Enabling the user interface

by S. Uhlir

Presenting a consistent interface to the user is one of the objectives of Systems Application Architecture (SAA). The development of SAA applications is simplified by providing enabling interfaces which help an application developer support the SAA user interface. Rather than providing a single-level enabling interface, SAA offers a spectrum of levels spread over two interfaces: the SAA Presentation Interface and the SAA Dialog Interface. This gives the application developer the freedom to choose the appropriate level of interface for the application.

Enabling the Systems Application Architecture (SAA) user interface for an application can be looked at from two viewpoints—that of the user, who determines the requirements, and that of the developer, who must satisfy them. This paper describes user requirements and the enabling interfaces that assist the application developer to satisfy these requirements.

The user view

The work of a user who interacts with multiple applications can be greatly simplified if the user interface across these applications is consistent. Every application must follow a set of rules, Common User Access, to achieve this consistency. To make it easier to develop applications which appear consistent to the user, the Common Programming Interfaces of SAA include interfaces which enable the user interface. Three different aspects of consistency across all applications in the interfaces are important to the user: *semantic consistency*, *syntactic consistency*, and *physical consistency*.[1] Semantic consistency refers to the meaning of the elements that make up the interface—for example, the result of invoking a particular command. Syntactic consistency refers to words used for commands and the sequence of the appearance of the elements comprising the interface, such as the particular word used to invoke a function. Physical consistency refers to the hardware and how it is used, e.g., what key is pressed for a particular command. The following subsections discuss each of these aspects in more detail.

Semantic consistency. To achieve semantic consistency, all aspects of the objects which are visible to the user, and the actions which can be performed on these objects, must be consistent. For example, an object that is a *table* of data contains elements called *rows*. Rows can be *added to* and *removed from* tables. An object that is a *file* of characters contains elements called *lines* of text. A user who has learned about adding and removing rows from a table will expect to be able to do the same to files. Semantic consistency allows a user to develop strategies for accomplishing tasks in one application and then apply the same strategies to other applications. This transfer of strategies makes the applications easier to use because the user already knows some of what can and cannot be done before using the application.

Syntactic consistency. Transferring a strategy from one application to another is simplified by syntactic consistency. To achieve syntactic consistency, the same terms should be used to express the same meaning (for example, the command "insert," which is used to add a new piece, should be consistent). Syntactic consistency allows a user to develop strategies in terms of the common terms that can be directly applied to each application. Without syntactic consistency, a user may develop strategies using

common concepts but may have to use different commands for similar strategies in different applications.

Physical consistency. Physical consistency is also important in simplifying the use of multiple applications. To achieve physical consistency, the same sequence of interactions should be used to request the same function (for example, to perform the "add a new piece" function, the user positions the cursor at the insertion point, presses function key *F10* to switch to the action bar, presses the *tab* key to move to the "insert" selection, and then presses the *enter* key).

Two distinct interfaces are provided by SAA to enable the user interface: the dialog interface and the presentation interface.

Because the user may utilize the functions of multiple applications, consistency in all of these aspects is important both within and across applications. Semantic consistency allows the user to develop strategies that are independent of the application boundaries. Syntactic consistency lets the user know what terms are to be used for commands, and physical consistency lets the user know how to use a command to request the desired action.

The application developer view

The application developer is responsible for implementing the consistency required by the user. This task is simplified to the extent that the enabling interfaces can be used to provide consistency. At the same time, the application developer's control and creativity are reduced to the extent that the enabling interfaces take responsibility for some of the consistency.

Rather than locking all programmers into a single level of support, Systems Application Architecture provides different levels of support to enable the user interface. These levels allow the application developer to choose the appropriate balance between implementation effort and control.

Two distinct interfaces are provided by SAA to enable the user interface: the dialog interface[2] and the presentation interface.[3] The dialog interface provides more consistency with less programmer effort, whereas the presentation interface allows the programmer more control.

The dialog interface and the presentation interface both provide a spectrum of services which allow a programmer to adjust the trade-off between consistency and control. Each interface can be divided into two sublevels, providing four logical levels of interface; these are discussed in the following sections.

