GRAPHICAL USER INTERFACES

HISTORY

The user interface is the vehicle by which the user specifies the actions that the computer program is to carry out. The computer program then conveys the results of carrying out this action to the user through the user interface. Early personal computers used character-based interfaces or interfaces that supported only text as input and output. Two widely used character-based interface styles are command language interfaces and question and answer interfaces.

Command language interfaces require that users type in textual commands. User feedback is given as text responses, in the form of results or error messages. In a command language interface, users compose commands and arguments using a grammar understood by that software application. Users of character-based command style interfaces have to rely on memory to recall the commands and objects needed to communicate actions to the computer. These commands are often cryptic, as is the feedback that users receive, including the error messages received if an incorrect command is given. Many experienced users prefer a command language style of interface, as this type of interaction is very efficient. Expert users can type in short sequences of commands that accomplish many tasks. However, command language interfaces are difficult for novices to learn and use because commands must be memorized and recalled.

A question and answer style of interface prompts users with a question. The user answers the question by supplying one of the choices requested by the software application. Depending on the user's response and the sequence of the question, the application issues another question or carries out the desired action. If the information is entered in an incorrect format, an error message is given, and the user is again prompted for the information. Question and answer interfaces are easy for novices to use as they are guided through a series of choices. However, experienced users find them tedious for prolonged use.

The SketchPad system by Ivan Sutherland (1) was the precursor of the modern graphical user interface. In Sutherland's application, the user interface consisted of line drawings. Because of hardware limitations, the graphical user interface (GUI) was not realized commercially until the 1970s with the Xerox Star system (2).

DEFINITION OF A GRAPHICAL USER INTERFACE

Graphical user interfaces (GUIs) use icons or pictorial representations of objects, such as files the user has stored in the computer, and display menus of commands so that user can recognize, rather than recall, actions and objects. Graphical user interfaces allow the user more flexibility in communicating with the computer than the traditional character-based interfaces. Users can communicate by selecting a desired object from a set displayed on the user's computer monitor and picking the appropriate action from the menu of choices. User feedback is also displayed graphically. For example, if a user specifies that a file is to be copied, the computer application might show an icon representing a copy of that file after the action is completed.

Figure 1. Typical desktop graphical user interface.

Graphical user interfaces commonly use bit maps to display graphical images. A bit-map system uses an array of data to represent images. By turning dots on or off and assigning different colors to the dots, different images are displayed. Another imaging system is postscript, which uses mathematical descriptions of images. Complex images described by mathematical formulas take longer to render on a display than the same bit-mapped images. However, images described by mathematical formula can be scaled easily whereas bit-mapped images cannot (3).

Direct Manipulation

User interactions with graphical user interfaces are commonly accomplished through *direct manipulation.* Shneiderman (4) used the term direct manipulation to refer to interfaces with the following properties:

- the objects and actions for a user task are continuously represented on the display
- the user communicates with the computer by selecting a menu choice or by moving an object on the display
- the user immediately sees the result of this action and has the ability to reverse or *undo* it if necessary

Icons are used to represent both physical and abstract objects. Users can directly manipulate objects visible on the screen by moving a graphical cursor and selecting the desired object. The cursor is moved using pointing and selection devices, such as computer mice, trackballs, or touch screens, or by using designated keys on a keyboard (5). In a direct manipulation system, users first select an object on which they wish to perform an action and then they select the action to be performed. This is considered an *object-oriented* style of interaction (6) or *object-action* style.

Menus are often used to present choices of actions to users. A list of items is displayed, and users select an item using the pointing device or by moving the cursor with the designated keys on the keyboard. Thus, the code for the user interface must allow continuous representation of graphical objects, display immediate feedback to the user, and present graphical menus to the user for selecting actions.

Display editors were one of the first uses of direct manipulation techniques. Display editors show multiple lines of text on the screen, and users are able to view tables, columns, page breaks, etc., as they will appear in the printed document. Uses can delete text, move text, change the formatting, etc. by highlighting the desired text and selecting the appropriate commands. The changes appear on the screen instantly and in the form that will be printed. This style has been labeled WYSIWYG, what you see is what you get.

