

Design Alternatives for User Interface Management Systems Based on Experience with COUSIN¹

Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner

Computer Science Department,
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

User interface management systems (UIMSs) provide user interfaces to application systems based on an abstract definition of the interface required. This approach can provide higher-quality interfaces at a lower construction cost. In this paper we consider three design choices for UIMSs which critically affect the quality of the user interfaces built with a UIMS, and the cost of constructing the interfaces. The choices are examined in terms of a general model of a UIMS. They concern the sharing of control between the UIMS and the applications it provides interfaces to, the level of abstraction in the definition of the information exchanged between user and application, and the level of abstraction in the definition of the sequencing of the dialogue. For each choice, we argue for a specific alternative. We go on to present COUSIN, a UIMS that provides graphical interfaces for a variety of applications based on highly abstracted interface definitions. COUSIN's design corresponds to the alternatives we argued for in two out of three cases, and partially satisfies the third. An interface developed through, and run by COUSIN is described in some detail.

1. Introduction

A large amount of recent work in human-computer interaction has been devoted to tools to support the design and implementation of user interfaces. A class of such tools, often called *user interface management systems* (UIMSs), is based on the idea that the user interface portion of an application program can be separated from the portion implementing its functionality. UIMSs provide facilities which allow the user interface portion to be specified at a level considerably abstracted from the primitive operations needed to realize it. The actual user interfaces are then provided automatically by the UIMSs on the basis of these abstract specifications.

¹This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM 0-89791-149-0/85/004/0169 \$00.75

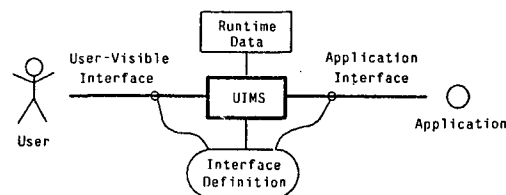


Figure 1: Architecture of a UIMS.

The architecture of most UIMSs can be described in terms of the block diagram shown in figure 1. A UIMS provides a complete user interface between an application and its user by supporting two communication channels — a *user-visible interface* which is the interface the user actually interacts with, and an *application interface* which the application uses as its channel of communication to the outside world. The UIMS bridges the gap between these two interfaces, acting as an intermediary between user and application. Both interfaces are specified (jointly or separately) in an *interface definition*, usually declarative in nature and external to the application. The block labeled *runtime data* represents the information that the UIMS maintains about the current state of the interaction between a particular user and application.

The advantages of providing user interfaces through UIMSs have been discussed at length elsewhere (e.g. [2, 3, 7]). They concern both the quality and the ease of construction of the resulting user interfaces. We summarize the main advantages below:

- **Less effort in interface construction:** Unlike ordinary programming languages, the interface definition language is typically based on abstractions of interface behavior. This allows UIMS interface definitions to be more concise and less complex than programming the interface directly.
- **More intelligent and supportive interfaces:** A corollary to the first point is that effort saved in the construction of interfaces to individual applications can be pooled in the construction of the UIMS to incorporate a high level of interface services, error correction, automatic help, etc.
- **Easier human factors involvement in interface design:** If the interface is defined externally to the application, the interface need not be specified by the application programmer, but can be handled by a person more skilled in human factors. The human factors specialist can also modify and experiment with the interface independently of the application programmer.
- **Consistency of interfaces across applications:** Since the low-level details of interaction are all handled by the same UIMS, they are the same for all applications.

- **Multiple interfaces for a single application:** If the interface definition is separate from the application, it becomes possible to have multiple user-visible interface definitions for a single application. This can be used to handle different types of interface hardware, non-interactive (shell script) use of applications, different user skill levels, and individual user preferences.

The advantages listed above apply to the UIMS concept in general. However, the design space for UIMSs is large and not all design choices are likely to offer the optimum realization of these benefits. This paper deals with three sets of design alternatives for UIMS:

- control architecture
- level of abstraction in I/O content definition
- level of abstraction in I/O sequencing definition

Subsequent sections consider these three design dimensions separately. In each case, we will argue in favor of a specific choice from the set of alternatives. We illustrate our arguments by references to existing UIMSs, particularly the Cousin UIMS [2, 3], or classes of UIMSs, particularly ATN-based UIMSs (e.g. [1, 4, 9, 11]). We go on to examine the Cousin UIMS in some detail, classifying it according to the space of design alternatives we identify, and presenting an interface implemented through it. Cousin follows the design choices we argue for in two out of three cases, and partially satisfies the third.

