Effective application development for Presentation Manager programs

by S. M. Franklin
A. M. Peters

The OS/2™ Presentation Manager™ provides an integrated graphical, windowing user interface to IBM's OS/2 operating system. This paper addresses a primary area of interest for Presentation Manager application developers: the use and development of user controls. A control in the Presentation Manager environment is a program object with a programming interface and application function. The structure and interfaces between controls and the system are described in order to provide an understanding of the correct procedure for programming the Presentation Manager efficiently.

With the introduction of a new generation of workstations and midrange computing systems, IBM has made several key strategic announcements which set the direction for both IBM products and applications developed for IBM systems. Systems Application Architecture™ (SAA™) and its end-user interface component, Common User Access (CUA), define how a system and applications running on it interact with an individual at a terminal.[1] Cooperative processing requirements among IBM's Personal System/2® (PS/2®) workstations, midrange systems, and large systems and IBM's plan to standardize user interfaces across hardware product lines have given the PS/2 workstation the key role in end-user interface support. The primary user interface component of the PS/2 multitasking operating system, Operating System/2™ (OS/2™), is the Presentation Manager™ which gives the user a windowed, graphical interface.[2] The Presentation Manager is the system-supplied CUA end-user interface tool that provides access to the file system, system services, and applications. Also available from IBM is a programmer's toolkit which allows application developers to develop new, graphically based applications.

Users who interact with multiple applications are more productive in their work if the user interface is consistent across applications.[3] This statement becomes increasingly true as cooperative processing requirements continue to grow and application developers build applications designed for multiple systems environments. To make applications consistent with one another, each application needs to follow a set of rules regulating the interface consistency.[4] Adhering to such rules is particularly important with the incorporation of a graphical user interface such as the Presentation Manager. The sophisticated capabilities of the Presentation Manager have promoted the growth of advanced user interaction styles and techniques and make the requirement for

a user interface definition all the more important.[5] CUA is IBM's solution to achieve this consistency for SAA-conforming applications.[3] Application sets may choose, as well, to follow more stringent conformance definitions.

As application developers implement many of these new concepts, the need for a consistent set of both graphical and text user interface services for applications has emerged. Some of this enhanced capability is provided by the basic services of the Presentation Manager and is accessible to the application developer through the toolkit. Other services must be built by the application itself to accommodate application-specific requirements. Further, these services will most certainly not remain static as the program evolves over time. Applications will have to evolve as user interface techniques advance, and sophisticated new display technologies will have to be integrated into existing applications if they are to remain competitive. To provide for consistency in the user interface and for evolution, these services should be implemented in such a way that they are easily reusable and can be modified and extended without changing existing program code.[6]

This paper discusses a strategy for providing such sets of services, or building blocks, referred to in Presentation Manager terminology as *controls*, which allow a consistent approach to end-user interface development. A control under the Presentation Manager is a user interface element with a unique programming interface and application function. Examples of system-supplied controls are menus and dialog boxes. Clearly, any attempt to standardize application development should not restrict the usage of the underlying system capabilities. Application-defined controls in no way inhibit any of the sophisticated capability of the Presentation Manager or the designated manner in which developers implement programs; all Presentation Manager function remains available to an application. Application developers can, in fact, enhance the existing function by developing controls for specific program purposes. Controls can also be structured so that the burden of change is isolated from the application and localized within the control, thus promoting an object-oriented design.

Making application-developed controls available has been a system design approach used in the development of IBM's OfficeVision/2™ by the Application Solutions Division. This product is a set of SAA-compatible, CUA-compliant applications that form an electronic office for system users. The programs exploit the capabilities of the Presentation Manager and at the same time provide a tool set that minimizes development effort, enforces user interface consistency, and maximizes code reusability. Interface consistency gives developers the flexibility to make internal control changes as CUA develops.

This paper explores the concept of application-developed controls by examining Presentation Manager-supplied controls and the design and structure of application-developed controls. We discuss how controls are used by applications and how developers can design and develop additional controls. We highlight aspects of our experience in control development and discuss future requirements for building controls as the Presentation Manager evolves.

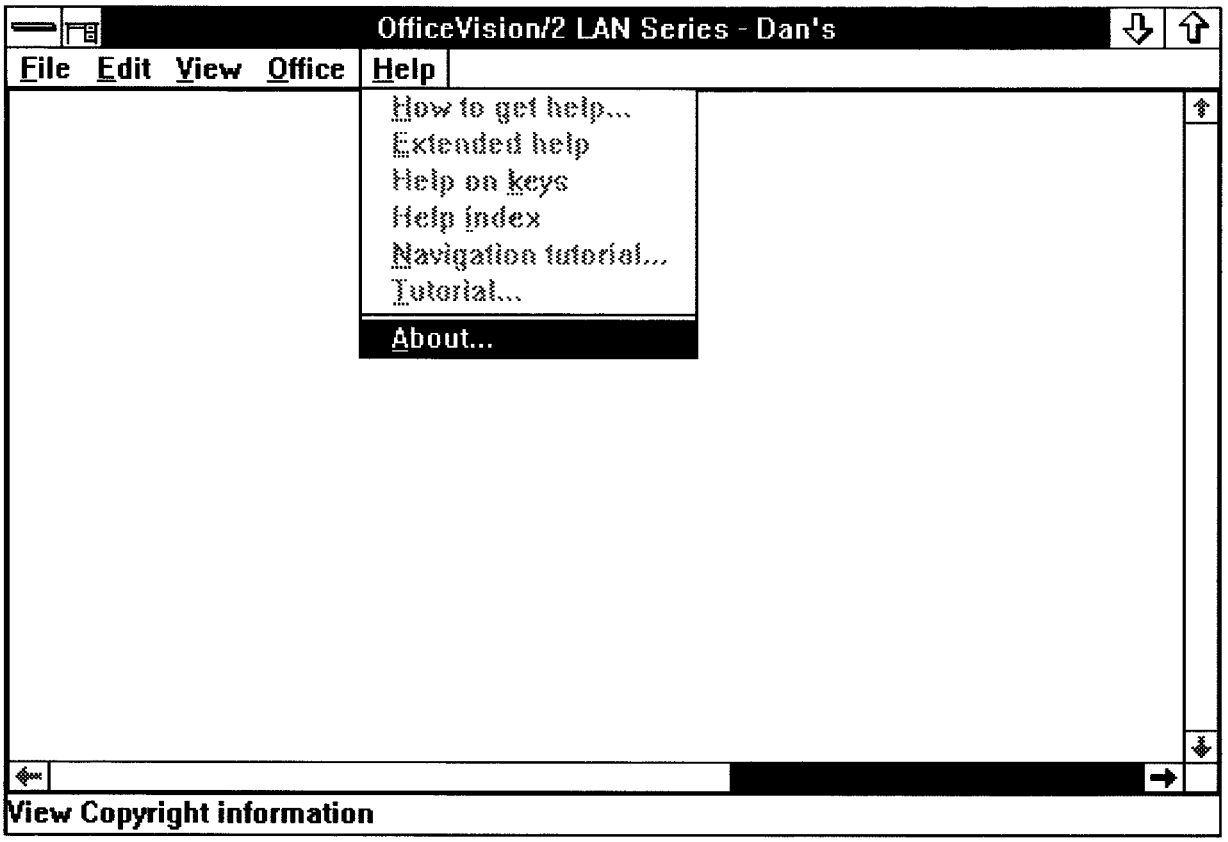
Presentation Manager user controls

The Presentation Manager implements a windowing interface by passing user-generated input events to underlying windows displayed on a workstation display screen. An input event is translated into a message and sent to the window procedure that processes input for a specific window. For example, if a user positions a mouse pointer over a button window and clicks the mouse, the window-processing procedure for the button window class receives a message indicating that a particular button has been selected on the screen by the user.