Figures 1 and 2 are examples of typical graphical user interfaces. Figure 1 is an example of a typical desktop interface, containing folders of information, an in-box, and a recycled

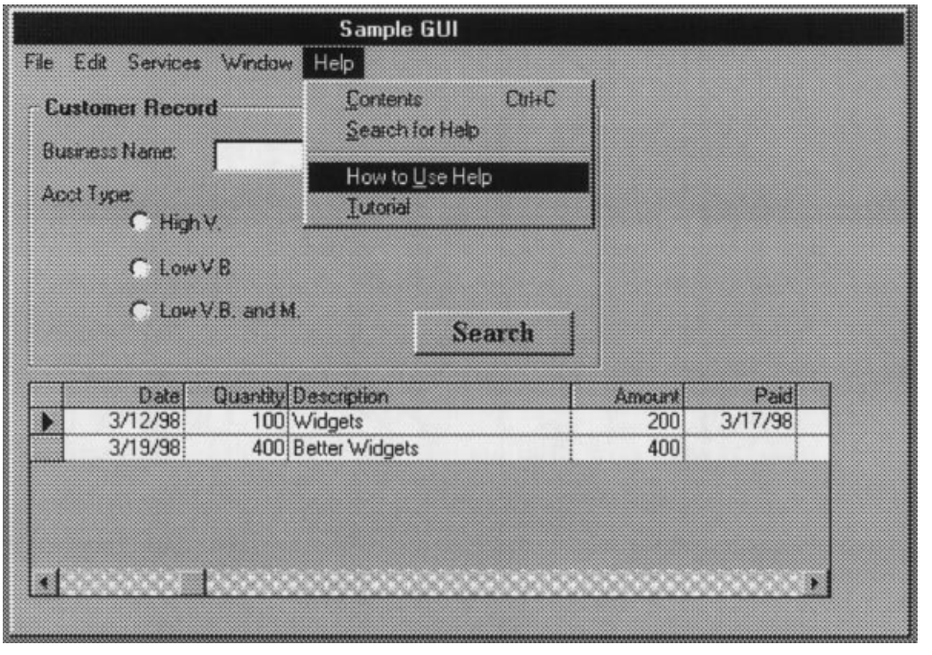

Figure 2. Application window with actions represented by menus.

bin. The user can move any of these iconic representations to different positions on the desktop, open the folders, and deposit any unwanted information in the recycled bin.

Figure 2 shows an application window with actions represented by menus. In the figure, the pull-down Help menu is open, and the menu item, Search for Help, is selected. The user can type a name into the text field, labeled Business Name, and select an account type. The user can then select the Search command button and the information about orders for that company will be retrieved and displayed in the bottom portion of the window. The scroll bar at the bottom of the menu indicates that not all the information is currently displayed. The user can view the remaining information by moving the arrows on the scroll bar.

These two examples illustrate principles of direct manipulation. If a user moves an icon on the desktop shown in Fig. 1, that icon will immediately appear in the new location. In Fig. 2, the actions a user can select are displayed in the menus or on command buttons in the window. The user can immediately see the business name that he/she entered and retype it if necessary. As the user selects a menu name from the menu bar, that menu is displayed (as with the Help menu shown in Fig. 2). Highlighting is used to show the menu item that the cursor is pointing to at any given time.

ARE GRAPHICAL USER INTERFACES BETTER THAN CHARACTER-BASED INTERFACES?

As computers moved out of the research labs into businesses and homes, new types of users appeared. Shneiderman (7) identified different types of users, including novices, experts, and intermittent users. Each of these user types requires different types of interface support to be as productive as possible in using software applications. Novice users know very little about an application in particular and computers in general. To be productive in a short period of time, novices benefit from seeing the functionality and interactions available to them. Novice users need guidance and handholding. Expert users already know how to use computers and this application. They want speed. They are not as concerned with making errors because they know how to recover from mistakes. Intermittent users are casual users of the application. They know what they want to do if only they can remember exactly how to do it. They want help to be available, but only when they need it.

Many studies have shown that graphical user interfaces with direct manipulation can benefit novice or casual users. Studies comparing direct manipulation text editors to nondirect manipulation text editors showed that the direct manipulation text editors are more efficient (8,9).

It has also been shown that direct manipulation interfaces improve learning by novices. Shneiderman and Margono's (10) 1987 study showed that novices were able to learn simple file manipulation tasks more quickly with the direct manipulation interface.

Graphical user interfaces are not appropriate for all types of users. People who are experienced typists type commands more quickly than they move a pointer to a command and select it. This problem has been somewhat alleviated by allowing shortcuts for experienced users. Interface designers often provide alternative ways for users to access frequent commands. Experienced users can use a keystroke or combination of keystrokes to select common actions.