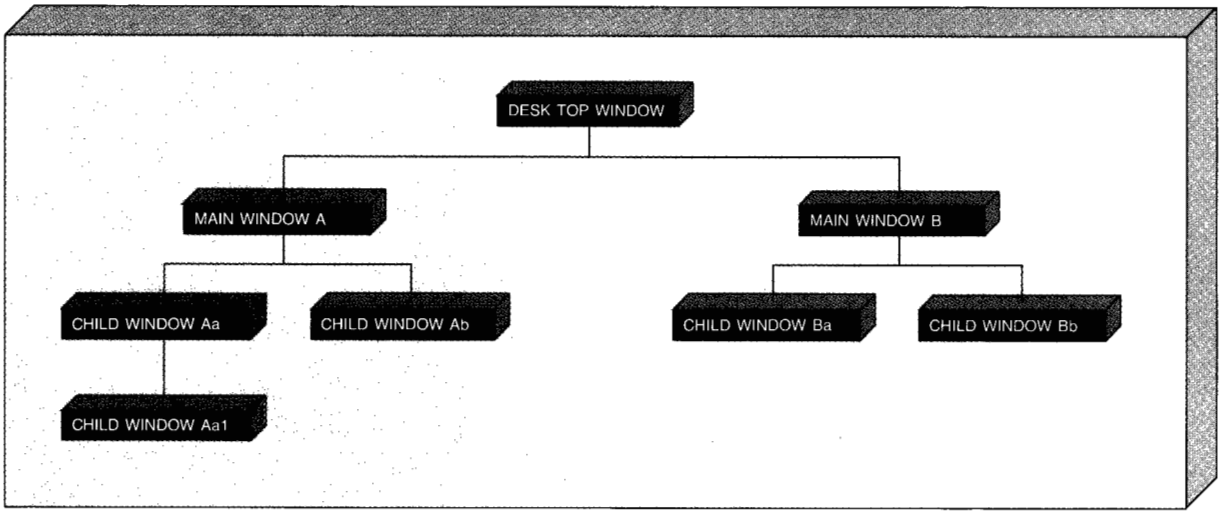
Presentation interface device independence. The level of interface which allows the application programmer the greatest control, and requires the greatest effort to achieve consistency, is the portion of the presentation interface which provides device independence only. This interface level supports a small amount of physical consistency by mapping the application's logical device requests into the capabilities of a particular display (for example, the physical realization of a selection device can be a mouse-driven pointer on some devices and a keyboard-driven cursor on others).

A basic element of the user interface which is defined by Common User Access is the window. A user interacts with an application through rectangular windows on the screen. Multiple windows can appear on the screen at the same time; one window may overlap another, obscuring the overlapped portion of the window underneath. The presentation interface assists the application in supporting this style of user interface by directly supporting overlapping windows.

Two kinds of windows can be created by an application: *main* windows and *child* windows. A main window is positioned relative to the *desk top*, which is represented by the display screen. Operations on one main window, such as moving it, do not affect other main windows. A child window is positioned relative to another window, its *parent*. Moving and sizing operations on a parent window affect its children.

Within implementation limits, an application can have as many main windows and as many child

Figure 1 Local hierarchy of nested windows



windows as it requires, nested to any depth. For example, if the logical relationship of a collection of windows is nested as shown in Figure 1, the display will appear as in Figure 2.

An application creates both main and child windows with the *create window* function. A main window is created by specifying the desk top as its parent; a child window is created by specifying another window as its parent.

Every window has associated with it a routine known as a *window procedure* that processes events related to that window. A window procedure may satisfy the processing requirements of more than one window. Windows with the same window procedure belong to a *window class*. When a window is created, the window procedure is indirectly specified by naming the class to which the window belongs.

An application supports the interactions defined by Common User Access by providing appropriate window procedures. A window procedure is invoked by the presentation manager when an event associated with the window it supports is detected. Information about the event is passed to the window procedure in the form of a message (a typical message indicates which key on the keyboard has been pressed). The application's window procedure contains the logic that reacts to the event, so the window procedure must be written to react as specified by Common User Access.

