Designing SAA applications and user interfaces

by W. P. Dunfee
J. D. McGehe
R. C. Rauf
K. O. Shipp

This paper describes a framework for developing applications that conform to Systems Application Architecture (SAA). The paper shows a high-level approach to creating a design; it gives examples of early modeling work with the user interface; and it appraises SAA through the eyes of several system designers. The usability of user interfaces has been evaluated through the modeling of office tasks. That experience is described, showing the influence of the SAA Common User Access (CUA) on the model and the influence of the model on CUA. Discussed is a design for distributed applications that fit within the SAA framework and the influence of SAA on the design of integrated distributed applications.

We consider in this paper first the creation of a major SAA application by a team of designers. We also consider how that team designs and evolves a user interface that serves each unique user yet conforms to the principles of Common User Access (CUA). The goals of our team and those of our customers were the same, as follows: Our customers are requesting

- Broad functionality across personal, office, enterprise, and industry-oriented user tasks
- Applications with easy-to-learn, easy-to-use characteristics
- Applications that coexist with their existing products
- Applications that share data

The design team, in addition to meeting these requirements, must design a homogeneous, usable application, as it would be perceived by a user. We also must build a set of cooperative-processing applications and services, as they would be perceived by a system designer who is using SAA services where available. We worked with four SAA environments—MVS, VM, OS/400™, and OS/2™—to design a set of replaceable, extensible applications and services, as they would be perceived by a systems integrator.

This paper describes a framework designed to meet these goals. It discusses some of the criteria one can use to determine whether an application conforms to SAA principles, and it explores ways to organize applications to address today's environment. It describes a high-level design—a structure into which to drop applications. We also show user-interface examples from early modeling work that we studied to make design trade-offs. The model illustrates some of the alternatives designers can consider in building a user interface. Finally, the paper describes some benefits of SAA and explores potential SAA extensions to be studied further. We discuss some of our experiences and assess the role SAA is playing in achieving our design goals.

The designer's framework

Our first design focus was to meet customer requirements for ease of learning and use, which we did by concentrating on a user interface. We asked the design group to create a user interface through which it would be obvious how to complete most tasks.

We surmised that we could design access to each function in such a way that executives, secretaries, and technical professionals could obtain their work result with no errors. To them the tasks would be straightforward. We also believed that if typical common functions were consistent across applications and across systems, the user would acquire skill within one application or on one system and would usually be able to transfer that skill to other applications and systems.

Another design focus was on using the evolving common communication services of SAA.

The methodology the group used was to develop a model and then measure user time on each task, the number of user errors, and user satisfaction with the model. Through iteration, the group produced a set of design principles that contributed to the original definitions of SAA Common User Access (CUA). We describe some of those principles later in this paper.

Our second design focus was on using the evolving SAA Common Programming Interface (CPI). Although as designers we were familiar with one or more of the systems environments that ultimately became SAA environments, we had little initial knowledge of the proposed SAA CPI services. Over time, we created a design structure that used the CPI services. The structure we chose allowed for replaceable, extensible components, and thus it allowed us to accommodate minor differences in CPI services as they were staged across SAA environments. This work continues to the present, as our detailed design makes effective use of the available services.

Another design focus was on using the evolving common communication services of SAA.[1] Application designers should use precursor components to the announced SAA components when they expect the communications requirements of an application to change or when they know of a need for future communication functions. A reasonable approach is to select system components that offer Advanced Program-to-Program Communication (APPC) functions so that applications contain a migratable, connectable interface. By using APPC, the program developer can move one end of the conversation to the CPI for communications and still maintain connectivity.

Our final design focus was to organize the internal design process so that development could be managed successfully. Because of the size and complexity of some SAA projects, managers may need to group functions into packages their development teams can control. As an example for this article, we have grouped office functions into three major applications—Personal Services, Decision Support, and Editing—and three major services—Library, Directory, and Distribution of Mail. (See Figure 1.) If required, the development of one or more applications or services could be dispersed geographically. However, the user must be able to go effortlessly to any function in such a way that our packaging is not apparent.

A reader may wonder whether the physical boundaries of a distributed design can be invisible when viewed from the end-user interface of an application. By grouping related functions, we must ask our designers and developers to focus on presenting the application functions to the end user as homogeneous functions rather than as a set of individual components. However, there are still boundaries, and we have had to develop tools (both for programming and for process) to ensure that developers write functions with consistent interaction styles and terminologies.

The concepts of SAA encourage us to use time-tested packaging guidelines[2] whereby environment-specific logic is isolated from an application's processing logic. Each application or service contains related functions. Each function segment depends only on the final state of another function. Each function is unique. If a function is needed by more than one application, it is packaged as a common tool for all applications. For example, we believe developers must use common functions for creating windows, action bars, and messages so that users see the same objects, take the same actions, and get the same results across applications.

Using the SAA Common Programming Interface (CPI). SAA governs software interfaces, conventions, and protocols for application development and pro-

Figure 1 Functions divided by management areas

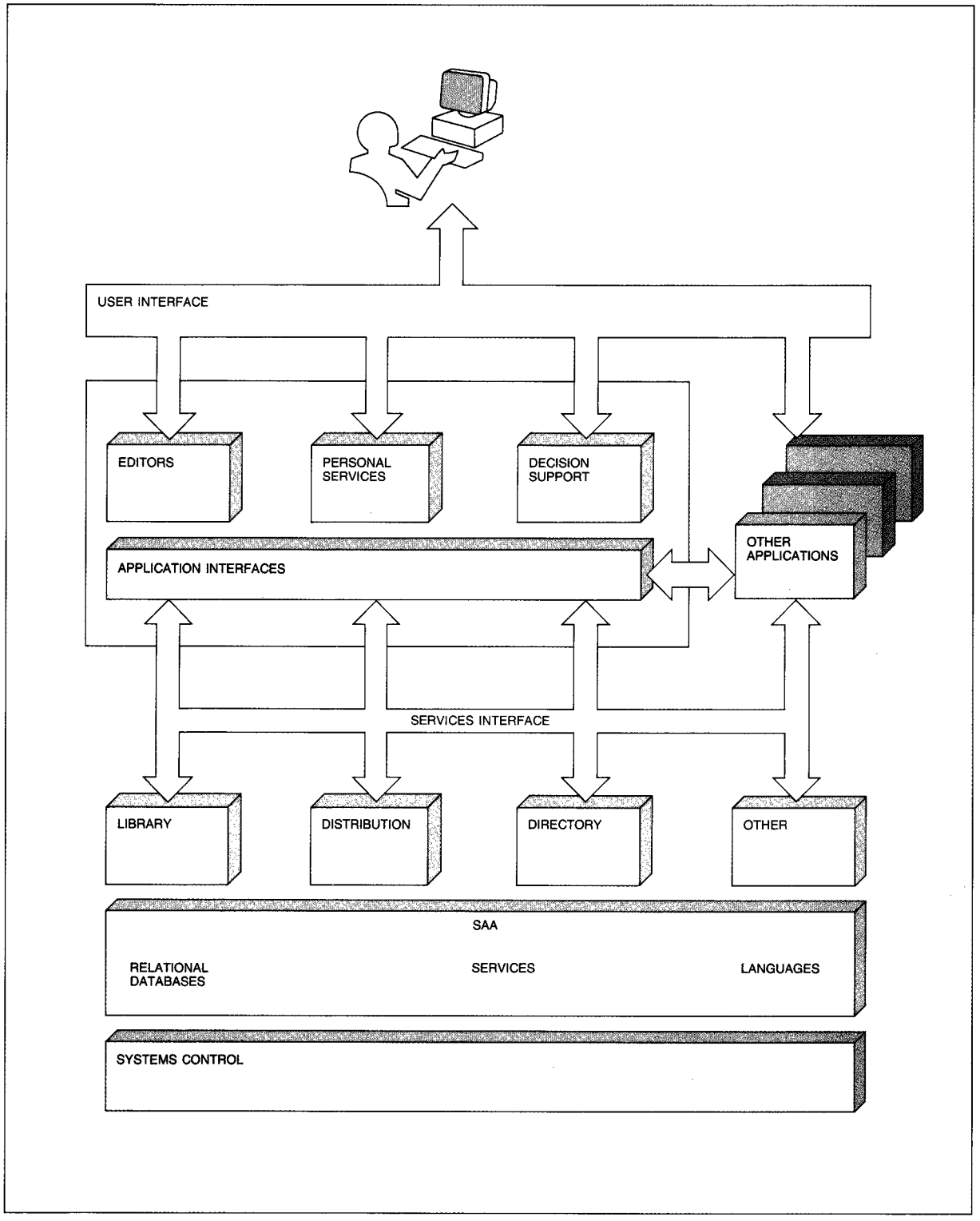
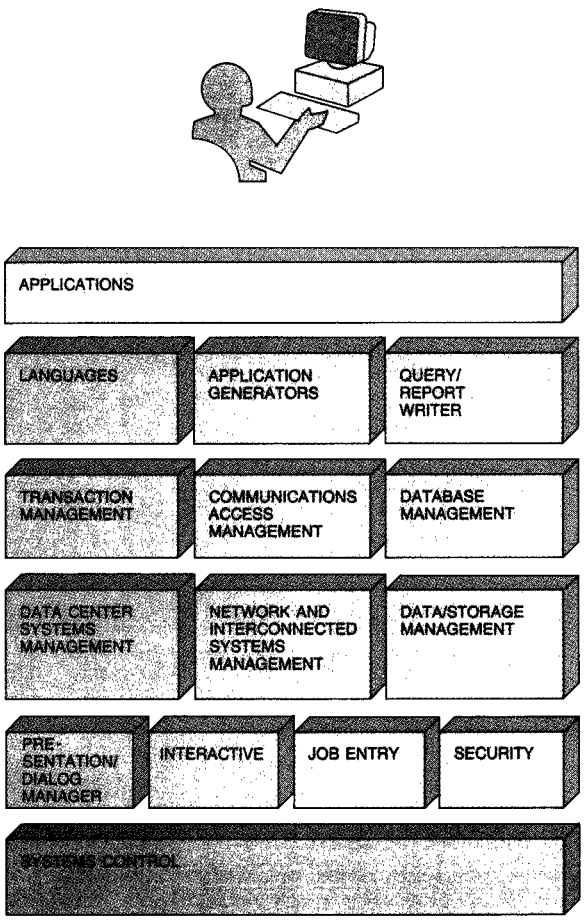


Figure 2 General software structure



vides consistent, durable interfaces in IBM and other software products as a means of achieving increased programmer and end-user productivity, enhanced ease of use through consistency across applications, improved communications capability and usability for enterprise-wide solutions, and increased return on a customer's information systems investment through a more effective utilization of programmer resources and user experience.[3]

This and other papers in this issue show that an application developer can use common services across the SAA environments. Thus, the application developer can write logic once, for example, to create, store, update, print, mail, and receive an object using similar steps (with similar results) and use that logic across all SAA environments.

