A Customizable Window-Interface To Object-Oriented Databases

E. Laenens - F. Staes

Philips PASS-AIT
KIWIS team
Building HCM5 P.O. Box 218
NL - 5600 MD Eindhoven, The Netherlands

D. Vermeir

Dept. of Math. and Computer Science
University of Antwerp, UIA
Universiteitsplein 1
B - 2610 Wilrijk, Belgium

Abstract

In this paper we present a user interface to the KIWI object oriented database system. The system combines ease of use with a powerful customization facility which greatly simplifies the development of end-user applications. This is achieved by using the underlying object-oriented language (OOPS+) features such as inheritance to define a consistent and orthogonal set of graphical object classes that can be used to attach a taylor-made external representation of (a view on) objects and/or classes. In addition, using the same tools, it becomes a simple matter to define general purpose browsing and manipulation tools which otherwise would have to be developed separately.

1 INTRODUCTION

The object-oriented paradigm has proven to be successful in a number of areas, such as programming languages, databases and user interface systems. In the latter area, the object-oriented philosophy naturally leads to the direct manipulation paradigm where the screen display units correspond to objects which can be manipulated directly, i.e. the user has the impression of performing actions on the representation, which immediately affect the underlying object.

This approach is appealing because the environment supported by the system bears a close resemblance to the "real world" that the system is supposed to model. Consequently, applications employing the direct manipulation paradigm are easy to learn. The now widely employed desktop metaphor is a classical illustration, albeit on another level, of this idea.

Another example of direct manipulation interfaces can be found in database browsers.
Here, the underlying knowledge representation model is often some kind of semantic network which determines the visualization of objects in the database. The user is then encouraged to perform exploratory searches using a predefined set of simple navigation commands, that allow the examination of objects in the "neighborhood" of the current object of interest. For instance, it is possible to focus on an object which is a property of the current object, or on its class etc. Some systems, notably KIVIEW [1], let the user specify his own view on the database by specifying virtual classes, which may be regarded as (answers to) queries.

Also programming environments for object oriented languages tend to support the direct manipulation paradigm, as is exemplified by systems such as Actor [2], Mjolner[3] and Smalltalk[4]. Typically, a class hierarchy browser is provided in addition to editing facilities that provide for the creation and manipulation of objects and classes.

From a certain point of view, object oriented programming environments and browsing systems such as KIVIEW are similar in that both support the direct manipulation paradigm: programming environments do this for knowledge engineers while browsers are oriented towards naive end-users. However, both kinds of systems are rigid in the sense that they are not easily customized. While this is not really a problem for programming environments where the application is always the same, i.e. software development, we feel that this is a serious drawback for knowledge base browsers. Such browsers could be called "knowledge-independent" since the way an object is represented and manipulated on the screen bears only a minimal relation to its semantics, as perceived by the end-user. Instead, the visualization is a consequence of the underlying knowledge representation model, of which especially naive users are not supposed to be aware.

As an example, one imagines that, to an end user, the representation of Figure 1 for a *student object is more meaningful than the "plain vanilla" one, which is typically*

provided by a browser such as KIVIEW, illustrated in Figure 2.

