11 Conclusions

We have described an object oriented UIMS that handles frame based data-representation in combination with binary relations. The UIMS handles simultaneous multiple (and different) visualisations of the same persistent object structure. It provides a full separation of user interface and functionality of the application, by defining that the UIMS should have knowledge about the storage layer of the application. The design has proven to support incremental development of a complex application at multiple sites.

The full separation of application from user interface and defining the "linkage" between them has long been a problem of UIMSs. This paper shows that separation is possible by adopting old technology (screen editors) to a new situation, whereas linkage can be centralised.

Further information on the UIMS and how it is applied in practice may be found in [3].

Acknowledgements

Claire Bouchet, Eric Brunet and Martin Wood participated in the development of Shelley and the design of the UIMS. Guus Schreiber commented on a draft of this paper. Reviewer's comments are gratefully acknowledged. The work reported in this paper has been supported in part by the Commission of the European Communities under contract ESPRIT Project 1098. Partners in this project are STC plc, SD plc, NTE GmbH, Cap Sesa Innovation, KBSC of TRMC and the University of Amsterdam.

References

[1] J. H. Alexander. Painless Panes for Smalltalk Windows. In *OOPSLA'87*, pages 287–294, 1987.

[2] A. Anjewierden. Knowledge Acquisition Tools. *AI Communications*, 0:29–38, 1987.

[3] A. Anjewierden, C. Bouchet, J. Wielemaker, and M. Wood. Shelley programmer's manual. ESPRIT Project 1098 Technical Report UvA-C2-TR-006a, University of Amsterdam, February 1989.

[4] A. Anjewierden and J. Wielemaker. Extensible objects. ESPRIT Project 1098 Technical Report UvA-C1-TR-006a, University of Amsterdam, March 1989.

[5] A. Anjewierden and J. Wielemaker. PCE 3.7 Reference Manual. ESPRIT Project 1098 Technical Report UvA-C1-TR-004, University of Amsterdam, February 1990.

[6] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[7] J. A. Breuker and B. J. Wielinga. Model Driven Knowledge Acquisition. In P. Guida and G. Tasso, editors, *Topics in the Design of Expert Systems*, pages 265–296, Amsterdam, 1989. North Holland.

[8] G. Cockton. Interaction Ergonomics, Control and Separation: Open Problems in User Interface Management Systems. Technical Report AMU8811/03H, Scottish HCI Centre, February 1988.

[9] M. Green. Report on Dialogue Specification Tools. In G. Pfaff, editor, *User Interface Management Systems*, pages 89–107, Berlin, 1985. Springer Verlag.

[10] S. E. Hudson. UIMS Support for Direct Manipulation. *Computer Graphics*, 21(2):120–124, February 1987.

[11] E. W. Myers and W. Miller. Row Replacement Algorithms for Screen Editors. *ACM Transactions on Programming Languages and Systems*, 11(1):33–56, January 1989.

[12] J. L. Sibert, W. D. Hurley, and T. W. Bleser. An Object-Oriented User Interface Management System. *Computer Graphics*, 20(4):259–268, August 1986.

[13] P. Szekely. *Separating the User Interface from the Functionality of Application Programs*. PhD thesis, CMU-CS-88-101, January 1988.

[14] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon. The Symbolics Genera Programming Environment. *IEEE Software*, pages 36–45, November 1987.