Because the MVS, VM, OS/400, and OS/2 operating systems provide the components of an SAA base—programming languages, enabling services, and communication services—application developers can offer applications with comparable appearance, operation, and results across SAA environments, using the same application logic. Thus it is to our advantage to use SAA services whenever they are available. Figure 2 shows this relationship between applications and the system services they use. The application layer uses the services provided by all other SAA layers. Consistent services across SAA environments allow us to offer consistent applications across environments.

Using the SAA Common User Access (CUA). We began our design under the assumption that the user interface at the workstation must look the same and work the same across all SAA environments. One methodology that we used to achieve this needed consistency was to conform to CUA rules, which specify the basic interaction techniques for user interfaces.

Although rules are effective for achieving consistent terminology and visual fidelity, they cannot overcome inconsistencies in user process (the sequence of user actions needed to achieve results) across the system environments. In general, these inconsistencies are caused by the following:

- Computer process differences (the internal logic structure of application and operating-system software)
- Hardware features (keyboard versus mouse, color versus monochrome, and screen and printer resolution)

An SAA goal is to remove process inconsistencies in common function. While work is underway to achieve this goal, we expect to minimize inconsistencies in our applications through the use of currently defined common languages (in our case, the C and REXX CPI), common database access, and common communications access across the SAA environments.

An SAA application. An application is an SAA application when it

- Conforms to CUA
- Uses applicable SAA interfaces and protocols
- Uses relational database
- Runs in applicable SAA environments

An SAA application is encouraged to

- Exploit cooperative processing principles
- Use programmable workstation to provide the user interface
- Share function and data with related SAA applications

When an application designer encounters a condition where an exception to these rules seems appropriate, the designer should be able to justify the exception on technical or business grounds. For example, a designer might want to substitute other code to perform a function provided by a CPI service. Some factors the designer should understand are the expected life of the function, the number of SAA environments involved, the potential cost (both immediate and long-term as the SAA environments evolve), and the implications for factors such as performance and security.

Because our goal is to build one application solution across four SAA environments, we prefer to avoid capabilities that are unique to a given environment. On the other hand, unique capabilities also have worked to our advantage. We use all of the extensive user interface capability of the intelligent workstation—some of which is above the SAA presentation interface base—to offer a user an effective interface for high-performance access to application functions. We also use the security features of the Resource Access Control Facility (RACF) in the MVS and VM environments, and comparable features in the OS/400. Few of these features are currently available in the OS/2 environment.

We plan to convert to new SAA services when they become available. In a few areas where critical functions are missing, we are writing temporary fillers, always trying to design the functions in such a way that they can fit within the SAA interface at a later time. Such efforts are used to provide important feedback which influences the evolution of SAA. They ensure that additional functionality is evaluated for inclusion in SAA as new requirements are identified.

Design concepts within the SAA framework

Designing for portability. New technology and changes in the business process can quickly make software obsolete. To help protect the customer's and our software investment, we are making every effort to design common services that are portable. An application or service is said to be *portable* when one can move it to a new environment with a minimum of time and expense without compromising its function. We encourage developers of common services to use only the set of services in the SAA CPI so as to have maximum code portability.

Designing for cooperative processing. Cooperative processing applications are those for which portions of the application or its services are executed in more than one processor. The designer's objective should be to use each processor for the operation it performs best. The workstation can be used for the highly interactive end-user functions, and the host (or a server on a local area network) can be used to handle shared data.

To meet the criteria for a cooperative processing application, we created the model in Figure 3. The application is built in two processors. The one entry point to the application is through the Application Programming Interface (API), which is shown shaded in the figure. The application logic determines what the end user or the caller wants to do and builds a complete request. If information is missing, the dialog logic presents screens to the end user to gather that information. In our model, the workstation contains all the dialog logic; no interactive services are allowed at the host. The application logic then passes the complete request to the requester, which manages the conversation with either a local service or a host service. When the conversation is complete, the results are returned to the application logic.

Public services, which also have an API, are shared services and are called up by the application as needed. A library and a directory are examples of public services. Public services, in principle, can be installed in the workstation, if they are used by only one end user. In our model, they must be installed in a host (or in a server on a local area network) when they are shared across multiple end users.

A *private interface* is allowed between a requester and a service. Other applications can use the private service by entering through the application's API, but they are prohibited from having direct use of that service. By packaging function as a public or private service, the application designer frequently improves the application's portability, cooperative-processing, and extensibility characteristics.

Designing for extensibility. One of our design goals is to build replaceable, extensible applications so as to integrate IBM software products with other IBM and customer-developed products. This goal can be achieved in part by building on a common base. In years past, this base was either provided by an operating system environment (VM/CMS, for example)

Figure 3 Cooperative processing concepts in which an application is divided between two processors

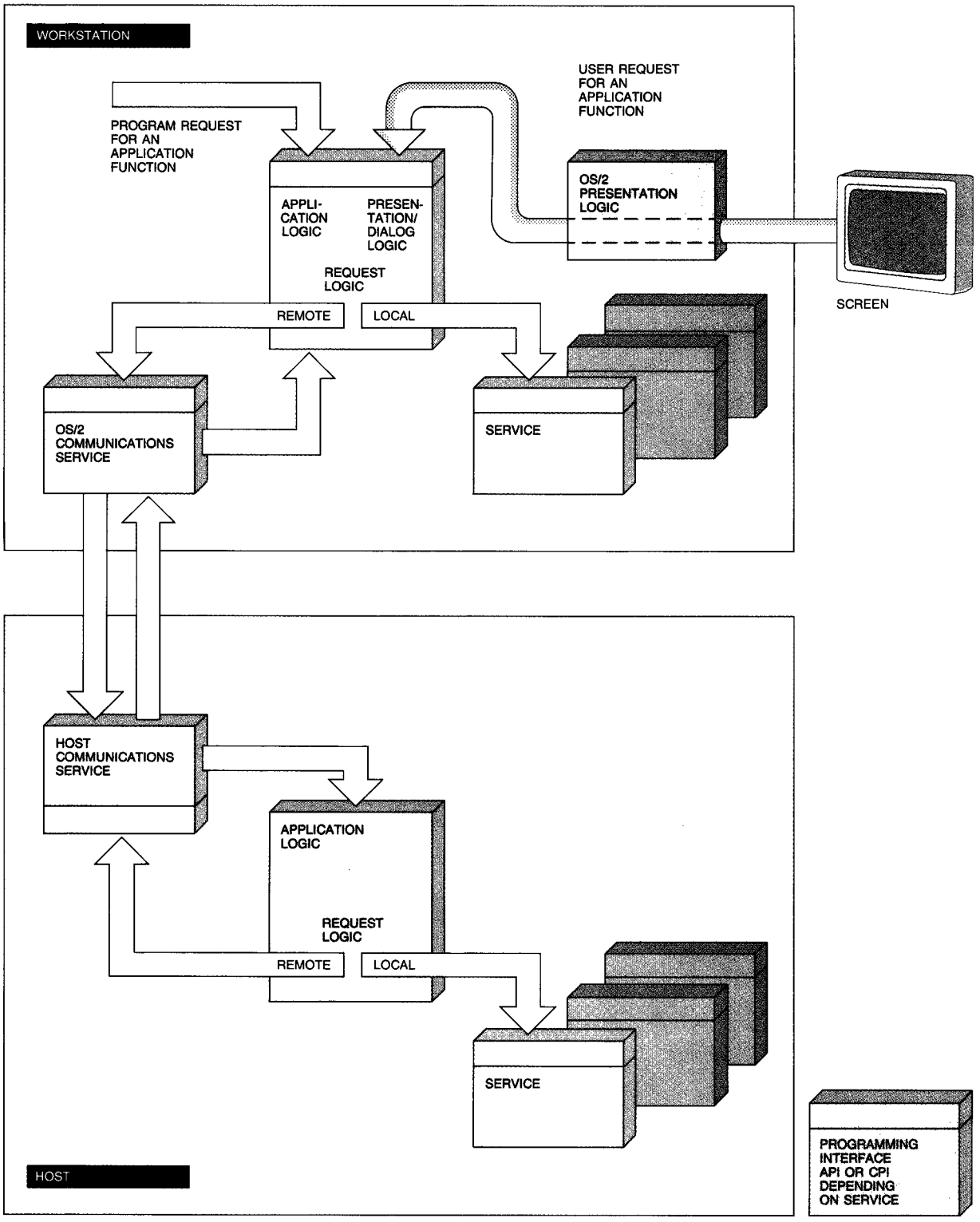
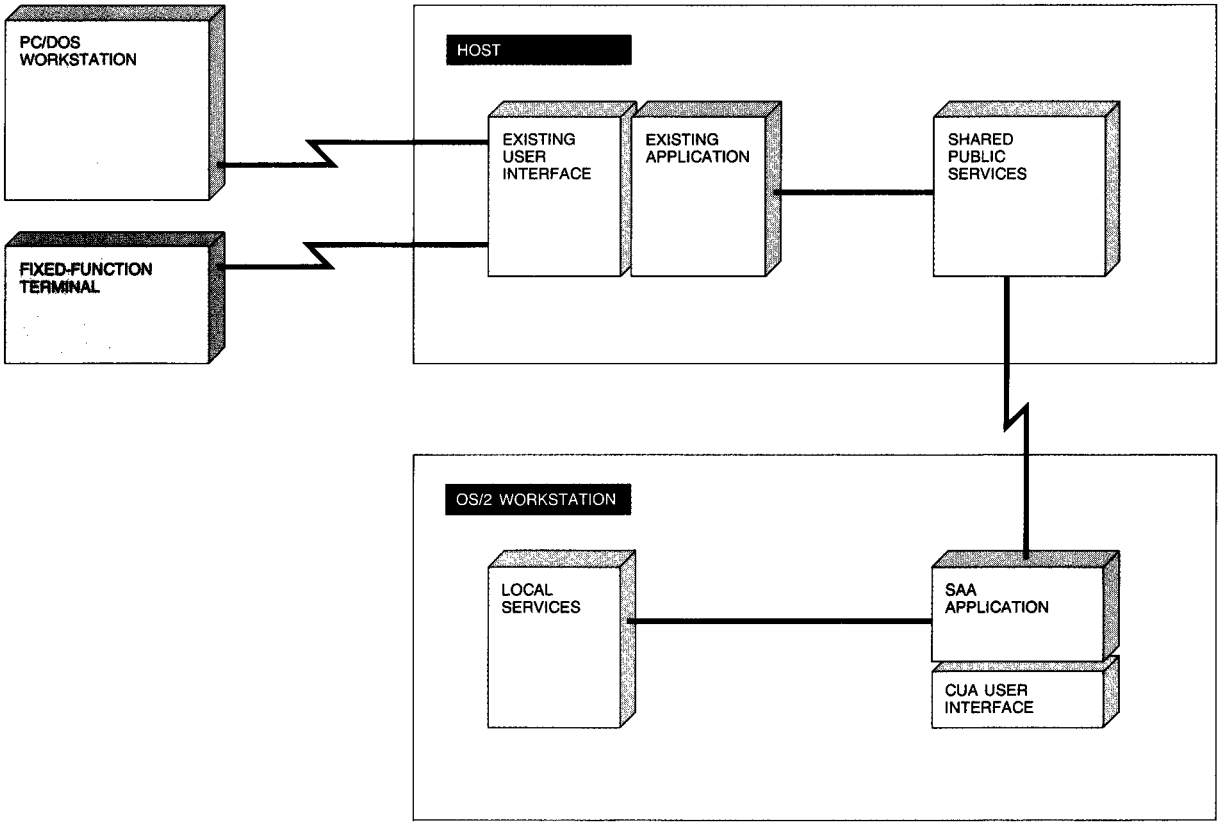


