The designer's model of the CUA Workplace

by R. E. Berry

This paper discusses the details, insights, and rationale of the Operating System/2® (OS/2®) Version 2 Workplace Model, an implementation of the user interface defined by the IBM 1991 Common User Access™ (CUA™) guidelines. The Workplace Model is described as an object-oriented user interface where objects represented by icons are manipulated by selection and movement, copying and creation of other objects, and by defining their behavior to accomplish the user's desired task.

Models are used in many different fields. Researchers in physics, chemistry, and molecular biology use models to explore relationships between atomic and molecular components of systems. Economists and city planners use models to analyze and predict the performance of complex economic and social systems. And teachers use models as an aid in explaining complex systems in a variety of fields. Whether the model is a plastic replica of an airplane, an exploded-parts diagram in a book, or an elaborate computer simulation model, the purpose of the model is to convey an understanding of the components that make up an object or a system and their interrelationships.

The user interface of a computer system can be described and analyzed by using models. The relationships among the interface, the programming system that implements it, and the users of the interface can be described and analyzed by using models. Furthermore, a model of the interface can be implemented as a prototype to support iterative testing with users.

This paper identifies three models that are relevant in the design of a user interface (UI). Each model gives a different perspective of the interface, including the end-user's perspective, the UI designer's perspective, and the implementing programmer's perspective.

Following the description of the three models, the designer's model of IBM's Common User Access* Workplace (CUA* Workplace Model) is described to provide an example of the types of concepts that UI designers must address. The Workplace interface is an object-oriented user interface recently introduced in IBM's Operating System/2* (OS/2*) Version 2.0.

This description is intended to provide an in-depth understanding of the concepts underlying the CUA Workplace Model. It provides detail, insight, and elements of rationale from a designer's perspective to supplement the information included in other publications.[1,2] A few of the figures from these publications are used here to provide the reader with points of reference to help position the supplemental information. Many of the concepts described in this paper are depicted in a demonstration program and videotape called *The CUA Vision.*[3,4]

Models for user interface design

We use *models* in user interface design to describe an interface in terms of objects, properties,

behaviors, and relationships between objects. A model provides a framework for analysis, understanding, and decision making.

A model does not need to address every aspect and feature of a system. A level of detail adequate to understand relationships of interest, explain observations, and make design tradeoffs is sufficient. In some cases it may be desirable to use several different models with various levels of detail for the same system. One model might be adequate for a salesperson to explain the system to a prospective customer. Another model of the system might be needed to help develop specifications for subcontracted components.

A model must be accurate at whatever level of detail is chosen. Models must be under constant scrutiny and should be changed to reflect varying requirements and explain observed behaviors.

There are three basic models on which the CUA user interface design is based:

- The user's conceptual model
- The user interface designer's model
- The programmer's model

While the particular details of these models as presented in this paper are specific to the CUA interface, the use of these models and the relationships among them apply to the design of user interfaces in general.

A diagram depicting how these models relate to each other is shown in Figure 1.

To show the relationships among these three models, we have drawn an analogy between the task of a designer who is designing a user interface and an architect who is designing a house. These tasks are similar in many respects because both of them require an understanding of all three models.

A user interface designer's job is to create a designer's model, or blueprint, of the user interface, just as an architect creates a blueprint of a house. To do this, the designer must:

- Understand the user's conceptual model. That is, just as an architect must understand a client's needs and expectations to design a house that pleases the client, the user interface designer needs to understand users, their tasks, and their expectations.
- Use the user interface design principles on which CUA is based. Architects use basic principles that apply to housing design. A good architect knows the environment in which the house will be built with regard to temperature, weather, humidity, and other factors, and successful designs that have been used in that environment. Accordingly, the user interface designer needs to have a knowledge of accepted and proven principles in the field of user interface design.
- Understand the capabilities and limitations of the programming environment, and the skills of the programmers who will be implementing the interface. Just as an architect must know the strengths and weaknesses of building materials and the skills of the tradespeople who will build the house, user interface designers must understand the capabilities and restrictions of operating systems, file systems, window managers, programming toolkits, and other components used to implement the interface.

The user's conceptual model. The *user's conceptual model* of a system is a mental image that each user subconsciously forms as he or she interacts with the system.

The user's conceptual model is based on each user's expectations and understanding of what a system provides in terms of functions and objects, how the system responds when the user interacts with it, and the goals the user wants to accomplish during that interaction. These expectations, understandings, and goals are influenced by the user's experiences, including interaction with other systems, such as typewriters, calculators, and video games.

Because each user's conceptual model is influenced by different experiences, no two users' conceptual models are exactly alike. Each user looks at a user interface from a slightly different perspective.

The problem for the interface designer is to design an interface that users find predictable and intuitive when each user is approaching the interface from a different perspective. To come as close as possible to matching users' conceptual models, designers should find out as much as they can about users' skills, motivations, the tasks they

perform, and their expectations. This process involves: using resources, such as task analyses, surveys, customer visits, and user requirements lists; incorporating information that users provide into the user interface design; and conducting usability tests.

This is an iterative process that may require many cycles. As the design progresses, users may identify aspects of the interface that are difficult to learn, that are counter-productive, or aspects they simply do not like.

Through interaction with the user interface, users' conceptual models may be expanded, which in turn may cause them to realize new requirements that they had not thought of before. As users provide this level of information, the picture of their conceptual models will become clearer.

Conceptual models of an object-oriented user interface consist of the objects, and the properties, behaviors, and relationships of those objects, that are involved in users' interactions with the interface.

When users first interact with a new interface, they are likely to attempt to understand its operation in terms of concepts already existing in their current conceptual models. Where their existing models lead to correct expectations, their models will be reinforced and the users will feel the interface is intuitive. When results are not as expected, users may rationalize by adding new relationships to their models to explain observed behavior. If the new extensions are accurate, they should be reinforced through interaction with similar aspects in different parts of the system. Sometimes users develop superstitions about the interface. These superstitions are incorrect rationalizations about the interface. They are likely to cause unexpected results and further contradictions of a user's intuition. The use of metaphors and consistency are two approaches that designers can use to build on users' existing conceptual models and create intuitive interfaces.

A new interface should resemble something familiar to help users get started, then allow them to explore new concepts. It is often said that a characteristic of a good user interface is that it is *intuitive*. Perhaps when used in this sense intuition can best be characterized as a good match between the user's conceptual model and the designer's model.

By using metaphors, designers can take advantage of a user's experience and allow a user to rely on intuition while expanding the user's conceptual model to take advantage of new capabilities provided by the interface. Interfaces that use metaphors and allow users to safely explore the computerized environment are popular for this reason.

For example, a computerized car dealership application might provide a *worksheet object* to be used by a salesperson in the task of selling a car. The computerized worksheet contains the same information and is used in the same way as a paper worksheet. Like the paper worksheet, the worksheet object allows the salesperson to enter the car's price and stock number, the customer's name and address, and information about the proposed terms of the sale.

However, the worksheet object also expands the salesperson's conceptual model by providing capabilities that go beyond those of a paper worksheet. Instead of typing information into the worksheet object one field at a time, the salesperson might simply "drag and drop" a car object onto the worksheet. The fields in the worksheet that are relevant for the car being sold are automatically filled in by the associated fields from the car object. Monthly payments and finance charges are calculated automatically. Instead of having to hand a paper worksheet to the sales manager for approval, the salesperson can drag and drop the worksheet into a specific mail out-basket to have it automatically sent to the sales manager through the dealership's computer network.

This worksheet object not only meets the salesperson's expectations, it goes beyond them. It is an object that the salesperson expects to use during the task of selling a car, it has behaviors and characteristics that the salesperson is accustomed to, and it provides additional value by use of the computer.

The worksheet object acts as a metaphor for an object that already exists in the salesperson's conceptual model of a car dealership and the task of selling cars. It is an object with which the salesperson is already comfortable, and it provides

additional capabilities that make the salesperson's job easier than it is using a paper worksheet.

Users' conceptual models constantly evolve as they interact with an interface. Just as users influence the design of a product, the interface design influences and modifies users' concepts of the system. Designers can help users develop an accurate conceptual model by using well-defined distinctions between objects and by being consistent across all aspects of the interface.

For example, given an object-oriented car dealership application, the salesperson opens and works with familiar objects instead of starting and running computer programs, opening files, and so forth. This object-oriented approach has fewer concepts for the salesperson to deal with and matches the salesperson's real world better than one in which a task is accomplished by starting applications and opening files. However, it may require a shift in the conceptual model of a salesperson who is already accustomed to a computer program-oriented type of interface.

Naturally, the conceptual model of a salesperson who is already familiar with using a graphical computer interface requires little modification. This salesperson would already know how to use icons, windows, menu bars, and push buttons.

In any case, the distinctions between objects must be clear and useful, and the interface must be consistent. Otherwise, the users' conceptual models will be modified in ways other than those intended by the interface designer.

The interface components and relationships intended to be seen by users and intended to become part of each user's conceptual model are described in the designer's model. This model represents the designer's intent in terms of components users will see and how they will use the components to accomplish their tasks.

The designer's model. The second useful model in user interface design is the *designer's model*. In the designer's model the user interface designer defines objects, how those objects are represented to users, and how users interact with those objects. User objects are defined in terms of properties, behaviors, and relationships with other objects. Differences in properties and behaviors are the basis for class distinctions, such as the distinctions between folders and documents. Relationships between objects affect how they are used in accomplishing users' tasks. For example, users can use folders to contain and organize

The designer's goal is that the user's conceptual model exactly match the designer's model.

memos, reports, charts, tables, and many other classes of objects. Users can discard an object by "dropping" the object's icon on a wastebasket icon, and users can print an object by dropping the object's icon on a printer icon. These actions are logical in that they maintain real-world relationships between objects.

By relying on a few basic classes and relationships, with well-defined distinctions based on user task needs, the designer's model should be easy for users to learn and understand. That is, users should quickly develop conceptual models that closely match the designer's model.

Reference 5 introduces the CUA designer's model, and this paper describes the model in detail. This model defines objects that are common to many types of applications. Designers must add objects that are needed by specific products. This is typically done by extending existing objects (creating subclasses) or by defining entirely new types of objects (creating new classes). Definition of the designer's model is crucial to developing products that are easy to learn and understand. Its definition should comprise the first series of steps during product design.

If the designer's model closely matches a user's conceptual model, the user should learn quickly and apply knowledge correctly in new situations. In other words, the user will feel the interface is intuitive. Designers can help users to develop a closely matching conceptual model by creating a clear and concise designer's model. A designer's model is clear and concise when it has made a

minimum number of distinctions among objects, the distinctions are clear and useful to users, and they are consistently conveyed throughout the interface.

For the designer's model to be consistent with the user's conceptual model, the designer must know the users, their tasks, and their expectations. If designers do not understand their users, the interface will not behave as users will expect it to. Also, if the system does not behave as users expect it to, their conceptual model will be different from the designer's model and misunderstandings will occur. Users can lose confidence in the reliability of their conceptual model, and thus in the system itself, when these misunderstandings occur. If users form an incorrect conclusion or a superstition to explain an inconsistency, they may try to apply it elsewhere in the system. This can lead to further misunderstandings and distrust of the system.