A control is a special kind of window. When a control is created, it is an instance of that special window class. Some controls are predefined by the Presentation Manager such as the title bar, iconic maximize and minimize arrows, menus, sizing borders, push buttons, scroll bars, and text entry fields. These types of controls are the basic user interface building blocks for Presentation Manager applications. A typical application consists of a frame containing an application-specific window known as the client window. The frame window itself consists of multiple controls such as the title bar, maximize and minimize icons, menus, sizing border, and scroll bars. Figure 1 shows an example of a frame window made up of frame window controls and the client window. The application selects the desired controls that make up the frame window at its creation. Each of these controls receives and processes messages from user-generated input events.

A window class definition is analogous to a definition for a new data type, just like a data-type class in an

Figure 1 Standard OfficeVision/2 frame window



object-oriented language.[7] Window class definitions describe class-unique data and specify the processing intelligence for the window class by indicating the window procedure for processing the class. The window procedure defines how the control appears to the user and is responsible for "painting" the control on the display screen. The procedure also defines how the control responds to user input because it processes the user's input, which comes to the control in the form of a message.

A simple example will help to illustrate how using controls impacts the structure of a Presentation Manager program. Suppose it is necessary for an application to display selection input fields such as the button selectors in Figure 2. The user indicates the desired action by positioning the cursor over the button selected and clicking the mouse.

The application could implement a button by drawing the oval outline for a button and displaying the text inside the outline. Whenever the application receives input from the mouse, the application could check to determine if the mouse was positioned over the oval when the mouse was clicked, and if so, process the command indicated by the text for the button. In this situation, the application creates and processes each button separately.

Instead, each button is defined as a "control" having particular button characteristics. The application creates the button by using the WinCreateWindow function call of OS/2, which specifies the size of the button, the text to appear in the button, and the location of the button. The WinCreateWindow function passes an application-defined amount of information to the new control when it is created, therefore allowing detailed information about button color, text fonts, etc., to be specified when the button is created.

Since the button is a predefined window class, the application need not draw the button. Instead, the window procedure of the button "paints" the button.

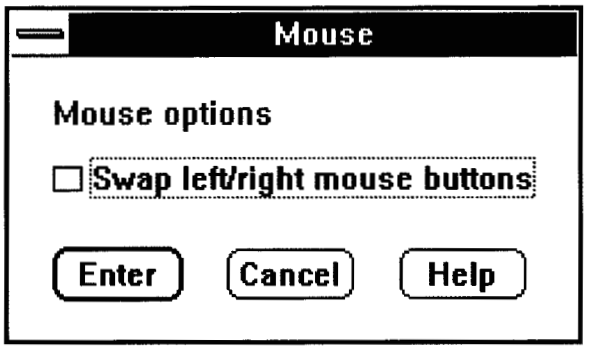
When the user selects a button, it is the button control that receives notification that the button was selected. The button control then sends a message back to the application program indicating that it was selected. To use buttons the application must only create the button with a single function call and accept notification when the button is selected. The control handles the mouse and graphics processing.[7]

The advantages of implementing the button as a control increase as the requirements for using buttons grow more complex. Consider that the application may be set up so that buttons are to be used in various parts of the application which require different forms of processing. It may be necessary for the application to use buttons both in the client window of the application and in dialog boxes. Client windows are normally created dynamically by the application program. Dialog boxes are normally created implicitly by defining the dialog box in an application resource file, including the components of the dialog box, which in this case includes a button control. If the button control is specified in the application resource file as part of the dialog box, the Presentation Manager automatically creates the button within the dialog box.

Further, the requirements for buttons may change after the application is developed, or different forms of buttons may be required in different computing environments. For example, a button might need to have both a short and a long form of text inside of the button, or possibly a variable shape. These changes are localized within the button control without having an impact on the application program.

We have already shown how the components of the frame window, exclusive of the client area, are primarily a collection of controls supplied by the Presentation Manager. Below we propose the same methodology within the client window based on an application-specific hierarchy of controls. Reusable, consistent program components can be structured and provided to calling applications as publicly defined window classes and then used as a control. Each component that is implemented as a control, regardless of its complexity, provides the same benefits for an application as the sample button control previously described. A control initializes itself, processes relevant information within itself, and interacts with the application with a minimal amount of messages. Changes and evolution within the sphere of a control are encapsulated within the control itself, making the remainder of the application immune from change occurring within the control.

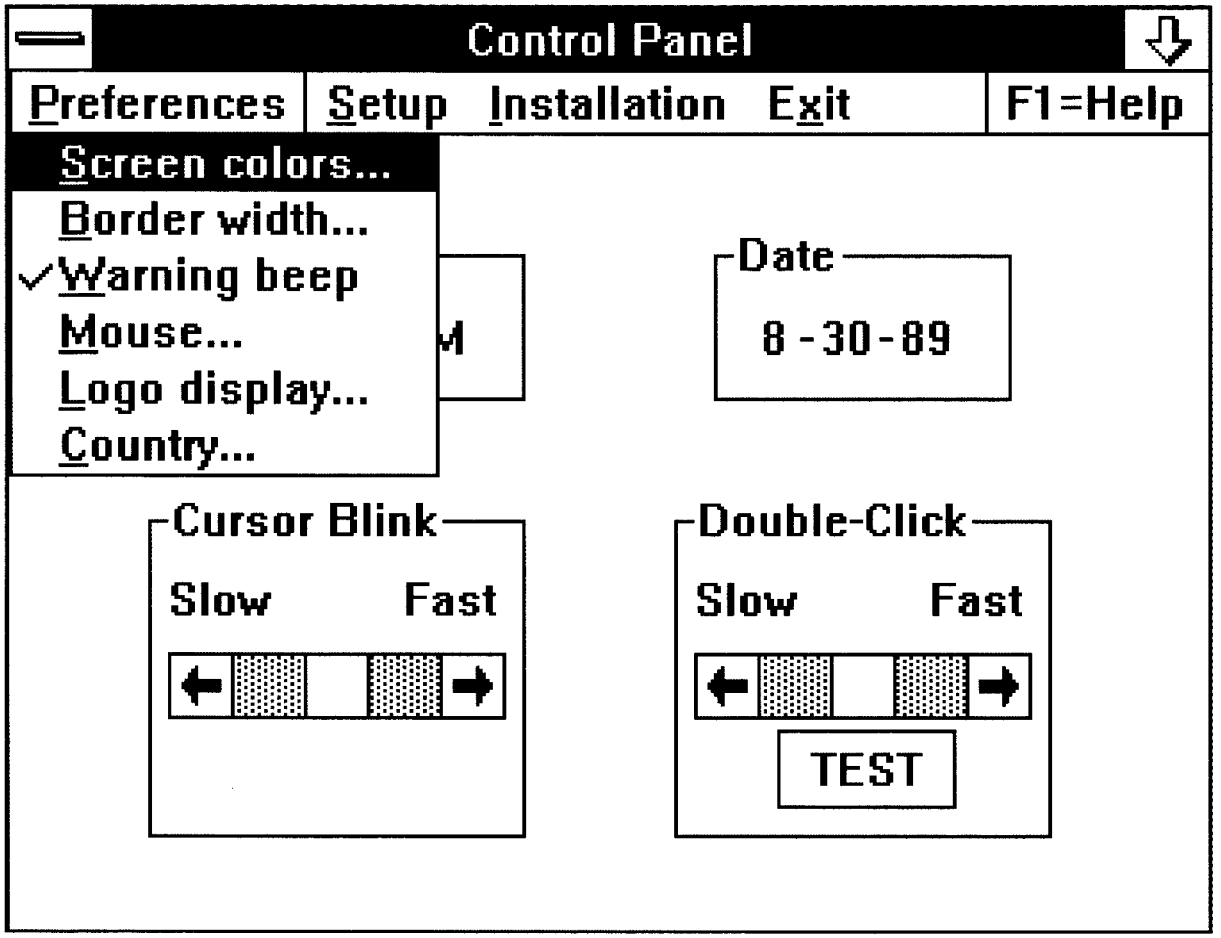
Figure 2 Button controls in a dialog box