2. UIMS control architecture

A major design decision for UIMSs concerns the division of control of the dialogue between the UIMS and the application. Following Tanner and Buxton [7], we identify three control structure choices:

- With *internal* control UIMSs, control always resides in the application. The UIMS is viewed as a subroutine package to be called at appropriate places in the application to perform the necessary communication with the user.
- In *external* control UIMSs, the situation is reversed. The application is viewed as a set of subroutines, and the task of the UIMS is to interact with the user and invoke the appropriate application subroutines at appropriate times.
- A *mixed* control UIMS allows control to reside in both the application and the UIMS. As in the external control case, the application can be viewed as a set of subroutines to be called by the UIMS at the appropriate time. However, in the mixed case, the application can take control of the dialogue and request or present data from or to the user through the UIMS. With mixed control, the UIMS and application can be coroutines or independent processes.

The remainder of this section examines the three alternatives in more detail. It concludes that the alternatives are listed in ascending order of desirability.

2.1. Internal control UIMS

The internal control approach to UIMS architecture is the simplest and most widely used. An internal control UIMS is essentially a collection of subroutines which perform I/O tasks (e.g. [6]). The tasks may vary in complexity from input of a single character to drawing and obtaining a selection from a menu of strings. However, they correspond to atomic I/O actions as far as the application using the UIMS is concerned. The definition of a user interface to a given application for such a UIMS merely consists of calls to the appropriate subroutines embedded at suitable points in the code of the application.

The primary disadvantage of this kind of UIMS control architecture is the lack of separation it allows between the interface definition and the application. Since the interface definition is embedded in the code of the application, it is not possible for an interface designer to construct or modify an interface without also changing the application. This makes it very hard in practice to involve a human factors expert or anyone other than the application programmer in the development of the interface. The lack of separation also makes it very hard to have multiple interfaces to the same application, or to use applications in both interactive and non-interactive contexts.

Interface consistency across a variety of applications or even within a single application is difficult to achieve with internal control UIMSs. For even minimal consistency, the I/O actions provided by the UIMS must be at a high level of abstraction, so that, for instance, all the details of a menu selection are handled within a menu-selection subroutine and will therefore be the same for all menu selections.

Finally, in the internal control architecture, the UIMS services are always very local in scope, concerned with obtaining a particular piece of input or giving a particular piece of output. This locality often introduces unnecessary modes into an interface. For example, suppose one of the subroutines comprising an internal control UIMS asks the user a yes/no question. Such a subroutine would probably never return until the user answered yes or no. Thus, once an application had asked a yes/no question via this subroutine, it would be impossible for the application to respond to any other input from the user. In particular, it could not respond to help requests, even though the information requested might help the user to answer the yes/no question.

2.2. External control

The external control architecture for a UIMS resolves the problems noted above for the internal control paradigm. Most importantly, it clearly separates the interface definition from the application. The application does not control the user-visible interface directly, but is modeled as a set of subroutines which can be invoked by the UIMS at appropriate points during the interaction with the user. This allows the user-visible interface to be developed independently of the application, possibly by a human factors expert different from the application programmer, and makes it possible to have multiple user-visible interfaces to the same application. Also, a suitably high level of abstraction in the interface definition language will lead to interface uniformity across applications. Finally, since an external control UIMS has access to the global context of the interaction, it is not subject to the scope constraints mentioned above for internal control architectures.

The most common style of external control UIMS reported in the literature (e.g. [1, 4, 9, 11]) uses augmented transition networks (ATNs) [10] as an interface definition formalism. To a first approximation, these systems operate as follows. Nodes of the network defining an interface correspond to different states or modes of the interface. Arcs linking nodes have one or more input events, output events, or application actions associated with them. The network is interpreted by the UIMS starting at some designated initial node and repeatedly traversing arcs to respond to input events, generate output events, or initiate application actions as appropriate.

