Standardizing the Interface Between Applications And UIMSs

Pedro Szekely[*]

USC/Information Sciences Institute
4676 Admiralty Way,
Marina del Rey, CA 90292
(213) 822-1511

Abstract

The user interface building blocks of any User Interface Management System (UIMS) have built-in assumptions about *what* information about application programs they need, and assumptions about *how* to get that information. The lack of a standard to represent this information leads to a proliferation of different assumptions by different building blocks, hampering changeability of the user interface and portability of applications to different sets of building blocks. This paper describes a formalism for specifying the information about applications needed by the user interface building blocks (*i.e.* the *UIMS/Application interface*) so that all building blocks share a common set of assumptions. The paper also describes a set of user interface building blocks specifically designed for these standard UIMS/Application interfaces. These building blocks can be used to produce a wide variety of user interfaces, and the interfaces can be changed without having to change the application program.

* The initial research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the: Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. Further work has been supported in part by the D.A.R.P.A. under contract number MDA 903-86-C-0178 and in part by the Air Force Logistics Command under contract number F33600-87-C-7047. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the office policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Logistics Command, or the US Government.

1 Introduction

The main goals of user interface management systems (UIMS) are to provide reusable interface building blocks that can be used to construct a wide variety of interfaces, and to facilitate composing and refining user interfaces using these building blocks. These building blocks have built-in assumptions about *what* information about application programs they need, and assumptions about *how* to get that information. The lack of a standard to represent this information leads to a proliferation of different assumptions by different building blocks, hampering changeability of the user interface and portability of applications to different sets of building blocks.

Most modern UIMSs communicate with application programs using external control[4, 7, 10, 11]. This means that the UIMS receives all input events, dispatches them to the appropriate user interface building block, which then calls application procedures when a response from the application is required. For instance, if the user clicks the mouse over a menu item, the UIMS sends the click event to the relevant menu building block, which highlights, and then calls an application procedure to execute the relevant action.

The procedures called by the user interface building blocks are usually called ***call-back procedures***, because once an application program starts, it gives control to the UIMS, which then calls the application *back*, when appropriate. The set of all call-back procedures is called the *UIMS/Application interface*. There are some variants of the UIMS control technique in which the user interface building blocks can directly access application data structures. Since the call-back procedure method is the most popular, straight forward and efficient technique [9], this paper assumes that UIMS/-Application interfaces consist solely of call-back procedures.

Writing an application for a UIMS that uses UIMS control consists of choosing the building blocks that will be used for the various user interface features, and then writing the call-back procedures [7]. The call-back procedures are dependent on the particular building block which will use them. Each kind of building block specifies how many call-back procedures it has, what the parameters for these procedures should be, and what the procedures should do when called.

UIMS/Application interfaces become highly dependent on the particular set of building blocks used in a program's user interface. Changing a building block for another (*e.g.* a pop up menu by a row of command buttons) could be troublesome because different building blocks could have different expectations about the results or parameter interpretation of their call-back procedures. Changing building blocks could require modifying the call-back procedures to fit the expectations of the new building block. This impedes the iterative refinement cycle required to build good interfaces, and hampers portability of applications to different sets of building blocks.

The dependencies between application programs and user interface building blocks can be minimized by standardizing the characteristics of the procedures in a program's UIMS/Application interface. It is probably impossible to define a standard for UIMS/Application interfaces that supports every imaginable style of user interface. However, this paper shows that a surprisingly simple standard supports a wide variety of graphical interface features. The standard for UIMS/-Application interfaces presented here is based on representing as objects the operations of a program, which are typically represented as procedures. The object representation of operations makes it easy to associate with the operation a variety of information needed for the user interface.

This paper is based on a new UIMS called Nephew [13], which is a descendent of Cousin [3] The paper has two main parts. The first part (section 2) describes the Nephew UIMS/Application interface, which defines a set of conventions for standardizing UIMS/-Application interfaces so that they can be used by a large variety of interface components. The second part (section 3) describes the interface components provided by Nephew. The components are structured so that they all use the same call-back procedures, but they can produce a wide variety of graphical interfaces. Sections 4, 5, and 6 contains some examples of Nephew interfaces, related work and conclusions.