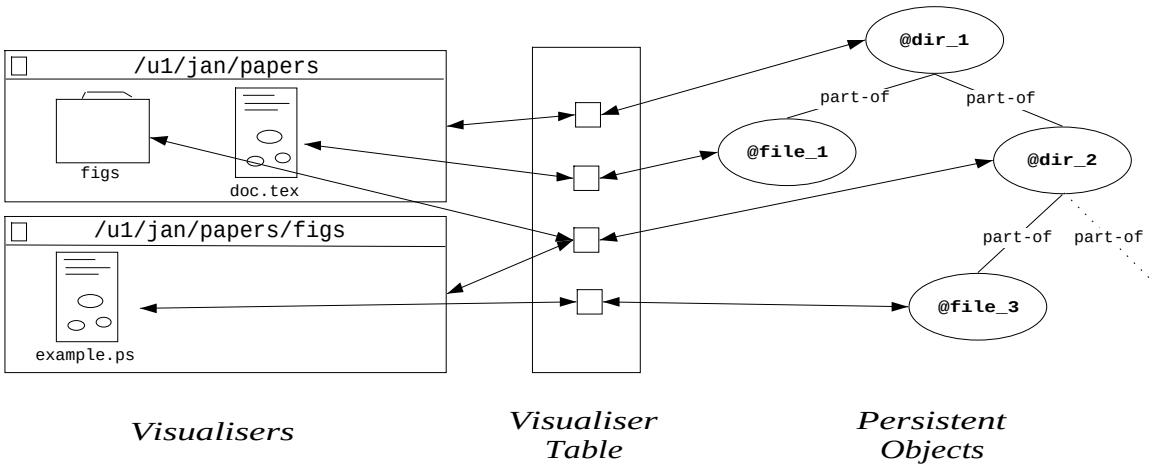

Figure 3: Example of visualisers and persistent objects

```
create_relation(V, consists_of, Sub) :-                    1
   sub_visualiser(V, Sub).                                 2

delete_relation(V, consists_of, Sub) :-                    3
   visualiser(Sub, SV),                                    4
   contains_visualiser(V, SV),                             5
   destroy_visualiser(SV).                                 6
```

Figure 5: Update section

sends a message to the directory_window visualiser. The Prolog code of figure 6 deals with this.

```
new_subdirectory(V) :-                                     1
   obtain_name(Name),                                      2
   create(SubDir, directory(name = Name)),                 3
   visualiser(Dir, V),                                     4
   relate(Dir, consists_of, SubDir).                       5
```

Figure 6: Create a subdirectory

2 Obtain a new name for the directory (this activates a dialogue box).

3 Create a new directory object with name *Name*.

4 Get the persistent directory object we are a visualiser of.

5 Relate the new directory to the current directory. The UIMS monitors this call and invokes create_relation on this, and other visualisers, which update the display.

10 Related Work

Szekely's thesis [13] gives an excellent overview of issues related to the design and implementation of user interfaces, in particular the separation of user interface and functionality in an application. His Nephew system provides a "weak methodology" for user interface development. The UIMS described in this paper ignores user interface design as such, and instead offers guaranteed consistency between what is displayed and maintained by the application.

Many others have developed UIMS models along the lines of the control flow diagram in figure 1, e.g., [9], [8], and [10]. They distinguish "presentation" (lexical), "dialogue" (syntactic) and "application" (semantic) components. Our model has the advantage of being very explicit what services the UIMS provides and where it is situated. Other techniques may be employed to create a linkage between visualisers and the user (e.g. [4]).

GWUIMS [12] is an object oriented UIMS that distinguishes: representation, interaction and application objects. The logical separation of the application components is determined by rules restricting how messages are passed between the types of objects. Active values are used to associate display and application attributes. GWUIMS places update control inside the objects. The Smalltalk MVC paradigm [1] also uses localised control. In our UIMS, updates are initiated in the storage layer and effectuated by the UIMS, resulting in centralised control.

The UIMS provided in the Symbolics Genera programming environment [14] is in many ways similar to the Shelley UIMS. Genera provides objects as the basic data representation and *presentation types* that act as visualisations of objects. The main extension of the Shelley UIMS, over Genera, is the ability to handle binary relations in addition to the frame based object model.

```
unrelate(P1, R, P2)
  remove_relation(P1, R, P2)
  foreach V ∈ V_P1 do
    invoke(V, delete_relation, R, P2)
  foreach V ∈ V_P2 do
    invoke(V, delete_relation, (R̄), P1)
```

Create

Creating a new object P_i has no implications for the visualiser world. The UIMS cannot know which visualisation needs to be created, nor where and how it should be shown to the user. A visualiser might be created if P_i is related to P_j and V_{P_j} is not empty.