While resolving the difficulties associated with internal control architectures, external control UIMSs have their own problems. The fundamental problem is that for the external control model to work properly, application actions must be truly atomic from the point of view of the user-visible interface. The application cannot communicate with the user during one of its actions without bypassing the UIMS and thereby ruining the UIMS's effectiveness as a mediator of the interaction.

Directing all communication through an external control UIMS means that all possible exchanges of information with the user must be foreseen and catered for in the interface definition. Given the range of problems that can arise for, say, a file transfer application (e.g. remote host down, incorrect password, etc.) or the amount of feedback the application needs to provide (e.g. print a character after every so many blocks transferred), the complexity of a complete interface definition would be very great. Imagine, for instance, an ATN network containing arcs for all possible outputs and for all possible user responses to those outputs, and so on.

2.3. Mixed control

The mixed control architecture is similar to the external control architecture, except that the application is no longer viewed as a set of standard subroutines. Rather, it is a set of modules which in addition to returning in a normal way to the calling UIMS, can return intermediate results reflecting error conditions or feedback, can make requests for additional information, and can also initiate communication with the UIMSs asynchronously (e.g. announcing the arrival of a mail message). These communication events in the application interface channel may result in interactions in the user-visible interface channel. This arrangement resolves the main objection to external control since the application's actions no longer need to be atomic.

The problems with internal and external control architectures described above make mixed control a highly attractive option. However, as we shall see in the next section, adopting a mixed control model places certain demands on the interface definition language of a UIMS.

3. Level of abstraction in I/O content definition

The second design alternative we consider concerns the kind and level of abstraction provided by the UIMS interface definition language for describing the information communicated between user and application. We distinguish two alternatives:

- **surface level** abstractions in which the communication is specified in terms of events at the user-visible interface. The events used in such specifications may be at varying levels of abstraction, ranging from primitives of the underlying graphics or other I/O package (e.g. picks, valators, strings) to more complex I/O events (e.g. menu selections, bargraphs, string editing buffers).
- **application level** abstractions in which the communication is specified in terms of objects or information that the application needs to obtain from or output to the user. An interface definition at this level may specify that a file description be output or a selection from a list of strings be obtained from the user, but it does not specify the I/O events at the user-visible interface necessary to achieve that exchange of information.

The following example illustrates the consequences of each alternative. Suppose the designer of the user interface for a file management application wanted to display information about a set of files in a scrollable table, and also wanted the user to be able to switch back and forth between a short format in which only file names were displayed and a long format in which size, creation date, and access rights were also displayed. The representation of a file in the application might be a record whose components are the name, size, creation date, access rights, etc. of the file. The application starts off with a set of instances of the record; somehow, appropriate parts of the contents of these records must be translated into a graphical representation and output to the user-visible interface. With an application level UIMS, the application would simply pass the record instances to the UIMS and expect it to take care of the details of displaying them. This implies that the interface definition would have to contain a

mapping between the application and surface level representation of a file. With a surface level UIMS, the application would have to do the translation from record instance to graphical objects itself, and request that the surface version be displayed.

The translation is not always trivial, especially for high resolution displays in which many details can be controlled. Using an application level UIMS, therefore, shifts work from the application to the UIMS, and more importantly, it specifies yet another aspect of the user interface behavior externally to the application itself. Note also that more than one mapping from application level to surface level can be contained in the interface definition. For instance there could be mappings from a file record into both the long and short representations for files. This means that the long vs. short format switching mentioned above could be handled purely within the UIMS without any need for involvement by the application.

Another advantage of application-level interface definitions is that they provide a better framework for integrating applications in a computing environment. In order to integrate two applications it is necessary for one of them to interpret the output of the other. The translation from the application level to the surface level representation generally combines the original data with formatting and other information (e.g. titles, units of numerical quantities, etc.) making it harder for other applications to interpret the data. In addition, applications which operate at the surface level generally expect to receive input in the form of keystrokes, and mouse movements. Integrating with such an application would require that the output from other applications be converted to the appropriate surface level notions (e.g. mouse movements and keystrokes). Performing such a conversion would be very difficult, at best.