A misunderstanding may be caused by inconsistency in objects' behaviors resulting from a particular action. For example, if a user learns that *double-clicking* the mouse button on an object opens a window on the object, and elsewhere in the interface the same action discards an object instead, the user may begin to distrust the system.

In summary, the designer's model is the model of objects, properties, behaviors, and relationships that the designer intends the user to understand. The designer's goal is that users' conceptual models exactly match the designer's model. Users who perceive the interface at this level have a precise understanding of the interface and can take full advantage of the capabilities provided by the designer.

The programmer's model. The third model of interest in user interface design is the *programmer's model*. The programmer's model is the system's implementation of the designer's model. The programmer's model includes details relevant only to the programmer. For example, the designer's model might include a directory object that consists of people's names, addresses, office numbers, and so forth. However, the programmer's model of the directory object might consist of records in a file, with one record for each directory entry; or, it could be a complex organization of multiple records from multiple files. These implementation details from the programmer's model should not be evident in the designer's model and are therefore transparent to users.

Figure 2 summarizes the three models and identifies factors that influence each model.

Getting users started—a kernel of knowledge

In addition to understanding which objects to use, users must understand the ways in which these objects can be used. Users unfamiliar with computers and graphical user interfaces may need some preliminary information on how to use the system. New users should be provided with enough information on how to use the system to get them started. This information establishes a base conceptual model and gets users started in the right direction. Both conceptual and procedural information should be included. Given this base, or kernel, of knowledge they can explore the interface to learn more about its capabilities and develop a more complete and accurate conceptual model.

The information user's need in order to get started using a computer includes concepts about objects and techniques for interacting with objects. Some of the concepts and techniques users must understand to use the CUA Workplace Model include:

- How to use the mouse
- Opening views of objects
- Manipulating windows and the views within them
- Accomplishing actions by dragging objects
- Copying and creating objects
- How various menus relate to objects
- How objects are composed of and contain other objects

The information in the kernel applies to all applications developed according to the CUA guidelines. This information can be presented in a tutorial for new users of a system. Rather than a complete tutorial about CUA concepts and mechanisms, it is just enough information to get users started. This information should be made available to users so that they can begin to explore the system. Designers may also use this information when developing product-specific tutorials for users, using product-specific examples.

Beyond the information found in the kernel, users can learn more about the system by trying the same or similar techniques on many different objects.

A common task is to look at information stored in the system. From information supplied in the kernel of knowledge, a salesperson understands the concept that the icons represent objects that can be opened to view their contents in a window. The kernel also provides procedural information about how the objects are opened.

From the kernel information, users can also learn about other mechanisms for handling objects such as selecting objects, requesting actions on objects, changing the view of an object in a window, and dragging objects. Users should be able to apply these mechanisms across many different objects and observe consistent results.

The designer's model of the CUA Workplace

The designer's model identifies objects, object relationships, how the objects are represented on the screen, and how users interact with the objects.

Figure 3 shows the various parts of the designer's model of the CUA Workplace. The figure also shows the relationships between these parts. Each of these parts is fundamental to the CUA Workplace user interface and is therefore common to all products.

The figure does not address product-specific objects, representations, or input mechanisms. These extensions to the model are left to the discretion of the product designer.

The figure is divided into three sections: one for user objects, one for visual representations, and one for interaction mechanisms supporting user actions. Two lines across an arrow means that the box the lines are closest to has a "more-than-one" relationship with the box at the other end of that arrow. For example, a container can contain more than one object and an object can be represented by more than one view.

Object classes in the Workplace Model

The top section of Figure 3 shows the classes of user objects that are fundamental to the CUA Workplace Model: data objects, container objects, and device objects. Objects that users work with on a computer should be designed in such a way that users can easily become accustomed to using them. The use of metaphors has become a popular method of helping users to relate these objects to objects that they work with in the real world.

The designer's model with objects, relationships, and interactions is the CUA Workplace.

When using metaphors, it is very important to preserve characteristic behaviors that distinguish objects in the real world. For example, a user object that emulates a folder on a computer should behave much like a real folder. Users should be able to open it to inspect its contents, add new items, and rearrange its contents.

Object behaviors. The ways in which objects can be used to accomplish users' tasks are called object *behaviors*. Three distinctive behaviors of objects can be identified through observation of real-world objects and typical user tasks. Individual objects may support behavior from one or more of these fundamental behavior classes:

- Data behavior

  Objects provide *data behavior* to communicate information. Data behavior includes presentation of views that show the composition of objects, and views that allow users to manipulate the information and the arrangement of objects that form the composition. For example, a composite memo might contain text, graphs, charts, and images. Views would typically be provided to allow a user to edit each of these objects as well as to allow rearrangement of their layout, or relationships with each other.

Data objects come in many forms, such as memos, business graphics, tables, music, recorded speech, animations, video clips, and various combinations.

- Container behavior

  Objects provide *container behavior* to store and hold other objects. Container behavior includes presenting a list of contained objects, allowing new objects to be added, and arranging the contents in various ways, such as sorting by object name or date created. Folders, trash cans, and in/out baskets are typical examples of objects providing container behavior.

- Device behavior

  Objects that have *device behavior* provide an interface to the world outside of the user's domain within their computer. Device behavior includes the ability to print and transfer to external media, such as a diskette.

Individual objects typically provide behaviors from one or more of these classes. For example, a *queued* printer provides both container and device behaviors. Its input queue behaves like a folder, but its primary purpose is to transfer information to an external medium—paper. Each object is typically intended to serve some primary purpose in terms of the data, container, and device distinctions. Objects in the CUA interface are classified with respect to their primary role in the interface as data, container, or device.

It is important for designers to understand the concept of object behavior because the behaviors of similar types of objects should be consistent and the behaviors of different types of objects should provide useful distinctions to users. An object's behavior determines such aspects as which views are provided, which user actions are supported, and what should happen in data transfer operations, such as when another object's icon is dragged to and dropped on that object's icon. The following contrast between a container object and a device object shows how the behaviors of these objects affect views and the results of direct manipulation.

A container object, such as a folder, is used primarily as a place to store other objects. Therefore, if an object is to be used primarily as a container, one or more views listing the container's contents must be provided. In addition, objects that are dropped on a container's icon are moved into that container.

In contrast, a device object is used primarily to provide communication between the user's computer domain and the world outside that domain. A printer, for example, is a device object that is connected to a real-world physical printer. It should provide at least a view of the current printer settings and a view of the printer's queue contents.

The settings view allows a user to ensure that the desired printer is correctly configured. The contents view allows users to inspect and change what is in the print queue. Even though a printer object is not used primarily as a container, its queue has some of the characteristics of a container, which means a contents view is also needed. This contents view could enable a user to see, for example, which object is currently being printed, how many objects are waiting to be printed, and what the status of those objects is (ready, on hold, and so forth). It might also allow the user to rearrange the order of the contents, for example, to move a particular object to the front of the queue.

To complete the comparison of containers and devices, we also need to look at the results of dragging another object to a printer. Objects that are dragged to a folder are *moved* from a current location to the folder. Objects that are dragged to a printer are *copied.* Copy is the default for drag and drop to devices because it is safe for the user. Objects will not be inadvertently lost by transferring them to printers, diskettes, or other destinations external to the user's workplace domain.

Whether an object is moved or copied during drag and drop operation (drag/drop) is an architectural distinction for consideration by designers as they define new actions. The intent is that users simply think of drag/drop to a printer as causing the print action, with the obvious consequence of preserving the object being printed.

Most objects provide more than one class of behavior. Therefore, in CUA, objects are classified

according to their primary behavior, even though they may support additional behaviors.

An analogy may be helpful in clarifying this concept. You can ride in both a car and a truck, but each has specific distinctions because each is designed for a specific purpose. Cars are primarily used to transport people, while trucks are primarily used to haul objects that are too heavy for, or do not fit inside, cars. Of course, trucks can carry people as well. Similarly, the following comparisons can be drawn for the data, container, and device behaviors:

- Objects that behave primarily as data objects may also provide some container behavior. Composite documents that contain embedded annotation text, business graphics, charts, and other objects are primarily data objects and provide data behavior. However, they may also provide a list of their contents.
- Objects that primarily behave as containers can also behave as data objects. Folders primarily provide container behavior. They are used to store, group, and arrange related contents. Like a data object, however, they can also be copied, archived, and mailed.
- Objects that primarily provide device behavior may also behave as containers. Printers, for example, are used for their device behavior, but may display a list of the jobs waiting to be printed (printer queue). The printer queue is displayed in a contents view, and may provide behavior that is normally associated with a container, such as filtering and sorting.

Establishing containment relationships. The primary behavior of container objects is to store and organize other objects. The primary behavior of data objects is to convey information. However, data objects can provide some container behavior and vice versa.

To help identify which should be an object's primary behavior, it is useful to consider whether the relationships between the contained objects are tight or loose. *Tight containment* relationships favor data behavior, while *loose containment* relationships favor container behavior.

Tight containment relationships. Tight containment identifies and preserves relationships between individual objects, such as text, tables, graphs, and charts, that are combined to form a single composed object, such as a composite document. The arrangement of objects in a composite document is typically very specific and contributes to the overall communication of information. Text is wrapped around related figures, which are kept on the same page when possible, and so forth, as shown in Figure 4.

The relationships between the objects that are combined to form a composite document are tight because they give added meaning to the document itself. If the objects in the composite document were arranged differently, the communication of the document's information could be affected.

The primary behavior of a data object is optimized around its composed views and the behaviors of the entire object as a composed whole. One aspect of this behavior is that the relationships between the collection of objects is maintained as the object is changed. For example, when a composite document is reformatted, the relationships between figures, captions, descriptive prose, and footnotes are preserved. These are examples of tight containment relationships.

Loose containment relationships. Loose containment does not rely on or preserve relationships between contained objects. The arrangement of objects in a container, such as a folder, is loose. The objects in a container will probably be related, such as those shown in the folder in Figure 5.

However, changing the arrangement of these objects within the folder does not change the overall meaning or purpose of the folder itself.

The user will typically arrange objects in the folder for convenient access, not to convey any particular meaning. This is the opposite of the relationship between objects in a composite document, in which changing the arrangement of the objects could alter the document's meaning.

As a result of the loose relationships between objects in a container, the views of containers typically provide a user with options for changing the layout of objects, sorting the objects, and so forth.