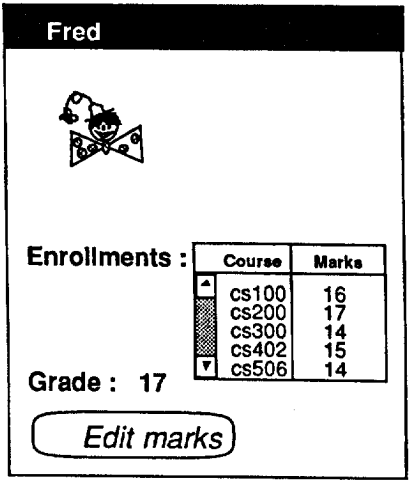

Figure 1 : custom representation

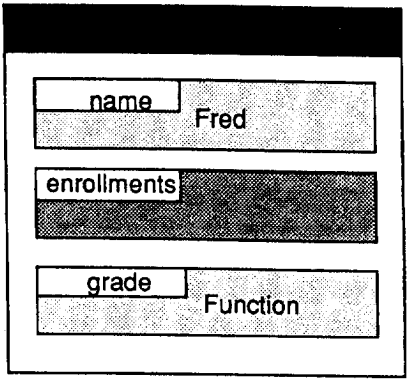

Figure 2 : standard representation

Unfortunately, the development of customized object representations for applications is not a simple matter. Even if, as is the case e.g. for Smalltalk, the language is equipped with graphical routines, these may be rather low level and cumbersome to use.

What we try to do is to use some of the experiences of User Interface Management Systems[9, 10] in our browser. We don't want to offer all the functionality of a User Interface Management System since we believe nobody wants to use e.g. animation to customize a browser. We based our design on the ideas existing in the User Interface Management systems about window-layout definitions[11] and dialogue management[12] although we limited the functionality in some ways to provide an easy system to the knowledge engineer.

In the present paper, we will describe an attempt to fulfill our requirements by extending the underlying object oriented language (OOPS+) with a limited number of predefined interface objects and types (classes) that have an interpretation in terms of the I/O facilities (mouse, windows, etc.). The package is called NAU (Not Another User interface).

Using these interface objects, it becomes straightforward to define a customized representation of an object (or a class of objects), which also includes dialogue features that define the possible interaction with an object through buttons, edit windows etc.

This is further facilitated by using other object-oriented features of the underlying language such as polymorphism and default values (inheritance). This allows e.g. to limit the number of functions that belong to the interface to just a few, which benefits the readability and correctness of the application.

These interface tools have by now been tested for a number of applications and the results indicate that programmer productivity is greatly increased and it becomes possible to develop prototype knowledge base applications with attractive user interfaces in a very short time.

Typically, applications developed using this interface can be regarded as customized browsers. In addition to the usual browsing activities, a number of predefined transactions that constitute the application are associated as methods with the proper objects. They can be called directly from the browser by activating the corresponding visualization, e.g. a button.

In order to facilitate the incremental development of such a customized interface to a knowledge base, we have provided a default browser (mostly written in OOPS+) that associates a standard visualization with each object. It is then possible to customize the presentation of the objects that are more relevant to the application. Due to inheritance, the browser run-time system will use the customized visualization whenever it exists and otherwise default to the standard one.

The customization itself can be done "without programming" by using an interface design tool, which is part of the designer version of the standard browser. Using this tool, one can interactively design hierarchical window layouts and associate these with the proper objects or classes.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of the relevant aspects of the OOPS+ language. Section 3 describes the design of the NAU objects and functions. In Section 4, we present an illustrative example application. Finally, Section 5 describes the present prototype implementation and further developments.

2 OOPS+

OOPS+[5, 6] has been developed as an object-oriented database programming language to be used in the KIWI system[7] as an interface to databases through an advanced deductive database environment. The language aims to provide a suitable tool for knowledge-based applications programming, integrating concepts from semantic modeling, databases and logic and object-oriented programming in a simple and orthogonal fashion. The basic concept in OOPS+ is the notion of (persistent) object. Several kinds of objects are recognized, the most important of which are

- Primitive objects such as `str` and `int` (the set of all strings and all integers respectively), integers, strings, `nil` and `*` (the object that has all objects as an instance).
- Records, which are labeled tuples of references to other objects. For example,

```
fred = (name = "fred"; age = 5)
```

defines an record object with two fields, labeled name and age.

- Sets and lists of (references to) objects.

```
persons = { john, joe, fred, jane };
ranks = < {fred, jane} , john>;
babies = { X | X : persons and X.age < 1 }
```

- Function objects. Functions may be written using a mixture of imperative and logic programming (for querying) constructs. They may also be set up for automatic execution whenever certain events occur. This trigger mechanism is useful, e.g. to support constraint enforcing.