4. Level of abstraction in I/O sequencing definition

The third and final design alternative we wish to consider also relates to the degree and kind of abstraction in the interface definition language, but concerns the ordering of I/O events rather than the content of the events themselves. Again, we identify two alternatives:

- **explicit** I/O event ordering. Here the interface definition explicitly specifies what sequences of I/O events constitute a valid dialogue between user and application. The interpretation of each user input and the corresponding system response is determined by the preceding dialogue context.
- **implicit** I/O event ordering. Here the interface definition specifies sets of I/O events without mentioning a specific ordering. Any ordering restrictions are implicit in the UIMS that interprets the definition.

ATN-based UIMSs and the Cousin UIMS are representative of the two approaches. ATN-based UIMSs use explicit I/O sequencing: an interface definition for an ATN-based UIMS specifies the possible dialogues between user and application for that interface through the topology of the ATN network. Each node of the network effectively defines a mode of the interface. The arcs leading from each such node specify the acceptable inputs in that mode, their treatment in the context of that mode, and the mode into which the system should go next.

Implicit I/O event ordering offers two main advantages over explicit ordering. The first concerns the specification of modeless interfaces, and the second relates to the handling of errors and other exceptional conditions.

Modeless interfaces can reduce cognitive load and offer the user maximum flexibility of action. The power of modelessness has been demonstrated through several excellent commercially

available interfaces (e.g. [5, 8]). With implicit I/O event ordering, it is straightforward for the UIMS to provide a modeless style by default (e.g. allowing the user to specify the parameters to a command in any order). With explicit I/O ordering, modelessness has to be "programmed" into the interface definition. In the case of ATN-based systems, this would mean very distorted networks with disproportionately few nodes (since they correspond to modes) and disproportionately many arcs. Such networks would be hard to construct and maintain.

We have already discussed in section 2.2 the various kinds of exceptional conditions that can arise during actual execution of an application. With the implicit control paradigm, the handling of such events in the user-visible interface can be specified independently of the mainstream (error-free) dialogue between the user and application. With explicit I/O ordering, all such events must be foreseen and dialogue paths to deal with them incorporated into the mainstream interface definition. The situation can easily become unmanageable, since all exceptional events must in effect be anticipated at all points of the dialogue.

Some attempts have been made to resolve these problems with explicitly ordered interface definitions. Jacob [4], for instance, suggests an extension to an ATN-based UIMS in which certain arcs are assumed by default to emanate from all nodes of an interface definition. These default arcs would then handle exceptional conditions or non-mainstream input from the user. Such modifications can be seen as a method of incorporating a limited degree of implicit I/O event ordering into a scheme fundamentally based on explicit ordering.

5. Design Choices taken in the COUSIN UIMS

In this section, we turn from consideration of the overall space of design choices for UIMSs to a specific UIMS — the COUSIN system (for cooperative user interface). COUSIN provides many of the advantages described in section 1 for the construction of graphical interfaces to a variety of applications in the context of a powerful personal workstation. We first locate COUSIN in the design space we have just established and then in the next section examine COUSIN in more detail through a specific interface implemented using it.

In terms of control structure, COUSIN implements the mixed control paradigm, running as an independent process from the applications it provides interface services to. In terms of interface abstraction, COUSIN provides implicit I/O event ordering, but requires I/O content to be specified at a level somewhere between the surface and application levels described in Section 3.

COUSIN's interface definition language is based on a highly abstract model of communication between user and application. In this model, communication takes place through a set of value-containing slots — one slot for each piece of information the user and application need to exchange. A simple print application might have slots for its parameters — file to print, number of copies, etc.. A more complex application, such as the file manager described in Section 6, would have slots for commands and feedback in addition to parameter slots. Since both COUSIN and the application can access and modify slots at any time, the control architecture is mixed.