To illustrate the point, Figure 3 and Figure 4 are examples of a relatively complex application which is implemented by constructing the application as a collection of basic controls. The purpose of the example application is to allow the user to define the colors and various screen components of Presentation Manager. Virtually every object on the screen is a predefined control supplied to the application from a previously existing source. In Figure 3 the frame window entitled "Control Panel" consists of a title bar, iconic minimize control, menu, scroll bar, and text input and output controls. When the user selects **Preferences–Screen colors ...**, the dialog box control labeled "Screen Colors" in Figure 4 is displayed, which consists of the same title bar, menu, scroll bar, and text input and output controls, as well as list, button, and window sizing controls, along with an application-defined color selection display control.

Not only does this example show how a relatively complex application is quickly implemented through the use of existing control components, but it is also an excellent illustration of how controls are able to provide user interface consistency. A new application, which provides an exact mockup of the user interface that the user can see and adjust, is created using existing user interface controls. This example provides a substantial amount of complex function in a consistent manner by simply specifying which controls are to be displayed and processing simple sets of messages sent to it by each of the controls. The controls handle most of the processing, without any involvement by the application. Of course, not every portion of an application can or should be structured into a control, but controls are a solution for those application segments that normally would be structured in a procedural manner.

Figure 3 Presentation Manager control-oriented application

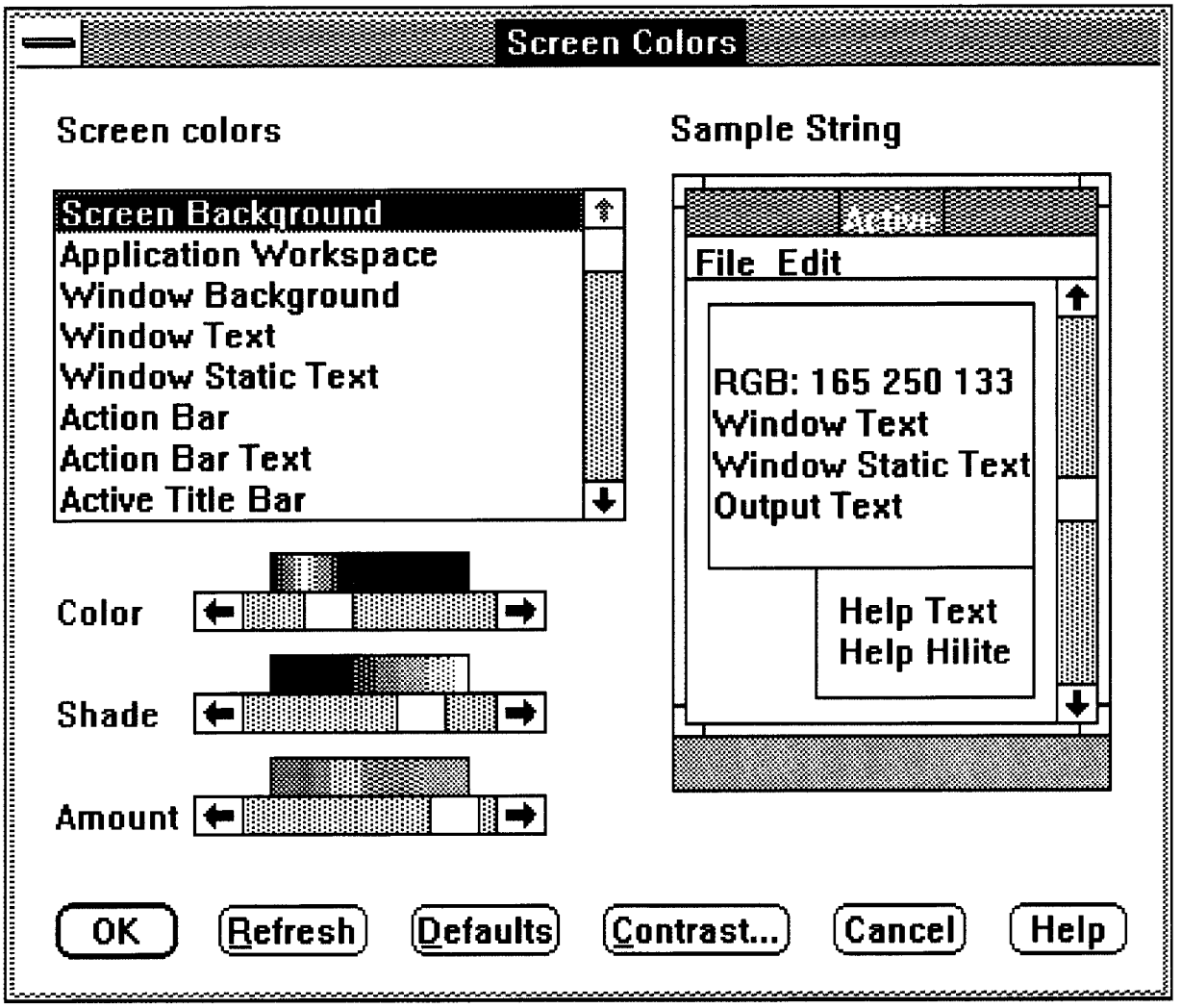


This concept can be extended to broad categories of applications. Within the office application several loosely coupled collections of data objects make up a substantial set of the office programs such as mail boxes, file cabinets, file drawers, and folders. Each of these programs keeps track of and maintains both similar and diverse sets of objects that have a common user interface format. This style is a list of graphics-based iconic representations of the objects within the collection. The exact style of the list is dependent on several criteria, including application requirements and user preferences. For example, the office window, which provides access to the office application set (Figure 5), is displayed in two distinct styles, one listed vertically and the other as a user-adjustable matrix.