Figure 4 Simplified shared-services structure



or by building an extension on the system environment (PROFS™, for example), or both. All of today's bases provide similar functions in dissimilar ways.

To meet our extensibility requirements, a common set of functions must be provided by SAA. It must be identical in all SAA environments; it must let the user select the applications to be used; it must let the user start, stop, or switch to any selected application; and it must let the user easily replace one application with another.

We found that these extensibility requirements were met by announced function within OS/2 Extended Edition, and other SAA environments are preparing to offer complementary function.[4] In effect, users would decide what applications and tools they wish to use, and then they would use OS/2 tools to help them build their lists of selected applications.

Because of the strengths of OS/2 (its user interface function, a multitasking SAA environment, connectivity to all other SAA environments, extensibility, and tools that help the user), we have chosen the OS/2 environment as our programmable workstation environment. We are designing applications to run in the workstation, and designing shared services to run in the attached host. Thus SAA applications can coexist with existing applications when both use common services. To ensure coexistence with current products, we allow the shared services to be used by those products. (See Figure 4.) Thus, anyone who uses an IBM 3179 display to obtain currently available application functions can access the same shared services as someone who uses a workstation.

The principles discussed in this section form the basis of our design. In the next section, we show examples of how our design reflects these principles, and discuss the evolution of CUA and our end-user interface. We also show a model of an application program which uses the SAA CPIs. Finally, we explore the influences of SAA on our future.

User's view of SAA

The objective of the user interface design is to present application functions to the user as homogeneous functions rather than as a set of unique components. The Common User Access (CUA) of SAA provides the initial concepts for developing user interfaces with these characteristics. In this section, we show how individual elements of CUA are combined by the application designer to meet the user's requirements for function and usability.

Our goal is to design the user interface early and let it influence the design of the application.

Methodology for understanding the user's view. It is generally accepted that an error in a software product is less expensive to change earlier rather than later in the development cycle.[5] Correcting a usability problem in the design of the user interface is no exception. Frequent diagnostic usability testing is important to the success of new applications; the usability test lets us evaluate how well we are doing in meeting our measurable objectives.

Software usability engineering has not evolved to the point where we have algorithms that predict least time on task, fewest user errors, and greatest user satisfaction. As a result, user interface design is iterative. Through iteration, we can successively hone our design until we meet or exceed our objectives.[6]

The output of the user interface design work is the input to the development of a model that simulates the user interface and the user–machine interaction. From our modeling experience, we are writing a set of user interface principles and guidelines. These guidelines form the base for defining the applicable terminology, the way the application actions should be grouped, and the objects (for example, inbasket and calendar) available at the workstation.

Our goal is to design the user interface early and let it influence the design of the application, not vice versa, as frequently happens. Expectedly, our modeling and usability testing influenced both our design and the CUA guidelines. We have made changes to the basic layout, the interaction style, and the user interface components on the basis of those results.

The model we used to evaluate the user interface was a representation of office function but was not a prototype, in that it performed no useful work. Instead, we used "canned" scenarios, so that users did not select functions that had no code behind them. Even with these limitations, we found the model valuable as an early design tool, because it gave us the appearance and operation of the future functions.

We used the model to provide feedback on both the function and the user interface, as shown in Figure 5. Customers and developers gave us valuable subjective evaluations. The testing of data from usability tests permitted an objective evaluation of the interface.

The results of each evaluation were used to update the design of the applications, and the model was retested. If the test results showed that the design met or exceeded our measurable objectives, we "closed" that section of the design. Of course, no design is ever closed because it passes its modeling tests. The design continues to evolve until the last function is added and the usability acceptance test is completed for the entire application (or set of applications).

We find it difficult to describe which change we made to improve or meet any specific usability requirement. Most decisions involve trade-offs, but diagnostic testing allows us to achieve a balance. Our examples in the following sections illustrate these trade-offs, because they are taken from some of the actual modeling work. They do not represent a final product, but show an interim step in the iterative process we used to explore issues. They have survived several usability tests, but they will have to pass many more tests as functions are added to the product.

Ease of learning. The first design objective, that is, making the application easy to learn, requires that the user's next step be obvious. A user should be able to use the function without reading publications and without extensive training. For example, our model evaluators ranged from experienced to first-time computer users. They were given less than 30 minutes of instruction before starting the usability test of the model. All instruction was on navigation,