Implications of tight and loose containment. Tight containment relationships tend to contrib-

Figure 6 What users see on the screen

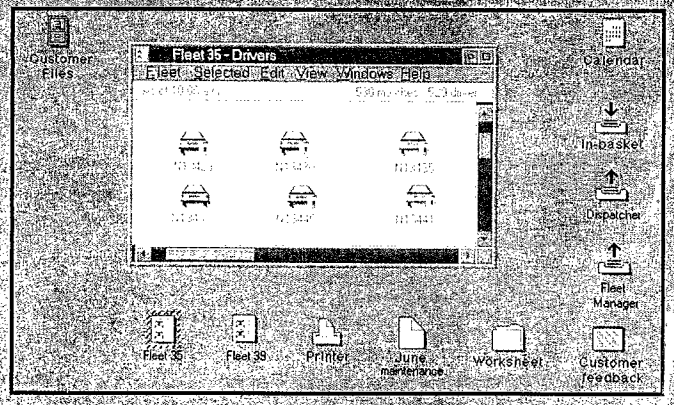


Figure 7 A contents view with an iconic layout

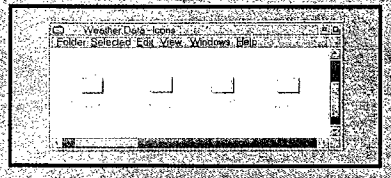


Figure 8 A contents view with a details layout

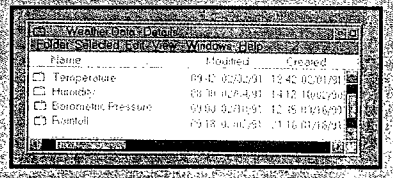


Figure 9 A settings view

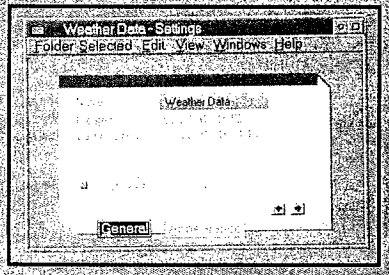


Figure 10 A help view

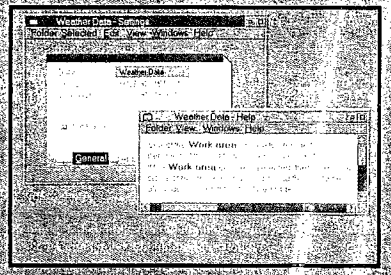


Figure 11 Composed and settings views

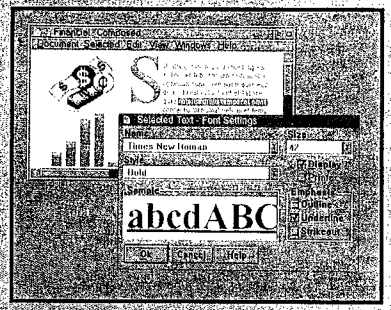


Figure 12 Drag and drop

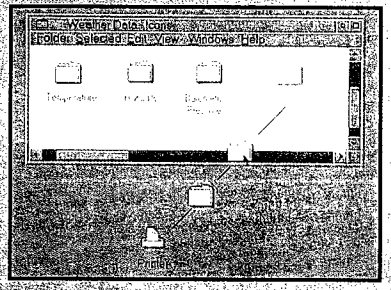


Figure 13 Spectrum of direct manipulation

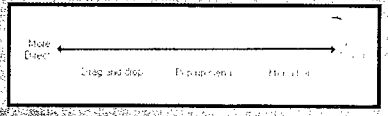


Figure 14 Selection cursor example

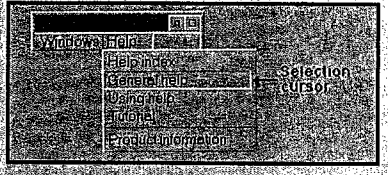


Figure 15 Text cursor example

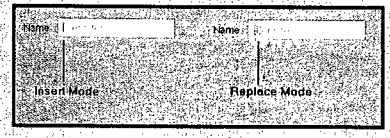


Figure 16 Cursor-driven scrolling

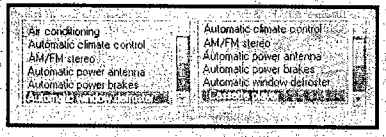


Figure 17 Types of menus

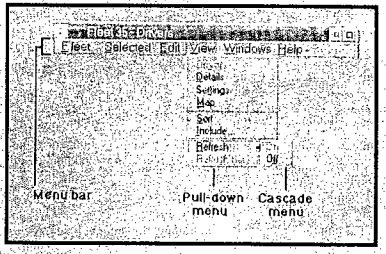


Figure 18 Three menu-bar styles

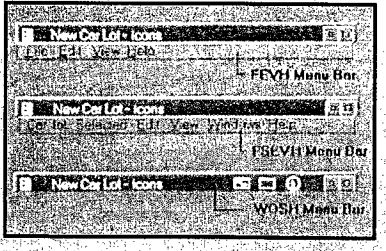


Figure 1 User interface design models

Figure 2 User interface design influences

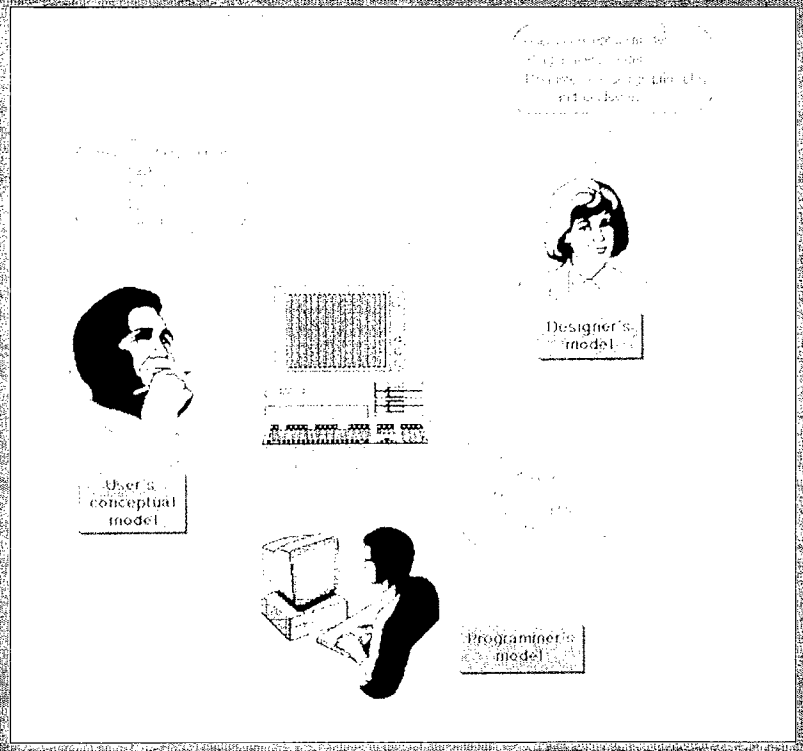


Figure 3 Designer's model of the CUA Workplace

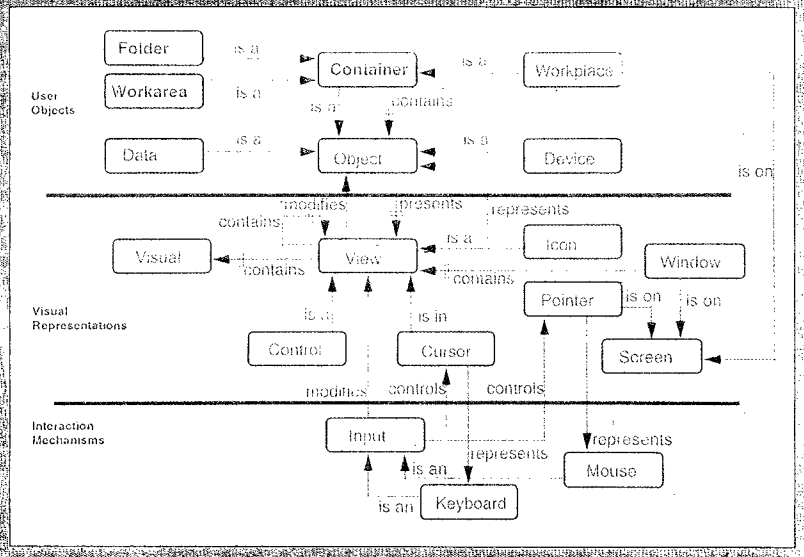


Figure 4 Objects in a composite document: tight containment relationships

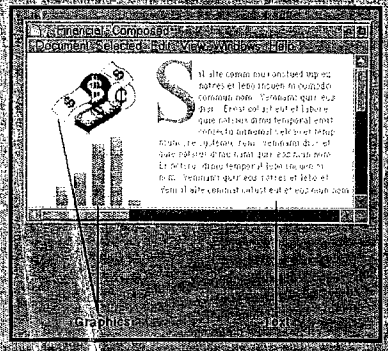
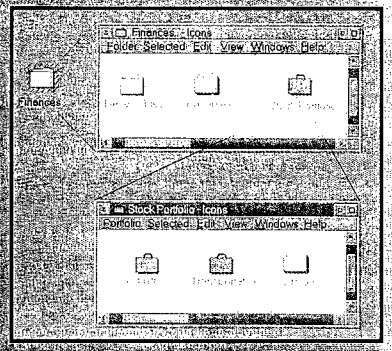


Figure 5 Objects in a folder: loose containment relationships



ute to the definition of object type because they describe an object's composition, an aspect on which type classifications are typically based. For example, consider the classification *automobile.*

CUA defines three general-purpose containers: the folder, the workplace, and the work areas.

An automobile can be classified as such based on its composition, which usually consists of an engine, passenger compartment, four wheels, etc. Its composition is *independent* of its contents, or passengers. An automobile is still an automobile regardless of whether it contains one, two, four, or no passengers. Taking away components of its composition may affect whether we still consider it to be an automobile, or at least whether it is a *complete* automobile.

Similarly, it is useful to consider aspects of composition when designing computer interface objects, such as folders, mail baskets, printers, and other objects that provide container behavior. It may prove useful to provide additional views that allow users to see and possibly manipulate components that compose the object distinct from and in addition to views and manipulation of its contents.

Data objects. *Data objects* are described in terms of materials and structure. Text, graphics, image, audio, and video are the materials from which data objects are composed. These materials are used in simple structures, such as strings, arrays, and records, which can be used to form more complex structures, such as reports, spreadsheets, and charts. This is analogous to the building materials wood, steel, concrete, and plastic, from which chairs, tables, houses, and schools are built.

The CUA interface does not define specific data objects. Data objects are typically product-specific and are therefore defined by product designers. However, general data object behavior is described in the CUA to achieve consistency within the Workplace Model. Examples of data objects are graphs, charts, spreadsheets, and composite documents.

Container objects. The primary purpose of *container objects* is to provide a place to store and group related objects. For example, CUA-defined folders can contain charts (data), printers (devices), and even other folders, all at the same time.

Container uses can range from general-purpose to product-specific. CUA defines three general-purpose containers: the folder, the workplace, and the work area. Designers can refine the CUA-defined containers to create containers for specific product needs. By using refinements of CUA containers, common container behavior is preserved. This allows users to take advantage of what they already know about containers while minimizing the effort to learn the new container's differences.

Folders and work areas. A *folder* is a container that provides a place to *store* a group of objects. A *work area* is a container that provides a place to *use* a group of objects to perform a specific task. Although these two containers have much in common, they provide optimizations for different roles in the interface. The difference between folders and work areas is based on how each is used in accomplishing users' tasks and in some special behaviors that each provides.