Between objects, an "instance-of" relation, denoted as ' : ' is defined which can be intentional (structural similarity) or extensional (set membership). E.g. we have that

```
fred : (name = str)
```

because the `name` field of `fred` refers to an instance of `str`.

On the other hand, we also have that

```
fred : persons
```

since `fred` is a member of the `persons` set object.

OOPS+ is strongly (but not statically) typed. Type constraints are specified by requiring that a record field always refers to an instance of an associated (type) object. E.g.

```
fred = (str name = "fred";persons father = joe)
```

requires `fred`'s name to be a string and his `father` must be an existing person. Any object can be used as a type, and thus types are first class objects which may evolve, e.g. when using a set as a type.

Functions are also (polymorphically) typed since the actual parameter (record) object must be an instance of the formal parameter object.

An OOPS+ program then is just a definition of a 'top level' record object, called the program object. The program below specifies a small knowledge base to be used in the sequel of this paper.

```
student = ( name = str;
      enrollments = [enrollmentType] default {};
      grade = () -> int default .. ;
```

```
        );
    enrollmentType = (
        course = courses;
        score = int;
        );
    courses = { "cs100", ... };
```

In the example, `[enrollmentType]` denotes a power object, the instances of which are all finite sets of instances of enrollmentType.

A record object may specify default fieldvalues for its instances. Together with the operation of type casting, this provides the usual inheritance facilities.
E.g. the expression `(student)(name="fred")` will equip `fred` with the default (empty) set of enrollments and (a reference to) the default grading function.

Finally, we mention the possibility of creating the union or intersection (denoted `a|b` and `a&b`) of two objects. Such objects have as instances the union (resp. intersection) of the sets of instances of their components. Union and intersection objects are useful as types, as in

```
person = male | female
student = person & (grade = int)
```

3 NAU: Not Another User Interface

Our requirements for a practical user interface module to be used for developing applications are the following:

- It should provide most of the capabilities of a windowing system but hide the underlying complexities.
- It should fit smoothly with the object-oriented and direct manipulation paradigms, e.g. using inheritance and polymorphism.
- It should be flexible, making it possible to develop applications incrementally in a very short time.
- Finally, there should be an "interface definition module" that gives the opportunity of specifying most of the interface interactively, without programming.

3.1 Standard browsing

The starting point for the design of NAU is the idea of browsing through a knowledge base consisting of complex objects. A simple browser is relatively easy to design: it is necessary to provide a standard visualization template for each kind of object that may appear in the knowledge base. In addition, the kinds of actions that may be taken must be specified. Typically, actions will allow the user to switch focus to another object that bears a certain semantic relationship with the present one. The