An interface definition in COUSIN's slot/value-based language suffices to specify both the application and user-visible interfaces which are part of the general UIMS model. The application interface is a message-based, interprocess communication channel that allows the application to access and update slot values. The user-visible interface is a graphical form displayed on a bit-map display, which allows the user to see slot values, and modify them using a mouse and a keyboard. Fields in the form correspond to slots in the interface definition. Details of both interfaces are specified as a set of slot attributes (see figure 6 in

section 6). The attributes for a given slot describe, among other things, the type (enforced by COUSIN) and default for the slot's value, how the application can find out about changes to the slot's value, and the appearance and behavior of the corresponding field in the user-visible form.

Implicit I/O event ordering comes naturally with COUSIN's model of communication. The user can, at any moment, modify fields in the user-visible form interface, ask for help, or use any of the other facilities COUSIN provides. COUSIN's interface abstractions are also closer in spirit to application rather than surface level specification of I/O content in that the set of slots with their types specify what information is to be exchanged by user and application rather than the I/O events needed to exchange it. However, the set of slot types supported by COUSIN does not cover a complete range of application level objects. Only primitive types such as strings, integers, and selections from sets of strings are supported; record structures, for example, are not supported.

We have found the slot/value abstractions of COUSIN's interface definition language appropriate for a broad class of applications requiring coarse-grained command interaction. Examples we have implemented include file management, electronic mail, process management, and file transfer. However, COUSIN's interface abstractions are not suitable for all applications. In particular, they do not fulfill the communication needs of applications requiring finer-grained interaction such as text editors or drawing packages.

6. An example COUSIN interface

This section examines a specific interface that we have constructed using the current implementation of COUSIN. This example should clarify the nature of COUSIN's interface definition language, and the way in which COUSIN uses it to mediate communication between user and application. The interface we will examine is for a File Management System (FMS). FMS allows a user to browse through the files in a hierarchical file system and to delete, copy, or rename them, individually or in groups. Figure 2 shows the user-visible interface for the COUSIN interface to FMS. The upper portion of the form is concerned with browsing and contains a table of file descriptions along with fields specifying the files which should be displayed in the table, and how they should be displayed. Below this are command buttons to manipulate the files, along with an area dealing with confirmation for destructive file operations. At the bottom of the form is a display showing the amount of unused file space.

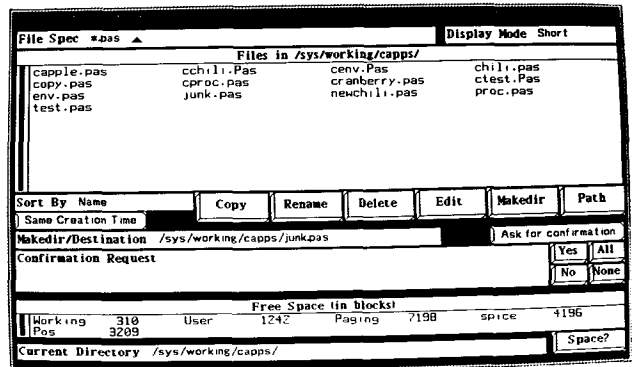


Figure 2: Form for FMS showing a list of files.

To see how FMS is used, let us look at a specific task. Suppose we want to delete some out of date Pascal files. We first specify the list of files to be displayed in the Files field by typing a pattern into the File Spec field, in this case "**.pas". FMS responds (figure 2) by displaying in the Files field all the files in the current

directory which end in .pas (i.e. all the Pascal files). We then select the files to be deleted by clicking at them with the mouse. If there are too many files to fit in the field, the contents of the field can be scrolled up and down using the scrollbar at the left edge of the field, or using keyboard commands.

Once we are satisfied with the selection of files, their actual deletion is started by pressing the Delete button with the mouse. The first press of the Delete button highlights the Files and Ask for Confirmation fields, which are the parameters to this command. If any parameter field has incorrect values (e.g. no files were selected in Files) the incorrect field and the button are highlighted in reverse video (figure 3). All incorrect fields must be corrected before a command can be executed. When all of the relevant fields are correct the Delete button is highlighted with a heavy border to indicate that it may be pressed a second time to execute the command.