Free

The pseudo-code for free is given below:

```
free(P_i)
  forall related(P_i, R, P_j) do
    unrelate(P_i, R, P_j)
  foreach V ∈ V_P_i do
    destroy_visualiser(V)
  remove_object(P_i)
```

8 Implementation

On the principles described above, a UIMS has been implemented on top of PCE/Prolog [4], [5]. PCE/Prolog consists of an object oriented system for building (graphical) user interfaces, connected to a conventional Prolog system. PCE/Prolog runs on Sun™ workstations. The storage layer, behaviour invocation and the UIMS are implemented in Prolog. The attribute/value pairs and relations associated with the persistent objects are stored in PCE, which offers more efficient storage and retrieval facilities.

Experience with Shelley indicates that the design is sound and facilitates incremental development in three countries with an integration effort of a few hours.

9 Example

We use a Macintosh style interface to a file system to illustrate application of the design. Unlike the Macintosh, we suppose the finder (visualiser of the file system) runs in parallel with other applications. We define the following persistent objects (see figure 3):

File
: A *File* F is a string of bytes. Files have a *name*. Files can have one to infinite "part-of" relations to *directories*.

Directory
: A *directory* D is a collection of files and directories. The only slot of a directory is its name. It can have 0 to infinite "part-of" relations with other (super-)directories and "consists-of" relations with 0 to infinite files and (sub-)directories.

A directory has two possible visualisations: a window displaying the files and directories that are part-of it and as an icon. Figure 4 shows the simplified *instantiate* predicate (called by create_visualiser/3) of the visualiser class *directory_window*.

```
instantiate(V, D) :-                          1
  create_window(V),                           2
  register_visualiser(D, V),                  3
  forall(related(D, consists_of, Sub),        4
    sub_visualiser(V, Sub)).                  5

sub_visualiser(V, Sub) :-                     6
  create_visualiser(SV, Sub, dir_icon),       7
  display_visualiser(V, SV).                  8
```

Figure 4: Instantiate a directory_window

3 Register V as a visualiser for D.

4–5 Create visualisers for all objects that are part-of the directory.

7–8 Create an iconic visualiser an object and register it to be contained in the directory_window. Directory icons are an example of a generic visualiser. This can be realised because all behaviour of the visualiser can be defined by issuing common behaviour on the associated persistent object (e.g. *name*, *image*, *open*).

Unlike the Macintosh finder, the directory window stays visible to the user while other applications are running. The directory window should therefore be updated if any application creates, deletes or renames files and directories. The UIMS notifies the finder using create_relation/3 and delete_relation/3. Figure 5 illustrates this.

4–5 Check whether V contains a visualiser for *Sub*. If so destroy this visualiser.

Note that changes in the name of a file or directory will be taken care of by the iconic visualiser for the file or directory whose name changed.

Finally we assume a menu has been attached to the directory window (in the *instantiate* predicate) which allows the user to create a sub-directory. The menu

Generic Visualisers

In many applications similar visualisers are used for a collection of classes. Examples are iconic visualisers for an object, the name of an object or the description of an object. The design should allow for the definition of visualisers that can be attached to objects of different persistent classes. As all communication between the visualisers, persistent objects and UIMS (see section 7) is done via message passing we can attach instances of one visualiser class to various persistent classes. For example, a generic iconic visualiser (displaying an icon and the name of its associated persistent object) with "open" behaviour (e.g. in response to a double-click) can use *name* and *open* behaviour to communicate with the persistent object. It is then left to the persistent object to respond appropriately.

7 UIMS

The communication between persistent objects and visualisers is controlled by the UIMS. The UIMS maintains a mapping between the visualisers and corresponding persistent objects called the *visualiser table*. We refer to the persistent object P of visualiser V as P_V. Similarly, the *set of* visualisers associated with P is referred to as V_P. The UIMS also maintains a table of *part-of* relations between visualisers. A visualiser is *part-of* another visualiser if it is displayed on another visualiser, e.g., a window can be a visualiser of a directory, while an icon displayed on the visualiser is a visualiser of a file or subdirectory. Knowledge of part-of relations between visualisers is necessary because a visualiser V_1 displayed on V_2 cannot exist without V_2.