2 The Nephew UIMS/Application Interface

The UIMS/Application interface for an application program defines the procedures that the user interface building blocks need to acquire the necessary information to display, procedures to inform the program about user actions, and type declarations that define the parameters of the procedures.

Nephew UIMS/Application interfaces are similar. Nephew provides two construct called **def-operation** and **def-object** to specify the procedures and objects of the application.

The **def-object** construct to define the application objects is similar to that of any object-oriented language, and hence is not discussed here. The only point worth mentioning is that objects only provide procedures to read the state of the object. The procedures to change objects are defined with **def-operation**.

The **def-operation** construct is different from the traditional facilities to declare procedures. **Def-operation** requires, in addition to the traditional declaration of the parameter types, the declaration of a host of other attributes that define the information needed by the user interface. Here is a detailed description of the **def-operation** construct:

:**precondition** – a function of no parameters that returns **nil** when the precondition is satisfied, or an object describing why the precondition is violated. The default returns **nil** (*i.e.* precondition satisfied).

:**cancelable** – a function of no parameters that returns **nil** if the operation can be canceled, or an object describing why the operation cannot be canceled. The default returns t (*i.e.* operation cannot be canceled).

:**preview** – a function that takes as many parameters as the operation has inputs, and returns an object, which the user interface interprets as an approximation of the effects of the operation, if the operation was called with the given set of inputs. The default returns **nil**.

The following information must be supplied for each operation input:

:**type** – the type of value acceptable as input.

:**validation** – a function of two parameters. The first one is the value to be validated and the second one is an object that can be queried for the values of the other operation inputs. The function should return **nil** when there are no errors, or a value describing the error. The default returns **nil** (*i.e.* valid input).

:**transformer** – a function of two parameters, the value to be transformed, and an object that can be queried for the values of the other operation inputs. The function returns the object into which

```
(def-operation move-piece-op
        :precondition is-game-not-over
        :inputs ((piece :type Piece
                        :validation is-movable-piece
                        :generator movable-pieces)
                 (location :type Location
                           :validation is-valid-location
                           :generator landing-locations))
        "body of move-piece-op")

(def-operation quit-op ...)
(def-operation save-op ...)

...
```

Table 1: Fragment of the Nephew UIMS/Application interface of a chess program. **Move-piece-op** moves a piece to a new location, and computes the program's response move. **quit-op** quits the program, and **save-op** saves the state of the game so that it can be resumed later. Other operations provided are not shown in the figure.

the parameter value is transformed. The default is the identity function.

:generator – a function of one parameter, an object that can be queried for the values of the other operation inputs. The function returns either a list of objects that are legal values for the input, or a function of no parameters that returns the next legal value each time it is called. The default returns nil.

:request – a boolean value. True means that this input should be supplied before the operation is executed, and nil means that the input might be requested during the execution of the operation.

Note that the Nephew UIMS/Application interface defines the parameters and return values for the procedures that fill the various attributes. This ensures that all user interface components will be able to use these procedures, and also allows the programmer to write these procedures before the user interface designer chooses the user interface components.

The **type** is the only attribute specified in traditional UIMS/Application interfaces. The other attributes, if specified at all, are specified as top-level procedures, at the same level of the operations. Since traditional UIMS/Application interfaces do not enforce any regularity in the specification of these procedures, there is no assurance that different user interface building blocks will be able to use the procedures.

Table 1 shows a fragment of the UIMS/Application interface for a chess program implemented with Nephew. The first call to **def-operation** defines the **move-piece-op** operation, which moves a piece to a new location, and computes the program's reply to the move. **Piece** and **location** are the inputs to **move-piece-op**. The specification of the inputs includes a **type** declaration, and **validation** and **generator** procedures, which the user interface building blocks need (see section 3). For instance, the **is-valid-location** procedure is called by the mouse handler to check whether the current piece can be moved to a location, so that the location can be highlighted accordingly.

Another benefit of the Nephew UIMS/Application interface is that it forces the programmer to declare the purpose of each procedure (*e.g.* **is-movable-piece** is used to validate the piece input to the **move-piece-op**). These declarations allow the user interface building blocks to automatically find the procedures needed to perform specific tasks. The procedures do not have to be explicitly stored by the programmer in each building block, as is necessary in traditional UIMSs.

The next sections describe the user interface components of Nephew. The components are designed to construct graphical interfaces to applications specified with **def-operation** and **def-object**. The sections illustrate how the components make use of the Nephew UIMS/Application interface to support different styles of graphical interfaces and different kinds of semantic feedback.

3 The Nephew Interface Building Blocks

The building blocks of Nephew are designed to be used with programs defined with the **def-object** and **def-operation** described above. Nephew provides three kinds of building blocks:

Commands. A command is an object that manages the dialogue for one operation. Each command stores the operation's inputs, their state and other information relevant to the invocation of an operation (see section 3.1). Commands manage dialogues by interacting with recognizers, which are the objects that receive events from the input devices, and presenters, which display objects to the user.

Recognizers. A recognizer is parser for a complete input gesture such as a mouse click or a mouse drag. While parsing a gesture, recognizers send messages to commands, for instance to set the value of inputs or to invoke the operations.

Presenters. Presenters are similar to Smalltalk Views [5]. They are used to display application objects, commands, and even other presenters.

Commands, presenters and recognizers are implemented as classes in the Lisp Flavors object-oriented programming package. Nephew provides default implementations of these classes, and programmers can define their own subclasses to modify the default behavior.

Figure 1 illustrates the architecture of Nephew applications, by using a chess program as an example. The chess program, like any other program implemented with Nephew, consists of four kinds of building blocks, called commands, recognizers, presenters, and the application objects.

Quit-command and save-command are commands for the quit-op and save-op operations provided by the chess program. The quit-presenter and save-presenter presenters display these commands as icons on the screen. Input events directed at these presenters are handled by the quit-recognizer and save-recognizer. For instance, if the user clicks the mouse over the quit-presenter, then quit-recognizer sends a message to quit-command to execute the quit-op.

The move-command manages the dialogue to invoke the move-piece-op, which allows the user to make a move. The figure shows a board with two pieces, displayed by the two piece-presenters. Each piece-presenter has a move-recognizer to handle drag gestures that start at the piece. While dragging a piece, move-recognizer sends messages to move-command to set the values of the piece and location inputs, and to execute the move-piece-op. Move-command also causes the locations under the mouse to be highlighted when the piece being dragged could be legally moved to that location. The details of how this works are described in section 4, following a more in depth discussion of commands, presenters and recognizers.

3.1 Commands

The structure of command objects is shown in Figure 2. All commands have this structure, but different classes of commands can manage the information in different ways in order to implement different user interface features. The difference between operations and commands is that operations just provide the relevant information (call-back procedures), and the commands manage this information. Commands decide when, how often, and what information to use. Different commands can implement different policies for using this information. In fact, many instances of commands for the same operation can be used simultaneously, each one keeping track of a different invocation of the operation. This is useful, for example, to implement interfaces where the user is allowed to first annotate many objects (*e.g.* messages) with commands to be performed on them (*e.g.* move to folder, delete), and then issue a single "execute" request to execute all the commands.

The command slots have the following meaning:

name – identifies the operation for this command.

active – either true or false. When a command is active, its recognizers are also active, allowing them to compete for input events that can affect the command. When the command is inactive, its recognizers are also inactive (except for the recognizer that activates the command). So, activation is a weak form of prompting, since it just enables the command's recognizers to acquire input. Several commands can be active simultaneously as long as the recognizers do not compete for events in the same area of the screen.

Consider a drawing editor that has a command to draw each type of shape (line, rectangle, polygon, circle, etc). When a drawing command is activated, it activates its recognizers, thus controlling how input events are interpreted over the drawing area. For example, the draw-line command would have a recognizer that waits for two click events, while the draw-polygon command would have a recognizer that waits for several clicks followed by a double click. Since only one drawing command can be active at a time, only one kind of recognizer will be active over the drawing area. So, events over the drawing area are interpreted differently depending on the active command.

executing – is true while the command's operation is executing, otherwise it is false. A presenter can use the information in this slot, for instance, to display an hour glass cursor while a lengthy operation is executing.

preview – the preview contains a value that represents an approximation of the effect that executing the operation would have given the current setting of the inputs. It is computed using the :preview attribute of def-operation.

In interfaces where the inputs to a command are specified by dragging the mouse, the preview can be used to display feedback about what would happen if the operation was executed with the current input values. The preview is useful because, being just an approximation, it can be computed and displayed faster than the actual operation results. For instance, in a drawing editor the shapes might have to be drawn under other shapes, with fancy borders, etc., but the preview could be drawn with xor mode, with thin lines, and hence would be drawn and erased very quickly.

presenters – the presenters displaying this command.

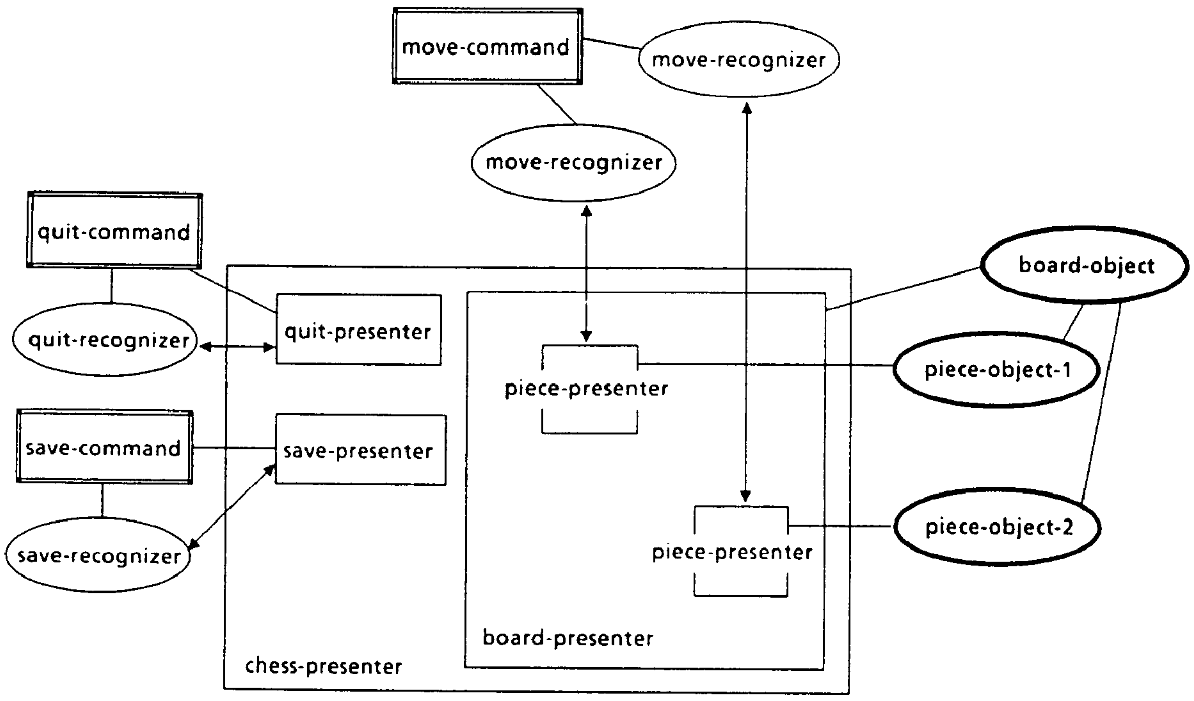

Figure 1: The architecture of a chess program implemented with Nephew.

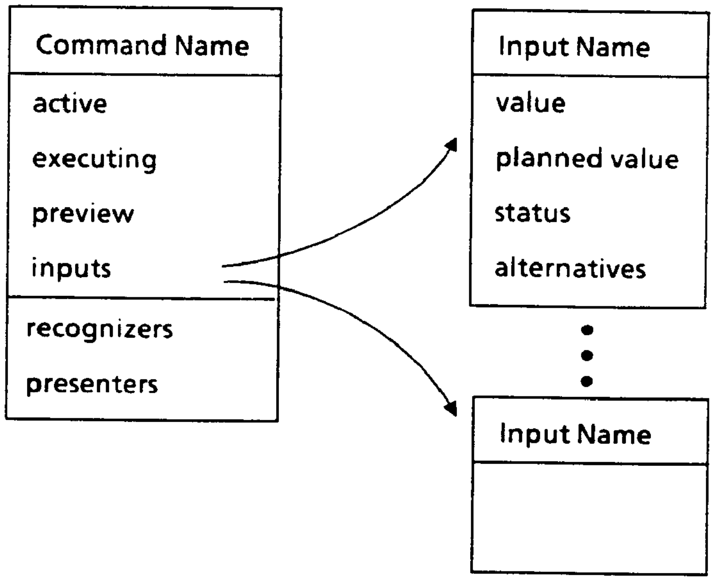

Figure 2: The structure of commands.

They are asked to update their display whenever the command state changes.

recognizers – the set of recognizers that interpret input events to change the information in this command.

inputs – the inputs store information for each of the operation's inputs. Their attributes are the following:

- name – identifies an operation input.
- value – stores the current value of the input.
- status – specifies whether the value is valid. as defined by the :validation attribute of def-operation.
- planned value – stores a value that is being considered as a possible filler for the value slot. The planned value is used to provide feedback while the user is specifying inputs.
- planned status – stores the status of the planned value, which is also computed with the :validation attribute of def-operation.
- alternatives – stores the possible fillers for this input that are consistent with the values of the other command inputs. The alternatives are computed using the :generator attribute of def-operation.

  The alternatives can be used, for instance, to display a menu of the values to fill a field in a form.

Commands classes provide two sets of methods. One set is for recognizers to set the value and planned value of inputs, and to activate, deactivate, execute and cancel (stop execution) commands. The other set of methods is for presenters to query the information in the other slots. Presenters use this information to display semantic feedback during the invocation of an operation.

The Nephew UIMS/Application interface contains all the information needed to compute the slots of commands. Commands determine *when* the information is used, and presenters determine *how* it is displayed. As will be illustrated below, by changing when, how and which command slots are displayed it is possible to construct a wide variety of interfaces styles.

The different behaviors of commands are programmed by defining subclasses of the basic command class provided in Nephew, and overriding methods and defining presenters and recognizers for it. For instance, to cause a menu of alternatives to appear after the user specifies an input incorrectly, the method to set that input is overridden. The new method first calls the default one, then tests the status slot, and if it is invalid, tells a presenter that displays the alternatives to display itself. Additional examples of how commands are used appear throughout the rest of the paper.

3.2 Recognizers

A recognizer is a parser for a complete input gesture such as a mouse click or a mouse drag. Nephew implements recognizers as classes, one class for each kind of recognizer. The behavior common to all recognizers, which includes activating and deactivating recognizers, connecting recognizers to presenters, and getting input events from a global event queue, is implemented in a class called Recognizer-Basic. The response to the input events is implemented by the subclasses listed below:

Click-Recognizer - implements the click gesture. The user interface implementor must specify in the handle-event method the messages to send to its command when the click starts and terminates.

The most common messages are to activate and execute commands, and to set input values. For example, in the chess program the quit-recognizer and save-recognizer are instances of Click-Recognizer. When the user clicks over the quit-presenter, the quit-recognizer sends an activate message to the quit-command.

Drag-Recognizer - implements the drag gesture. The default Drag-Recognizer is programmed to drag an outline of its presenter while the gesture is in progress. The implementor must specify in the handle-event method the messages to send to the command when the drag gesture begins, each time the mouse moves, and when the gesture terminates.

For example, in the chess program the move-recognizers are instances of Drag-Recognizer. Their behavior is described in detail in section 4.

Recognizers are programmed by defining subclasses of the above recognizers and overriding their handle-event method. The code to map screen coordinates into the objects used as input values is included in this method. These values are a function of the object displayed by the recognizer's presenter.

3.3 Presenters

A presenter displays objects, which can be application objects, commands, or other presenters. Nephew implements presenters as classes, one class for each particular way of displaying each class of object. Nephew provides presenter classes to display a variety of generic application classes such as lists, structures and arrays, and also to display commands and even presenters themselves.

In Nephew complex presentations are constructed by connecting several presenter instances to form a tree. Presenters which are children of other presenters are called *sub-presenters*. For instance, in the chess example the pieces are sub-presenters of the board presenter.

The following are the presenter classes in Nephew's library:

Presenter-Basic provides the basic facilities to link presenters to the objects they present, and all the hooks into the graphics package. Presenter-Basic displays its object as a string identifying the type of object (*e.g.* a chess-piece). This is useful in the initial stages of the implementation of a user interface.

Borders-Mixin provides the facilities to define the borders, background and foreground of presenters.

Structured-Presenter provides the facilities for a presenter to have sub-presenters.

Record-Presenter, List-Presenter, Array-Presenter are sub-classes of Structured-Presenter that can present records, lists and arrays. They provide the methods that know how to construct and update sub-presenters for their respective data structures.

Rectangular-Alignment-Mixin provides definitions to align sub-presenters in columns or rows.

String, Icon, Bitmap and Color-Presenter provide commonly used presentations.

The chess program uses many of these classes. For example, the piece-presenters are instances of Icon-Presenter, board-presenter is an instance of Array-Presenter, and it uses the Borders-Mixin to define the borders of the board. The menu containing the save and quit commands is presented using a List-Presenter to present the list of save-command and quit-command, and uses the Rectangular-Alignment-Mixin to align the presentations in a column, left aligned, and with some space between the items.

Redisplay is handled as in Smalltalk, by sending a changed message to changed objects, which causes an update message to be sent to the presenters attached to those objects.

4 An Example

This section contains a detailed example to illustrate how different interface features can be implemented with the Nephew components. The example describes how the dragging of a chess piece in the chess program is implemented, and how semantic feedback is provided while the piece is dragged.

The move-recognizer, which is used to drag a piece is implemented as follows:

Mouse button press – when the mouse is pressed over the piece presenter, move-recognizer sends move-command a message to set the value of the piece input. If the value is invalid move-recognizer waits for the mouse button release, clears the piece input and exits. If the value is valid, move-recognizer handles the next events, mouse movement and mouse button release.

When the piece input of the move-command is set, the piece-presenter is asked to update its display, which it does by highlighting the piece to show whether it is a valid input or not.

Mouse movement – each time move-recognizer detects that the mouse crosses from one board square to another, it sends a message to move-command to set the planned-value slot of the location input to the board location under the mouse.

The board-presenter is asked to update its display to show whether the location is a valid one. The net effect is that as the user drags a piece around, the locations under it highlight when they are a valid landing location. This is an example of the kind of semantic feedback that Nephew can generate based on the information in the UIMS/Application interface.

Mouse button release – when the mouse button is released, move-recognizer sends move-command a message to set the value of the location input the location under the mouse. Then move-recognizer sends a message to execute the command, and exits.

If the value of the location input is valid, move-command executes the move-piece-op, and clears the inputs. If the value is invalid, the operation is not executed, and the bell is sounded to inform the user about the error. When the piece is moved, the piece-presenter is asked to update its display, which causes the piece to be shown at its new location.

The interface to the chess program can be easily modified so that rather than highlighting the legal locations while dragging a piece, all the legal locations are highlighted when the piece to be moved is initially selected. The modification involves changing move-recognizer so that it does not send messages to move-command when the mouse moves, and changing move-command so that when the piece input is set, the board-presenter is asked to display the alternatives for the location input.

5 Related Work

In the Seeheim Workshop on UIMSs it was determined that no notations existed for specifying the application interface model [2]. The Nephew UIMS/Application interface is one such notation with features that go beyond most of the requirements outlined by Green [2]. Nephew does not address the problem of representing global state and sequential relationships between operations. UIDE [1] provides good solutions to these two problems. UIDE explicitly represents global state, and defines transformations to transform a program into an equivalent one that does not use global state. The problem of sequential relationships between operations is addressed by defining pre- and post-conditions on operations, which give UIDE the ability to reason about the effect of a sequence of operations.

Several systems define richer UIMS/Application interfaces than traditional application interfaces. Reid G. Smith *et. al.* [12] developed a system that uses a frame-based knowledge representation system to represent information about objects needed for the user interface. They represent information such as defaults, number of inputs, data types, validation predicates, error messages and order in which inputs should be requested. In Nephew, the error messages and the order in which to request inputs are considered user interface attributes, and are specified in commands. In addition, the Nephew UIMS/Application interface provides additional information needed to support graphical interfaces and semantic feedback.

The Cousin UIMS [3] enforces a strict separation between user interface and application. The Cousin UIMS/Application interface does not support the representation of all the information needed to provide semantic feedback in direct manipulation interfaces. Cousin only supported coarse-grained interfaces, where the application and user interface communicate infrequently (*e.g.* form-based interfaces).

The object-oriented architecture of some toolkits such as the X Toolkit [7] alleviates the proliferation of call-back procedure conventions because similar building blocks are defined as subclasses of a common class, and thus share some of their call-back procedure conventions. However, building blocks with radically different appearance are not defined under the same class and thus do not necessarily share call-back procedure conventions. In addition, since the purpose of the different call-back procedures in not declared in the UIMS/Application interface, the building blocks cannot automatically find the relevant call-back procedures.

Nephew presenters, commands and recognizers are similar to Smalltalk's views and controllers [5]. The main difference is that the role of the controllers is played by commands and recognizers in Nephew. By separating gesture handling (recognizers) from dialogue control (commands), Nephew simplifies the design of the controllers.

Presenters, controllers and recognizers are also similar to MacApp's [10] views and commands. The role of commands in both systems is similar, serving to collect the inputs for the program operations. Nephew takes the idea one step further by using commands as object representations of the program operations, and allowing commands to be used anywhere where objects can be used. For example, the commands can be displayed with a presenter. Separating out the recognizers is advantageous because the recognizers are hard to write, and if separated they can be reused [8].

The EZWin system [6] provides presentations, commands and trackers, which are similar to Nephew commands, presenters and recognizers. The main difference is that EZWin has no UIMS/Application interface. All the information contained in the Nephew UIMS/Application interface is spread out in the methods of the various components, thus making it harder to change the interface for the application.

6 Final Remarks

Three applications where implemented using the Nephew prototype, the chess program used as an example in this paper, an icon editor, and a Macintosh-like interface to the Symbolics file system. The experience with these applications, although limited, suggests that the Nephew UIMS/Application interface indeed provides the information needed to support a wide variety of interface features and semantic feedback.

The interfaces for these programs featured menus and various kinds of buttons with non-selectable items dimmed, dragging with semantic feedback (appropriate highlighting of objects under mouse), gridding, rubberbanding. All these made use of the application-specific information contained in the UIMS/Application interface. The author's dissertation [13] describes in greater detail the Nephew UIMS/Application interface, commands, presenters and recognizers, and how they are put together to construct a variety of interfaces.

A new implementation of Nephew is currently underway that will allow interface designers to specify the interface by interactively connecting the various components, rather than by writing code as was necessary in the implementation described here. The goal is to allow designers to specify interfaces with diagrams like the one in Figure 1.

This Nephew UIMS/Application interface is better that traditional UIMS/Application interfaces from the modularity and code reusability point of views. The user interface can be changed without affecting the application portion of the program, and the interface building blocks can be reused because they are plug-compatible with the application portion of the program.

Acknowledgments I wish to thank Bob Neches, Neil Goldman, John Granacki and Brian Harp for useful comments on previous drafts of this paper.

References

[1] J. Foley, C. Gibbs, W. C. Kim, and S. Kovacevic. A knowledge-based user interface management system. In *CHI'88 Conference Proceedings*, pages 67–72. ACM, May 1988.

[2] M. Green. Report on dialogue specification tools. In G. E. Pfaff, editor, *User Interface Management Systems*, pages 9–20. Springer-Verlag, 1983.

[3] P. Hayes, P. Szekely, and R. Lerner. Design alternatives for user interface management systems based on the experience with COUSIN. In *CHI'85 Conference Proceedings*, April 1985.

[4] K. Kimbrough and L. Oren. Clue: A common lisp user interface environment. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 85–94, October 1988.

[5] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–41, 1988.

[6] H. Lieberman. There's more to menu systems than meets the screen. In *ACM Computer Graphics*, pages 181–189. ACM, July 1985.

[7] J. McCormack and P. Asente. An overview of the x toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 46–55, October 1988.

[8] B. A. Myers. Encapsulating interactive behaviors. In *CHI'89 Conference Proceedings*, April 1989.

[9] B. A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15–23, January 1989.

[10] K. J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, 1986.

[11] A. Schulert, G. Rogers, and J. Hamilton. ADM - a dialog manager. In *CHI'85 Conference Proceedings*, April 1985.

[12] R. Smith, G. Lafue, and S. Vestal. Declarative task description as a user-interface structuring mechanism. *Computer*, pages 29–38, September 1984.

[13] P. Szekely. Separating the user interface from the functionality of application programs. Ph.D. thesis CMU-CS-88-101, Carnegie-Mellon University, January 1988.