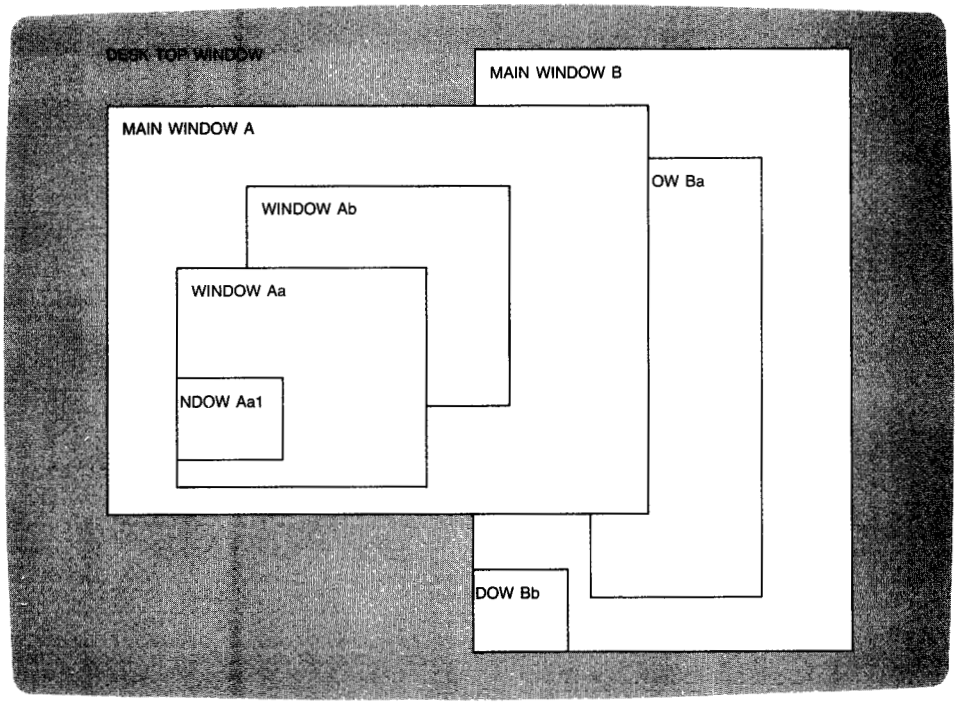
An application becomes, in effect, a set of window procedures which react to user input. The style of programming for such an application is different from that used for an application which controls the input allowed from the user. This reacting style of programming makes it easier to implement applications which conform to Common User Access. If application designers begin to think in terms of windows reacting to user input, their designs will be more consistent across systems.

Most of the physical consistency and all of the syntactic and semantic consistency is left to the application. This level of interface is appropriate for a highly interactive application (for example, an application used to compose free-format pictures). The user interacts with the graphical image on the screen which is translated by the application.

Presentation interface controls and templates. The next level of interface allows an application to delegate more responsibility to the presentation interface through the use of *controls* and *templates*. A control is a logical description of the information which the application needs from the user and the style of interaction which should be used (for example, the user can choose from a set of mutually exclusive options by pushing a *radio button*).

A template is a collection of related controls which are presented to the user together. Controls can be combined to create a panel to present to a user in a

Figure 2 Appearance of nested windows on the screen



window. Some types of panels are common across many applications—for example, panels for selecting choices. Common User Access defines some standard panel types, which are composed of fundamental elements such as panel titles and scrolling information. The presentation interface supports controls (Table 1) which make it easier for an application to support the standard panels defined by Common User Access.

An application can build a panel by combining the appropriate controls. This approach allows an application to control the contents of the panel at run time while still taking advantage of the Common User Access conformance supported by the controls.