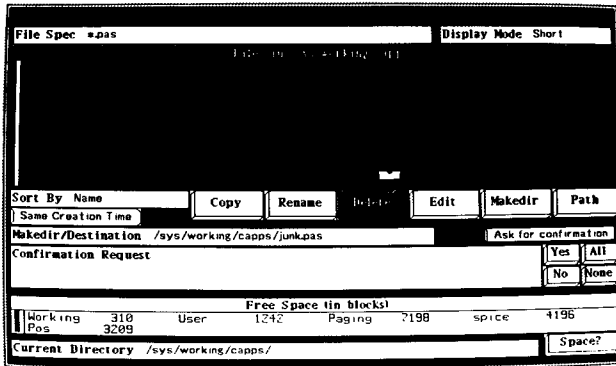


Figure 3: Pressing the Delete button without selecting any files.

At this point, however, we are not committed to issuing a Delete command. We may, if we wish, alter a parameter field (e.g. Files, or Ask for Confirmation), an unrelated field (e.g. DisplayMode), or issue a different command (e.g. Rename). Suppose we push the Delete button again: as shown in figure 4, FMS asks us to confirm the operation, by placing a question in the Confirmation Request field. We reply by pressing one of the Yes, No, All or None reply buttons on the right hand side of the form.

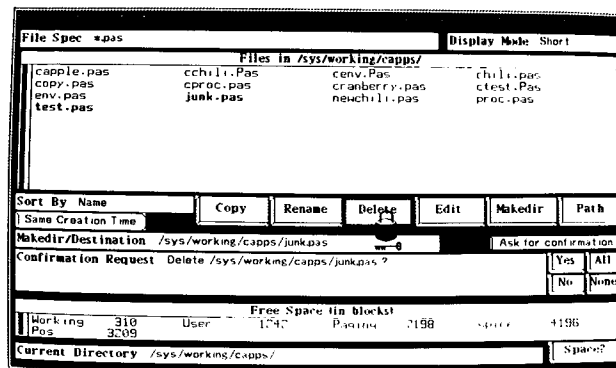


Figure 4: Two files selected, the Delete button pressed, and confirmation requested.

The other fields in the form are used to perform various operations. The Sort By field, for example, controls the order in which the list of files is displayed. This field may be set to one of Name, Last Change Date, Last Access Date, and Size. Since this field has a limited enumeration of values, we may either type the value into the field (with keyword completion and spelling correction), or select the value from a pop-up menu (figure 5).

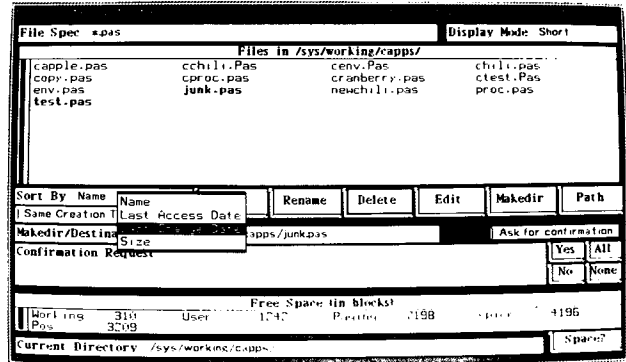


Figure 5: Changing the Sort By field with a menu.

At any given moment, we may request help. For example, pressing the Help key when the mouse is over the DisplayMode field, causes a help message with a brief explanation of the purpose of the DisplayMode field to appear².

It is important to note that in all these interactions, COUSIN takes care of form display, field scrolling, field editing, selection of table entries, pop-up menus, and help displays, without any involvement by FMS. The only task FMS has to perform is to update the relevant slots (e.g. Files and Confirmation Request) response to changes in other slots (e.g. Delete, File Spec, Sort By, and Display Mode).

A portion of the Cousin interface definition from which the FMS form is derived is shown in figure 6. All interface definitions have the same format: they contain a header, and an entry for each slot. Each slot is described in an attribute/value formalism, with attributes corresponding to the various aspects about slots needed by COUSIN to construct the application and user-visible interfaces. Figure 6 shows the slot attributes for the Files and Sort By slots.

```
[
  FormName: "File System Manager"
  Purpose: "Utility for browsing and managing files"
]

[
  Name: Files
  ValueType: String
  DefaultSource: NoDefault
  MinNumber: 1
  MaxNumber: 1000
  InteractionMode: Table
  NumColumns: 4
  ChangeResponse: Passive
  Purpose: "List of files specified by File Spec"
]

[
  Name: "Sort By"
  ValueType: String
  DefaultValue: Name
  InteractionMode: CycleButton
  ChangeResponse: Active
  EnumeratedValues: (Name, "Last Change Date", "Last Access Date", Size)
  Purpose: "Specifies the sorting order for the files table"
]
```

Figure 6: Partial Interface Definition for FMS.

² A more sophisticated help facility exists for Cousin. It uses the representation of the state of the interaction, and the interface descriptions to semi-automatically generate more comprehensive help.

The **ValueType**, **MinNumber**, **MaxNumber**, and **EnumeratedValues** attributes define the type of value that can go in the slot, the number of values the slot can have, and in some cases, an enumeration of the possible values. These attributes constitute the application level description of the slot.

The description of the user-visible interface is centered around the **InteractionMode** attribute, which defines how the field corresponding to a slot is displayed and how it can be manipulated. This attribute defines a mapping between the application level representation of data and a surface level representation. In effect, each interaction mode defines a display format for the corresponding values, and also defines the meaning of the low level user inputs (keystrokes and mouse movements).

There are five basic interaction modes — *push button*, *cycle button*, *text*, *table*, and *window*. *Push buttons* (e.g. **Delete** and **Rename**) look like physical buttons in a home appliance, and can be pushed by clicking at them with the mouse. *Cycle buttons* (e.g. **Ask for Confirmation**) display one of an enumeration of values; the next value in the enumeration will be displayed when the mouse is clicked over the field. *Text fields* (e.g. **Sort By** and **File Spec**) display values in a text buffer, and can be directly edited by the user. *Tables* (e.g. **Files**) are two dimensional menus, and *windows* provide applications with a sub-area of the form in which they can perform graphics operations which are not supported by COUSIN.

As discussed in Section 5, a COUSIN interface definition specifies the application interface as well as the user-visible interface. In essence, FMS's interface to COUSIN is the set of value-containing slots named in the interface definition. However, the precise method by which an application interacts with a slot is determined by the slot's **ChangeResponse** attribute — either *active* or *passive*. The application is asynchronously notified whenever a new value is placed into an *active* slot, but must ask COUSIN for the value of a *passive* slot. A *push button* is normally *active* so that the application is notified when it has been pressed. Occasionally, other fields are also designated *active*. For example, the **File Spec** field is *active* so FMS is notified when the user changes it, allowing it to update the **Files** slot with the corresponding files. This mechanism can also be used to allow the application to do specific type checking which COUSIN is not programmed to handle, such as checking whether a value represents a valid mail address in a mail application.

```

Procedure DoFileSpec;
begin
  DispMode := GetValue(form, DisplayMode);
  SortMode := GetValue(form, SortBy);
  ... compute enumeration for files table ...
  SetEnumeration(form, Files, enum);
end;

begin {Main routine}
  InitCousinInterface;
  while True do
    begin
      WaitForActivation(form, field);
      case field of
        FileSpec:   DoFileSpec;
        Copy:       DoCopy;
        Delete:     DoDelete;
        Rename:     DoRename;
        Edit:       DoEdit;
      end; {case}
    end; {while}
  end.

```

Figure 7: Code fragment for FMS.

The simplicity of COUSIN's application interface means that little code in the application itself is needed for interface purposes. The code fragment in Figure 7, from FMS, is typical of applications using COUSIN. After initializing its interface with COUSIN (**InitCousinInterface**), an application waits for an activation message from COUSIN (**WaitForActivation**). When one is received, the application executes the routines appropriate to the activated slot. If parameters are needed, remote procedure calls to COUSIN are used to obtain the value of the slots containing the parameters. Other calls can be used to set the values of slots with any results the application needs to communicate to the user. For example, the **DoFileSpec** procedure is called, when FMS receives an activation on the **File Spec** slot; it reads the values of the **Display Mode** and **Sort By** slots and updates the enumeration of the **Files** slot with the appropriate list of files. This causes COUSIN to redisplay the **Files** field, enabling the user to see the new files.

7. Summary and future plans

The UIMS approach to interface construction offers major benefits in terms of reduced implementation effort, easier involvement of human factors expertise, greater consistency across applications, and an overall higher level of interface quality. In this paper, we examined the impact of three major design choices for UIMSs on the realization of these potential benefits. We argued in favor of UIMSs with a mixed control structure, application-level specification of I/O events, and implicit specification of I/O event sequencing. The COUSIN UIMS incorporates the first and third of these design choices along with a limited version of the second.

COUSIN runs on the Perq — a powerful personal workstation with a high-resolution bit-map display. We have constructed interfaces for several applications including a file transfer program, a mail program, a version control system, and a process manager, as well as the file management application described above. To date, no formal assessment has been made of any of these interfaces, but user reaction has been largely positive.

Despite the favorable reactions to COUSIN, our experience with the interfaces constructed through it suggests there is still considerable room for improvement in COUSIN's UIMS model. Areas we have targeted for attention include:

- **Complete support for application level I/O content specification:** Some of the arguments we gave in Section 3 for preferring application level specifications over surface level specifications were based on negative experiences with COUSIN in this regard. For instance, with application level definitions, it would be possible to handle the switch between long and short format in the **Files** field of FMS internally to COUSIN and not require the application to recalculate the strings to be displayed each time the user wants to change format.
- **Providing more control over details of the user-visible interface:** Although basically adequate for the communication needs of coarse-grained command interaction, straightforward use of COUSIN's current interface abstractions does not always produce an interface that allows the user to do things in the most rapid or natural way (e.g. changing to a new directory in the file management system simply by pointing to its name). To provide conveniences of this kind to the user, finer-grained control of both the layout and interaction details of the user-visible interface is required.

- **Interactive interface specification:** Although COUSIN makes it much easier to construct interfaces than programming them from scratch, we have found the interface development procedure still to be much slower than we think appropriate or necessary. We currently support this task with some simple interactive tools, but much more interactive methods of interface definition are required. Such tools would also be useful for the end user to personalize the user-visible interface.

Overall, our experience with the COUSIN implementation supports our more abstract arguments about the various design choices for UIMSs. We find that the mixed control architecture, implicit I/O sequencing and (close to) application level specification of I/O content allow us to construct high quality graphical interfaces at relatively low cost. Resolution of the problem areas identified above should improve quality and reduce cost still further.

References

1. Bleser, T. and Foley, J. D. Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces. Proceedings of Conference on Human Factors in Computer Systems, Gaithersburg, Maryland, March, 1982, pp. 309-314.
2. Hayes, P. J. Executable Interface Definitions Using Form-Based Interface Abstractions. In *Advances in Computer-Human Interaction*, H. R. Hartson, Ed., Ablex, New Jersey, 1984.
3. Hayes, P. J. and Szekely, P. A. "Graceful interaction through the COUSIN command interface." *International Journal of Man-Machine Studies* 19, 3 (September 1983), 285-305.
4. Jacob, R. J. K. An Executable Specification Technique for Describing Human-Computer Interaction. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed., Ablex, New Jersey, 1984.
5. Macworld. The Macintosh Magazine. April, 1984.
6. Shafer, S. A. C Shell Command Interpreter. In *Unix Programmer's Manual (CMU additions)*, Bell Labs, 1983, ch. 3.
7. Tanner, P. and Buxton W. Some Issues in Future User Interface Management System (UIMS) Development. IFIP Working Group 5.2 Workshop on User Interface Management, Seheim, West Germany, November, 1983.
8. Teitelman, W. A Display Oriented Programmer's Assistant. Proc. Fifth Int. Jt. Conf. on Artificial Intelligence, MIT, August, 1977, pp. 905-915.
9. Wasserman, A. I. and Shewmake, D. T. The Role of Prototypes in the User Software Engineering (USE) Methodology. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed., Ablex, New Jersey, 1984.
10. Woods, W. A. "Transition Network Grammars for Natural Language Analysis." *Comm. ACM* 13, 10 (Oct. 1970), 591-606.
11. Yuntan, T. and Hartson, H. R. A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed., Ablex, New Jersey, 1984.