The folder can be thought of as a central storage place for objects that have a common theme, such as a group of worksheets, memos, voice messages, and video sales brochures related to a particular customer account.

The work area is a place where objects are brought together to perform a particular task. A work area might be used to contain mail trays, a sales catalog, a printer, and a folder that contains worksheets, used collectively to perform an account billing task. Work areas provide window management assistance for windows of objects that reside in the work area. For example, when a work area window is closed the windows of all objects opened from that work area are also closed. Likewise, when the work area is reopened those windows are also reopened. By providing

separate places to perform specific tasks, work areas can help users manage multiple concurrent tasks, while still providing a useful separation between them.

However, when the user closes a folder, any windows that were opened for the objects in that folder remain open. This allows the user to remove the unnecessary clutter of the open folder from the screen while continuing to work with the object.

Suppose a salesperson in a car dealership is using a work area with several open windows to prepare a report to the sales manager. When interrupted by a potential customer the salesperson can simply: close the report work area, open a car sales work area, complete the sales task with the customer, then resume the report task.

All windows related to the report task are closed when the report work area is closed. Similarly, all the windows used in selling a car would appear automatically when the car sales work area is opened. These windows would typically allow the salesperson to enter information about the customer, search for a car that the customer is interested in, and ultimately complete the sale. When the sale is complete and the customer leaves, the salesperson closes the car sales work area and reopens the report work area to continue the report task from the point of interruption.

Objects in multiple work areas. A user will typically find it useful to have some objects in more than one place. To explain, we can draw on an analogy from a business environment to show that the same objects need to be available from many different places. If you work in an office, your desktop computer may be attached to a larger computer in another location. The larger computer contains product information that you need when talking to clients. However, you may also need to have access to this information when you go out of town to visit a client. You could take a laptop computer and a modem with you, thus giving you access to the information through a telephone line instead of having to take a copy of the information with you. Thus, although the product information (the object) is still in the same place, you have access to it from your client's office (a different work area).

Similarly, a folder might contain a set of charts, all of which are related to the same subject, such as monthly sales data for a particular business account. But a user may also need one of the charts while performing two different tasks. The Workplace Model allows more than one icon to represent an individual object. Therefore, an icon for the chart can be in the folder and in two different work areas at the same time. Each icon represents exactly the same object, and actions on the object can be performed from any one of the icons. Each icon is a reflection of the object. For most purposes the user need not care that an object has multiple icons. In OS/2 2.0 these additional icons are called *shadows*.

For example, one task might be to create a monthly report for selected accounts. The other task might involve doing a year-to-date sales analysis. By creating a work area for each task the user can group the objects required to accomplish each task, including the chart, while preserving the storage relationship established by the folder. By representing the chart with more than one icon, there are convenient access points for the chart from each place that the user might need it.

The CUA intent is that users typically need not be aware of which icon represents the "original" object. Except for deletion, users need not know or care that an object has multiple icons. For deletion, users are provided with choices to allow deletion of individual icons or the entire object with all of its icons.

Users, work areas, and products. Some users, particularly those who perform one task at a time, may not create work areas. Other users are likely to create their own work areas that are tailored to the tasks they perform often.

Some products may provide ready-to-use work areas for certain tasks, such as a programmer's work area that contains standard libraries, debugging tools, and so forth. A car dealership might provide salespeople with ready-to-use work areas for preparing standard reports, selling cars, and determining commissions.

The primary means for distinguishing folders from work areas is a visible difference that should be designed into their icons, and the icon labels and window titles that further help users identify objects.

For example, in a car dealership application, the salesperson's work area might be titled "Selling Cars" to reflect the task that the salesperson would perform when using that work area.

The workplace. The third type of container is the *workplace*, which contains all objects accessible by a user whether local or remote to the user's system. In hierarchical storage systems it represents the apex of the user's storage hierarchy. The workplace is represented by the computer screen. Users typically leave objects that are used for many tasks on the workplace, such as a wastebasket, a telephone, an address book, and a calendar.

Device objects. The primary purpose of *device objects* is to provide an interface between objects within the computer system and the world external to the computer system. For example, a user can mail an object to another user by dragging the object's icon to a mail tray, or print an object by dragging the object's icon to a printer.

Device objects tend to provide specific functions, but can often operate on many different types of objects. In the preceding examples, a user could drag almost any object to a mail tray or to a printer.

Device objects are typically used in conjunction with other objects for which they provide some function. These associations might be specified in advance, but are typically done during a user's task. For example, a user might associate a low-resolution printer with a document to establish a default printer, but can drag the document's icon to the icon of a high-resolution printer when higher quality printing is required. Besides printers, examples of other devices are a diskette drive, a keyboard, a mouse, and a plotter.

How objects are represented to users

In terms of representing objects to users, the most important questions are:

- What aspects of the object does the user need to see for each task to be performed?
- How will those aspects be represented?
- What types of actions must users be able to accomplish and which techniques can be provided to accomplish them?

CUA uses views and icons to represent objects. Views are displayed in windows. The middle section of Figure 3 shows how these elements of the designer's model relate to user objects and interaction mechanisms. Figure 6 shows how users see them on the screen.

Icons represent objects on the workplace, in work areas, and in folders. Common object behaviors, such as creating, copying, moving, printing, and deleting, are provided by the objects that the icons represent.

Users can access additional aspects of an object by opening a window. A window contains a view of the object. Many objects can provide more than one view, showing different aspects, or the same aspects in different formats.

Icons. An *icon* is a small graphic image that represents an object. Icons can appear on the workplace, and can appear in contents views of work areas and folders, as well. They usually have a label to identify the object they represent. Figure 5 shows a contents view with icons representing the objects in a folder.

An object's icon depicts its class. For example, the icon used to represent a folder conveys its class by resembling a typical office folder. An icon should also depict other important aspects of an object. For example, the icon that represents a mail in-basket should change in some way to show that new mail has arrived. An icon need not convey all of an object's properties; however, it should convey those that are useful without confusing the user. The key is to design each icon's image so that a user can immediately recognize the type of object the icon represents, and thus recognize the basic properties and state of the object.

Windows. A *window* is a space on the screen in which a view of an object is displayed, in which choices associated with an action are presented, or in which a message is displayed. Users can control the position of the information on the screen and how much is visible by moving and changing the size of the window.

A window for an object can be opened from the object's icon. For example, opening a window from the icon of a folder allows a user to see the objects that the folder contains.

A window for presenting additional choices associated with an action is displayed when a user selects an action choice, such as from a menu or a push button. For example, selecting a "Print . . ." choice from a menu would display a window in which a user could specify desired print options.

Views can be larger than the windows in which they are displayed. When this is the case, the view is either scaled to fit in the window or it is clipped. If the view is clipped, the user sees only a portion of the view. Scrolling techniques are provided so users can control the portion of a view displayed at any one time.

A window can also be split into *panes*, which are used to display different portions of the same view concurrently. Or, a user can display multiple views of an object concurrently by opening more than one window.

The CUA interface classifies windows in two ways based on the kind of information presented in a window and how a window relates to other windows on the screen. Windows are used to present three basic kinds of information:

- Views of objects, such as the contents of a folder, the formatted text of a memo, the settings for a printer, or a video sales brochure
- Options for an action request, such as the number of copies and range of pages for a print request
- Messages, such as a message that indicates the printer is out of paper

The CUA guidelines for the use of standard push buttons, window title text, and so forth are based on these distinctions. For example, the guidelines specify which push buttons to use with action choice windows. Different push buttons, appropriate to the types of messages CUA defines, are used with message windows.

CUA defines two types of windows based on how a window is related to other windows on the screen. A *primary* window is one in which the main interaction with the user takes place. Views of objects are typically presented in primary windows. Opening and closing of primary windows is not dependent on opening and closing of other windows.

A *secondary* window is dependent on a primary window. A secondary window is used for information that supports the use of a primary window. Each secondary window is associated with a particular primary window. A secondary window is closed if its primary window is closed.

CUA guidelines encourage designers to make all windows movable, sizable, and as modeless as possible.

For example, a primary window is typically used for the view of a document that is shown when a user "double-clicks" to open a window from a document's icon. However, a secondary window would typically be used to display action options associated with a Search or Print request for the document.

The primary-secondary window distinction describes a relationship between two windows. Each window may have different relationships with other windows. For example, a Print Options window would typically be a secondary window; however, it could also be a primary window for any *message* windows displayed that pertain to the print option.

Any window can be primary for another window, making that window secondary. A primary window can have more than one secondary window, but each secondary window has only one that is primary, which controls its Close and Open behaviors. The CUA guidelines currently only address Close and Open behaviors. However, the primary and secondary relationship should be thought of as a primitive window grouping mechanism that can provide users with explicit control of window grouping, as well as applicability of other actions to the groups, like moving a group of windows while maintaining a particular spatial arrangement.

The CUA guidelines encourage designers to make all windows *movable*, *sizable*, and as *modeless* as

possible to give users the feeling of being in control. Differences in these capabilities are typically not a sound basis for distinctions between window types. Primary and secondary relationships should be established on the basis of how the information in each window is used to support the user's tasks.

Because the CUA guidelines recommend that each window be sizable, they also recommend that each window provide support for the Maximize function. However, some simple analysis can be performed to decide whether a window should provide the Minimize, or Hide, function.

We characterize the Minimize and Hide actions as requests to "put aside temporarily," while Close is thought of as "put away." The advantage of putting a window aside is that the window is kept close by and handy. It can be easily redisplayed. A disadvantage is that having too many minimized or hidden windows can cause confusion that outweighs the advantages. Windows used to present views of objects are often opened from objects that reside somewhere in the user's storage hierarchy, such as from an icon in a folder. The icon's location may be many levels deep within the hierarchy and the user may have closed the windows used to access the icon, thus making it difficult or tedious to redisplay the object if its window is closed. Therefore, the ability to put a window aside is an important function for windows showing views of objects. However, a window that displays options for an action request, such as "Print . . . ", can be easily redisplayed from its menu choice. Therefore, a put aside action is not recommended for these windows.

Views. A *view* is intended to convey certain aspects of an object to its users. In the Workplace Model, views resemble as closely as possible real-world counterparts of the objects. This resemblance helps users recognize objects and understand how they are intended to be used. Each view provided by an object:

- Displays particular aspects of an object, such as the contents of a container
- Supports a set of actions related to those aspects, such as moving an object from one container to another

Users can interact with objects through object views displayed in windows. Designers must know what information users need and which user interactions need to be supported in order to design useful views. Often, there are so many aspects of an object that multiple views must be provided. Groups of related aspects are then shown in different views. For example, a composite document might provide a composed ("what you see is what you get") view, an outline view, a print preview, and a settings view.

Each view is provided in support of particular user tasks. For example, a settings view might allow a user to change the font type and size for a string of selected text. Also, each view can be presented in a separate window. In other words, opening a different view of an object need not cause the contents of the current window to be replaced by the new view.

Types of views. CUA identifies four common types of views:

- Composed view
- Contents view
- Settings view
- Help view

Composed views use visible representations that show important relationships within an object. The intent is to convey the meaning of the composed object as a whole.