Graphical user interfaces pose other use problems. Although these problems (7) were recognized from the beginning, some have become more pronounced. If graphical representations are to be used in an interface, users must understand the meaning of these representations. Ideally, the meanings would be intuitive. However, abstract concepts and actions are difficult to represent using icons. Anyone who uses current word processing software can attest to the difficulty of comprehending the multitude of icons that appear on the various menu bars. All of these visual representations take space. Text consumes less space on a display than a series of icons. Users are left with less workspace in graphical user interfaces. As functionality increases in software applications, more space on the computer monitor is needed to display iconic representations of the objects and actions available in the application.

Graphic representations are most useful when users easily understand the representations. Design of individual icons is difficult. Moreover, the individual icons can be more easily understood if they are interpreted relative to the presentation of the software application as a whole. Designing large sets of icons to represent a wide range of actions and objects is a complex task. This is often accomplished by using a metaphor or analogy to help user comprehension. The icons used are those that fit the metaphor. For example, file folder icons are used to represent documents on the desktop metaphor. The use of metaphor is discussed in more detail in the section on screen design.

EFFECTS ON SOFTWARE DEVELOPMENT

Developing graphical user interfaces is much more difficult and time-consuming than developing simple command-based interfaces for software applications. Moreover, the amount of code needed for graphical user interfaces is much greater than the code for character-based interfaces. One survey (11) found that in applications for artificial intelligence, around 50% of the code was for the user interface. A survey conducted by Myers and Rosson (12) found that 48% of the code in an application dealt with the user interface. In addition, approximately 50% of the time and cost of software development was devoted to the user interface. Myers (13) lists several reasons why programming graphical user interfaces is difficult. The following are among these reasons:

1. Issues of running multiple processes such as deadlocks and synchronization, must be dealt with. Users can input information that has to be processed while other processing, such as printing files, is going on.
2. Feedback for direct manipulation involves displaying an object to users as they are moving it across the screen. This requires redisplaying an object as many as 60 times per second.
3. The user interface has to be extremely robust. Although there are numerous combinations of commands and objects that a user can select, some valid, some not, the system must never crash but should provide the user with informative feedback.
4. Testing all combinations of actions and objects is not possible. Automated testing is not feasible in many in-

stances because the feedback from the screen has to be observed and factored into the success of the action.

5. Many tools have been provided to help programmers implement the user interface. However, these tools are complex and involve much effort on the part of the programmer to learn.

New disciplines have been incorporated into software development to facilitate development of GUIs. Additional considerations need to be given to the appearance of the user interface, including icon design and screen design. Graphic artists are needed to contribute design knowledge. Cognitive psychologists have contributed knowledge in reducing the complexity of displays and ways to design large systems so that users do not become confused in trying to carry out tasks. Another section of this article discusses issues of physical design and navigation.

ARCHITECTURE OF GRAPHICAL USER INTERFACES

Windowing Systems

Most graphical user interfaces are developed on top of what is called a *windowing system.* A *window* is the term denoting a section of the computer monitor containing the user interface for a particular application. Windowing systems actually consist of two parts (14): the windowing system and the *windowing manager.* The windowing system is used to obtain the input and the output for the application program. The windowing manager handles the presentation of the windows to the user and allows the user to control windows via special window commands. The user can have several applications running at the same time, each of which has its own window for input and output.

All windowing systems provide a basic set of controls and interaction techniques, although the look and feel of these controls are distinct for the different platforms and windowing systems. The window is the main control provided. There are different types of windows, but all windows are used as containers for communication between the application and the end user. Users can close, open, move, and resize windows. Users may also be able to control how multiple windows appear on their screen. Typical options are tiled, overlapping, and cascading.

Current windowing systems include a full set of graphic tools and drawing tools that a programmer can use for displaying output. Programmers can use the functionality provided by the windowing system for input and output. This ensures that the visual representations of application objects appear within the window for that application. Some windowing systems allow application programs to directly draw the output on the user's screen, but this is an exception that should rarely be used.

An application can have multiple windows open simultaneously if there is a need for the user to switch among various pieces of functionality. Only one window can receive input from the user at any one time. The window receiving input is called the *listener* or the *active window.*

Device drivers are library routines that manage input and output devices. This code exists in the windowing system portion. The window management system keeps track of the active window. Window management systems are used to coordinate the input and output between applications and the users of those applications (15). Window management systems are associated with operating systems; that is, a given window management system is built on top of a specific operating system. Examples of window management systems are X Windows, OpenLook™, Motif™, Microsoft Windows, Windows 95™, and MacApp. All window management systems support several shapes and types of windows. The relationship of the various types of windows is defined by the particular system. A popular relationship is that of parent–child windows. Child windows can be displayed only within the parent window. Each child window can have only one parent window. Child windows depend on the parent window. If the parent window is resized or closed, the corresponding adjustment is made to the child window.