To achieve user interface consistency, we decided that the user interface for each office application should be based on a common set of functions. We knew early within the development cycle that our user interface would be developed in an iterative fashion in conjunction with substantial human factors testing and would therefore be subject to considerable change throughout development. Development costs could soar unrealistically if every application within the product set was obligated to "understand" the intricacies involved in developing the graphical interface for the entire set of applications for the product.

Accordingly, we implemented the primary user interface components as controls. Figure 5 shows two

Figure 4 Dialog box from the control-oriented application

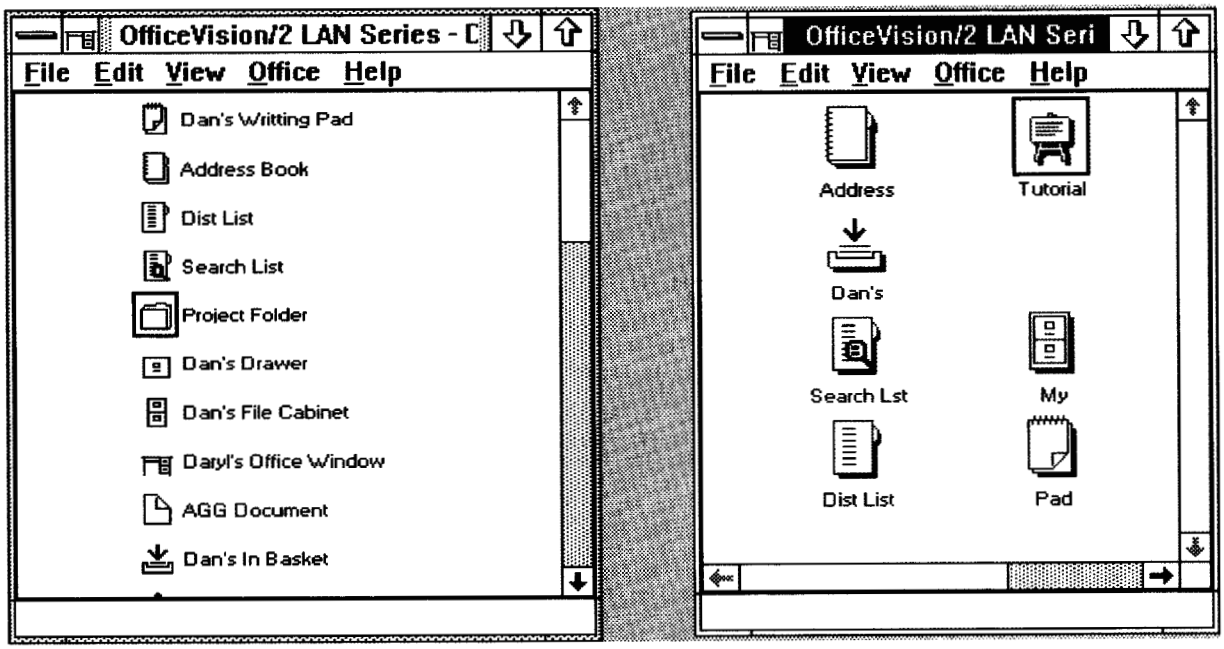


instances of the office window using the same control to produce different variations of the iconic interface. This control also provides other office programs with a simple interface for their own graphical user interface components; each application programmer need not understand the complexities of the graphical iconic interface. As changes surfaced and the user interface evolved, we changed only the controls that did not involve individual applications. This strategy ensured that individual application philosophies were superseded by those of the application set as a whole. These controls also make it possible for non-IBM developers to build applications that run as part of OfficeVision/2 and have the same user interface as the IBM-supplied applications which make up the office system.

An example user control

As illustrated previously, it is possible for an application to provide substantial function simply by using system-supplied controls. When these controls are not appropriate for a specific task, the application developer may implement the unique requirement as a "user control." If the code is structured so that the function is packaged as a unique window class, the control may be used in many different places throughout the application, as well as in other appli-

Figure 5 Office controls



cations. In developing an application control, the logic and data of the control must be structured and maintained such that each instance of a control maintains its own set of specific data, while responding to user interaction in a consistent manner.

We will create a user control called "value set" as an example of the use and development of application controls, but first, we explore the requirements for the value set control.

Function of a value set control

Often, the user of an application must make a selection from among a list of graphical elements. The Presentation Manager has controls for choosing items from lists of words. The value set control provides a mechanism for discrete, single selections depicted by icons, text, numeric values, patterns, or color. When a program requires the user to make a visual, single selection choice, a value set may be used to provide the interaction and selection.

For example, consider an application that displays a dialog box prompting the user for a color selection. This choice could be depicted using Presentation Manager buttons labeled "red," "white," and "blue," but the value set could also present this choice by displaying a palette containing red, white, and blue items. Use of the value set in this situation provides a more visual choice and may save valuable screen area.[8] It also eliminates the necessity of translating text as the product is developed for foreign countries. Figure 6 shows a dialog box containing several different styles of value sets.

Application users may interact with the value set by using a mouse, keyboard, or a combination of both. When using the mouse, the user selects an item by pointing to it and clicking with the mouse button. The value set control notifies its owner with a message whenever one of its items is selected or deselected. It also provides visual feedback of the selection state by drawing a heavy black line around the value set selection. The user may deselect items by clicking a second time, or by clicking on another item.

Use of the keyboard to navigate through the value set requires information about the current cursor position. When a value set receives attention from the keyboard, it reflects both the current selection and the current cursor position. The cursor position is indicated with a broken line drawn around the value set item. (See "Pie" in the "Chart" value set in Figure 6.) As the user navigates through the value