Composed views are provided for *data objects*, in which relationships of the parts contribute to the overall meaning. For example, documents, graphs, and charts provide composed views because the relationships of the components in each determine the meaning of the object as a whole. This is the tight containment relationship described earlier. Figure 4 shows an example of a composed view.

Contents views list the objects within a container. The intent is to convey the contents of an object in a way that helps understanding which objects it contains. Rearranging the order or changing the layout in which the contained objects are displayed or grouped in a contents view has no effect on the overall meaning of the object that contains them. This is the loose containment relationship described earlier.

Contents views are provided for *container objects*, such as work areas and folders. They can also be provided for any object that has container behavior. For example, *data objects*, such as composite documents, and *device objects*, such as queued printers, can have contents views.

Contents views can have various layouts, depending on a user's needs. One commonly used layout is the iconic layout, shown in Figure 7.

Layouts other than iconic layouts are sometimes preferred because icons take up more space than text, thus allowing a user to see fewer objects than the user could see if icons were not used. Three commonly used contents layout views are:

- Iconic layout, which uses icons to represent objects
- Small-icon layout, which uses small icons accompanied by text descriptions to the right of the small icons
- Details layout, which uses small icons accompanied by text descriptions plus additional details about the objects

See Figure 8 for an example.

The distinction between composed and contents views cannot be rigid. The relationship is more like a spectrum with contents views at one end and composed views at the other. For example, an outline view of a document might list the sections, but it also shows them in order. Designers should consider users' needs in deciding which views to provide. Our intent is to raise an awareness of the potential for each object to provide different types of views to support various user tasks. In particular, providing views that treat objects as compositions of other objects can potentially provide very powerful, flexible, and extendable capabilities to users.

Settings views provide a way to change the properties associated with an object. Settings views are typically provided for all types of objects. This way, users can change settings such as fonts, font sizes, colors, and so forth in a document, or output quality and destination of printed output for a printer device.

By convention, CUA specifies that settings views be presented using a "tabbed" notebook metaphor, typically provided as a control in the programmer's toolkit. Use of this control provides quick access to all of the properties of an object. Related properties are grouped together in tabbed sections for easy access. Use of the notebook shortens pull-down menus because most of the settings choices no longer need to appear in the menus. Figure 9 shows an example of a notebook used to display a settings view.

Help views provide information to assist users in using an object. Help views should be provided for all objects. The type of information that a particular help view contains depends on the choice that a user selects from the Help menu. These choices are described in Reference 2. Figure 10 shows an example of a help view for the work area setting.

Using different views. Users can learn the capabilities of an object by exploring its different views.

Users learn object behavior by observing the results of actions performed on the object when menu choices are selected, when the object is dragged, and so forth.

When an object is opened, a user can perform actions that are not available directly from the object's icon. For example, by dragging an icon that represents a graph, a user can move the graph to a different folder, copy it, print it, and delete it. But to change its color or shape, or to delete part of the graph without deleting all of it, the user must open a window that contains a view of the graph.

To use a particular aspect of an object, users must look through the available views to find a view in which the aspect is represented. To allow users to change an aspect, some view must support interaction with that aspect. For example, to determine which font a text title uses, some view must represent the font, the title, or both. A composed view may show the usage of different fonts, but not provide a way for users to change them. A settings view is typically used for such things as changing fonts, colors, and sizes. Figure 11 shows a composed view with a string of text selected. A settings view is open on top of the composed view and shows the different settings that a user could select to change the way the selected text string is displayed on the screen or printed.

The designer's challenge is to provide the right combination of views, each representing logically related aspects, and supporting interactions required by users to accomplish their tasks. Designers can help users associate different views with what those views contain and the tasks for which they are intended through careful selection of names used for views in menus and elsewhere in the interface.

User distinctions between types of views. Users do not have to consciously think about distinctions between the types of views to use the interface. They need only find a view that presents the aspects of the object they want to use.

The distinctions between the view types are presented to help designers develop views based on logical groupings of object aspects. The actual terms used in the interface should be appropriate for users and should describe as accurately as possible the role of the view. For example, "Composed" is not intended to be a user term. A designer might call a composed view of a document "Formatted Text" or "Print Preview." Also, "Contents" is not intended to be a user term. A designer might call a contents view of a folder "Icons" or "Details."

The usefulness of providing these different types of views can be demonstrated through a simple analogy. For example, suppose you purchase a component stereo system with a glass-front cabinet. The user's manual typically contains a figure showing the system completely installed, with the components in the cabinet and the speakers on either side. This is a composed view showing all of the components in appropriate relationships. If there are several possible arrangements of the components, several composed views might be shown.

The packing slip typically lists the components, and the quantities when multiples of certain components are used, like speakers. Sometimes pictures of the components are shown beside the names to help you identify them. These are examples of contents views of the stereo system.

If the user's manual contains a listing of the programmable remote control functions, these are examples of settings views. The specifications for the system are also a settings view but since users can't easily change the specifications they aren't quite as interesting.

Finally, the Quick Reference Chart for the controls and In Case of Trouble pages are both examples of help views. Each of these views serves a specific purpose and supports certain user tasks in installing and using the stereo system.

Pointers and cursors. *Pointers* and *cursors* provide visible connections between input devices and representations of objects. Each pointer and cursor is associated with an input device. Pointers and cursors show a user where the next interaction will occur when the respective input device is used.

There is typically only one pointer. It is associated with a pointing device such as a mouse, track ball, or joy stick. If a system were to support more than one pointing device at a time, there could be a pointer for each pointing device. Pointers can be moved over the entire screen. User actions, such as mouse button clicks, are transmitted to the object over which the pointer is positioned. If that object can accept input from the input device that controls the cursor as well, and it should if possible, the cursor is moved to the pointer's position.

Cursors associate an input device such as a keyboard with a particular view. A cursor moves within its associated view, between the objects that are presented in that view. Keystrokes are transmitted to the cursor's current position. However, a cursor cannot move from one view to another. Therefore, each view has its own cursor. This means that the keyboard can be associated with only one view at a time; only the view that has the input focus shows its cursor. If a system supported more than one keyboard at a time, there could be a cursor in each view for each keyboard, but, again, the cursors would only be visible in the views that have the input focus.

The shapes displayed for pointers and cursors give the user information about the current state of the object and which actions are available. For example, the pointer is normally displayed as an arrow, which shows that the pointing device can be used for selection and dragging. Another common shape for the pointer is an I-beam for positioning a text cursor.

CUA defines two types of cursors associated with the keyboard: a selection cursor and a text cursor. A *selection cursor* is used to select objects, choices in menus, controls in action option windows, and so forth. It typically appears as a dotted-outline box around a control, such as a radio button or check box.

A *text cursor* is used to type text. It typically appears as a vertical line between text characters during insert mode, or as a bar of color during replace mode. Refer to Reference 2 for more information about pointers and cursors.

How users work with objects

CUA identifies six general types of actions that users can perform on objects:

- Copy
- Create
- Move
- Connect
- Change
- Discard

These actions are enabled or initiated through icons and views.

Copy and create. Both the copy and create actions allow a user to create a new object from an existing one. Because these two actions have similar but different results some examples of usage are first provided to give an appreciation of the possible benefits users may derive from the differences between the two.

Taking advantage of the differences. Users can generate new objects from existing objects by using either create or copy. In the most simple case the copy action results in an exact replica of the object for which copy was requested while the create action results in an initialized, "empty" object. In this simple case the object designer has decided not to use the object's current context or some other information to tailor the newly created object. The created object contains no information other than initial settings of its properties, which may or may not match those of the object from which the create action was requested.

However, object designers can provide users with powerful and productive capabilities by designing the create action so it creates new objects dynamically tailored to specific user needs. For example, invoices are typically numbered with a unique sequential invoice number. The create action for an invoice object can be designed so that it automatically generates the next valid invoice number. This saves the user time and avoids potential errors if the invoice number uses some special format or numbering sequence.

Using the copy action on an existing invoice will result in a new invoice object that has the same invoice number, customer name and address, and item list as the invoice being copied. Using the create action would result in a new invoice with the next valid invoice number, no items, and potentially a customer address already filled in, based on the folder the invoice is in. Tailoring information might also come from connections, or links, between an invoice and other related objects, such as customer records and history data. The amount of tailoring done on create is entirely up to the designer of the object. Designers can provide significant work-saving assistance to users by thoroughly understanding users' tasks and designing "intelligent" create actions. These actions can use information within the current context to help users be more productive.

Copy. The copy action creates a new object that is an *exact replica* of an existing object. In cases where objects must have unique names, such as in a folder within some file systems, the names of the two objects will be different.

Create. The create action is used to make a new *initialized* object from an existing one. The new object is the same type as the existing object. The new object may inherit properties and content from the existing object, from other related objects, from the current context, or from whatever source the object designer deems useful and meaningful to the users of the object.

Templates. Designers are encouraged to provide objects specifically designed to be used as templates for creating new objects of the same type. For example, folder and work area templates should be provided by the system. Users can then create their own folders and work areas from these templates.

Products should provide versions of product-specific templates, such as documents, charts,

graphs, and spreadsheets, that users can use to create new objects in an identical manner. Templates provide basic settings and content. They do not have to be complete, final-form objects. For instance, a template of a document might provide basic settings for document format, fonts, style, and print options. It might also contain some standard paragraphs of text. A new document created using the template document would have the same settings and would contain the same paragraphs of text.

Designers are encouraged to provide objects specifically designed to be used as templates.

Any object can be used as a template to create new objects of the same type. For example, a user might want to create a memo template using a personalized letterhead and logo. The user could edit any memo to contain the letterhead, logo, and desired settings. New letterhead memos could then be created from this memo template using the create action.

Furthermore, since creation of new objects is typically a very frequent action, the CUA interface provides a shortcut technique for creating objects using drag and drop. *Create-on-drag* is a property that a user can set for any object. When users choose this property the object's icon changes to indicate that one of its drag and drop behaviors will be the create action.

Systems often have various naming requirements for objects, and names often must be unique within some scope, for example within a given folder. The copy and create actions should generate a default name and ensure its uniqueness for objects when this requirement exists.

Move. Users can use the move action to change the location of an object. They can move objects on the workplace, between folders and work areas, and to any location that is acceptable for the object.

Connect. Objects can be connected, or linked, to provide inter-object relationships. For example, the model name of a car in a view that lists cars in stock can be connected to a picture of the car. When a user activates the connection, which might be done by "double-clicking" on the model name, the picture of the car appears in a window. In addition, once a connection is established between two objects, data can be transferred between the objects. For example, certain cells of a spreadsheet might be connected to the bars in a business chart. When users type new numbers in the cells the bars would change. Likewise, if users could change the size of the bars by some direct manipulation technique the numbers in the cells would change at the same time.

Designers determine the connection relationships that need to be supported for each object they design. They enable the connections by following CUA-specified guidelines for the user interactions and through object-to-object communication, using methods such as named *pipes* and *dynamic data exchange*. The Workplace Model establishes the direction for generalized and consistent object connections. Over time, CUA will evolve to include these capabilities and the necessary supporting tools to make object connections a generally available capability.