Figure 5 A methodology for managing end-user interfaces

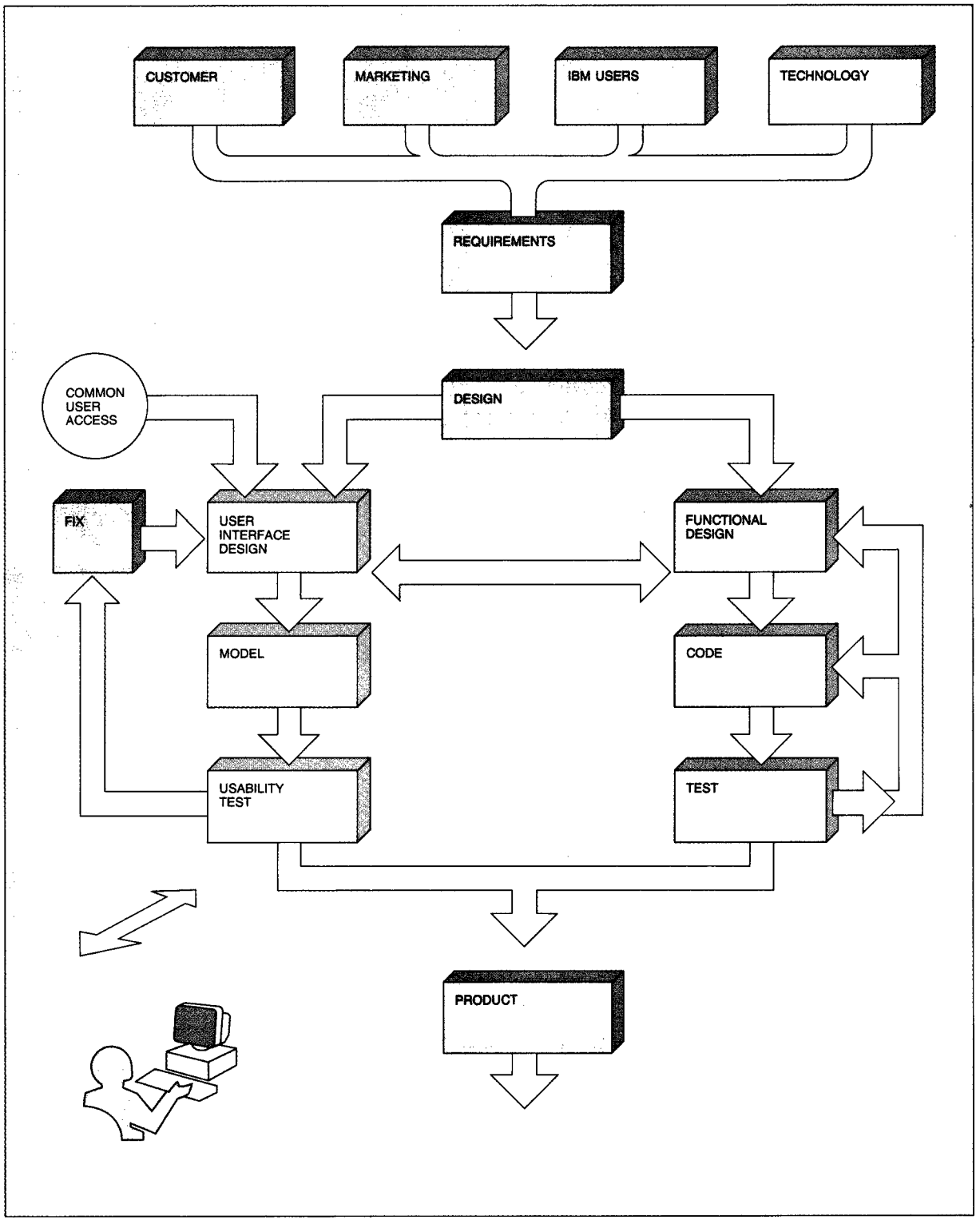
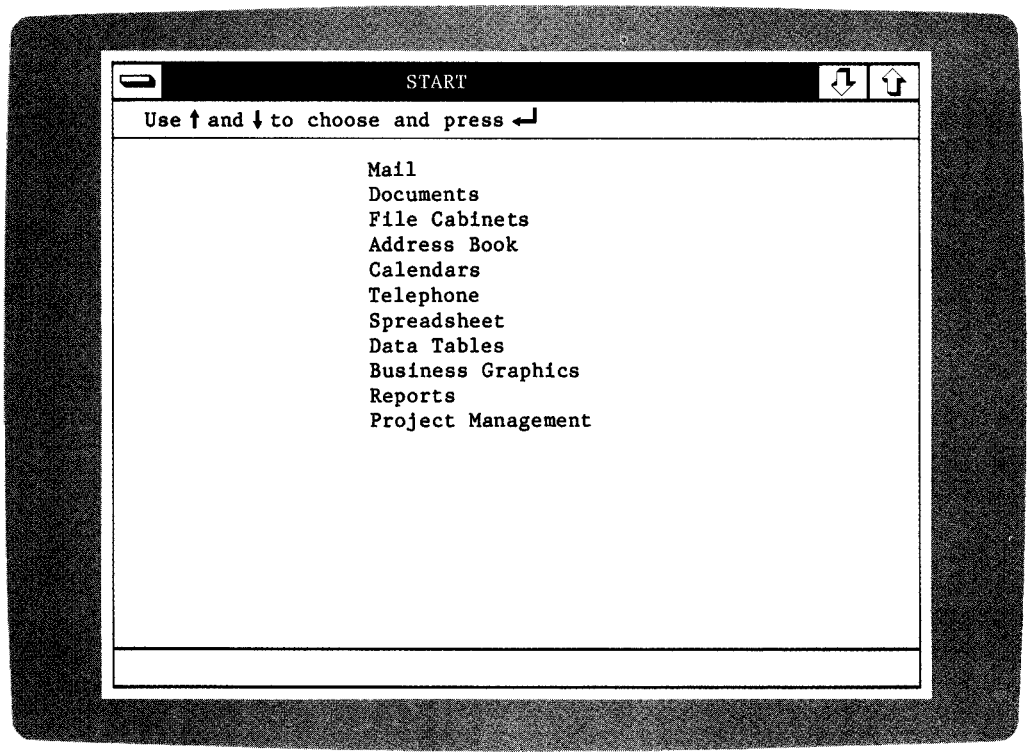
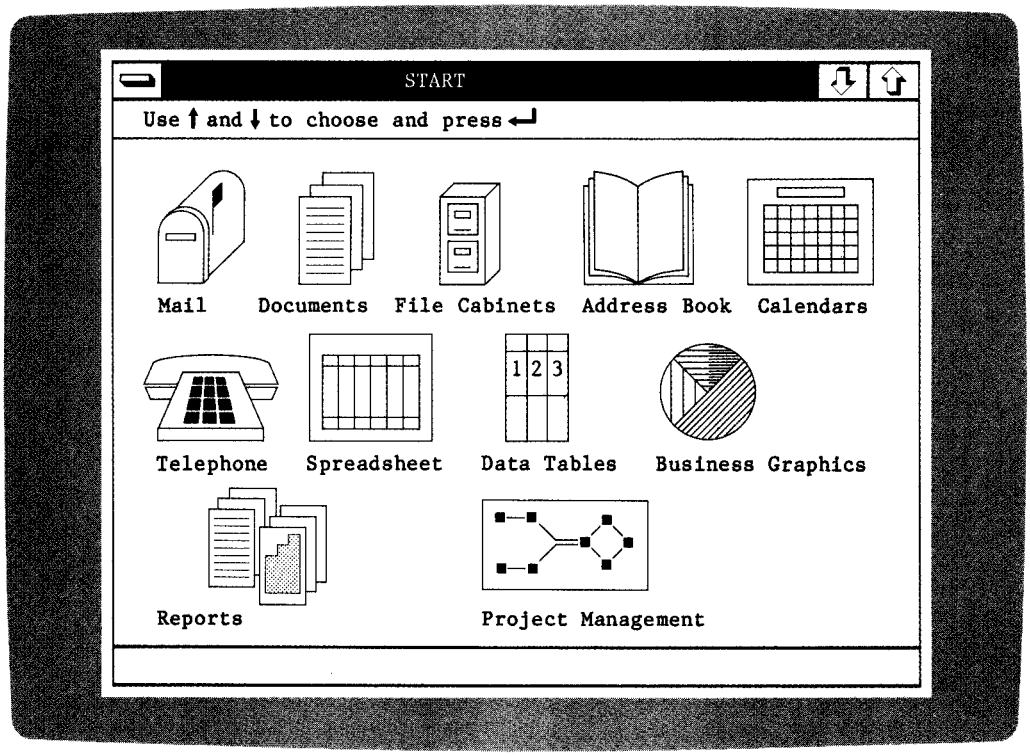


Figure 6 Major application function panels





such as the use of the keyboard to get from one function to another, so that use of the hardware did not significantly affect our results.

The ease-of-learning requirement has been addressed in several ways. One of our approaches was to take advantage of CUA's object–action interface style; that involved having new office applications reflect access to functions by representing them as easily understood objects and related actions. A user's initial recognition of a function (i.e., selecting an object and taking an action on the object) seems related to his or her ability to correlate the object and action to a familiar event, from past experience with the function or a similar function. Figure 6 (top) shows a panel used to select major application functions. The items in the list represent easily recognizable office objects rather than abstract functions. This approach can also be extended into other interface styles, such as the use of icons in place of text, as shown in Figure 6 (bottom). Figures 6 through 13 are an artist's simulation of selected screens from the model and not an exact representation of CUA.

We concluded from the results of our user interface testing that those who clearly understood the objects and actions made fewer errors. The designer's challenge is to find those terms and icons that are obvious for the largest number of users. We also discovered that users migrated well from text to icons, as long as we used the same underlying objects and actions.

We also tried to solve a problem that confronts today's designers, the number of user choices to show on a panel. The following are the guidelines we used:

- Make all function accessible.
- Offer consistency across many application functions.
- Avoid filling the screen with actions or information that might cover the user's data.
- Make the interface extensible without major change.

The problem can be seen in the mail function. When working with an incoming mail item, a user may want to take any one of a large number of possible actions on one item or a collection of items. These actions might include viewing an item, its history, or its status; copying an item or all items; adding reminders; deleting items; or searching for a particular item. Several approaches are used in current products to address these types of requirements.

One historical approach is to perform the selection by using function keys, as shown in Figure 7. The steps followed might be first to select mail (F1), then select the first mail item (F1), and then the send function (F7). At each step a new menu is displayed on which to make the next selection. In this approach, the function keys represent different functions, as the user moves from panel to panel, and the number of panels needed to accomplish the task is greater than with other approaches.

Effective use of the action bar makes the system easier to learn.

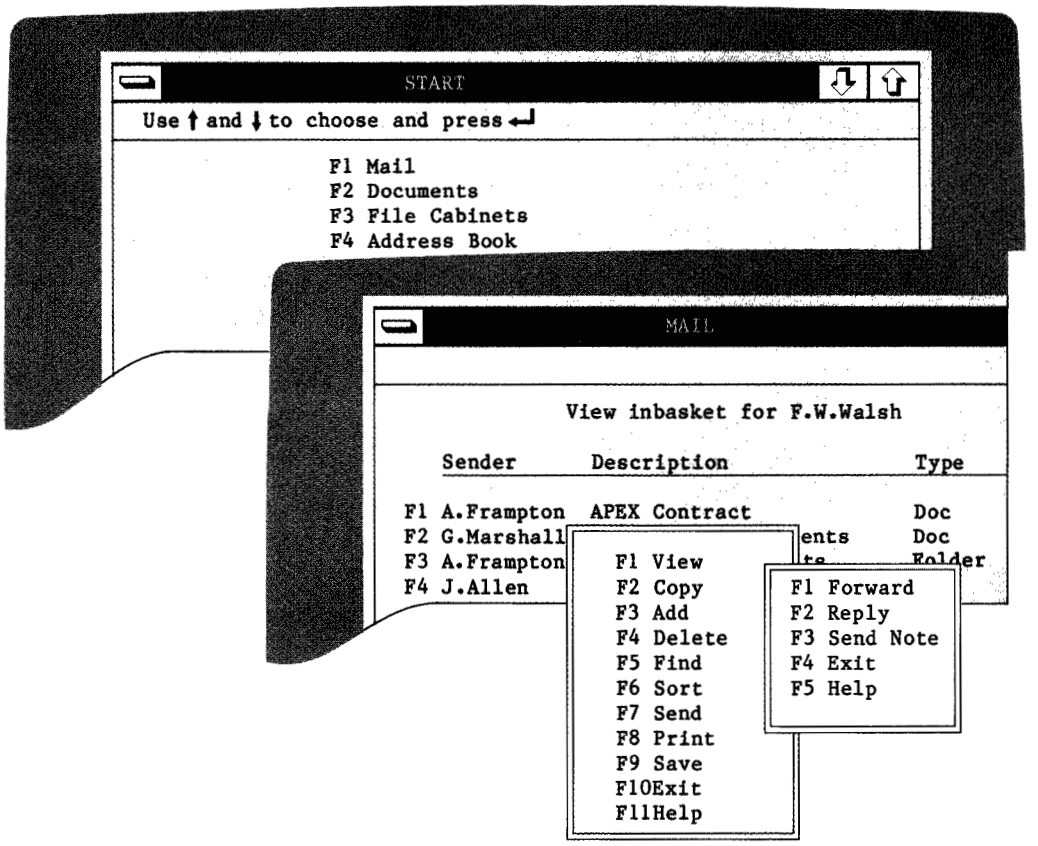
Another approach is to show all the actions to the user at one time. However, too many choices can make the panel unusable (for example, the Forward function in Figure 8 is difficult to find). Also, in this approach, additional space is needed on the panel when a new function is added.