Figure 6 Dialog box containing various value set controls

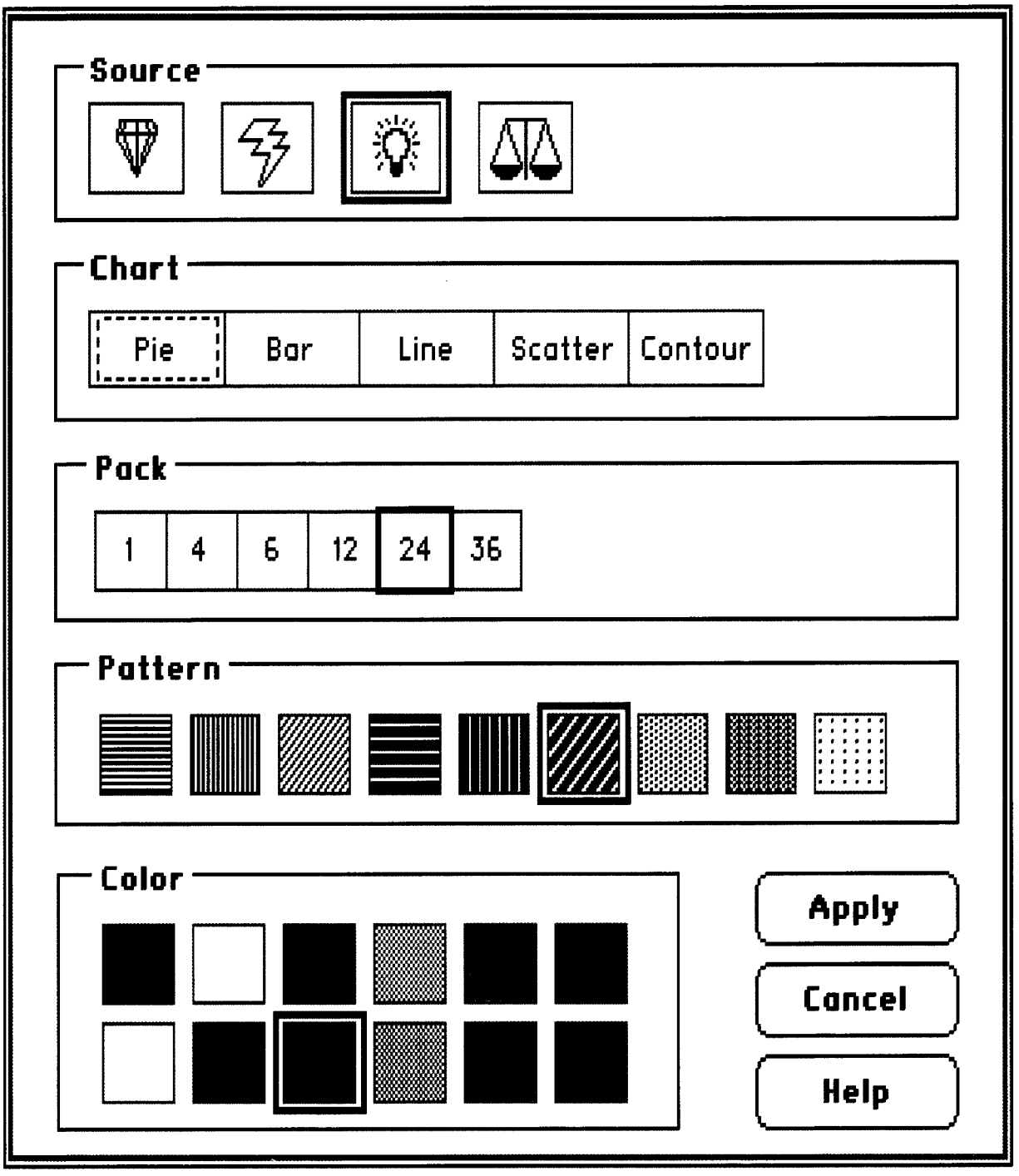


Figure 7 Sample value set invocation by calling application

```
WinRegisterClass((hab)NULL,          /* anchor block handle */
                 "ValueSet",         /* window class name   */
                  ValueSetWndProc,   /* window procedure    */
                  CS_SIZEREDRAW,     /* class style bits    */
                  4);                /* bytes of storage    */

hCtl = WinCreateWindow(hwndParent,        /* parent window */
                       "ValueSet",        /* window class  */
                       "Value Set Text",  /* window text   */
                       style,             /* style bits    */
                       x, y,              /* position      */
                       cx, cy,            /* size          */
                       hwndOwner,         /* owning window */
                       HWND_TOP,          /* z order       */
                       id,                /* window id     */
                       (PVOID)NULL,       /* optional      */
                                          /* presentation  */
                                          /* parameters    */
                       (PVOID)&CtlData);  /* control data  */
```

set items using the cursor keys, the broken outline follows. When the desired selection is reached, the Enter key is used to select the item. If the user presses Enter a second time, the item is deselected.

Although implementation of the value set selection mechanism could be undertaken as an application-supplied function, it would not facilitate code reuse. Other applications with similar requirements would have an individualized implementation of the value set which could result in different, perhaps confusing, interaction styles among similar functions. By structuring the value set function as a control, the function may be reused throughout the application and in many related applications.

User control invocation

The two primary requirements of an application program in creating a user control are to register the window class for the control and create the window. Each control is an instance of a particular window class. Registration tells the operating system what window procedure to call when a window of that class receives a message. Registration also specifies parameters such as storage requirements and the actions that the operating system is to perform when moving or sizing operations occur on that window. Once a window class has been defined to the operating system through registration, the program can create as many controls of this specific class as necessary.

Once the window class has been registered, the application creates the control by issuing a WinCreateWindow call specifying this class. With the value set serving as the example, it can be seen that the application issues the WinCreateWindow using the registered class of *ValueSet*.

Four basic types of information must be passed when the control is invoked:

1. *Information describing the ownership and parentage of the control.* This information is necessary in defining the messaging matrix to the system for communication between the owning window and the subordinate controls.
2. *Size information during the creation of the control.* This information can be omitted at creation time and dynamically supplied by the application when the control is displayed.
3. *Application-specific parameters.* In the value set, these parameters describe the row and column structure of the items within the control.
4. *System-required parameters.* Applications must supply identification (ID) for a window, identifying the control to the operating system.

When a window is created, the operating system returns a unique window identifier to be used by the application in communicating with the control. This identifier is called a *window handle.*

Figure 7 shows the sample calling sequence for invoking a value set.

General requirements for user control implementation

Although the implementation of a specific Presentation Manager user interface control will vary depending on the requirements, there are general guidelines for implementing a control. Successful development of a control begins by understanding the general characteristics and the skeletal structure of a control. This template can then be expanded to include the specific functional requirements of the control.

Message handling. From an implementation viewpoint, a control is nothing more than a specialized window class which is expected to field certain messages and return the expected values. The internal structure of a control is simply a window procedure. There is no main program or invoking routine; these tasks are performed by the application creating the control. The functionality of a control is determined simply by the types of messages that are accepted.

A control generally accepts two types of messages: window messages predefined in the Presentation Manager and new messages defined by the control. A subset of the former group is fielded by all controls. For example, the control must always be prepared to redraw its contents when the WM_PAINT message is received. Likewise, the contents of the control may need to be repositioned or resized when the WM_SIZE message is encountered. WM_CREATE processing gives the control a chance to initialize data and set up storage blocks, whereas the WM_DESTROY message is the appropriate time to release all resources allocated for the control. Beyond these four messages, the specific purpose of the control determines what additional system window messages (such as WM_BUTTON1DOWN) or specific control-defined messages must be fielded.