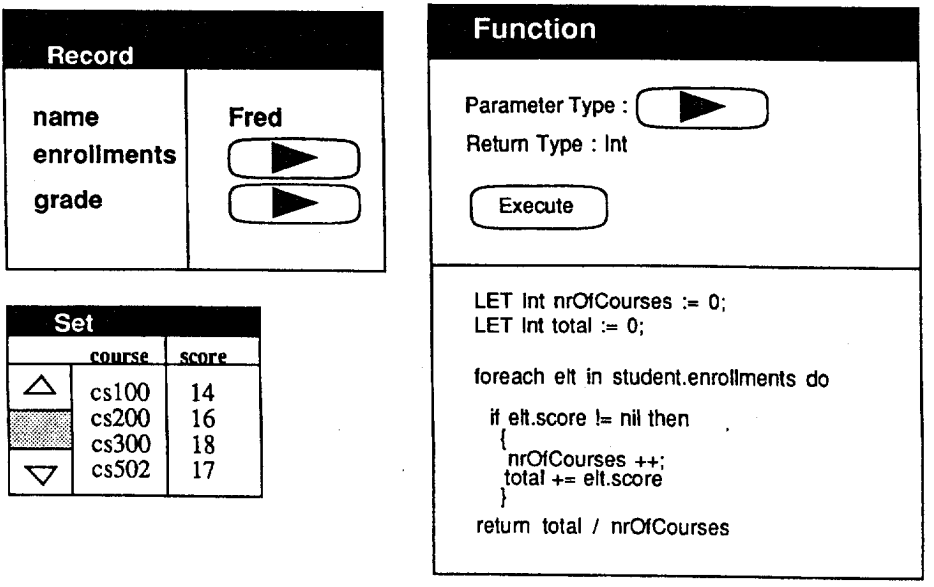

Figure 3 : The standard OOPS+ object representation

standard visualizations of some OOPS+ objects are illustrated in Figure 3.

Note in figure 3 that the field of a record representation either contains the object itself (when it is primitive) or a button (when it is a complex object), which when pressed will zoom into the object the field points to. A similar rule holds for the presentation of members of a set object. Scrollbars are automatically added whenever the size of the window would become too large.

Actions are associated either with the window or a part of it. They are activated by clicking the mouse button in the appropriate area. For instance, with each object window is associated a menu that allows the user to request the display of the type of the object, its instances etc. It is also possible to update the knowledge base, e.g. by creating new objects, modifying or adding fields or members etc. This is illustrated in Figure 4.

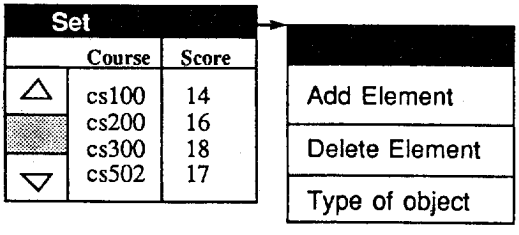

Figure 4 : possible actions on a set object

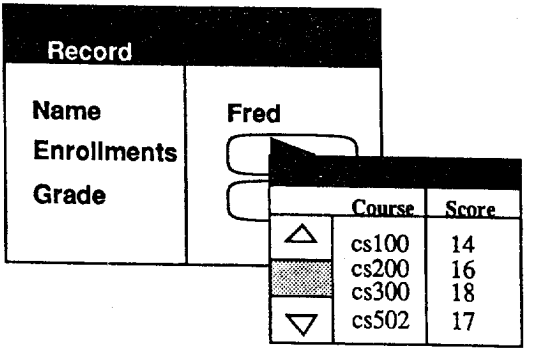

Figure 5 : zooming into a field of a record.

In addition, one can click on member (for set objects) or field buttons (for records) in order to display the window associated with the member/field. (See figure 5)

Functions can be either browsed (i.e. the text is displayed in an edit window) or activated, in which case a parameter object is automatically created which is then further specified by the user, before the function is actually executed.

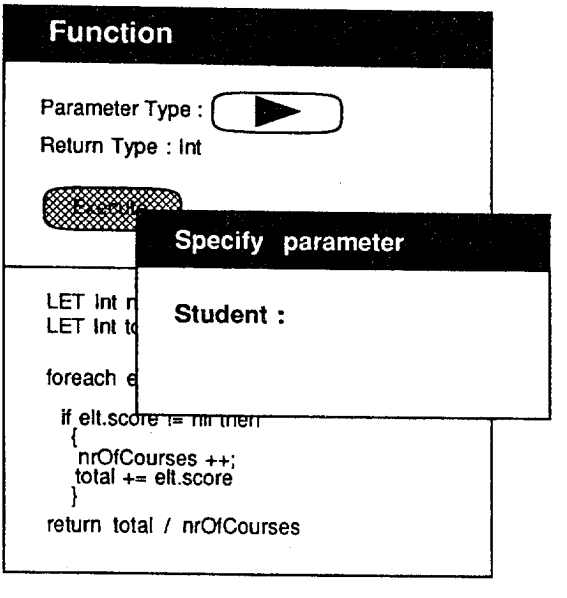

Figure 6 : Executing a function.