We found that the CUA approach—using the action bar—improves usability. User actions are grouped by category. In Figure 9, the user selects the View category to see the valid actions related to that category. If the user selects the Send category, the View actions disappear and the Send actions appear. This creates an interaction style that is consistent, extendable, and easy to learn.

In joint work with the CUA area, we found that effective use of the action bar is one of the best approaches to making the system easier to learn. This conclusion agrees with the results of user interface activities at Xerox's Palo Alto Research Center (PARC) and the interface of the Apple® Macintosh™.[7] Developing the data, however, allowed us to extend our exploration to the interface questions which are discussed later.

We believe that a consistent placement of actions on the action bar across families of applications speeds ease of use. At a minimum, it improves the percep-

Figure 7 Selection using function keys, a traditional technique



tion of usability. The action bar creates a consistent means for users to see the actions they can perform, and our test users noted consistent placement of actions as one reason for their opinion that the functions were integrated.

To minimize the need for a new or infrequent user to read a manual, we used the CUA message area to display instructions on what to do next. Obviously, extensive instructions become "busy" and take too much panel space. However, we found that short phrases on a single line (i.e., prompts) are quite helpful. As the user completes each step, the prompt changes. This technique allow most users with little or no familiarity with the function to use the application productively.

Occasionally, the user needs more help and selects Help on the action bar. The data derived from usability testing and customer feedback about early versions of the model indicated that our best professional judgments on wording were frequently wrong. Thus, we worked closely with information developers so that text on the message line would lead into the detailed help logic for the model. We concluded that information developers and early testers should help us select and test the prompts and the help panels.

Ease of use. To make the user interface friendly, we studied the following choices:

- Simplifying the interface for all functions
- Allowing the interface to be tailored to individual choices and skill levels
- Ensuring that skills learned in one application can be applied when using another application

Simplicity. Many applications provide functions with so much capability that users can become lost in selections, parameters, and options. Consider, for example, an office systems electronic filing function.

Such a function may provide capabilities for searching office correspondence and computer files, and may do it more efficiently than a manual filing system. The menu for searching for correspondence might be long and complex, but it typically contains only a few critical questions:

- What is the subject?
- Who is the author?

Questions such as the following are often repetitive or of little interest to a particular user:

- What folder do you want to look in?
- What file cabinet do you want to search?
- What language are you using?

The design we propose is to show only frequently changed parameters on the first panel seen by the user. In the search example in Figure 10, seven parameters are shown, and two (Folder and File Cabinet) are prefilled. The prefilled parameters, as well as other preferences such as choice of language, have been set previously by the user or installation. For most operations, the prefilled entries do not need to be changed, and the user need consider only values for the blank fields.

When a change to an option is required, the user needs a predictable technique used consistently across all applications to display and change the preset defaults. Our model used the F9 function key. When the user presses the F9 key in Figure 10, a window containing preset values is displayed. (See Figure 11.) A change to any preset value is a temporary change for this execution of the Search. Otherwise, the user could use the Save function shown in the action bar and make a permanent change to the defaults.

The CUA List function is another memory-aid technique that reduces the need to remember and enter information. The List capability allows the user to display the potential inputs for a parameter on a parameter-entry panel. The user may want to change the folder to be searched but may not know the valid folders that could be searched. In our model, if the F4=List key is pressed with the cursor in the Search-

Figure 8 Selection by showing every choice, a difficult-to-use technique

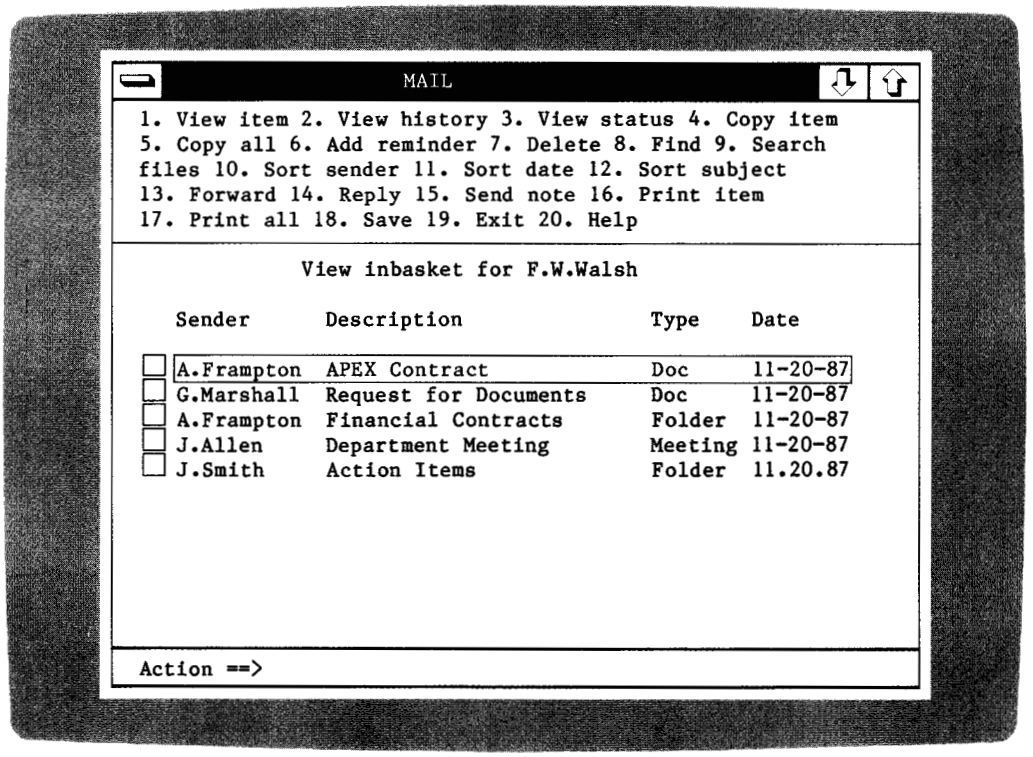
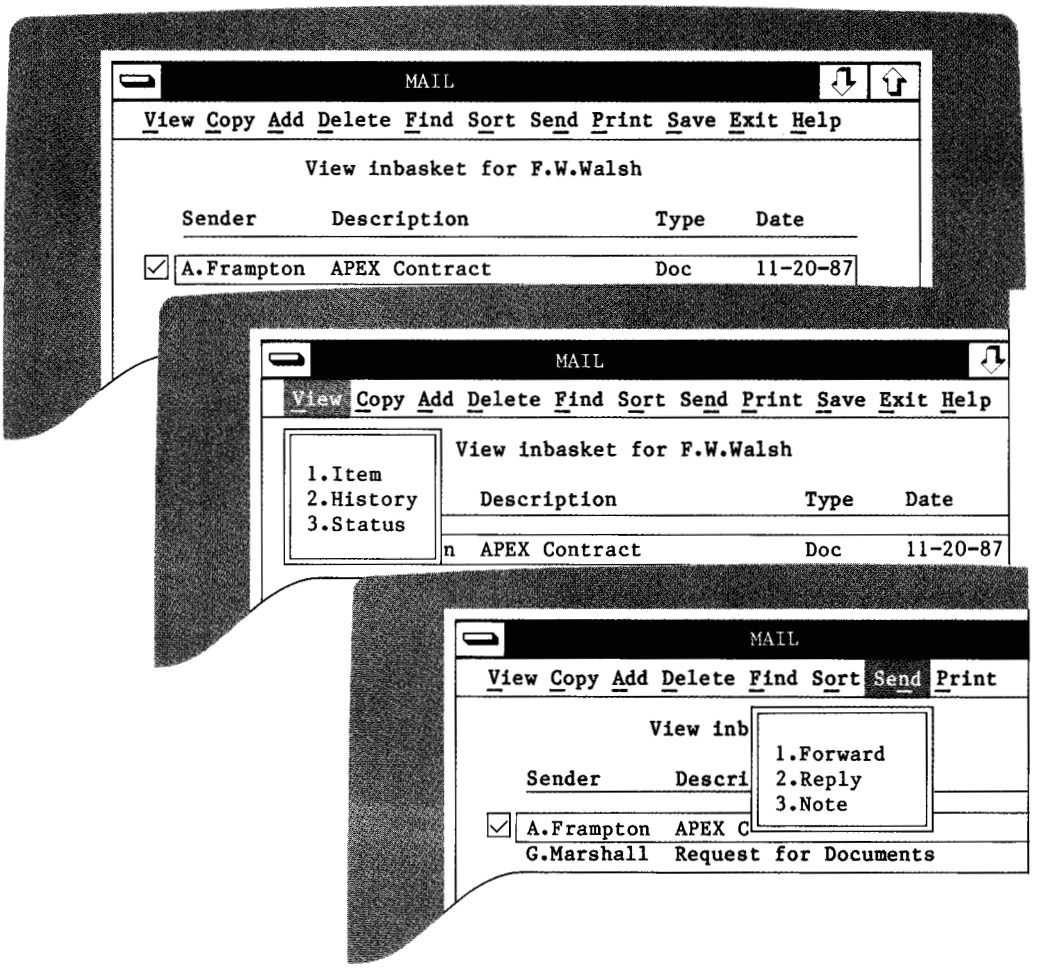


Figure 9 User actions grouped by category



in-Folder field, a list of folders that are valid entries in the field is displayed in a pop-up window, as shown in Figure 12 (top). The user can then select the desired folder, such as APEX Contracts, and the data are copied automatically to the parameter entry field when the pop-up window is exited (see bottom of Figure 12).

Tailoring. The features that make an application usable to one user can make the same application seem difficult to other users. Personal preferences, job assignments, and skill levels are some of the factors that affect a user's productivity with a given application. Most user-interface components should be changeable by the user or installation, including color combinations, the words used on panels, and the language.

One of the main considerations is how to adapt the user interface to the skill level of the user. As users gain experience with an application, they require less prompting and help to complete their work. As they become proficient, users want faster ways to execute a task, and they do not require the step-by-step approach provided by menus. The goal is to deliver a smooth progression of techniques to match the increasing skill levels of users, not separate interfaces. Application developers should consider the new or novice user, the experienced user, and the expert user as they design the user interfaces.