These tables are maintained by a small number of predicates that form the interface between the visualisers and the UIMS:

create_visualiser(*−V, +P, +Class(+Arg,...)*)
: Create *V* as a visualiser of *Class* for persistent object *P*. Use *Arg* as initialising arguments for *V*.

destroy_visualiser(*+V*)
: Destroy visualiser *V*. If the visualiser contains other visualisers these are destroyed first.

display_visualiser(*+V1, +V2*)
: Display visualiser *V1* on visualiser *V2*. This updates the visualiser part-of table.

The primitive functions in the storage layer call the UIMS. And, in turn, the UIMS invokes update behaviour on the visualisers using the visualiser/persistent and visualiser part-of tables.

The following paragraphs describe the actions undertaken by the UIMS on the various interface functions of the storage layer. The retrieval functions (e.g. fetch/3) do not have consequences for visualisers, and are omitted.

Store

Storing a new value X in a slot S of a persistent object P is the simplest of the actions. All visualisers of P are notified of the change. Visualisers that display S respond by updating their appearance. Visualisers that do not display S ignore the change. The pseudo-code for store is given below.

```
store(P, S, X)
  P.S <= X
  foreach V_i ∈ V_P do
    invoke(V_i, update, S, X)
```

If X is an object itself and V_i displays S then V_i destroys the old visualiser for the slot, creates a new visualiser of the appropriate type and displays it using display_visualiser/2. Note that only V_i knows which of the possible visualisations (e.g. textual, iconic, ...) is appropriate. This implies the UIMS should pass X rather than a visualisation for X and leave the decision on the correct visualisation to V_i.

Relate

Slots are normally used to store primitive data. Relations are a more powerful mechanism to indicate a semantically meaningful relation between two persistent objects. As we have seen in section 4 relations have different name in both directions. This has consequences for the update scheme used:

```
relate(P_1, R, P_2)
  make_relation(P_1, R, P_2)
  foreach V ∈ V_P1 do
    invoke(V, create_relation, R, P_2)
  foreach V ∈ V_P2 do
    invoke(V, create_relation, R̄, P_1)
```

where $\bar{R}$ is the inverse relation of R. The persistent object P_1 (P_2 respectively) is passed to V. This enables V to decide what type of visualisation to create.

Unrelate

Deleting a relation is very similar to creating one.

Identifiers starting with a capital letter denote variables. Identifiers starting with a lower case letter denote constants. The arguments in a predicate description are preceded with a "−" if their value is delivered by the predicate (output), a "+" if their value is input to the predicate and a "?" if either input or output is allowed (logical predicates).

create(*−P, +Class(+Arg, ...)*)
Create *P* as an instance of *Class* using *Arg* as the initialising arguments. The arguments are pairs of the form *Name* = *Value*. Example:

```
?- create(P, file(name = doc.tex)).
P = @474733
```

`@474733` is a *reference* to the object created.

instance(*?P, ?Class*)
Is true if *Class* is the class of persistent object *P*.

free(*+P*)
Remove *P* from the persistent object base.

fetch(*+P, +Slot, −Value*)
Return the value currently associated with *Slot* in persistent object *P*. *Value* is either a reference to an object or a primitive data type.

store(*+P, +Slot, +Value*)
Associate *Value* with slot *Slot* in object *P*.

The following predicates manage relations between persistent objects.

relate(*+P1, +Relation, +P2*)
Relate *P1* using relation called *Relation* with *P2*. Relations are bi-directional. Asymmetrical relations have different names depending on the viewpoint:

relate(*P1, part-of, P2*) ===
relate(*P2, consists-of, P1*)

Relations are specified separately, stating the names of the persistent classes and the allowed cardinality.

unrelate(*+P1, +Relation, +P2*)
Remove a relation from the database.

related(*?P1, ?Relation, ?P2*)
Succeeds if *P1* is currently related to *P2* over *Relation*.