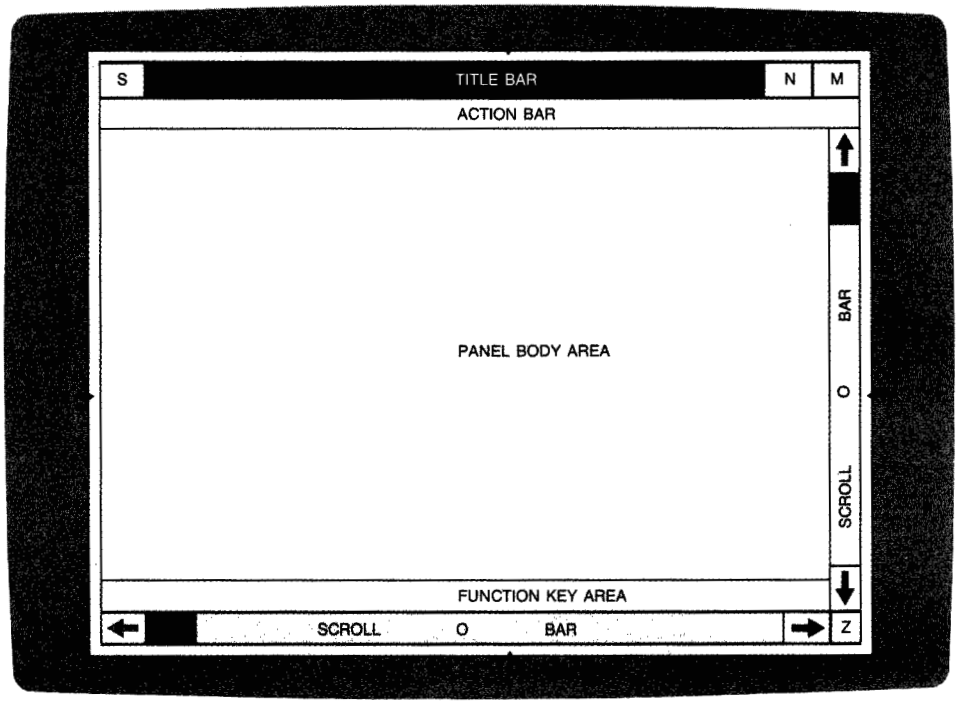
A template can be created to combine a set of controls in a predefined fashion, and an application can build a panel using such a template without having to consider the layout of every control in the panel. This approach reduces the complexity of the application, as it deals with a single template rather than multiple controls. Of course, the application loses some flexibility when this is done.

Table 1 Presentation interface controls

Button Control	Presents a fixed set of choices from which the user can select. The control has options to support single-choice and multiple-choice selection fields.
Edit Control	Presents a single-line text-input field
Static Control	Presents constant information
List Control	Presents a scrollable list of items
Menu Control	Presents action bar choices or pull-down menus
Scroll Control	Presents scroll bars which allow the user to request scrolling of the screen contents and associated area
Size Control	Presents a window border which allows the window size to be changed
Title Control	Presents a window title

Certain combinations of controls are common to many types of Common User Access panels. The presentation interface provides a *create standard*

Figure 3 A standard window



window command which automatically generates the controls for a window with the structure shown in Figure 3.

An application can fill in the panel body area with application-specific information. The presentation interface provides the physical consistency within a control, such as the way a set of buttons are displayed and how they are activated. The application is responsible for the labels on the buttons and for the action taken when a button is activated. Thus, the application is responsible for syntactic and semantic consistency.

This level of interface is appropriate for an application which is still very visually oriented but has more structure than an application for creating free-format pictures. The main part of a business graphics application is an example of this type of application. The output of a business graphics application is a picture determined by the data being presented and the options that have been selected. Controls and templates can be used to collect values for the options to be used in drawing the data, such as the type of graph to produce.

The example of a business graphics application also shows why the presentation interface provides a spectrum of interfaces rather than discrete levels. Much of the input to a business graphics application involves specifying values for predefined options, but there is a need to interact directly with the image to position chart annotations. The application can use the range of support provided by the presentation interface to handle both types of interactions.

Dialog manager panel layout. The next level of interface allows an application to delegate the layout

of its panels to the dialog interface. The application determines the variable information which is presented to the user and the sequence of panels to present. The dialog interface controls how the panel appears on the screen and how the user interacts with the panel.

All physical consistency is provided by the dialog interface. All syntactic consistency which is defined in Common User Access to be standard across all applications is provided by the dialog interface. Semantic consistency for a small number of functions that apply to all applications is also provided by the dialog interface. (For example, the *keys help* command displays a list of the keys used by the program and their functions. The processing of this request is transparent to the application.) The application is responsible for the application-specific syntactic consistency and semantic consistency.

This level of interface provides and enforces much of the Common User Access conformance. It is appropriate for an application such as data entry, which has a fixed set of panels to present in a sequence determined by application processing of the user's input.

The panels to be presented to the user can be determined in advance. The user must complete the input to a panel before the application can respond, because the next panel to be presented depends on the values entered by the user. For example, the application may present one panel when the user attempts to insert a record with a unique key value and a different panel when the user inserts a record with a key value that matches an existing record.

An application which uses the dialog interface specifies only the name of the panel to be displayed. The dialog interface determines what is to be presented on the basis of the definition of the named panel. Since the panel definition is outside the application, it can be changed without changing the application.