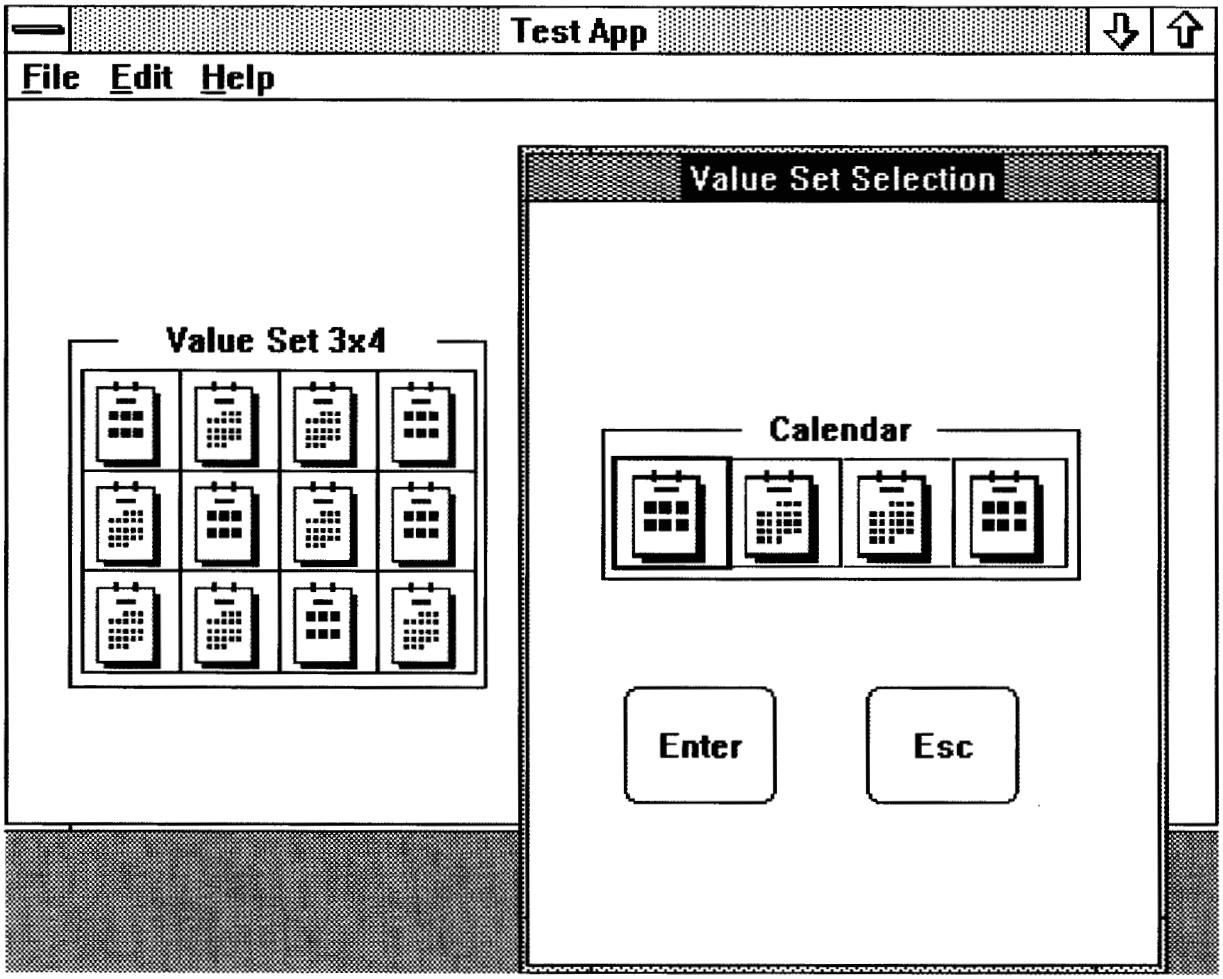
The value set control processes system messages such as WM_BUTTON1DOWN and WM_CHAR in order to determine mouse and keyboard navigation and selection. The remaining messages fielded by the value set are control-specific. The control provides a message set that allows the application developer to dynamically add, delete, and alter selection items in the value set control. The text and graphics inside the value set items may also be queried. Table 1 summarizes the value set messages.

Control parent and owner relationships. In addition to fielding certain messages and returning values, controls often post messages when certain specified events occur. The value set generates messages as an item is selected or deselected. Notification messages are posted to a window known as the owner of the control, the owner being specified during creation of the control. The only logical relationship between a control and its owner is the fact that notification messages are posted to the owner. It is the parent of the window that determines positioning of the control. The parent and owner may be the same window but do not necessarily have to be. For example, Figure 8 shows an application which has placed the

Table 1 Value set messages

Message	Cause and Processing
VSM_INSERTITEM	Inserts an item into the value set.
VSM_DELETEITEM	Deletes an item from the value set.
VSM_SETTEXT	Sets the text of the control.
VSM_DELETEALL	Removes all items from the value set.
VSM_SELECTITEM	Sets the selection state of the item.
VSM_QUERYSELECTION	Returns the selected item.
VSM_QUERYTEXTLENGTH	Gets the size of the value set field text in bytes.
VSM_QUERYTEXT	Copies field text of the control.
VSM_QUERYITEMCOUNT	Returns a count of the number of items in the value set.
VSM_QUERYITEMPOSFROMID	Returns the zero-based index of an item given its item identifier.
VSM_QUERYITEMIDFROMPOS	Returns the item identifier of an item given its zero-based index.
VSM_SETITEMSTRUCT	Resets the structure of the specified item to the new structure supplied.
VSM_QUERYITEMSTRUCT	Fills the structure supplied with the current information for the specified item.

Figure 8 Value set in dialog box signaling client window



value set control inside a dialog box. In this case, the dialog box is the parent of the control, although the client window will receive notification messages and act on selections that occur within the dialog box. In order to receive notification messages, the client window of the application is specified as the owner of the control. The role of the parent and owner with respect to a control is often muddled by control implementations. A simple rule is that all outgoing messages from a control are posted to the owner. There should never be any reason for a control to communicate with its parent.

Control status and instance data. Since a user interface control is a resource made available to all applications, a control cannot make any assumptions regarding its origin. For example, the control may be invoked several times by the same application or by many different applications. Associated with each invocation of a control is a particular state. For example, each value set contains a certain number of items positioned in particular locations within the control. Data which describe such information are called *instance data* and must be stored such that each individual instance of a control can access its instance data at all times during execution of the control.

Storage of instance data can be accomplished by storing a pointer to the data in a Presentation Manager *window word*. During class registration, an application may specify a specific amount of data to

Figure 9 Value set input structure

```
typedef struct _VALUEITEM {
  USHORT              idItem;          /* unique id for data item    */
  LONG                cBytes;          /* length of data in pGPI     */
  USHORT              rgfFormat;       /* data format flags          */
  char                szItem[64];      /* string name of item        */
  unsigned char far *pGPI;             /* pointer to bits or orders  */
  PBITMAPINFO         pbmapinfo;       /* pointer to bitmap table    */
  HBITMAP             hBitmap;         /* bitmap handle              */
} VALUEITEM;
```