One of the items discussed earlier, the message line, is intended to address the needs of the novice user. In many cases, this information becomes unnecessary to a user who is experienced in the application

functions. Users should be given the choice of requesting or suppressing both the help function and the function area.

Another option—the CUA command line—can be turned on by experienced users as they become expert in the use of the system. Our model contains a command line to give the expert user the ability to bypass the menus and go directly to a function. Users are exposed to command usage by seeing the command (or accelerator function) next to the selections they make while completing a task.

The command line can be used to enter commands in several ways. One way is to enter an action (or abbreviation of an action) and the object on which to perform the action. For example, the user can print the first item in a list by entering the command PRINT 1, or can print the first three items by entering PRINT 1,2,3. All commands take the user directly to the function, bypassing both the action bar at the top of the screen and any intervening panels.

Consistency. As stated earlier, applications must be consistent, but an object, action, or panel can be measured as "consistent" only after a satisfactory answer to the following questions: Consistent with respect to what feature? How do those features relate to overall system goals?

We used several tests when evaluating the model for consistency. We looked for a positive transfer of user knowledge as the user moves among applications and environments. We tested for terminology, style, and procedural exactness both within a function and across all functions of the model. We looked for examples of ambiguity, where the same term or process was used in different functions; we compared the two usages to understand the similarities and differences. Finally, we looked at how the user's perception of these functions changed as the user became more experienced.

Many of our insights helped refine the CUA rules, which define the design elements of a user interface.

Figure 10 Search example

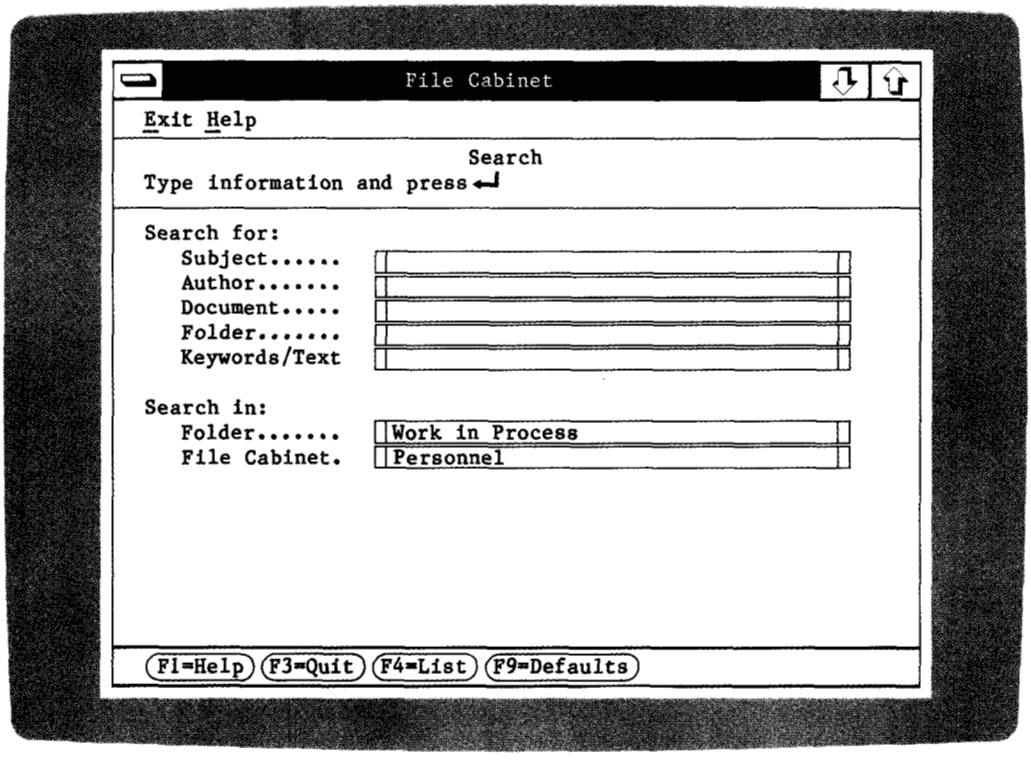
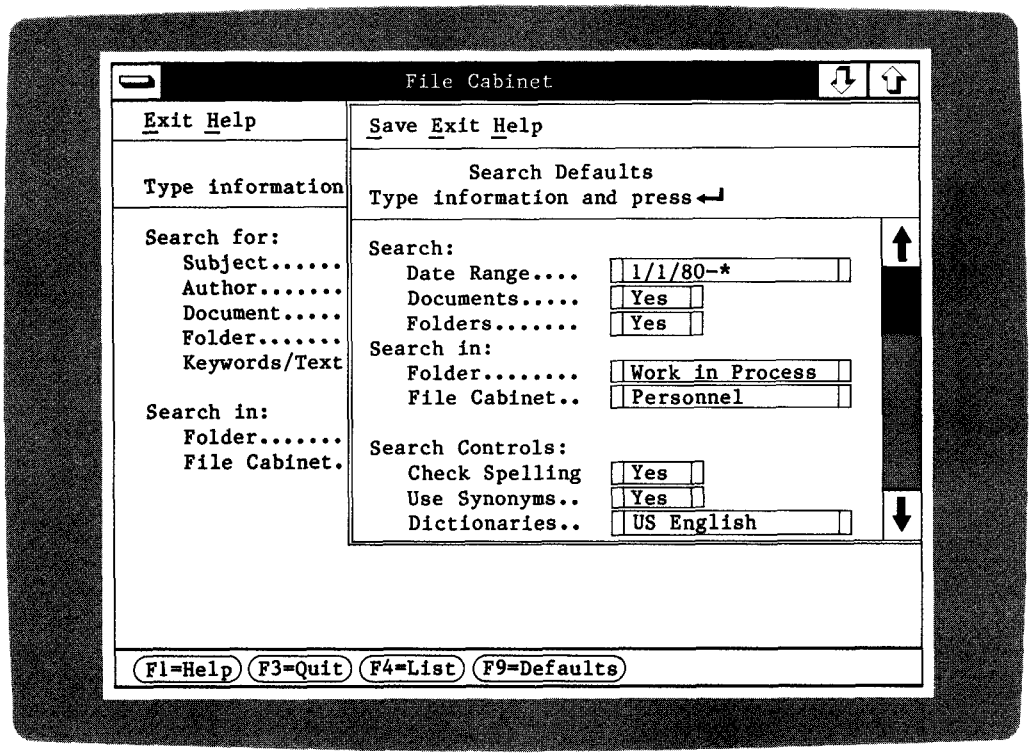


Figure 11 Preset values for Search example



However, it is the application designer who must assess those elements and select the proper combinations.

Some office applications, such as electronic mail, require a critical mass of users who must be using the application before it becomes effective as a means of communicating in an organization. Such persons tell us that they want a positive transfer of learning. When they travel, they want the application to have the same appearance and operating characteristics as it did at home. When they change jobs or move from area to area, they want training on new equipment or applications to be minimal.

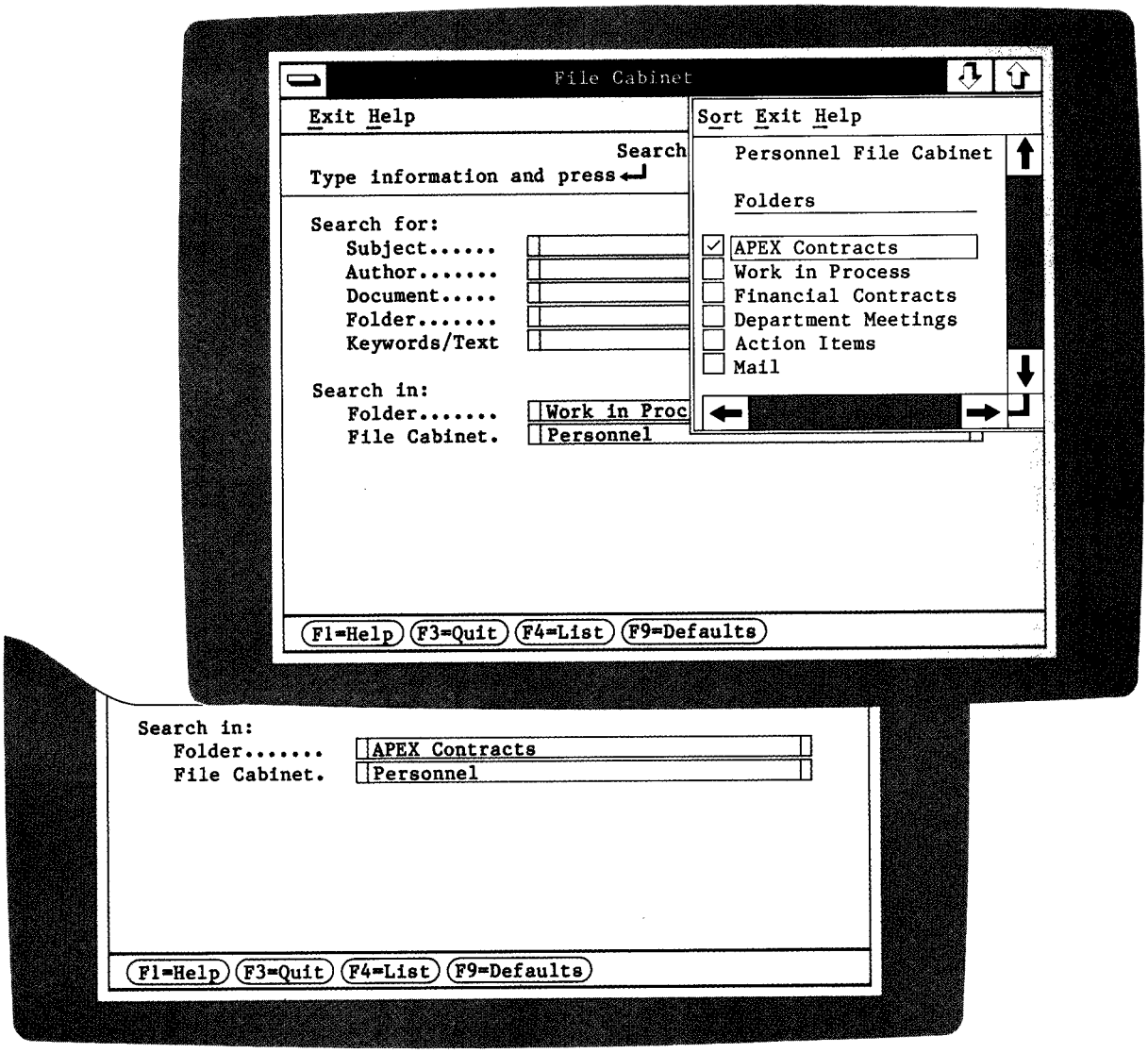
The objective of both CUA and our model is to ensure that transfer between devices is easy, but equipment differences keep them from being identical.

Figure 13 shows the model's workstation Inbasket compared with its fixed-function terminal equivalent. The function is presented in similar forms, and the terminology is identical. The use of the action bar and the pulldown for Send operate similarly. To move forward or backward in the list of mail items, the users press F7 or F8. The workstation user can use the mouse or space bar to make a selection, but both users can cursor through the list to make a selection. When additional capabilities are available on the workstation, they are usually allowed.

Above all, we want to ensure that the use of a function is predictable across applications. A design that takes advantage of the state-of-the-art features of a workstation differs from the design for a fixed-function terminal. These differences can be allowed when users are not expected to move between workstation and terminal. However, they should not be allowed to diverge to such a point that they become unpredictable to the user. Some of the areas we feel important are the following:

- Terminology used for objects and actions
- Categories of actions—how they will be ordered in the action bar as well as what actions are grouped under each category
- Similarity of look, feel, and results of the same action in different applications running on the same workstation

Figure 12 Valid folders for Search in Folder function



We believe we have achieved similarity in "looks" in our model, and we have defined similar interaction styles for similarity of "feel" between workstations and fixed-function terminals. In the following section, we describe some of the activities that help achieve similarity of "results."

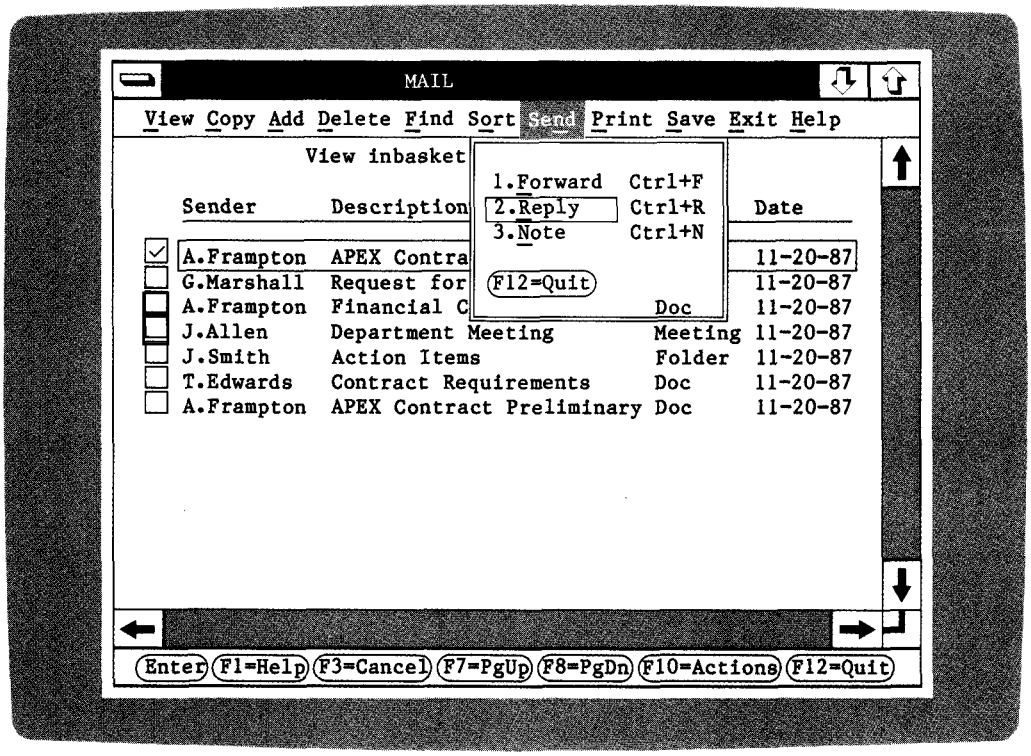
SAA and the application developer's view

From yesterday's stand-alone products we have evolved to today's integrated applications, and we are now designing the cooperative-processing applications to meet a user's enterprise-wide requirements.[8] In this section, we identify the key steps in this evolution.

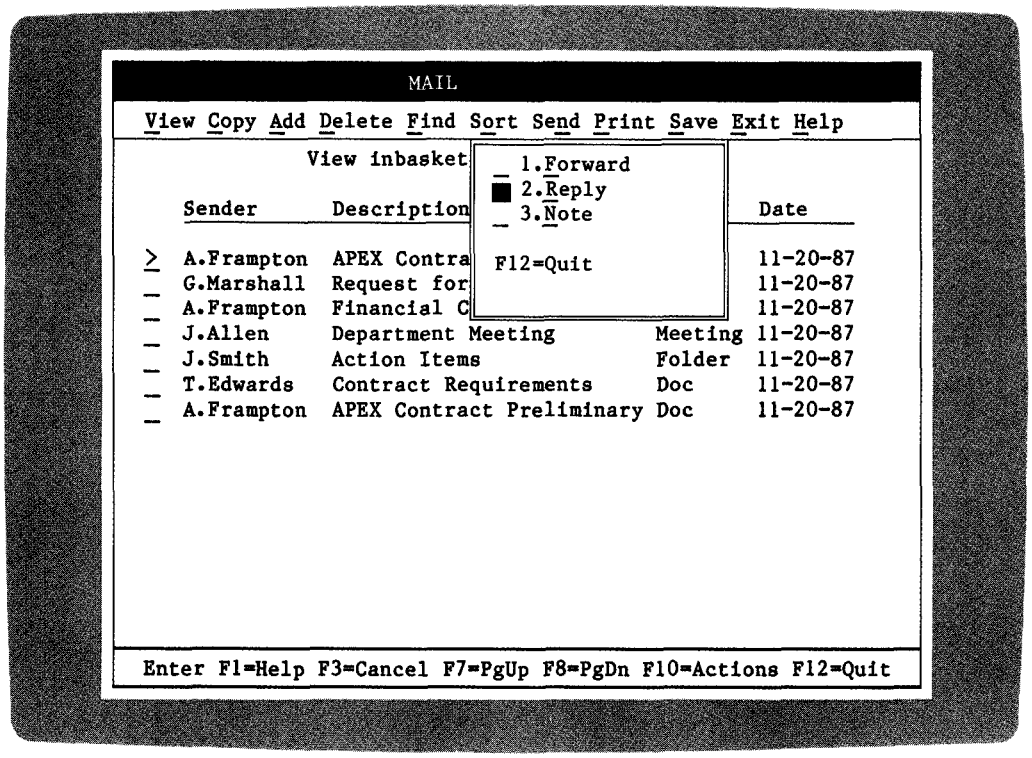
Application integration. In our vision of integrated applications, a user is not necessarily aware of moving from one application to another when multiple applications are needed to accomplish the user's task at hand. To the user, the applications have the same appearance and operation.

Figure 13 Comparison of programmable workstation and a fixed-function terminal equivalent

PROGRAMMABLE WORKSTATION



FIXED-FUNCTION TERMINAL



We have just discussed our view of integration with respect to appearance. We now present our view of integration with respect to application logic. We approach this discussion by assuming that programmable workstations are attached to any SAA host.

Applications that are integrated have several common characteristics. They appear "seamless" to end users; they share or exchange data; and they offer most, if not all, of their functions as callable services. From a user's perspective, once one application has been given information, the user who entered the information should never have to re-enter that information elsewhere. For example, if a user creates a meeting notice, the information supplied can be used to update the user's calendar as well as to notify each meeting participant; when the participant receives the notice and commits to attend the meeting, the participant's calendar is also automatically updated.

Applications can share data through several techniques, such as common data blocks, common variables, and parameter passing. Another technique of data exchange in OS/2 is the clipboard, a service of the OS/2 presentation interface. Thus, when a user wishes to move an object from one application to another, the user can select the object in one application, move it to a "clipboard," and then move it to the next application. This is a standard technique used today by Microsoft® Windows, Apple Macintosh, and others.

We believe SAA enhances today's clipboard techniques by allowing SAA applications to offer users data-sharing advantages over other applications. SAA, by standardizing the object architectures, enables applications that support the clipboard to pass an object that can be widely used. It is even possible to design applications to work together so that the clipboard is unnecessary.

Beyond these data-exchange methods, we are investigating how data, if changed in one application, are automatically reflected in the results of another application. For example, when a user changes data in a spreadsheet window, a chart in a business-graphics window is automatically redrawn on the basis of the new data.

Our proposed mechanism for providing callable application function was shown in Figure 3. Each application has a public Application Programming Interface (API) that other applications may use. As an example, a mail application might provide a callable interface to send an object to a user in the network. The developer of a word processor application might then provide a Send function on its action bar. When the user selects Send, the word processor might have been designed to perform one of the following actions:

We want to select designs that allow us to write code once and use it anywhere.

1. Call the mail application, passing no data, thus requiring the user to identify what to mail and where to send it.
2. Collect all the needed mailing information from the user, then call the mail application with a complete request.
3. Call the mail application, passing information as to where the object can be found, and let the user identify where to send it.

Option 1 violates our principle of not asking the user to re-enter information the application already knows, but it may have to be used for applications that are unable to share data. Option 2 makes the word processor application duplicate code that is required in the mail application, but this option is viable if the developer has a reason for not allowing the user to see mail application panels or to change mail options. Option 3 appears to provide the most function and application integration with the least logic and the least user entry of data. Of course, no one option is right for every situation.

As the word processor uses the mail function, so the mail function may use the directory function, if the addressee is not in the form of a system address. For example, if the mail request is to "send x to y," then the mail function must call the directory function, requesting "provide system address for y." Each application is added to our design in this manner.

Portable applications. We want to select designs that allow us to write code once and use it anywhere. The

model shown in Figure 3 allows us to separate the common service code from user workstation interface code, thus improving the probability that service code can be reused at a tolerable cost.

Also, we want to write application logic for the workstation and recompile it on the host to perform those same functions. In designing the application, however, an equally important goal is to produce the best possible user interface for workstation users. Although the user interface code would be difficult to reuse, we can reuse those parts of the application that perform the data-handling functions, provided they are designed to be independent of the user interface functions.

We expect to code the services in the host, assuming that they will execute in all SAA environments. Where practical, we have designed each service so that one part has all the functions that are common across SAA environments (the kernel) and the other part has the environment-unique functions (the shell). Nearly all service logic is in the kernels, including data management. Resource management and system-unique services are in the shells. On average, we suspect that the kernels will contain 50 percent or more of the service's code. This percentage should approach 80 to 90 percent as SAA matures.[2]

Cooperative processing. Our cooperative processing strategy is to place function in the processor that handles it best. These "what's-best" decisions are the results of many "what's-available" questions, and the solutions chosen must be reviewed continuously. One of the final tests must be an affirmative answer to the question: Does the user consider the performance to be acceptable?

As we evolved our design we were always conscious of this evolution, and we strove to segment the applications and services so that all major parts of the services could be run on workstations, on a host from a fixed-function terminal, or on a host from a workstation. At present, the long-term implications of this generality are not well understood.

Those functions that are shared by users are being placed in the host, but as recovery, security, access control, and availability concerns are addressed in the workstation, designers can consider the placing of shared function in OS/2 servers.[2]

System connectivity. Previously we said that the logic used to process user requests (the application logic) should be independent of the logic used to present panels to the user (the dialog logic). The application logic also should be independent of the logical and physical links used to connect processors (the connectivity logic).

The connectivity logic in our design is managed by the service requester, as shown in Figure 3. If the service is remote (i.e., in another processor), the requester uses OS/2 communication logic to handle the peer-to-peer communications. Thus, as common communications support becomes available in each SAA environment, we have only to convert the one requestor—as opposed to each application—from today's Advanced Program-to-Program Communication (APPC) interface to the communications interface of the CPI.

All office workers need access to information that is shared among them.

The physical links can be any link technology supported by the product providing the APPC interface for an environment, but the parallel sessions function is required for satisfactory performance. We expect to use SAA Common Communications Support, as it becomes available, because it offers a more comprehensive solution than our requester logic.

Local–remote transparency. Cooperative processing applications that support the distribution of data are frequently complex applications in heterogeneous networks. Much of this complexity can be hidden from the application programmer by services that provide local–remote transparency so that the application programmer is not necessarily aware of whether the data being used are local or remote, whether the service being called is local or remote, or what connectivity path is being used.

Our requester makes the location of the service transparent to the requesting application. However,

there may be cases, such as performance, where the application chooses to be aware of whether the service is local or remote. The application logic converses with services using a begin-conversation, middle-of-conversation, end-of-conversation technique. The begin function returns a signal to the application when the conversation is remote.

Our interfaces today provide a specialized solution in that they support the types of data objects and services we require for our set of applications, not the general set needed across all SAA environments. But the data streams are architected, thus allowing us to interchange data in heterogeneous networks with end-to-end fidelity.

Access to public services. All office workers need access to information that is shared among them. The programmer can package this information in a data store organized into objects, information about objects (attributes of the objects), and collections of objects (lists or groups of objects). However, the user sees this information through a construct, such as a directory, that contains the names of people and their telephone numbers, or a library that contains folders and documents.

Our model contained examples from three of these public services—a directory, a library, and mail distribution. (See Figure 1.) Another example of a datastore service is a calendar that can be used to manage personal events, shared events (such as conference rooms), and time-initiated events (such as overnight printer runs for low-priority documents).

Office workers also need access to tools shared among them, which the system designer can package as public services. One example is a service that allows users to share a printer. The application programmer must know how to begin and end a conversation with a shared service, but need not be aware of the location of the service when writing the application logic. This approach obligates the system designer to provide an administrative function that defines and maintains the electronic address of the services. When a service cannot handle a request, we allow it to send the request to a second service. Although, to be precise, that is a new conversation, from the user's point of view it is a repackaging of the original request.

Public services are excellent vehicles in which current products may coexist with SAA products. Consider the following two choices: We can leave the user interface unchanged and allow access to the new services, or we can update the product throughout, providing both access to the new services and a new interface that conforms to CUA. The interfaces of the public services appear to be good candidates for study as interface extensions to the SAA CPI. One aspect of such a study would be to determine whether such a public service had wide usage potential such that it could be considered as a building block by the development community.

Public services are excellent vehicles in which current products may coexist with SAA products.

Concluding remarks

During our design, we found functions that were needed by a single application or a group of closely related applications, and we chose to define a private interface for each of these functions. Although they may be used frequently by our applications, they are unlikely candidates for SAA CPIs because they are specialized functions. We foresee application designers encountering many situations where creating common services makes sense. When designed properly, common services are portable and can be candidates for moving to other environments. Those common services that have broad applicability may become candidates for addition to the SAA CPI.

The SAA design criteria we have discussed in this paper are neither new nor revolutionary. However, these criteria do focus on ease of use, return on investment, and flexibility. The ease of use emphasizes the personal comfort and general well-being of the user. We believe this allows for higher productivity, and it should increase the user's acceptance of the system. As systems become more extensible, applications become more extendable, and logic becomes easier to move between SAA environments.

We believe another step is being taken to make software design and development more of a science than an art.

Finally, added flexibility releases us from past design constraints. Freedom from device differences, database differences, and connectivity differences leads our list of contributors to flexibility. We believe that soon we will be able to distribute across SAA environments the design and production of application code. This capability seems as desirable for programmers who sit in adjacent offices as it is for departments that are located in different countries.

As SAA matures applications and services that are developed can be expected to execute without significant change in the future.

Acknowledgments

The authors wish to thank John Bennett and Clarence McGee for their insights in usability engineering and systems design. Skip McGaughey and Dick Washington provided much of the model that was used to make the usability evaluations.

Cited references

1. V. Ahuja, "Common Communications Support in Systems Application Architecture," *IBM Systems Journal* **27**, No. 3, 264–280 (1988, this issue).
2. *IBM SAA: Writing Applications, A Design Guide*, IBM Corporation, SC26-4362; available through IBM branch offices.
3. E. F. Wheeler and A. G. Ganek, "Introduction to Systems Application Architecture," *IBM Systems Journal* **27**, No. 3, 250–263 (1988, this issue).
4. S. Uhlir, "Enabling the user interface," *IBM Systems Journal* **27**, No. 3, 306–314 (1988, this issue).
5. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15**, No. 3, 182–211 (1976). See also R. W. DePree, "The long and short of schedules," *Datamation* **30**, 263–280 (December 1983).
6. J. L. Bennett, K. A. Butler, and J. Whiteside, *Usability Engineering*, Tutorial No. 23, presented at CHI'88, Conference on Human Factors in Computing Systems, Association for Computing Machinery, Special Interest Group on Computer and Human Interaction, Washington, DC, May 16, 1988.
7. C. Clanton, "The future of metaphor in man–computer systems," *Byte* **8**, No. 12, 260–263 (December 1983). See also B. F. Webster, "The Macintosh," *Byte* **9**, No. 8, 238–251 (August 1984).
8. A. L. Scherr, "SAA distributed processing," *IBM Systems Journal* **27**, No. 3, 370–383 (1988, this issue).

General reference

Systems Application Architecture Common User Access Panel Design and User Interaction, SC26-4351-0, IBM Corporation (December 1987); available through IBM branch offices.

William P. Dunfee *IBM System Products Division, 44 South Broadway, White Plains, New York 10601.*
 Mr. Dunfee joined IBM in 1963 as a junior programmer. He advanced through technical and managerial positions in operating system design, development, and testing, including MVT/360, SVS/370, MVS/370, and DPPX/8100, and was promoted to senior programmer in 1973. In 1980 he joined the communications systems business staff organization, where he was responsible for providing technical support for the executive management in the System Communications Division (SCD). Mr. Dunfee assumed management responsibility for JES2, JES3, and MVS/370 design and test in 1981; in 1982, these responsibilities were enlarged to include design and development as well as testing of these programs. In 1983, Mr. Dunfee became the MVS design and performance manager, and in 1985 he was promoted to manager of office systems development, in which position he was responsible for software (both operating systems and applications) and for the IBM office-systems strategy. In 1987, he was appointed manager of System Products Division system development software, where he is now responsible for application-enabling software. Mr. Dunfee graduated from the University of Massachusetts, Amherst, in 1963 with a B.S. degree in mathematics.

J. David McGehe *IBM Application Systems Division, Neighborhood Road, Kingston, New York 12401.*
 Mr. McGehe joined IBM in 1967 at Kingston, New York. He is a systems planner who is currently responsible for planning product implementations of SAA. Before his current assignment, Mr. McGehe planned and released products in the office-systems family. Prior to that, he had worked on several enhancements to Systems Network Architecture (SNA) and managed departments responsible for publications on the IBM 3600 Finance System and the IBM 3790 Communications System. Mr. McGehe received his B.S. degree in English from Iowa State University, Ames, in 1961.

Robert C. Rauf *IBM Application Systems Division, Regency Park, Cary, North Carolina 17511.*
 Since joining IBM in 1964, Mr. Rauf has had several technical programming design and development assignments. He has also managed various groups involved in the design and performance of large operating systems. For the past several years, Mr. Rauf has managed areas of responsibility for the design of office-systems products. Most recently, this work has been related to the integration and usability of office systems.

Kenneth O. Shipp *IBM Application Systems Division, 220 Las Colinas Boulevard, Irving, Texas 75039.* Mr. Shipp joined IBM in 1974 as a programmer in Austin, Texas. In 1980 he was appointed a first-line development manager on the IBM DisplayWriter product, and in 1983 he became application development manager for the DisplayWrite product. He is currently systems design manager for the Application Systems Division in Irving, responsible for the design of future office systems across all SAA environments.

Reprint Order No. G321-5329.