3.2 Customizing the visualization

The next step in the design of NAU is the realization that, using the inheritance and default mechanisms of OOPS+, most of the simple browser interface described above can actually be written in OOPS+.

The main idea is that the association between an object and its external representation can be specified in the language itself. In NAU, an object is represented by a window, which is itself an object. A window typically contains information on the object with which it is associated, a description, possibly in terms of subwindows, of its layout (size, position etc.) and a reference to a function that is to be activated when an event (e.g. the clicking of a mouse button), to be handled by the window, occurs. When displayed, a window occupies a rectangular area on the screen. The precise contents of this area depends on its kind (see below) and layout and on the object with which the window is associated.

There is a predefined function winDisplay that displays a window on the screen, waits for events, and transfers control to the action function associated with the event occurrence, implementing the usual control structure of window packages such as SUNVIEW.

```
str aText = "Hello world";
WINDOW win = TEXT ( txt = aText;
      label = "TEXT WINDOW" );
     /* window of program object */
```

The program above defines a program object with two fields: a string and a window. The NAU run time will find a window associated with the program object and will call winDisplay for this window, as is illustrated in Figure 7.

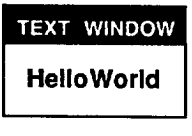

Figure 7.

When the user causes any kind of event for this window, the default action function will be activated.

3.3 Frames and windows

There are a number of different kinds of windows supported by NAU. Some information, such as the size, which is common to all kinds is stored in a "frame", i.e. a "window without contents". The type FRAME is defined in OOPS+ as follows:

```
FRAME = ( label = str default nil;
     size = COORDINATE default nil;
```

```
        location = LOCATION default down;
        action = EVENT -> nil default ... ; );
```

The meaning of the various fields is as follows: if label is not nil then the window will be displayed with a labeled border. If size is not specified (as a width and/or a length), the window will fit its contents. Specifying the size may cause a scrollbar to be automatically added whenever the size of the contents would exceed the given size. The location field may be used to specify the position of the window frame within an encompassing super-window (see below).

```
LOCATION = COORDINATE | left | right | up | down
```

Besides an absolute location (using a COORDINATE), it is also possible to specify a location relative to the previous window in a layout sequence (see below). This is similar to the approach taken in picture definition languages such as pic[8].

The action function specifies what happens upon the occurrence of an event that is to be handled by the window. Relevant information about this event will be stored in an EVENT object that is passed to the action function.

```
EVENT = ( kind = { leftButton, .. };
        position = COORDINATE;
        win = WINDOW );
```

Note that an event also provides a reference to the associated window, making it easier for the handling function to get access to the relevant objects.

A complete window consists of a frame and a specification of its contents. Below is a summary of the kinds of windows that are available in NAU.

- A FORM window serves as a container of other windows. It has a layout field that consists of a sequence of windows, any of which may again be FORM's.

  ```
  FORM = (layout = <WINDOW>)
  ```

  Upon display, the form is filled with the subwindows as specified in the layout field.

- Text windows for displaying text.

  ```
  TEXT = (txt = str )
  ```

- Edit windows that allow update of a text.
- Pictures that display raster images.
- Buttons which are typically associated with actions that perform some application function.
- Tables that provide automatic alignment and resizing (both horizontally and vertically) between rows of subwindows. Note that there is no restriction on the kind of subwindows that are supported by tables.

- Choice windows which allow an event that corresponds to the selection from a collection of alternatives.
- Finally, canvasses are equipped with graphics methods that allow the program to construct arbitrary drawings.