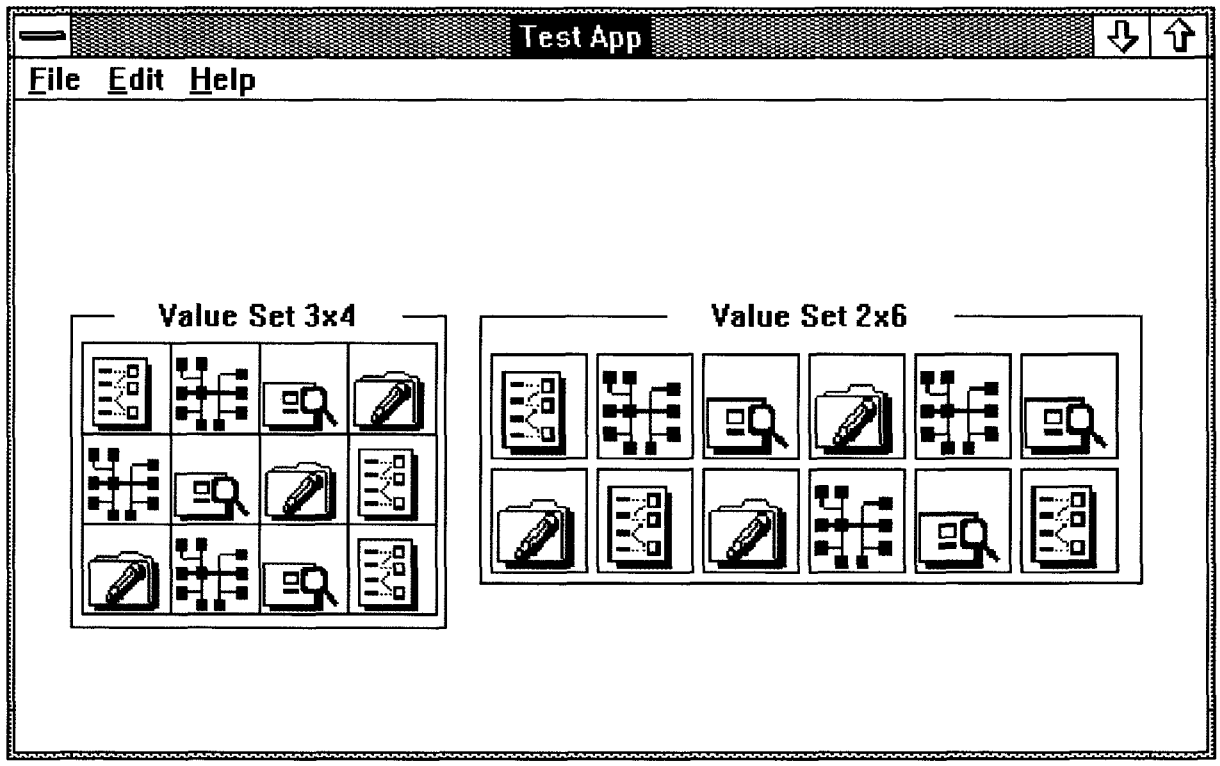
be reserved for each instance of the window class. These data blocks, or window words, are accessible through a standard Presentation Manager application program interface (API) call. Given any window handle, the Presentation Manager can return a pointer to any of the requested window words. The window class of the value set is registered with an additional four bytes of window word data in order to maintain and access instance data. This window word is requested in the last parameter of class registration as illustrated in Figure 7. It is used to hold a long pointer to the instance data block of the control and may be accessed at any time during execution of the control. Since one of the parameters to a window procedure is the handle of the window receiving the message, there is no confusion as to which invocation of the control is executing. This handle may be used to access the pointer to the correct instance data block.

With use of this design approach, each value set control has a standard four-byte window word that points to its unique instance data block. The size and contents of the instance data block will vary widely in different controls. In the value set this block contains information concerning the number of items present, the size of the items, the presentation format of the items, and pointers to the data concerning each individual item. The contents of the instance data block are likely to change as the developer iterates on the implementation of a user interface control. These iterations have no effect on any other applications, since the instance block pointer remains the same.

Control input structure. Each user control generally has a specialized data structure which is used by the application to supply information regarding the objects in the control. The value set defines a VALUEITEM structure containing the text and graphics that comprise an element in the value set. Figure 9 illustrates the VALUEITEM structure. A primary goal of the value set implementation is to provide a flexible set of input formats for the graphical data, so that the supplying application may use any drawing method to enter the graphics. Figure 10 shows a client window which has two value set controls containing graphics in raster and vector format. The value set accepts graphics as both bitmaps and drawing orders deposited into a memory block.

Bitmaps are a commonly used input format for standard Presentation Manager user interface controls such as buttons and menus. Bitmaps provide a fast mechanism for drawing graphics, but they are very device-dependent. Bitmaps may change somewhat in size and appearance, depending on the particular hardware display device that is being used. However, they provide a simple way for applications to load or create graphics and pass them to a control. The value set provides two methods for supplying bitmap data. First, a bitmap handle created by the system may be supplied. This method is used by the menu and button controls. An application utilizes system calls to load a bitmap from resources or to create one dynamically. The system returns a unique handle describing the bitmap. This handle may be used within the OS/2 process that created the bitmap but may not be shared with other processes. Since an application and its control execute in the same process, this format is usually acceptable. However, in some cases it may be necessary for the application to use graphical data supplied by another independent application. To facilitate the input of bitmaps

Figure 10 Value sets containing multiple data formats



received from outside sources, the value set also accepts bitmaps in a raw format consisting of a Presentation Manager BITMAPINFOHEADER structure which describes the bitmap followed by the actual bit settings for each pixel. In this case, the value set generates its own private bitmap handle.

Applications may provide graphics in a format that is independent among devices. In such a case, the Presentation Manager Graphical Programming Interface (GPI) may be used to generate drawing orders that describe the picture in vector format, as opposed to the raster image described by bitmaps. The standard Presentation Manager controls do not accept drawing orders as inputs. If applications are to utilize the GPI interface to draw control items, they must explicitly request that the control notify the application when the item is to be drawn. This method is called *OWNERDRAW*. If a control item is set with the OWNERDRAW style bit, the owning application will receive a message every time the item must be painted. The application is provided with a presentation space for drawing and a description of the size and location of the item and may then use the GPI interface or any other preferred drawing APIs to draw the item. This method requires that the application process the WM_DRAWITEM message in order to draw any item as owner-drawn.

The value set expands the notion of owner-drawn items by providing an input format which consists of drawing orders written into memory. The application may draw its control item into a presentation space and then use system-supplied calls to write the drawing orders to memory. These drawing orders are then saved by the control and re-executed each time that the item must be drawn. By using this input format, the application need only draw the item once and pass it to the control. When redrawing is necessary, the control executes the stored drawing instructions, rather than requesting that the application redraw the items.

Keyboard support. Any function provided by a user interface control must be accessible through a keyboard as well as a mouse. Although actions such as selection or direct manipulation may seem more natural using a mouse, the control must provide

equivalent function for the keyboard user. This implies that the control instance data block must contain information concerning the current cursor position along with other status data. Even if the items in the control have no real logical ordering, the control must order the items such that keyboard navigation permits access to all of the objects in a control.

User control expansion. Understanding the basic guidelines in designing and implementing a user control is the first step in developing application code which can be shared and reused. Once the basic structure for a control is designed, the function of the control may be extended by processing a greater number of messages and adding information to its instance data or input structure as application requirements change. These changes can be made without dramatically altering the original structure or the processing performed by the control. If the control message interfaces and input structures remain stable, applications may obtain expanded function without any modification.

Concluding remarks

Our development of controls for the IBM office environment product has proved to be a successful method for both propagating consistent user interaction requirements throughout all components of the office system and reducing the programming effort for the individual applications. The development of specific controls as well as the use of the system-supplied controls encapsulates very specific functional requirements into specialized objects that can be used by all applications that are a part of the office environment. The user is presented with a visual interface that behaves consistently across office components, and the application programmer benefits from the availability of these controls when designing and programming an office application.