Change. Users use a variety of actions to alter settings and otherwise modify the contents of objects, for example, typing text, changing a font, and changing colors. These are collectively called change actions.

Discard. The discard action removes an object from the computer system. When an object is discarded, its visible representation disappears and it is no longer accessible by users.

The term *discard* is used here in a generic sense to represent any action that removes an object from the system. Delete and Clear are the names of two CUA-defined discard actions that appear on menus and in push buttons. While both actions remove an object from the system there are differences at the point formerly occupied by the discarded object. These differences are described in Reference 2.

Multiple levels of discard should be provided to create a forgiving environment for users. For example, the first discard action for an object might simply move the object to a container for objects to be discarded, such as a wastebasket. The user can retrieve objects from this container at any time until a discard action is taken on this container.

How users interact with objects

Users' interactions with objects are described in several models of interaction:

- *How information is displayed* describes how users accomplish tasks by interacting with views of objects displayed in windows.
- *Direct manipulation of objects* describes how actions are accomplished by using drag and drop, and pop-up menus.
- *Selection of objects* describes how users select objects on which to operate and the effects of view layout on selection techniques.
- *Moving the cursor in object views* and *scrolling an object view* describe how users move the cursor within a view and scroll a view within a window.
- *Role of the menu* describes how menu organization relates to selection and the information in a view in an object-oriented interface.

In the following discussion of each topic the emphasis is on describing concepts that will help designers understand and accurately implement the CUA Workplace interface.

How information is displayed. Each view of an object serves a particular purpose by displaying certain aspects and enabling certain user actions. Controls such as menus, entry fields, buttons, and others can be used to display information in a view and enable user actions. Some user actions are common to all views, such as selecting, moving the cursor, and scrolling. Users may also find it useful to have more than one view of an object open at a time.

By convention, views are rectangular so they have top, bottom, left, and right edges. Cursor and scrolling functions are bounded by these edges, and some of these functions are designed specifically to allow users to move quickly to an edge of the view.

Views are displayed in windows. The Open action opens a window on a particular view. The window provides an area of the display screen in which a portion of the view is displayed. Views often contain more information than can be displayed in a window at one time. If the view is small enough to be completely contained within the window the user can see all of the information at one time. If the view does not fit entirely within the window the user must scroll. Scrolling actions allow users to control the portion of the view that is visible in the window.

Different views for different tasks. Object designers provide views that allow users to perform tasks using individual objects and groups of objects. Designers should look for situations in which the same information is used in several tasks. The goal is to provide as few views as necessary to support the tasks while not compromising optimizations that allow each task to be performed efficiently. This is a design tradeoff that is made based on an understanding of users, their skills, and their expectations. For example, CUA identifies Icon and Detail layouts as useful for content views. Both of these two layouts show the contents of a container, such as a folder, but show different levels of detail to accommodate various user tasks.

CUA identifies the four types of views—composed, contents, settings, and help—based on the kind of information contained in them. This is a coarse distinction and for many objects designers may need to provide views with finer degrees of distinction.

Distinctions between views are based on differences in the type of information in the views and hence in the tasks supported by them. For example, a contents view of a compound document would identify the individual objects included within the document. This view is useful for determining in which documents a particular object is used, and for accessing that object as an entity, for example to copy or delete it. In a composed view, such as a print preview, the individual objects may not be apparent. A composed view is useful for formatting and otherwise assessing the appearance of the final form of the document as a whole.

This distinction between types of views should not be confused with the need to provide different

views that vary in layout but otherwise display the same type of information. For example, a contents view of a folder might provide options for an iconic view and a text view. Both views display the contents of the folder and thus enable basically the same actions. They are both contents views.

Views can often be tailored in other ways as well. Options to display information in different sort orders, to include only certain objects, and to show more or less detail are common. Since these options affect only the layout or amount of information displayed but not fundamentally a different kind of information, essentially the same user actions are enabled in each view.

The CUA guidelines for menus reinforce distinctions between types of views and options within a view. The intent is to help users quickly locate a view having the kind of information needed for the task desired. Users can then select layout options based on personal preference and optimization for a particular situation.

Multiple concurrent views. Users can open multiple windows containing the same type of view. This can be useful, for example, to look at different pages of a document at the same time. Since both views contain portions of the same object, changes in either window may affect what is seen in the other. When this is the case the changes should appear simultaneously to reinforce that both views are representations of the same object.

Multiple windows can also be used to display different views of an object concurrently. This can be useful, for example, to observe the effect of changing formatting properties in one window while the document is displayed in another. Again, the changes should appear simultaneously in both windows.

Using visuals and controls in views. Views are composed of visuals. *Visual* is the term used to refer to any drawing on the display regardless of whether it is text, graphics, image, or video. Audio can also be a component in a view and is not meant to be excluded by use of the general term visual.

Designers use visuals to convey object information to users. Whether a visual is a string of text or a picture is up to the designer. Designers should decide which forms to use by understanding what information users need, which forms of presentation might be most easily recognized, and which forms might be most efficient in supporting user interaction.

Controls are special views that are provided as part of a programmer's toolkit because they are common across a wide variety of applications. Designers can use controls to provide views of specific aspects within a larger view. For example, a check box control can be used to provide a view of the Bold property in a settings view for a document. Similarly, multiline entry field controls can be used to provide views of the header and footer text.

Direct manipulation of objects. CUA defines direct manipulation actions as those actions in which users interact directly with the desired object, for example, by clicking on or dragging an icon that represents the object. Such actions are contrasted with actions accessed by using the menu bar, in which the object is implied by prior selection.

CUA specifies two direct manipulation techniques:

- Drag and drop
- Pop-up menus

Drag and drop allows users to "pick up" an object and drop it on another object to accomplish some action involving the two objects. For example, dragging the icon of a spreadsheet to the icon of a printer would cause a view of the spreadsheet to be printed. Also, dragging a string of text to an entry field would cause the string to fill in the field. Dragging a numeric string in the proper format to a telephone icon could cause the number to be dialed. Figure 12 shows an example of drag and drop.

Pop-up menus are accessed by directly interacting with an object. A Menu action is defined to provide access from both the mouse and the keyboard. Pop-up menus dynamically appear beside the object and contain only actions pertinent to the particular object in its current context. The context is affected by factors such as the type of container within which the object resides, the state of the object, and the contents of the object itself.

The use of direct manipulation techniques reinforces the object model for users. When it is necessary to design object-specific interaction techniques it is helpful to consider how they relate to the direct manipulation and other action techniques specified in the CUA.

The degree of direct manipulation is probably best thought of as a characteristic of interaction techniques. The degree to which any technique reflects this characteristic can be mapped across a spectrum of possibilities. For example, typed commands are at one end of the spectrum, providing little if any direct manipulation "feel." Drag and drop techniques are at the opposite end of the spectrum, providing much direct manipulation feel. Pop-up menus do not provide as great a degree of direct manipulation as do drag and drop techniques, but they provide more direct manipulation feel than does the menu bar and associated pull-downs and cascade menus. Figure 13 depicts this spectrum.

Drag and drop actions usually involve a source object and a target object. For example, when a spreadsheet icon is dragged to a printer icon the spreadsheet is the source object and the printer is the target object. The result of drag and drop depends on the classes of the source and target objects.

CUA specifies a general paradigm for drag and drop on the workplace, based on the container, data, and device distinctions described earlier. The paradigm is based on the principles that:

- Drag and drop should provide, as much as possible, results that are intuitive for the source and target involved.
- The results should be comparatively safe, not allowing users to unexpectedly lose information.
- Overrides should be available to allow users to explicitly request useful alternative results.

These principles provide the basis for the CUA paradigm for drag and drop of objects that can exist on the workplace. During drag and drop the source object is displayed with source-emphasis as a reminder of which object is being dragged. The pointer remains visible and the source object is dragged with the pointer. While the pointer is over a target object the object being dragged and the pointer are changed in appearance to show whether the result of dropping will be a move, a copy, a link, or whether no action will occur. The target object is displayed with target emphasis while the pointer is over it, to help users discriminate between overlapping target objects.

CUA defines direct manipulation actions, e.g., dragging an icon that represents the object.

Dragging a data object, container object, or device object to a workplace container results in *moving* the source object into the target container. For example, dragging a document, a printer, or another folder to a folder on the workplace causes the object being dragged to be moved into the target folder.

Containers that exist on the user's local fixed disk and fixed disks in network servers to which the user has access are considered to be *within* the user's workplace. Containers on removable media are considered to be *outside* the user's workplace and are treated like devices.

If the target is a device that provides containment behavior, the source object is *copied* into the target container. For example, dragging an object to a printer or a folder on a diskette causes the source object to be copied into the target object. The source object remains in its location within the workplace containment hierarchy. This protects users from inadvertently losing information from the workplace environment.

Users can override the impending result, shown by the appearance of the source outline and pointer, to explicitly cause a move, a copy, or a link.

The drag and drop technique is used only with pointing devices, such as a mouse. Equivalent actions are available using menus, which can be accessed by using the keyboard as well as the mouse.

Selection of objects. Selection encompasses a set of techniques and an object state with which users can indicate objects and actions they want to manipulate. In a point-select object-action interface, users identify an object, then an action. Selection is necessary because the same mechanism, for example a mouse and pointer, is used to do both. The mechanism is first used to select an object, which puts the object in the selected state, then the same mechanism is used to identify the action to be performed. The action is applied to the selected object. When an object is in the selected state it is displayed with a form of visual emphasis called selected emphasis.

In the real world, users manipulate objects directly. The concept of selection is implicit in the manipulation. Likewise, direct manipulation in the user interface does not require selection. In the CUA, selection and direct manipulation are independent techniques. With one exception, objects need not be selected to be dragged, and dragging does not alter the current selection. For example, a document need not be selected to print by dragging it to a printer, and dragging does not affect what is selected. The exception to this independence of dragging and selection is when users want to manipulate a group of objects. Selection is used to identify the group. For example, three documents can be printed with one drag operation by first selecting the three documents as a single group and then dragging the group to the printer icon.

Pop-up menus also provide a degree of direct manipulation, and access to them is independent of selection to the same degree as is dragging.

Scope of selection. A selection scope identifies a domain of objects from which users can make selections. Objects from several different selection scopes can be selected concurrently. For example, each primary window establishes a selection scope. Objects in several primary windows can be selected at the same time. Selected emphasis is displayed only for the window that has the keyboard focus, but the selections in each window are preserved and selected emphasis is redisplayed as users switch between them. When a window is split, each resulting pane establishes a separate selection scope. Similarly, list boxes, groups of radio buttons, entry fields, and other controls that support selection each establish a selection scope.

Within a selection scope two important aspects define how users can select choices:

- The number of choices that can be selected at one time (one or many)
- The fewest number of selected choices allowed (zero or one)

These aspects are the basis for different types of selection.