Hence the NAU definition of window can be summarized in OOPS+ by:

```
WINDOW = FRAME & (TEXT | EDIT | BUTTON |
      FORM | TABLE ..)
```

As an example, the window of Figure 1, which is associated with a student, can be specified in OOPS+ by:

```
( label = fred.name;
      layout = <
      PICTURE (pic=fred.picture);
      TEXT (location = down;
      txt = "Enrollments:" );
      TABLE (size = (height=3);
      rows = <<(txt="Course");(txt="Mark") >> +
      <(txt=e.course);
       (txt=e.score)> | e : fred.enrollments);
      TEXT ( location = down;
      txt="grade:" + fred.grade );
      BUTTON  ( student=fred;
      location = down;
      txt="Edit marks";
      action = nil
      {
      /* edit marks function */
      ...
      }();
      );
      >
      );
```

Note that in the BUTTON subwindow, fred is referenced (using the field "student") such that the action function, when activated, will have access to all relevant data through the parameter event. This is an example of the use of polymorphism.

4 Case study

This section presents an actual small NAU application concerning a knowledge base of articles (papers) that have associated search keywords. The listing in the appendix contains the query part of the program where all papers dealing with a certain subject are retrieved. Figure 8 shows a snapshot of the running system where the user has asked for a list of all papers concerning OO (object oriented programming) after

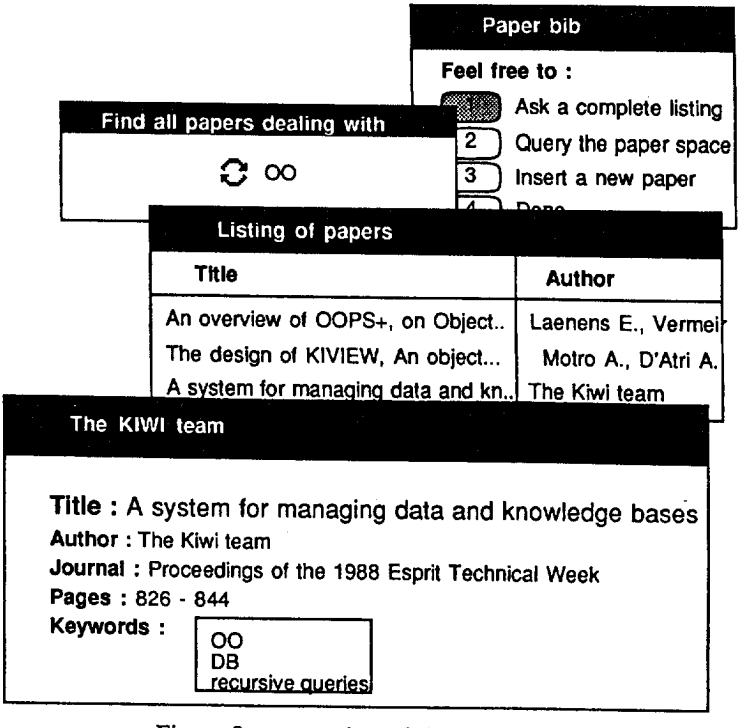

Figure 8 : a snapshot of the papers-demo

which he asked more information on the paper about "A system for managing data and knowledge bases" by the Kiwi team.

5 Further developments

At present, a NAU prototype is running on top of the SUNVIEW window package. We have also integrated NAU with the prototype KIVIEW browser. Currently, we are developing a new browser using NAU default object- window associations. As window are themselves objects, it should be possible to specify a suitable window to present window definitions. Together with appropriate methods, we then obtain a "window definition module" that can be used to design interface objects without programming. Such a module is currently under development.

6 Bibliography.

1. A. Motro, A. D'Atri, and L. Tarantino, "The Design of KIVIEW: An Object-Oriented Browser," in *Proc. of the 2nd Intl. Conf. on Expert Database Systems*, pp. 17-31, 1988.

2. W. Bliss, R. Kimmel, J. Leahy, B. Newburger, and M. Solinkski, *ACTOR language Manual, The Whitewater Group, 1987.*

3. G. Hedin and B. Magnusson, "The Mjolner Environment: Direct Interaction with Abstractions," in *Proc. of the European Conference on Object-Oriented Programming*, pp. 41-54, 1988.

4. A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983.

5. E. Lanenens and D. Vermeir, "An overview of OOPS+, an Object-Oriented Database Programming Language," in *Proc. of the European Conference on Object Oriented Programming*, pp. 350-373, 1988.