5 Behaviour Invocation

The OOPS defines the message passing algorithms resulting in the ability to invoke behaviour on persistent and visualiser objects. The UIMS only needs the most basic feature of an OOPS: encapsulation. Encapsulation ensures separation of invocation (i.e. the behaviour name: *selector*) and implementation. More advanced object oriented mechanisms, inheritance and delegation, are not necessary for the UIMS.[3] Two predicates to invoke behaviour are provided:

invoke(*+Obj, +Selector, +Arg...*)
Invoke behaviour identified by *Selector* on object *Obj*. *Obj* can be either a visualiser or a persistent object. *Arg ...* are additional arguments for the behaviour.

retrieve(*+Obj, +Selector, ...+Arg..., -Result*)
Obtain status information from object *Obj*. The answer is unified with *Result*.

The data associated with persistent objects is managed by the storage layer. Predicates implementing behaviour of persistent objects use the interface functions to the storage layer to manipulate the data associated with the object. They invoke behaviour on other (related) persistent objects to manipulate them.

6 Visualisers

A persistent class can have one or more *visualiser* classes associated with it. A visualiser class defines one visual appearance of a persistent class. A visualiser class defines how a new visualiser for an instance of its corresponding persistent object needs to be created and how it responds to various changes in the persistent object (i.e. the *update* code, see figure 5 for an example).

If the visualiser can respond to user actions the visualiser also defines a command interpreter. The command interpreter monitors the user actions on the visualiser. If a command is recognised the arguments are collected and behaviour on the corresponding persistent object is invoked. The UIMS invokes update behaviour if the user action resulted in modifications to the storage layer. Note that a user action on a certain visualiser can have propagating effects on other visualisers (see section 7).

Visualisers which contain visualisations of other persistent objects create a visualiser of the appropriate type for those persistent objects and display them.

[3] Although the mechanism described in this article will apply to systems providing inheritance and/or delegation as well

