Olsen et al., 1992) have investigated the use of editing UIMS for direct manipulation of general object collections.

The idea of encapsulating an editing paradigm in a development environment has also been tried in other areas. The Dost system (Dewan and Solomon, 1990) is a programming environment where the resulting applications are generalized editors. This approach makes it possible for the programming environment to better support user interface development, even though the Dost system cannot be considered a UIMS by traditional standards.

Regarding the encapsulated hypertext-style browsing, it is instructive to consider recent work in the area of hypermedia. Features often listed as desirable in hypermedia systems include transparent, object-oriented information storage and ways of managing the users' interaction history (Haan et al., 1992; Nielsen, 1990). Our framework, based on an object-oriented database with temporal reasoning capabilities, is an excellent substrate for providing such features.

5.4 Conclusions

The work presented in this paper is an example of the tradeoff between generality and level of support for the user interface developer. By confining the scope of our UIMS to object-oriented database applications, we have been able to

- identify a suitable level of separation between user interface and application and implement a UIMS based on a logical model where the amount of application information in the user interface specification is well-defined and manageable;
- manage multiple interfaces to the same application objects and the dynamic selection among these interfaces in a declarative way;
- provide encapsulated management of the end-user's interaction with the application in terms of editing and hypertext-style browsing of the objects in the database.

To summarize, we have investigated the use of an editing UIMS model and arrived at a framework which answers to the requirements posed by general object-oriented database applications as well as hypermedia applications. Our claim is that this is a valuable alternative to the approach of developing maximally general user interface management.

Acknowledgments

This research is supported by funds from CENIIT (Center for industrial information technology) and from the Swedish Board for Technological Development (STU) under grant # 8802819P.

A considerable number of people have been involved in the design and implementation of the current system. Some of these are Ralph Rönnquist, David Partain and Michael Jansson. The medical prototype application was developed primarily by Patrick Olnäs and Christian Ökvist.

References

Bancilhon, F. and Kim, W. (1990). Object-oriented database systems: In transition. *SIGMOD Record*, 19(4):49–53.

Beech, D. and Mahbod, B. (1988). Generalized version control in an object-oriented database. In *Proc. 4th IEEE Conf. Data Engineering*, pages 14–22.

Cattell, R. (1991). *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, MA.

Coutaz, J. and Balbo, S. (1991). Applications: A dimension space for user interface management systems. In *Human Factors in Computing Systems (CHI'91 Proceedings)*, pages 27–32. ACM Press.

Dewan, P. and Solomon, M. (1990). An approach to support automatic generation of user interfaces. *ACM Trans. Programming Languages and Systems*, 12(4):566–609.

Edmonds, E. and Hagiwara, N. (1990). An experiment in interactive architectures. In *Human-Computer Interaction — Interact'90*, pages 601–606. Elsevier.

Green, M. (1985). Report on dialogue specification tools. In Pfaff, G. (ed), *User Interface Management Systems*, pages 9–20. Springer Verlag.

Haan, B., Kahn, P., Riley, V., Coombs, J., and Meyrowitz, N. (1992). IRIS hypermedia services. *Communications of the ACM*, 35(1):36–51.

Hall, T. (1992). Command history in LINCKS. Master's thesis, Dept. of Computer Science, Linköping University, Sweden.

Hudson, S. (1987). UIMS support for direct manipulation interfaces. *Computer Graphics*, 21(2):120–124.

Kim, W., Bertino, E., and Garza, J. (1989). Composite objects revisited. In *Proc. Conf. Management of Data — SIGMOD 89 [SIGMOD Record 18(2)]*, pages 337–347.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The ObjectStore database system. *Communications of the ACM*, 34(10):50–63.

LINCKS Manual (1988). *Application Interface to LINCKS on Sun: Programmers Manual.* LINCKS Project Group, Dept. of Computer Science, Linköping University, Sweden.

Myers, B. (1990). A new model for handling input. *ACM Trans. Information Systems*, 8(3):289–320.

Nielsen, J. (1990). The art of navigating through hypertext. *Communications of the ACM*, 33(3):297–310.

Olsen, D. (1992). *User Interface Management Systems: Models and Algorithms.* Morgan Kaufmann, San Mateo.

Olsen, D., McNeill, T., and Mitchell, D. (1992). Workspaces: An architecture for editing collections of objects. In *Human Factors in Computing Systems (CHI'92 Proceedings)*, pages 267–272. ACM Press.

Padgham, L. (1988). NODE: A database for use by intelligent systems. In *Proc. Third Int. Symp. Methodologies for Intelligent Systems*, pages 190–199. Elsevier.

Szekely, P. (1990). Template-based mapping of application data to interactive displays. In *Proc. ACM SIGGRAPH Symp. User Interface Software and Technology (UIST'90)*, pages 1–9. ACM Press.