As workstation interaction techniques continue to mature, the need for user interface consistency across applications will become increasingly important. Definition of a uniform interaction style is the framework for this consistency. However, this definition is no guarantee of conformance unless tools are developed that enable an application to conform to user interface definitions while preserving the freedom of the application. Controls serve as flexible application enabling tools by providing programs with a set of user interface building blocks for application development. Properly structuring our controls allows us to take a first step toward an object-oriented application programming interface by hiding the data and internal control processing mechanisms and surfacing only a message set to the applications. The object-oriented approach to Presentation Manager application development can be expanded by developing base classes of control objects upon which can be built more specialized controls that inherit behavior from the base classes.

We have seen how the implementation of application function as user controls has many practical advantages. In addition, the evolution of controls toward objects in an object-oriented environment provides an opportunity for continuing research and development in the use and design of Presentation Manager controls. Controls may be designed and implemented not only as individual building blocks for specific application function, but as integrated sets of objects related in a hierarchical fashion and sharing certain levels of behavior.

Acknowledgments

The authors wish to acknowledge the support of Sandy Cureton, Bill Braley, and Dorene Palermo in writing this article. We also acknowledge Nancy Jackson, Tony Temple, Ellen Cohen Sonenthal, and Art Goldstein for their support of the OfficeVision/2 Presentation Manager platform. We extend special thanks to Dave McGehe, Randy Black, and Dan Kardell for their help with the manuscript and to Mark Estes for his programming efforts.

Cited references

1. E. F. Wheeler and A. G. Ganek, "Introduction to Systems Application Architecture," *IBM Systems Journal* **27**, No. 3, 250–263 (1988).
2. M. Vellon, "OS/2 Windows Presentation Manager: Microsoft Windows on the Future," *Microsoft Systems Journal* **2**, No. 2, 13–18 (May 1987).
3. R. E. Berry, "Common User Access—A Consistent and Usable Human–Computer Interface for the SAA Environments," *IBM Systems Journal* **27**, No. 3, 281–300 (1988).
4. S. Uhlir, "Enabling the User Interface," *IBM Systems Journal* **27**, No. 3, 306–314 (1988).
5. S. Franklin and T. Peters, "Graphical Interface Services for Application Integration," *Graphics Interface '89 Proceedings*, Canadian Information Processing Society and Canadian Man-Computer Communications Society (June 1989), pp. 105–112.
6. B. Cox, *Object-Oriented Programming*, Productivity Products International, Inc., Sandy Hook, CT (1986).

7. C. Petzold, "Object Oriented Programming," *PC Magazine* **8,** No. 1, 317–324 (January 17, 1989).
8. *Systems Application Architecture Common User Access Advanced Interface Design Guide*, SC26-4582, IBM Corporation; available through IBM branch offices.

General references

D. M. Chess and M. F. Cowlishaw, "A Large-Scale Computer Conferencing System," *IBM Systems Journal* **26,** No. 1, 138–153 (1987).

W. P. Dunfee, J. D. McGehe, R. C. Rauf, and K. O. Shipp, "Designing SAA Applications and User Interfaces," *IBM Systems Journal* **27,** No. 3, 325–347 (1988).

S. Franklin and T. Peters, "A Technical Study of Dynamic Data Exchange Under Presentation Manager," *Microsoft Systems Journal* **4,** No. 3, 1–16 (May 1989).

E. Iacobucci, *OS/2 Programmer's Guide*, McGraw-Hill, Inc., Berkeley, CA (1988).

"IBM Operating System/2," *IBM Personal System/2 Seminar Proceedings* **5,** No. 5, 32–45 (May 1987).

IBM Operating System/2 Internals Volume 2: Presentation Manager, IBM International Technical Support Center, Boca Raton, Florida (1988).

IBM Operating System/2 Version 1.1 Programmer's Toolkit, Part No. 6280211, IBM Corporation; available through IBM branch offices.

IBM Operating System/2 Version 1.1 Technical Reference, Part No. 6280212, IBM Corporation; available through IBM branch offices.

"IBM OS/2 Standard Edition Version 1.1, IBM Operating System/2 Update, Presentation Manager (Part 1)," *IBM Personal System/2 Seminar Proceedings* **6,** No. 1, 13–62 (April 1988).

"IBM OS/2 Standard Edition Version 1.1, Presentation Manager (Part 2)," *IBM Personal System/2 Seminar Proceedings* **6,** No. 2, 2–41 (April 1988).

M. S. Kogan and F. L. Rawson III, "The Design of Operating System/2," *IBM Systems Journal* **27,** No. 2, 90–104 (1988).

C. Petzold, "The Graphics Programming Interface: A Guide to OS/2 Presentation Spaces," *Microsoft Systems Journal* **3,** No. 3, 9–18 (May 1988).

C. Petzold, "OS/2 Graphics Programming Interface: An Introduction to Coordinate Spaces," *Microsoft Systems Journal* **3,** No. 4, 23–40 (July 1988).

C. Petzold, *Programming the OS/2 Presentation Manager*, Microsoft Press, Redmond, WA (1988).

K. Welch, "Creating User-Defined Controls for Your Own Windows Applications," *Microsoft Systems Journal* **3,** No. 4, 54–66 (July 1988).

K. Welch, "Inter-Program Communication Using Windows' Dynamic Data Exchange," *Microsoft Systems Journal* **2,** No. 6, 13–23 (November 1987).

Susan Franklin *IBM Application Solutions Division, 5 West Kirkwood Boulevard, Roanoke, Texas 76299.* Ms. Franklin is a senior associate programmer who joined IBM in 1987. She received a B.S. degree in computing science from Texas A&M University at College Station in 1987. Since joining IBM, she has worked at the ASD Software Development Laboratory in Westlake, Texas, in user interface design and platform development of OfficeVison/2. Ms. Franklin has received two Invention Achievement Awards and one Outstanding Technical Achievement Award for her work in enhancing OS/2 interapplication data transfer techniques.

Tony Peters *IBM Application Solutions Division, 5 West Kirkwood Boulevard, Roanoke, Texas 76299.* Mr. Peters is a senior programmer who has been with IBM since 1982. He received a B.S. degree in mathematics and computer science from the University of Tennessee at Nashville in 1977 and an M.S. degree in computer science from the University of Tennessee at Knoxville in 1979. Prior to joining IBM, Mr. Peters was a member of the technical staff at Bell Telephone Laboratories in Naperville, Illinois. From 1982 to 1984, he was a project leader on the UNIX® development effort in Dallas, Texas. In 1984 and 1985 he was on international assignment in Böblingen, Germany, where he worked on the IX/370 project. Since 1986, he has been working on the OfficeVision/2 project in the Westlake ASD laboratory. Mr. Peters has received five Invention Achievement Awards and two Outstanding Technical Achievement Awards for his work in developing the OS/2 Dynamic Data Exchange (DDE) protocol for the OS/2 Presentation Manager and for his work on the Office Presentation Manager user interface platform.

Reprint Order No. G321-5385.