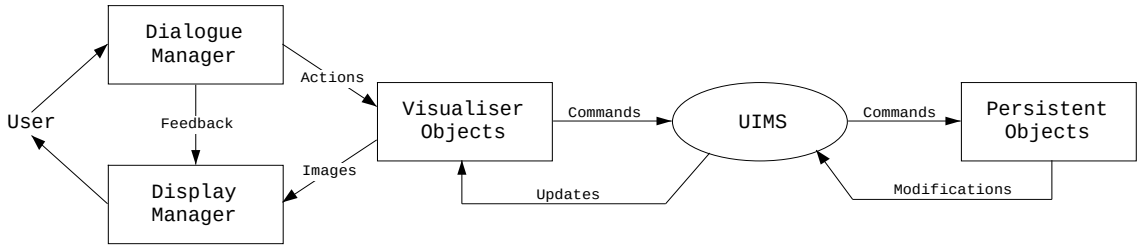

Figure 1: Flow of Control

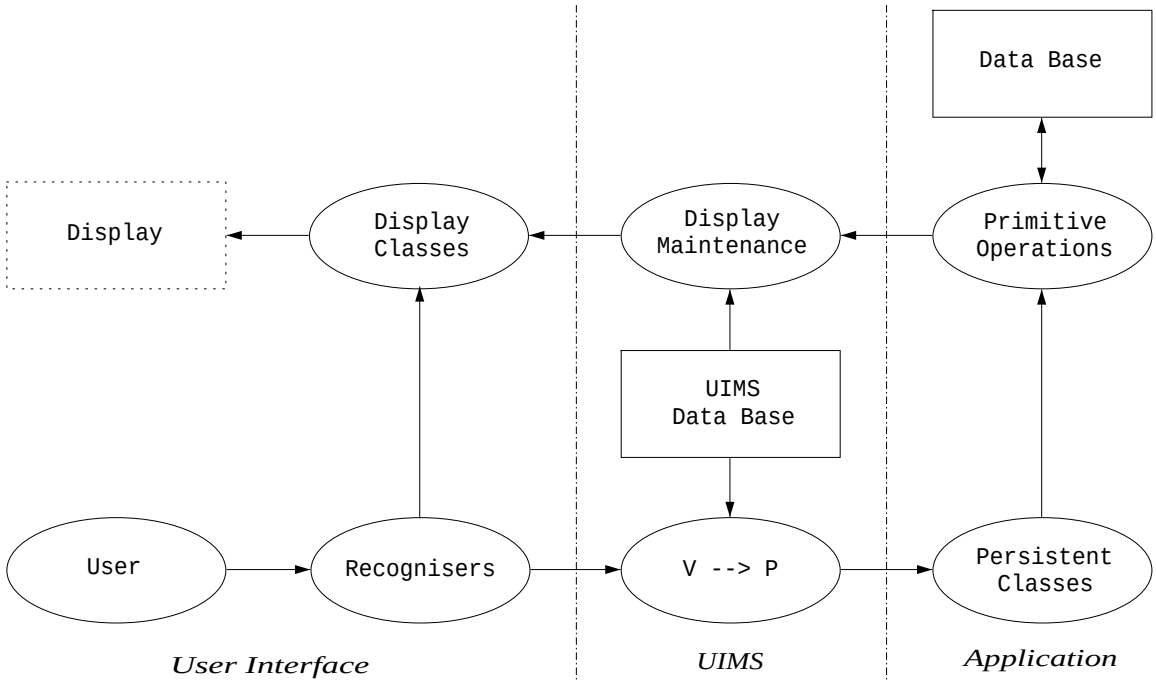

Figure 2: The UIMS Architecture

from the visualisers to the related persistent objects (the "$V->P$" ellipse).

A visualiser defines how it interacts with the dialogue and display managers. Each visualiser defines a set of interaction techniques it responds to (*recognisers*), and a set of display objects. Recognisers take care of lexical feedback (e.g. echoing keyboard input, dragging an object's outline). If a user action affects the persistent world a message is sent (via the UIMS) to the related persistent object. Note that this design completely separates invocation of behaviour on persistent objects ("$V->P$") from updating the display.

4 Storage Layer

The storage layer comprises the *Primitive Operations* and *Data Base* components of figure 2.

The data model is based on frames with attribute-value pairs, extended with binary relations between frames. A relation is a named link between two frames. Although relations are not strictly necessary as they can be emulated using value sets,[2] they simplify maintenance of consistency and are conceptually more clear.

The interface to the storage layer can be compared to the *Primitive Operation* component of the screen editor example. It is used by the UIMS to guarantee consistency between the display and the data base. The actions undertaken by the UIMS on the various functions provided by the storage layer are described in section 7.

Prolog is used to define the interface predicates because its declarative nature allows for a concise description.

[2] In KL-ONE [6] we could express the relation beween the parts of a bike and the bike itself as follows. *Bike* is given an attribute *consists-of* with value restriction *bike-part*. *Bike-part* is made the super-type of *wheel*, and all the other (potentional) part-types. Type *bike-part* is given an attribute *part-of*. Finally the user is made responsible of maintaining the consistency between the bike and its parts.

Command interpreter
Reads commands by the user and maps these on functions that implement a specific command.

Functional component
This component is activated by the command interpreter. It translates the commands to a sequence of *primitive* operations.

Primitive operations
The primitive operations (insert, delete, replace, etc.) modify the internal text buffer. The primitive operations also keep track of what part of the text buffer is modified.

Screen update
The screen is updated by comparing the modifications to the internal text buffer with the contents of the screen buffer. This generates device dependent redisplay commands.

This architecture results in a clear separation of control and interpretation: a user action only indirectly affects display contents. The command interpreter can be changed without consequences for any of the other components. Changing the functional component only has consequences for the command interpreter, i.e. providing access to the changed commands. The primitive operations allow an efficient and portable data representation. The screen update component facilitates porting the editor to any screen driver.