Types of selection. There are three types of selection with respect to the number of choices that can be selected at one time: single-choice selection, multiple-choice selection, and extended selection. *Single-choice selection* is used when only one choice within a selection scope can be selected. Selecting a choice in a single-choice selection scope supersedes any previously selected choice. *Multiple-choice selection* is used when more than one choice within a selection scope can be selected. Selecting a choice does not affect other previously selected choices. *Extended selection* is used when it is likely that users will select one choice but occasionally may need to select more than one. This type of selection is intended to accommodate less experienced users who are likely to select and act on choices one at a time, as well as more experienced users who may want to act on several choices at once. Extended selection behaves like single-choice selection unless the user decides to override this behavior and select multiple choices. Specific keyboard and mouse override techniques are specified in the CUA guidelines.

Each type of selection must also specify the minimum number of selected choices allowed as either zero or one. In zero-based selection it is valid to have no choices selected. In one-based selection at least one choice must always be in the selected state; having nothing selected is not allowed. For example, zero-based single-choice selection does not require that a choice be selected at all times, but if one is selected it must be the only one. In one-based single-choice selection one choice is always selected.

The menu bar contains one group of choices and supports single-choice selection. Pull-down, cascade, and context menus can contain one or more groups of choices. Each group of choices is either single-choice or multiple-choice even though only one choice can be selected at a time because these

menus disappear each time a selection is made. When these menus contain multiple-choice groups or more than one group, users can access the menu as often as necessary to make multiple selections.

Selection techniques. CUA specifies interaction techniques to support the types of selection. Each selection technique must address two aspects of selection:

- Which choices are to be selected
- Whether the selected state of other choices is affected or not

The CUA *mouse selection* techniques use a Select button on the mouse and keyboard modifier keys. The selection techniques follow a general selection paradigm that is common across different types of objects and view layouts. In general, the techniques support selection of an individual item, selection of groups of individual items, and selection of areas and ranges of contiguous items.

Selection of an individual item is provided by clicking a mouse Select button. Selection of a group can be accomplished by pressing a keyboard key while clicking the mouse Select button, or by touching each individual selection while the mouse Select button is held down. Selection of areas and ranges of contiguous items can be done by clicking at begin and end points, or by *swiping* between the two points. A "stretchable" outline box called a *marquee* box is shown in some types of views.

The CUA mouse button mappings support the use of different buttons for selection and direct manipulation. This allows a variety of selection techniques to be provided, some of which require moving the mouse with the Select button pressed, as well as supporting drag and drop of selected groups of objects by moving the mouse with the Drag button pressed.

Keyboard selection techniques parallel those for the mouse in most respects. For example, CUA specifies that a Select key on the keyboard and the Shift key provide a range selection capability similar to that provided using a mouse. Single-choice and multiple-choice selection is performed by moving the cursor to the desired choice and pressing the Select key. A range can be selected by moving the cursor to the desired end point and pressing Shift plus the Select key.

CUA also specifies two selection techniques optimized for keyboard use:

- *Mnemonic selection* allows users to select a choice by typing a single character, which is typically one of the letters of the choice text.
- *Automatic selection* using the cursor's position is provided when only one group of choices is available and the group allows only one selection. This saves the keystroke of pressing the Select key.

Mnemonic selection should not be confused with a technique of moving the cursor to the first letter of each choice in a list. This latter technique simply moves the cursor and does not necessarily select the choice, depending on whether the list also uses automatic selection.

View layout affects selection. The selection techniques made available to users depend on the type of information to be manipulated and the layout of the view in which it is displayed. CUA defines various selection techniques optimized to different information organizations and view layouts. Selection techniques such as swipe range, swipe touch, and marquee are available in various types of views. Information order and overlap are some of the aspects of a view that determine which selection techniques can be provided. There are typically three types of view layouts that determine which selection techniques are supported:

- Strings, such as text
- Arrays, such as lists and tables
- Free-form layouts, such as fill-in-forms and graphics

For selection, a string can be thought of as a linear ordered set of selectable objects. For example, a text string is a linear ordered set of characters. The order of characters is important to the semantics of the string and remains fixed even though the viewing layout may change. Text views typically flow a text string from left-to-right, top-to-bottom in one or more columns.

Range selection depends on the definition of some order between the selectable objects. Since text is ordered, views of text typically support the range selection techniques. Users can select one point

in the text, move to another point in the text and select a range of characters between the two points regardless of how the text flows.

Arrays typically support range selection by defining a range as a rectangular group of array cells.

CUA defines selection techniques optimized to different information organizations and view layouts.

Users can select one cell as the corner of a range and another cell as the opposite corner. Single-column lists are simple cases of arrays and similar selection techniques apply.

Marquee and swipe touch selection are most useful in unordered views such as user-arranged icon views of folders, fill-in-forms and graphics drawings.

Information characteristics, such as order, and view layout are important factors that influence the decision of which selection techniques to provide. Over time, and with increasing emphasis on the use of image and video, new uses for existing selection techniques as well as new techniques are likely to be identified.

Moving the cursor in object views. A cursor provides a visual connection between an input device, such as a keyboard, and the information in a view. When input is entered it appears at or affects the information at the cursor location. CUA specifies guidelines for several aspects related to cursors such as types of cursors, what cursors look like, where cursors can be positioned in views, how cursors are moved, what happens to cursors during scrolling, and how cursors relate to pointers.

Cursors are visuals that show points within information being viewed where the effects of interaction from a related device will occur. CUA defines two types of cursors based on the type of information in which the cursor exists:

- *A selection cursor* is used within groups of selectable choices.
- A *text cursor* is used within textual information.

Designers may need to define other cursor types for object-specific information manipulation. For example, if graphics are to be manipulated using a keyboard, a graphics cursor is needed.

The standard selection cursor visual defined in CUA is a dashed-outline box. Some of its uses are around icons on the workplace, in views of folder contents, and around choices in property views and action windows.

Menus typically use a selection technique called automatic selection in which the choice indicated by the cursor is automatically selected. When automatic selection is used, the selection cursor is not shown. Selected emphasis provides both cursor location and selected state indications. Figure 14 shows an example of automatic selection in a menu.

A text cursor represents a point within textual information where new text can be inserted and existing text can be changed. A vertical between-characters bar is used for text insertion and editing. When entry field controls are used in a window the text cursor identifies the point where keyboard interaction will occur. A separate selection cursor is not shown. For example, the selection cursor is typically a dashed-outline box around controls such as radio button and check box choices, but when the cursor is moved to an entry field the dashed-outline box disappears and the text entry cursor appears within the field. Figure 15 shows an example of text cursor.

Cursors show points within the information being viewed where user interaction can occur. Therefore, cursors can only be positioned at valid interaction points. For example, a selection cursor can only be positioned on choices. It cannot be positioned between choices, on field prompts, or on column headings. Similarly, text cursors can only be positioned at valid text entry and revision points. This behavior is based on a design principle of error avoidance. Users are prevented from positioning the cursor at invalid points and errors that would result from attempted entry are avoided.

Users can move the selection cursor and text cursor by using keyboard cursor movement keys.

When using the mouse the select action causes the cursor to move to the pointer location. This is called pointer-cursor *join*. For example, when the user selects an icon on the work area the selection cursor moves to that icon.

This joining of the cursor and pointer keeps both devices focused on the same point in the view. This allows users to switch back and forth between the two devices to take advantage of whichever device is the most convenient and efficient in each situation.

A *first-letter cursoring* technique allows users to quickly move the cursor to a choice by typing the first character of the choice text. For example, in a lengthy list such as the states of the United States, a user could cursor from Alabama directly to Hawaii by typing the letter "H." The cursor is positioned on the choice but the choice is not selected, unless the field also uses automatic selection. Each time a letter is typed the cursor is moved to the next choice starting with that letter.

When a cursor is used in a view that can be scrolled, the cursor may be scrolled out of sight. Because cursors are only positioned on valid information interaction points, if the point on which the cursor is positioned is scrolled out of sight the cursor can disappear with it. The scrolling technique used determines whether the cursor disappears or not.

Scrolling an object view. Users can see the entire view of an object only if the window can be made large enough to contain the entire view, or if the view can be scaled, or reduced in size, to fit within the window. Otherwise, users must scroll the view. CUA identifies two categories of scrolling actions, based on the role of the cursor:

- Cursor-driven scrolling
- Cursor-independent scrolling

In *cursor-driven scrolling*, cursor movement causes automatic scrolling when the cursor meets a window border. For example, if a user is moving a selection cursor down a list of choices that extends beyond the border of the window, the view will be scrolled up to reveal the next choice in the list when the cursor meets the window border. Figure 16 shows an example of cursor-driven scrolling.

Actions that move the cursor cause cursor-driven scrolling. These actions include: Up Arrow, Down Arrow, Left Arrow, and Right Arrow cursor keys; Beginning of Data (Ctrl + Home) and End of Data (Ctrl + End); and Beginning of Line (Home) and End of Line (End).

In *cursor-independent scrolling* the view is scrolled without affecting the cursor position. For example, when users scroll a view using a mouse and scroll bars the cursor stays in its position within the information. It will disappear from view if the information it is on is scrolled out of the window. The Select button on the mouse causes the cursor to join with the pointer and can be used to bring the cursor back into view. Cursor movement keys on the keyboard cause the view to be scrolled to make the cursor visible again.

If designers find it necessary to provide additional scrolling techniques, they should decide what the effects on the cursor position should be, if any. For example, sometimes it is desirable to scroll the view while maintaining the position of the cursor in the window. The appearance is that the cursor is fixed in the window and the information is scrolled underneath it. This requires that each scrolling action result in a data point underneath the cursor, which is typically only possible in information views that have a repeating structure such as in lists and tables.

Role of the menu. Menus have been a mainstay of various user interfaces for many years. They have been used to present lists of objects, actions, properties, and various other types of choices. Menu forms have also been varied, ranging from full-screen to dynamic pop-up menus. In point-select object-oriented interfaces the primary role of menus has been to present action and property choices for objects. With the evolution to direct object-oriented manipulation, the role of menus needs to be re-examined.

Types of menus and menu choices. CUA specifies the use of menus for three types of choices: actions, routings, and settings. *Action choices* allow users to perform actions on selected objects, such as when the user deletes a memo. *Routing choices* result in continuation of the dialog by presenting additional information, such as "Print . . . ", which leads to a window in which users specify printing options, or Open View, which leads to a cascade menu containing view

choices. *Settings choices* allow users to change the properties of selected objects, such as setting Bold and Underline for text.

CUA specifies the use of the notebook control in settings views for object properties. Therefore, the use of settings in menus should decline over time. Settings notebooks offer the advantage of having many related properties visible concurrently and they can accommodate types of controls, such as entry fields, not permitted in menus.

CUA also specifies several types of menus: a menu bar, pull-down menus, cascade menus, and pop-up menus. Figure 17 shows some examples.

The menu bar is a list of choices that appears across the top of many windows, just below the window's title bar. It is used only for routings to pull-down menus. A pull-down menu is displayed below a menu bar choice when the menu bar choice is selected. The choices in pull-down menus can represent actions, routings, and settings. A routing choice leads to a cascade menu or causes a window to be displayed. Routings to windows are used to present action options. A cascade menu is displayed beside another menu from which it is displayed when a routing choice is selected. Cascade menus contain the same types of choices as do pull-down menus.