The panel language allows the application developer to specify the logical contents of a panel. For example, an application which offers the user a choice from a set of mutually exclusive options would use the panel language to define a single selection field. The choices are described as part of the definition of the selection field. The dialog manager determines how the choices are presented to the user and how the user makes a selection. The current definition of Common User Access results in a set of radio buttons being presented to the user. As Common User Access evolves, this mapping may be changed.

The panel language allows field validation rules to be specified. For example, a field may require numeric values within a specified range. Another field may require the input to match a specified pattern (e.g., a social security number must have three digits, a dash, two digits, a dash, and then four digits).

When the validation rules are defined in the panel, the user can be prompted for correct input by the dialog manager. Since the prompting and error responses conform to Common User Access, the application developer does not have to provide this code. The result is an application that is highly interactive, with no need to implement the details of the interactions in the application.

The panel language permits definition of fields through which arrays of information are presented. If the area within the window is sufficient, the entire array is presented. If the area is insufficient, the dialog manager automatically introduces scrolling controls.

In all of these examples, significant programming effort is moved from the application to the dialog manager. The panel language is used to specify the application's logical requirements. The dialog manager maps these requests according to Common User Access to take best advantage of the specific display device which is used. The result is a highly interactive application with much of the interaction delegated to the dialog manager.

Dialog manager panel navigation. The final level of interface adds navigation independence. The application determines what variable information is to be presented to the user. The dialog interface determines how the panels appear on the screen, how the user interacts with the panels, and the sequence of panels presented to the user.

This level of interface is appropriate for an application with a fixed set of panels presented in a sequence which can be predetermined. An application which leads a user through a set of steps to perform a complex task is an example of this type of application. An application which requires more input parameters than will fit on a single panel can also take advantage of this level of interface. All of the input parameters are collected before the application begins its main processing, even though multiple panels must be displayed.

There is one type of panel navigation which Common User Access requires that an application support—this is the navigation to help information.

When a user is not sure what to do, Common User Access specifies a number of levels of help information which should be available. The panel language allows help information to be associated with specific fields, panels, or an entire application. The dialog manager directs the navigation to the appropriate help information and the navigation within a set of help panels.

When a user is not sure what to do, Common User Access specifies a number of levels of help information.

An application becomes essentially a set of panels which collect user input and a set of application-specific processing routines. The dialog manager is used to connect these pieces together. This is similar to the presentation interface concept, where the application is a set of window procedures which react to user input. However, where a presentation interface window is a blank slate which the application must fill in, a set of panels is predefined for a particular application, and no application code is required to tailor their appearance further.

All physical consistency is provided by the dialog interface. All syntactic consistency defined in Common User Access to be standard across all applications is also provided by the dialog interface, as well as all semantic consistency related to panel navigation. The ability to move forward and backward through a sequence of panels and the state of information on the panels are also supported by the dialog interface. The application is responsible for application-specific syntactic and semantic consistency.

The panel definition language allows navigation to be developed independently of application logic. The application names the first panel to be used, and the dialog interface uses the panel definitions to determine what panels should be presented and when to return to the application. The panel definitions can be changed to modify the navigation without changing the application.

An action field is used to define the navigation performed by the dialog manager. The action is based on the user's input, and can be the display of another panel or the invocation of a command or program. As long as the action is the display of another panel, the dialog manager remains in control. The invocation of a command or program is used at a point where application-specific processing is required.

Portable applications. To provide portability of applications across SAA environments, the same layers of interface must be supported in each environment. Differences in presentation capability across environments have required careful design of the interfaces, particularly the lowest-level presentation interface.

The personal computer allows for high interactivity, in which an application can react to each keystroke. A remote display connected to a host system cannot support this type of interactivity, but the presentation interface must be the same across this range of displays. For this reason, the presentation interface provides input to the application as a series of messages representing events. A portable application must be written with the understanding that some events can never occur on some devices (for example, the individual keystrokes will not occur on a remote display, but the "Enter" event which submits a pull-down window or an entire panel to the program for processing occurs in all environments). By reacting to the events presented to it, the application can run unchanged across the range of environments.

The levels of presentation interface and dialog interface allow an application developer to decide how much of the user interface consistency will be provided by the application and how much will be provided by the enabling interfaces. The lower-level interfaces give the application developer more control and require more implementation effort; they also require more care when a portable application is being developed.

Possible extensions. The higher-level interfaces decrease the application implementation effort and

make it possible for an application to take advantage of new user interface techniques without change to the application. For example, a set of pushbuttons implemented by the application as a set of mouse-selectable images using the lowest-level presentation interface may look exactly the same today as a set of pushbuttons implemented using the presentation interface pushbutton control. In the future, the presentation interface can be extended to support voice input to push a button without changing the application. The pushbuttons implemented by the application would not take advantage of voice input until the application was modified. The higher the level of interface used by an application, the more changes can be accommodated without changing the application.

The Presentation Manager in OS/2 supports application management functions which are extensions to the SAA presentation interface.

Over time, the common programming interface can be extended to move more functions outside the application. Even with the highest level of the dialog interface today, most of the semantic consistency is the responsibility of the application. It is possible to standardize more of the operations on objects and decrease the amount of function that must exist in each application.

One example of this can be seen if an application is considered to be an object with which the user works. A user may need more than one application to complete a job. These applications may run on more than one system. There are operations on applications which are needed to accomplish this—in particular, the user must be able to start applications, switch between applications, and stop applications.

Systems Application Architecture provides the interfaces that are needed to take advantage of the one SAA environment which does have application management functions—Operating System/2™.

The Presentation Manager in OS/2™ supports application management functions which are extensions to the Systems Application Architecture presentation interface. Applications written in any of the SAA environments can take advantage of these application management functions by using the SAA communications interface to connect to an OS/2 workstation.

An application written in this cooperative fashion has its user interface running on the OS/2 workstation. The remainder of the processing takes place on the host system.

With this approach a user has a consistent view of application invocation. The Application Manager in OS/2 presents the user with choices of applications, and the user is not aware of which applications run locally and which run on any of the remote hosts to which the user is connected.

When a cooperative application is started, it establishes a communication session with the appropriate host. The user is not aware of the steps involved in performing authorization and initialization of the host portion of the application. The work to accomplish this is contained within the application.

A cooperative application can take advantage of the interactive capabilities of the workstation; in fact, the user interface can look the same as the user interface of a local application. This provides consistency for the user regardless of the system on which the processing occurs.

Over time it is possible to extend the Systems Application Architecture Common Program Interface (CPI) to decrease the implementation effort required for such a cooperative application. For example, services could be provided to set up the environment for host code automatically.

As with any higher-level interface, this would also decrease the flexibility of the application. Systems Application Architecture has started with the most general functions and can be extended over time with more specific functions as the need develops.

Summary

The user requires semantic consistency, syntactic consistency, and physical consistency across all ap-

plications. Every application must follow a set of rules, Common User Access, to achieve this consistency. To the extent that enabling interfaces are provided to support and enforce this consistency, the application developer's job is simplified.

Systems Application Architecture provides two distinct interfaces to enable the user interface: the dialog interface and the presentation interface. Each of these interfaces comprises a spectrum of services that allow the programmer to adjust the trade-off between consistency and control. The higher the level of interface used by an application, the more changes can be accommodated without changing the application. The lower the level of interface used by an application, the more control it has over its appearance. The presentation interface and the dialog interface work together in an application to provide the full spectrum of user interface enabling.

The application developer benefits from the increased productivity and consistency of the higher-level interface, and is free to choose the level of interface that is appropriate at each point in the application.

Cited references

1. *Systems Application Architecture Common User Access: Panel Design and User Interaction*, SC26-4351, IBM Corporation; available through IBM branch offices.
2. *Systems Application Architecture Common Programming Interface Dialog Reference*, SC26-4356, IBM Corporation; available through IBM branch offices.
3. *Systems Application Architecture Common Programming Interface Presentation Reference*, SC26-4359-0, IBM Corporation; available through IBM branch offices.

Steven A. Uhlir *IBM General Products Division, 555 Bailey Avenue, San Jose, California 95141.* Mr. Uhlir received his B.S. in physics and M.S. in computer science from Stanford University, both in 1977. He joined Bell Labs in 1977 as a software developer on a microprocessor-based PBX project at Holmdel, New Jersey. Mr. Uhlir joined IBM in 1978 at Santa Teresa, California, and has worked on a number of database-related projects. His most recent assignment has been a technical staff position with the business professional products manager responsible for improving interaction of products to satisfy users' needs.

Reprint Order No. G321-5327.