6. E. Laenens and D. Vermeir, "A language for Object Oriented Database Programming," *Journal of Object Oriented Programming*, Vol. 1, No. 5, pp. 18-28, 1988.

7. The KIWI team, "A System for managing Data and Knowledge Bases," in *Proceedings of the 1988 Esprit Technical Week*, pp. 828-845, 1988.

8. B. W. Kernighan, "pic - A Language for Typesetting Graphics," *Software - Practice and Experience*, vol. 12, no. 1, pp. 1-21, 1982

9. D. J. Kasik, "A User Interface Management System", *Computer Graphics*, vol. 16, no. 3, pp. 99-106, 1982

10. P. Hayes, "Design Alternatives for User Interface Management Systems Based on Experience with COUSIN", in *Proc. of the ACM CHI conference*, pp. 169-175, 1985.

11. P. Barth, "An Object-Oriented Approach to Graphical Interfaces", in *ACM transactions on Graphics*, vol. 5, no 2, 1986

12. Hill R., "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction, the Sassafras UIMS", in *ACM transactions on Graphics*, vol. 5, no. 3, 1986

```
paperType =
        (
        title = Str;
        author = Str;
        journal = Str;
        pages = (beginPage = Str;
                 endPage = Str
                );
        keywords = [Str]
        );

[paperType]          paperSet = {..};

[Str]       keywordSet = {..};

mainWindow = FORM
        (
        label = "Paper bib";
        layout =
                <
                TEXT     (txt = "Feel free to :";
                          font = bigbold);
                BUTTON   (location = (X=0;Y=down);
                          txt = "1";
                          action = nil
                                   {
                                   winDisplay(listingWindow(papers=paperSet));
                                   }());
                TEXT     (txt = "Ask a complete listing");
                BUTTON   (location = (X=0;Y=down);
                          txt = "2";
                          action = nil
                                   {
                                   winDisplay(queryWindow);
                                   }());
                TEXT     (txt = "Query the paper space");
                BUTTON   (location = (X=0;Y=down);
                          txt = "3";
                          action = ..);
                TEXT     (txt = "Insert a new paper");
                BUTTON   (location = (X=0;Y=down);
                          txt = "4";
                          action = ..);
                TEXT     (txt = "Done")
                >
        );

paperWindow = FORM
        {
        return (label = paper.author;
                layout = <
                         TEXT     (txt = "Title:" + paper.title);
                         TEXT     (location = (X=0;Y=down);
                                   txt = "Author:" + paper.author);
                         TEXT     (location = (X=0;Y=down);
                                   txt = "Journal:" + paper.journal);
                         TEXT     (location = (X= 0;Y=down);
                                   txt = "Page:" +
                                         paper.pages.beginPage + " - " +
                                         paper.pages.endPage);
                         TEXT     (location = (X=0;Y=down);
                                   txt = "Keywords:";
                                   font = bold);
                         TABLE    (size = (height = 3);
```

```
                                        rows = << >> + <<(txt = k)> | k:paper.keywords >)
                                >
                )
        } (paper = paperType);

listingWindow = TABLE
        {
        return
                (
                label = "Listing of papers";
                rows = <<(txt= "Title");(txt= "Author")>> +
                        <<(ref=p;
                            txt=p.title;
                            action = nil
                                {
                                winDisplay(paperWindow(paper=ref));
                                }0);
                          (txt=p.author)>
                                | p:papers>;
                )
        }(papers = [paperType]);

queryWindow = CHOICE
        (
        label = "Find all papers dealing with";
        choices = keywordSet;
        selectedItem = nil;
        action = nil
                {
                let selectedPapers = {};
                let paper p;
                foreach p in paperSet do
                        if selectedItem:p.keywords then
                                selectedPapers += p;
                winDisplay(listingWindow(papers = selectedPapers));
                }0
        );
```