A pop-up menu is dynamically displayed when a user points to an object and clicks the Menu button on the mouse or presses the Menu key on the keyboard. The choices in pop-up menus typically represent actions and routings. The choices are arranged in the following three groups, from the top to the bottom of the pop-up menu:

- View choices
- Data transfer choices
- Convenience choices

View choices provide alternative views of the object. *Data transfer choices* result in moving, copying, connecting (linking), and creating. These actions provide a menu-driven alternative to corresponding direct manipulation actions. *Convenience choices* are frequently used choices, such as Print and Delete, that can be placed in the pop-up menu for the user's convenience. Frequency of use for each choice may vary with different users. Therefore, user customization of this section of the pop-up menu is encouraged. User customization should support choices for settings as well as for actions and routings.

Styles of menus bars. Menu bars are popular in graphical user interfaces because they provide users with visual cues to the actions that are available. The menu bar and its pull-down menus provide a domain in which users can explore to determine what functions are available. The menu bar provides a "home base" from which users can venture out and to which they can return when they begin to feel lost.

The CUA-defined menu bar has evolved based on requests for greater consistency across products. However, with the evolution to an object-oriented interface, further evolution is necessary. The traditional application-oriented File, Edit, View, Help (FEVH based on the first letter of each of the choices) menu bar cannot adequately cope with the rich object-oriented environment typified by composite documents. In response to these evolutionary pressures CUA is designing an object-oriented menu bar. This new menu bar style directs the user's focus to four objects associated with each window: the window itself (W), the object (O) being viewed in the window, objects that are selected (S) within that view, and Help (H). This new style is called WOSH, based on the four objects it addresses.

The evolution of existing applications and development tools to the WOSH style will take time. To aid this migration an intermediate style called FSEVH has been defined. This style is similar to FEVH with an added choice (S) for actions affecting selected objects within the current window. The FSEVH style can be created using existing development tools, yet it begins to focus users on working with objects. Figure 18 depicts the current FEVH and FSEVH styles, and it shows a potential configuration of the WOSH style in which the menu bar has been combined with the window's title bar. The following paragraphs provide additional detail on the characteristics of each style and the pressures contributing to this evolution.

The FEVH menu bar. For the Graphical Model, CUA specifies standard menu bar choices of File, Edit, View, and Help (FEVH). Window actions are available using a system pull-down menu from the

window title bar. The FEVH style has evolved with traditional interfaces oriented to application programs and concepts of starting programs, finding and using objects to be processed, and exiting the program.

Problems associated with using the application-oriented FEVH menu bar with the object-oriented Workplace Model have led to the evolution of alternative menu-bar styles. Some of the problems associated with the FEVH style have been:

- Difficulty in achieving adequate consistency across applications
- Inadequate support for container objects
- Inability to gracefully handle large numbers of actions found in composite objects
- Ambiguity of the relationship to object-based pop-up menus

Despite the efforts of style guidelines, arbitrary inconsistencies between the menu bars of different applications that provide similar functions continue to occur. Guidelines can only address pervasive functions, like the File, Edit, View, and Help actions specified in the FEVH style. Product-specific actions such as Format, Color, Style, and so forth often appear in quite different places in the menus of different products. It is not practical to address this level of consistency through guidelines. The FEVH style does not provide designers with a clear and adequate basis for distinctions between menu-bar choices and pull-downs.

When used with container objects the FEVH style can support actions on the container, or its contents, but not both. For example, Print is one of the actions specified in the File pull-down menu. When an FEVH menu bar is used on a view of a container, such as a folder, the Print action might be made to apply to either the folder itself, or the selected object within the folder. Most implementations today apply it to the selected object. Therefore, in order to print a listing of the contents of the folder the user must navigate to the folder's container, open a view if one is not already open, select the folder, and request Print. It would be more natural to allow users to print the contents view of a folder while they are looking at it.

The object-oriented Workplace interface provides the opportunity for users to nest objects within objects as deeply as they desire, and to use any combination of objects they find useful. The FEVH menu bar is a collection of actions that can be applied to the objects within the window to which it applies. Actions that are not applicable to the selected object are typically shaded in gray. When the set of objects becomes large and unpredictable, the FEVH menu bar becomes impractical. Work-around techniques, such as changing the menu bar for each different selected object, can be employed, but these techniques typically detract from the stability and familiarity that makes users comfortable with menu bars in the first place.

Finally, with the advent of pop-up menus that relate directly to a particular object, there is a question of relationship between these menus and the menu bar. Are they two independent mechanisms? Or, should users continue to be able to explore the interface by using the menu bar and transition to pop-up menus as they become more familiar and comfortable with the functions provided?

These problems and questions are addressed by a new object-oriented menu-bar architecture called WOSH.

The WOSH menu bar. Problems with the FEVH menu style can be solved while still providing the advantages of a menu bar by using object-orientation in the menu-bar organization. The WOSH style (the window itself, the object being viewed, the selected object, and help) is evolving with the trend toward object-oriented interfaces where concepts associated with starting and running programs are replaced with concepts such as object views and object containment.

The WOSH style establishes a clear and distinct framework for the location of actions. Actions appear in the menus of their respective objects. Thus, in contrast to the FEVH style, using the WOSH style allows a user to Print both the folder and a document in the folder from the same window. The Print choice will appear in the menu for the folder and in the menu for the selected document. For users experienced with FEVH style menus, having the same action appear in two different places may seem confusing at first, but when users realize the object-based distinction it becomes very natural. Current interfaces using the FEVH style may have the same actions in adjacent windows, and pop-up menus provide the

same actions on different objects as well, so the concept is not entirely new to users.

When used with composite objects, the WOSH style enables graceful and consistent decomposition to any desired level. For example, while viewing the contents of a folder the Object menu contains action choices for the folder, such as alternative views, Print, Send, and Delete. The Selected menu contains action and view choices for the selected object contained within the folder, such as a selected document. When the user chooses one of the view choices for the selected document a new window is opened containing a view of the document. The WOSH menu in the new window applies to the document. That is, the Object menu applies to the document and the Selected menu applies to any objects selected within the document, such as a business graph.

This decomposition could continue indefinitely. At each level of the decomposition the user has a clear indication of the objects involved and the associations between actions and objects, and the technique is consistent regardless of object types.

The WOSH style also establishes a clear and useful relationship between the menu bar and pop-up menus. The Selected menu is essentially the same menu as the pop-up menu for the selected object. Just as the user might see a different pop-up menu using different objects, the contents of the Selected menu may vary depending on which object is selected. As new users who begin by using the menu bar become more comfortable with the interface, they can make the transition to using pop-up menus and will find the same actions available for each object regardless of the menu they use.

The actual appearance of the WOSH style is yet to be determined. It is likely that both iconic and text versions will be provided. An additional goal of the WOSH menu style is to merge the menu bar and title bar, thus providing more space for the display of the user's information. This evolution is not yet complete and several alternatives for presentation and interaction using a converged menu bar and title bar are being evaluated.

The FSEVH menu bar. In the interim, the CUA guideline published in 1991[1,2] has taken a first evolutionary step toward the WOSH style by specifying an addition to the FEVH menu bar. This addition provides some of the advantages of the WOSH orientation. The addition consists of a Selected choice, added between File and Edit, for objects that are containers of other objects. This extension of FEVH is called FSEVH, again based on the first letters of the standard menu choices. Using the FSEVH style, window actions are still accessed using the system menu pull-down. Containers are accommodated by using the File menu for the object actions. The File choice can also be renamed to the class of the object being viewed, such as Calendar, or Folder. The Selected menu provides access to actions on selected objects within the view. The Edit and Help menus remain essentially unchanged from FEVH to ease user migration.

Object-specific choices are allowed in addition to the CUA-defined standard choices. CUA also specifies guidelines for the types of choices in the pull-down menus for each of the standard menu-bar choices in each menu-bar style. These are described in Reference 2.

Concluding remarks

Models can be very useful in user interface design. They can help us better understand our users, concisely define the concepts we want users to understand, and match our intent to the programming capabilities that are available.

The CUA interface has evolved considerably since its introduction in 1987. The CUA Workplace Model is the latest stage in this evolution, and the evolution will continue. We must fully exploit the potential of composite objects and object connections. We must also integrate new capabilities from the realm of multimedia and new interaction technologies such as pens and handwriting recognition.

When a model is implemented in a prototype it provides us with a proving ground to explore new ideas, potential relationships between objects, new presentation approaches, and new interaction techniques. It can can also help us understand users' expectations and measure their reactions to new concepts.

Through the use of models, designers are encouraged to think more explicitly about the elements of their design and the relationships between the elements. Only through a clear and concise un-

derstanding of our design intent can we begin to evaluate the capabilities and abilities to extend an interface with respect to users' needs, and to assess its acceptability by those users.

Acknowledgment

I would like to express my appreciation to Tony Temple for providing the inspiration and initial impetus that led to the creation of the CUA Workplace Model; to Dave Roberts and the CUA team who completed its design; to the OS/2 Version 2.0 programmers who implemented the first product version; to Fred Brown and Jennifer Adame who helped in the preparation of this paper; and to John Bennett whose devotion to excellence and the methods necessary to achieve it have provided an unending source of motivation.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. *Systems Application Architecture Common User Access Guide to User Interface Design*, SC34-4289, IBM Corporation (October 1991); available through IBM branch offices.
2. *Systems Application Architecture Common User Access Advanced Interface Design Reference*, SC34-4290, IBM Corporation (October 1991); available through IBM branch offices.
3. *The CUA Vision: Bringing the Future into Focus*, G242-0215 (a DOS-compatible demonstration program and brochure), IBM Corporation (October 1991); available through IBM branch offices.
4. *The CUA Vision: Bringing the Future into Focus*, GV26-1003 (a VHS format videotape), IBM Corporation (October 1991); available through IBM branch offices. (GV26-1004 is in PAL format, GV26-1005 is in SECAM format.)
5. R. E. Berry and C. Reeves, "The Evolution of the Common User Access Workplace Model," *IBM Systems Journal* **31**, No. 3, 414–428 (1992, this issue).

Accepted for publication March 17, 1992.

Richard E. Berry *IBM Personal Systems Programming, 11400 Burnet Road, Austin, Texas 78758.* Mr. Berry is a Senior Technical Staff Member in the object technology area in IBM's Personal Systems Programming group in Austin, Texas. He joined IBM in 1968 in the Albuquerque, New Mexico, branch office where he performed a variety of programming maintenance and systems engineering duties. Since moving to programming development in 1971, he has held various technical and management positions including: lead programmer and chief designer of the user programming facility for the IBM 3650 Retail Store System; programming development manager for the Retail Store System; lead architect for the IBM 5520 Administrative System Files Processing, and architecture manager for the IBM 5520 Externals Design. Throughout his career Mr. Berry has concentrated on defining product function and user interfaces. In 1982 he was appointed lead architect of IBM's User Interface Architecture which became the Common User Access (CUA) component of Systems Application Architecture (SAA) in 1987. He was one of the codesigners of the Workplace Model.

Reprint Order No. G321-5481.