Text editors manipulate only one data structure (a linear string of characters) and usually do not show different representations of the data simultaneously. This paper defines an object oriented design which has the same properties as the screen editor design, but operates on a richer data model: frames (objects) with attributes and values, and binary relations between frames. It also maintains multiple (and different) representations of the same application object. The design aims to support:

Incremental application development
Large applications in new domains cannot be designed completely top-down. The design should therefore support incremental development of user interface and functionality. This can only be achieved by a rigid separation of user interface and functionality.

Reusability and consistency of user interface
Developing and testing user interface code is expensive. Reusability and consistency of such code is desirable.

Multiple representations
In many applications multiple representations of the same underlying object need to be supported. An example is the simultaneous display of a folder icon and a folder window on the Macintosh™.

2 Model

The functionality part of an application is represented by a set of *persistent* objects.[1] Some persistent objects may have display representation. The objects that achieve this representation are called *visualisers*. For each visualiser there is precisely one persistent object. For each persistent object there may be zero or more (different) visualisers. The distinction between visualisers and persistent objects assures that the functionality does not directly interact with the user.

The UIMS (User Interface Management System) translates actions on visualisers to commands on persistent objects. The resulting modifications on the persistent objects are reported back to the associated display objects. The relation between a visualiser and a persistent object is of a declarative nature: V visualises P means that V is updated by the UIMS each time P changes.

The components, and the control flow between them, for our model of a UIMS is shown in figure 1.

A visualiser has two additional aspects: the interaction techniques [13] defined on it ("Dialogue Manager") and its display representation ("Display Manager"). We assume these to be provided by the environment in which the UIMS is embedded. The UIMS described in this paper does not depend on any specific definition of dialogue and display management.

3 Architecture

The relation between the UIMS and the persistent world on one hand and the visualiser world (user interface) on the other hand is summarised in figure 2. Both the display classes and the persistent classes are implemented on top of a small object oriented programming system (OOPS). The most basic aspect of OOPS – separation of behaviour from its implementation – ensures the UIMS can activate both display and persistent objects independent of their class.

The UIMS maintains a database of the displayed visualisers and the persistent objects represented by them. Using this database the UIMS translates the *Primitive Operations* on the *Data Base* into update messages for the visualisers. A visualiser object is responsible to define appropriate behaviour to handle these messages. Besides this service the UIMS forwards messages

[1] The name persistent refers to the characteristic that they are permanent even after quitting the application.

Separating User Interface and Functionality Using a Frame Based Data Model

Jan Wielemaker, Anjo Anjewierden

University of Amsterdam
Department of Social Science Informatics
Herengracht 196
1016 BS Amsterdam
The Netherlands

(jan,anjo)@@swi.psy.uva.nl

Abstract

The separation between user interface and functionality found in many screen editors is generalised to handle a data model based on frames and binary relations. This paper describes a User Interface Management System (UIMS) based on the data model. The UIMS is capable of maintaining different and simultaneous representations of the same application data objects. The functionality and user interface are implemented on top of a small object oriented programming system. This allows the UIMS to be simple and independent of the graphics software and hardware as well as the data representation used by the application programs.

1 Introduction

We are currently involved in the development of a workbench that supports a methodology for knowledge engineering (KADS, [7]). The workbench, called *Shelley*, supports a wide variety of activities to be performed by the knowledge engineer. Shelley supports representations as diverse as documents, compound objects and diagrams. Our experience ([2]) indicates that the methodology and therefore the functionality supported by Shelley is a moving target. This requires a flexible design, especially regarding the user interface and integration.

Due to the lack of good mechanisms to separate the user interface from the functionality the development of (graphical) user interfaces for applications that manipulate various types of data and relations is considered difficult. A clean separation is necessary to simplify porting of the application to a different hardware or software infra-structure as well as to enable incremental development of both application and user interface.

Several schemes have been proposed for a clear separation of user interface and functionality. Screen editors are a classic example [11]. An editor typically consists of the following components: