A Programming Language Basis for User Interface Management

Dan R. Olsen Jr.

Brigham Young University
Computer Science Department
Provo, UT 84602

Abstract

The Mickey UIMS maps the user interface style and techniques of the Apple Macintosh onto the declarative constructs of Pascal. The relationships between user interfaces and the programming language control the interface generation. This imposes some restrictions on the possible styles of user interfaces but greatly enhances the usability of the UIMS.

Keywords: User Interface Management Systems, User Interface Specifications, User Interface Generation.

Introduction

User Interface Management Systems (UIMS) have been a research topic for quite some time. A number of models have been presented for specifying human / computer interfaces in a fashion suitable for generating some or all of the user interface code. Early systems based their models on transition diagrams [Jacob 85, Olsen 84], grammars [Olsen 83] or menu trees [Kasik 82]. More recently attention has been paid to the power of object oriented languages to model the processing of interactive events [Schmucker 86]. Such approaches have concentrated on modeling what to do with input events such as key presses or mouse clicks with a great deal of success. Object-oriented approaches have been particularly successful in implementing direct manipulation interfaces. Our research has attempted to exploit the descriptive rather than the processing models of programming languages as a vehicle for specifying user / computer interfaces.

This paper describes the Mickey UIMS which takes the interactive capabilities of the Apple Macintosh and maps them onto the declarative facilities of Pascal. Although this implementation will be used in the examples throughout the paper these ideas are neither restricted to the Macintosh nor to Pascal.

Language-Based User Interface Specifications

Our first attempt at building a language-based UIMS was the MIKE system[Olsen 86]. The basic metaphor for this system was that all user interfaces were modeled by a set of object types and a set of procedures and functions that operated on or returned information about such objects. These were coupled with a set of base level interaction techniques and a default interaction style from which user interfaces were produced. These interfaces could then be refined via a profile editor.

Our experience with MIKE has produced the following insights into the efficacy of language-based user interface specifications. The first was that by using a user interface specification based on terms familiar to programmers we were able to overcome the programmer resistance that plagued our earlier UIMS development efforts. MIKE interfaces are described in terms of what they are supposed to do rather than in terms of how events are handled. By adopting a high-level description of the interface we were able to generate default interfaces from very early specifications of an interface design. Such a high level description allows us to create tools for measuring interface usage to aid in refining the interface [Olsen 88b]. Foley [Foley 88] has developed a range of interface analysis techniques based on interface models similar to MIKE's. Because of the language basis of MIKE, a Macros by Example [Olsen 88a] facility was added which allowed end users to extend and modify their user interfaces.

In spite of the advantages described above, we discovered two problems in MIKE's model of user interfaces. The first was that the dialog style was dictated by the UIMS to a large degree. The style could be adapted to a variety of applications but the fundamental style could not be changed. A second problem was that MIKE's view of user interfaces weighed heavily in favor of command-style interactions which the community in general is moving away from.

The Mickey Approach

The Mickey system was designed both as an attempt to simplify the user interface specification process and to broaden the scope of interfaces that can be modelled in a language-based UIMS. The primary thrust of this project was to extend the user interface specification model to include a wider range of descriptive facilities than simple types and procedures. Rather than creating a new specification language or editor the descriptive facilities of the Pascal source were used so that programmers need learn little new syntax in creating user interfaces. Recognizing that language-based UIMSs of necessity carried heavy stylistic biases we choose to implement in the Macintosh environment because of the stylistic biases shared by that community. The stylistic requirements of the Macintosh and other environments turns the fact that Mickey will not generate all styles of user interface into an advantage rather than a disadvantage. This is so because Mickey generates exactly those kinds of interfaces required by the target environment.

Our design approach was to examine the fragments of user interface style provided in the Macintosh toolbox and then find corollary constructs in the Pascal language. The UIMS is then built around these mappings between standard Macintosh interaction techniques and Pascal facilities. These mappings greatly simplify a programmer's view of the user interface implementation.

The approach described in this paper requires a unique UIMS implementation for each combination of a user interface look and feel with a specific programming language. We find the integration of the UIMS with the programming environment no more restrictive than integrating compilers, editors and debuggers into a uniform environment. It is a price that must be paid for ease of use.

Mickey User Interface Specifications

As with many other UIMSs, Mickey views the application as a set of services that are to be made available interactively. In Mickey this set of services is defined as a Pascal Unit, which is a separately compilable module. Mickey parses the interface portion of such a unit to extract the necessary information about the application's interactive services. The interface portion of a Pascal unit provides sufficient information to access that unit interactively but it does not provide information about such issues as menu placement, external names, function key equivalence or other interactive presentation information. In order to supply this presentation information, any symbol declared in the interface unit can have a set of properties associated with it. These properties are specified using the (* *) form for Pascal comments. The following fragment of an interface unit will illustrate this.

```
UNIT SimpleDraw;
INTERFACE
    USES MickeyServe, MKW, DrawUtil;
    PROCEDURE NewDrawing (
        (* Menu=File Name='New...' Key=N*)
            DrawFile : OutFileDesc);
    PROCEDURE OpenDrawing (
        (* Menu=File Name='Open...' Key=O*)
            DrawFile : InFileDesc);
IMPLEMENTATION
    . . . .
END.
```

The NewDrawing procedure declaration is augmented by the property list (* Menu=File Name='New...' Key=N*) which states that NewDrawing is to be invoked from the File menu using an external name of New... or the command key N. Other than the mappings between interactive techniques and various Pascal constructs, the only other syntax that a programmer must learn is to associate property lists with symbols.

The Macintosh / Pascal mapping

The key to Mickey's specification model is the mapping of user interface techniques to specific Pascal constructs. For purposes of this discussion these mappings will be grouped around menus, dialog boxes and windows.

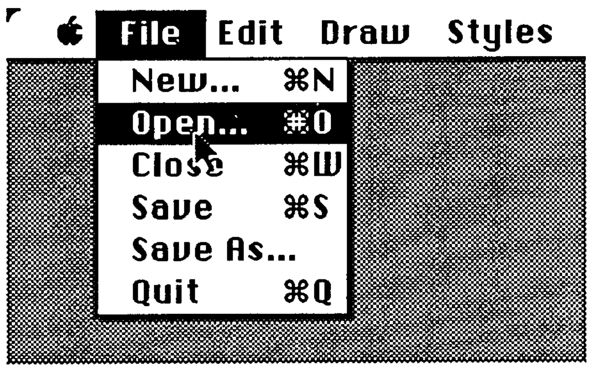
Menus

Menu items are handled in one of two forms. An item either represents a procedure invocation or a variable modification.

Simple Procedures

A procedure is entered into the menu simply by specifying the menu that it should belong to and by giving an optional external name and command key. If the external name is omitted, then the name of the procedure is used. The following is an example of a menu for a simple drawing program which has been constructed in this fashion.

```
PROCEDURE NewDrawing (
    (* Menu=File Name='New...' Key=N *)
        DrawFile : OutFileDesc);
PROCEDURE OpenDrawing (
    (* Menu=File Name='Open...' Key=O *)
        DrawFile : InFileDesc);
PROCEDURE CloseDrawing;
    (* Menu=File Name=Close Key=W *)
PROCEDURE SaveDrawing;
    (* Menu=File Name=Save Key=S *)
PROCEDURE SaveDrawingAs (
    (* Menu=File Name='Save As...' *)
        DrawFile : OutFileDesc);
```



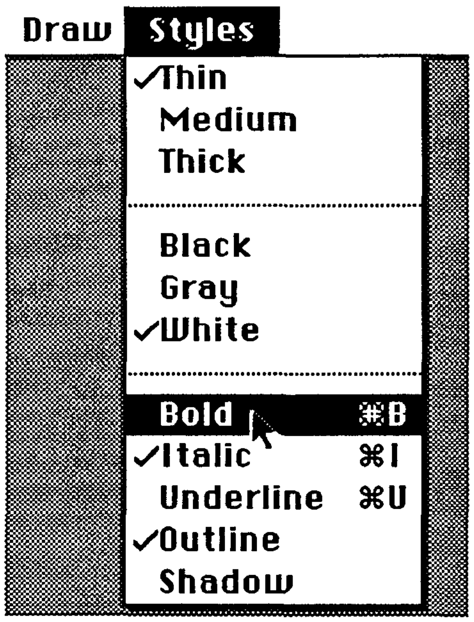
Many of the procedures, such as CloseDrawing and SaveDrawing, do not have arguments. Such procedures are immediately invoked when the corresponding menu item is selected. Others such as NewDrawing, OpenDrawing and SaveDrawing do have arguments which must be drawn from a set of primitive argument types. Each primitive argument type corresponds to a specific interactive technique. This is similar to MIKE with the exception that a much wider range is possible. The set of primitive techniques types is:

Point	input a 2D geometric point
RubberLine	input the end points of a line using a rubber band line
RubberRect	input the corners of a rectangle using a rubber rectangle
InFileDesc	select a file to be read using the standard Macintosh dialog
OutFileDesc	specify an output file using the standard Macintosh dialog

In addition to the primitive techniques described above any record data type can be specified for an argument in which case a dialog box is used, as will be described later.

<u>Shared Variables</u>

A second form of menu item are variables from the interface unit which are shared with the interactive users. There are variety of techniques whereby such variables can be modified from the menu.



The preceding menu is created by the following Pascal declarations:

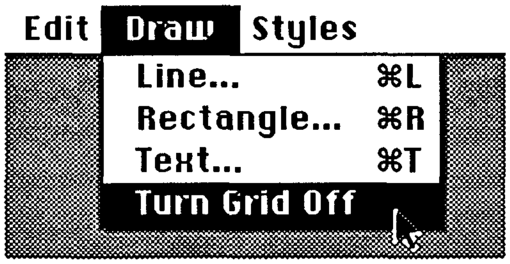
```
TYPE
    LineWidth = (ThinLine (*Name=Thin *),
        MedLine (*Name=Medium*),
        ThickLine (*Name = Thick *) );
    FillColor = (BlackFill (*Name=Black*),
        GrayFill(*Name=Gray*),
        WhiteFill (*Name=White*) );
```

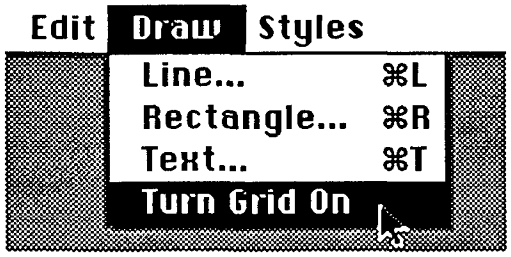
```
VAR
    LineSize (* Menu=Styles *): LineWidth;
    ColorSetting (* Menu=Styles *): FillColor;
```

```
VAR
    BoldText (* Menu=Styles
        Name=Bold Key=B*) : Boolean;
    ItalicText (* Menu=Styles
        Name=Italic Key=I*) : Boolean;
    UnderLineText (* Menu=Styles
        Name=Underline Key=U*) : Boolean;
    OutlineText (* Menu=Styles
        Name=Outline *) : Boolean;
    ShadowText (* Menu=Styles
        Name=Shadow *) : Boolean;
```

Because the variables LineSize and ColorSetting are defined as enumerated types they are represented in the menu as check lists. For example, selecting Medium from the menu causes the variable LineSize to be set to MedLine. LineSize is a variable in the interface unit which can then be freely accessed by the application code. The BoldText variable is declared as Boolean and thus becomes a checked item in the menu. Selecting Bold from the menu will set BoldText to True and will mark Bold with a check mark. Selecting Bold again will set it to false and remove the check mark. Toggled items which switch between two settings are specified using enumerations with only two choices, as in the case of the GridSetting variable shown below.

```
TYPE Gridding = (
        GridOn  (* Name = 'Turn Grid On' *),
        GridOff (* Name = 'Turn Grid Off'*) );
VAR GridSetting
        (* Menu = Draw *) : Gridding;
```

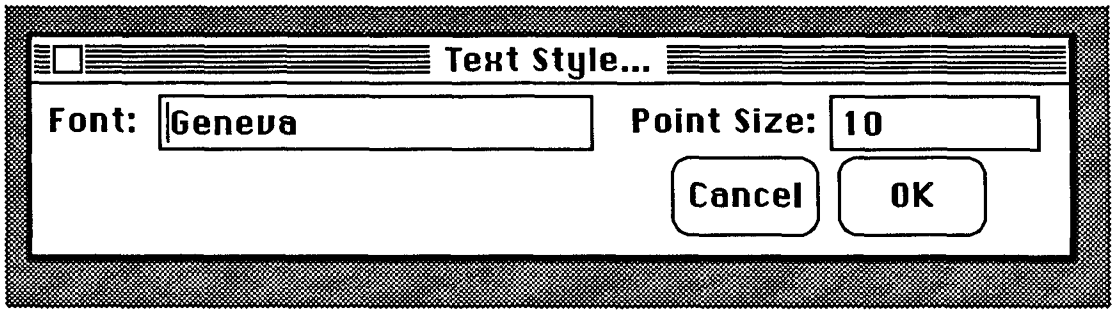




More complicated global settings are handled using variables whose types are Record data structures and then generating dialog boxes for those variables. When a dialog item is selected from the menu, its corresponding dialog box appears with the current values for that variable in the dialog fields. The user then interacts with the box in the standard way. Selecting OK will then set the variable that corresponds to the dialog box while selecting Cancel or the close box will remove the box without modifying the variable. The following StyleOfText variable is an example of this

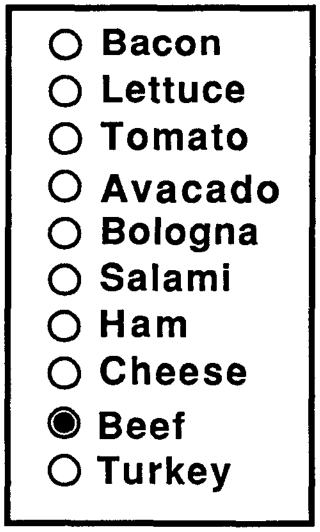
```
TYPE Str40 = STRING[40]
    TextStyle = RECORD
        Font : Str40;
        Points (*Name='Point Size'*) : Integer;
        END;

VAR   StyleOfText(*Menu=Styles
          Name='Text Style...'*): TextStyle;
```



Dialog Items

As shown in the previous example the types of variables are used to determine the appropriate interactive technique. The most complex technique supported by Mickey is the dialog box. Dialog boxes are defined by Record data types with the dialog fields being drawn from the fields of the record. Fields which have integer, real or string types are represented by fields to be typed in, Boolean fields become check boxes and fields with enumerated types become banks of radio buttons such as is shown below



```
Type SandwichItems =
        (Bacon, Lettuce, Tomato, Avacado,
        Bologna, Salami, Ham, Cheese, Beef,
        Turkey);
    Sandwich = Record
        Items: SandwichItems;
    End;
```

Not only can dialog boxes be created for global menu variables or arguments to menu procedures, they can also be created by the application code itself. For example, the type TextStyle shown above could have the (* DialogWindow *) property associated with it which would specify that dialog windows could be generated for TextStyle values. The application would create such a window by calling a service routine, passing the name "TextStyle" and a pointer to a TextStyle value. Whenever the fields in that window are modified interactively the corresponding fields in the value associated with the window would be changed.

Application Windows

The metaphor of equating Macintosh interface constructs with Pascal constructs starts to break down when addressing the actual windows in which the application is to display its information.

Even in the area of application windows the Macintosh is highly stylized in what can happen. Within a given application there may be a potentially large number of windows but a limited number of window types. Each window type is characterized by the way that the window is handled by the application. For each of the standard kinds of things that can be done to a Macintosh window there is a property that can be associated with the window type which specifies an application procedure to call in response to that user interface activity. In addition to handling actions such as resizing, closing, selecting or scrolling windows there are the basic interactive events which can be directed at a window such as mouse clicks, key input or dragging operations. Properties associated with each of these events specify application routines to be called when such an event occurs. The following is an example window type declaration.

```
(* Window=DrawWndw ScrollH=ScrollHProc
    ScrollV=ScrollVProc
    Redraw=RedrawDrawing
    Select=SelectDrawing
    GoAway=DrawingGoAway
    MouseDown=StartDragSelect
    MouseStillDown=DragSelect
    MouseUp=EndDragSelect *)

PROCEDURE ScrollHProc (Drawing : WindowNum;
    OldX, NewX : Integer);
PROCEDURE ScrollVProc (Drawing : WindowNum;
    OldY, NewY : Integer);
PROCEDURE RedrawDrawing(Drawing:WindowNum);
PROCEDURE SelectDrawing (Drawing:WindowNum);
PROCEDURE DrawingGoAway
    (Drawing : WindowNum);
PROCEDURE StartDragSelect
    (Drawing : WindowNum; Y, Mods : Integer);
PROCEDURE DragSelect
    (DrawWn : WindowNum; Y, Mods : Integer);
PROCEDURE EndDragSelect
    (DrawWn : WindowNum; Y, Mods : Integer);
```

The routines StartDragSelect, DragSelect and EndDragSelect support dragging a rectangle to select items from the window.

Application / UIMS Interface

The two primary vehicles for interfacing between the UIMS and the application are procedures which can be called and variables or other data values which are shared. In the case of procedures the interface is a simple invocation. The case of shared variables is more complex. If for example the variable BoldText is set by selecting it, then any future text entry procedures can simply look at BoldText to determine how the text is to be drawn. If, however, selecting BoldText also means to change the bolding of some currently selected piece of text, then a simple changing of the variable is insufficient. It may also be the case that the application only wants BoldText changed when a text item has been selected. These two problems are handled by means of guards. A guard is simply a Boolean function which can be associated with a variable or a field declaration. The guard function accepts the new value and can look at the variable or field for the old value to a) perform any actions associated with the change in value and/or b) return false if for any reason the change should not be made. If the guard returns false, then no change is made.

In addition to the declarations and property specifications found in the application's interface unit, there are a number of service routines for managing windows and doing the actual drawing in the windows. Any symbol declared in the interface unit can be disabled or enabled by calling a service routine which will cause the corresponding menu items to appear gray and for any interactive technique associated with the symbol to become inactive. This enable/disable facility gives the application more complete control over the user interface.

Implementation

Mickey functions by parsing the interface unit's source code, extracting from it the declarative information as well as the symbol properties of its interface. The parsed information is then traversed in order to construct the menu definitions, dialog box layouts, and window type definitions. As part of this construction phase a variety of error and consistency checks are made. The symbol, dialog and menu information is then generated as Macintosh resources and a unit of code is generated to provide the interface between the generic UIMS code and the application's interface unit. The generated code, the application code and the standard Mickey code are then all compiled together and attached to the resources to form a complete interactive application. Dialog box layouts are automatically computed and stored in standard Macintosh dialog resources. An interface designer can then use any of the Macintosh resource editors to refine the visual layout of the dialog boxes.

Summary

The Mickey system has extended the language-based user interface metaphor by adding typing information and variables shared with the user interface as models for extended capabilities. This form of specification leads to a very tight binding between a particular user interface style and a particular programming language. The virtues of this approach are that programmers can readily create new user interfaces for interactively sophisticated environments without learning an excessive amount of new notation or terminology. Our limited experience has shown that in less than two days a good Pascal programmer can have a working user interface without any prior experience programming the Macintosh. This system has been used with great success by a class of graduate students to generate dialog layout editors.

References

[Foley 88] Foley, J., Gibbs, C., Kim, W. C. and Kovacevic, S. A Knowledge-based User Interface Management System. *CHI'88 Proceedings* (May 1988).

[Jacob 85] Jacob, R.J.K. A State Transition Diagram Language for Visual Programming. *IEEE Computer*. 18, 8 (August 1985).

[Kasik 82] Kasik, D.J. A User Interface Management System. *Computer Graphics* 16, 3 (July 1982).

[Olsen 83] Olsen, D.R. and Dempsey, E.P. SYNGRAPH: A Graphic User Interface Generator. *Computer Graphics* 17, 3 (July 1983).

[Olsen 84] Olsen, D.R. Push-down Automata for User Interface Management. *ACM Transactions on Graphics*. 3, 4 (July 1984).

[Olsen 86] Olsen, D.R. MIKE:The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics* 5,4 (Oct 1986).

[Olsen 88a] Olsen, D. R. and Dance, J. R. Macros by Example in a Graphical UIMS. *IEEE Computer Graphics and Applications* 8, 1 (Jan 1988).

[Olsen 88b] Olsen, D. R. and Halversen, B. W. Interface Usage Measurements in a User Interface Management System. *ACM SIGGRAPH Symposium on User Interface Software* (Oct 1988).

[Schmucker 86] Schmucker, K. J. MacApp: An Application Framework, *Byte* 11, 8 (Aug 1986).