A specialized parent–child relationship is the multiple document interface or MDI windows made popular in modern word processing systems. A single toolbar is displayed with the functionality of the word processor, but users may have several documents open simultaneously. Users can switch back and forth between these documents, but the toolbar remains fixed.

Dialog windows or *information windows* are also used by applications to request information from the user or to inform the user about the status of the application program. These windows normally appear on top of the main application window. These windows are one of two types: modal and nonmodal. Modal windows stay on the screen until the user completes an action to dismiss them. The windows are commonly dismissed when the user provides the requested information or acknowledges that the information has been seen. When a modal window appears, the user is unable to input information into another window until the modal window has been dismissed. Nonmodal windows are used to display progress in an action to the user, such as copying a file to a floppy disk. These windows disappear automatically when the action is completed.

The actual presentation of the input and output is not determined by the window management system but by the application. This functionality is managed by a *user interface management system* (UIMS). Myers (16) distinguishes between user interface development systems (UIDS) and UIMS, by noting that the UIMS is associated only with the run time portion of the interface. A UIDS contains tools to help with interface design and interface management.

The first graphical user interfaces were part of the application code, and each application had its own windowing system (16). The application and interface components were implemented in a single unit. This type of architecture makes modification and debugging very difficult, especially if the application is of any size. Porting the application to a different platform is also difficult and usually results in completely rewriting the application. It is difficult to separate the code for the user interface from the code for the application. Therefore, writing a similar application usually involves rewriting the user interface portion of the code also. Another problem with each application having its own windowing system is that there can be little or no overlap in how the GUI looks or behaves from the user's perspective.

Interface Architecture

There are four basic architectural styles (15) that application developers have historically used in designing applications

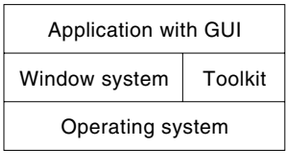

Figure 3. One type of window system architecture: The windows library or client server.

and their graphical user interfaces. The style used depends largely on the size of the application, the window management system and operating system for the platform on which the application is being developed, and the suitability of the application to components provided by different user interface toolkits.

A monolithic architecture is built on top of a windowing system, but all of the user interface management routines are intertwined with the application code. For all but the smallest of applications, this type of architecture is not recommended. It is difficult to debug, and the developer cannot easily reuse the code for the interface. The development task is complex and error-prone. Monolithic architectures are not suited for use with modern window management systems, because this would involve adding application code to the window management architecture.

A client server architecture or toolkit architecture (Fig. 3) separates the components of the user interface from the rest of the application. The client-server relationship in this architecture means that individual workstations are the servers where the code resides and they send data or events in the interface to the client, which is the remote handler for user interfaces.

Toolkits contain procedures that applications call to provide different GUI components (16). Object-oriented toolkits contain classes that define basic interface objects. Then programmers can use these classes to create specific user interface objects in their applications. Although toolkits provide for code reuse, the programmer must become very familiar with the different procedures and classes provided to locate the appropriate code.

X Windows, developed at MIT, was an early standard Unix-based windowing system. Its architecture is based on the client-server architecture. Application programmers use the Xlib toolkit to provide interface components.

The Seeheim (17) architecture separates the application code from the user interface code. Moreover, two separate modules are provided for the user interface. One component deals with the way the objects are physically displayed in the interface. The other component is used to define and manage the dialog between the application and the user. Figure 4 illustrates the various modules and layers in this type of architecture. Modularization is good because it facilitates code reuse. However, this type of separation is difficult to achieve in practice, and this type of architecture also involves much communication between the different components.

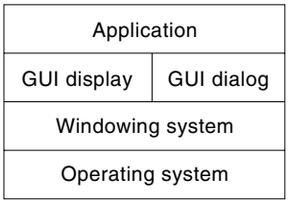

Figure 4. Seeheim architecture.

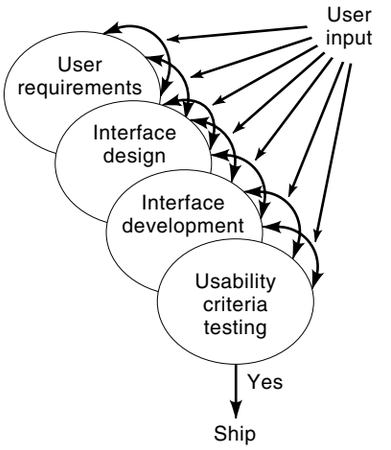

Figure 5. Iterative design process for user interfaces.

Figure 5 illustrates the model view controller architecture. Again, this architecture is based on modularization. Like the Seeheim architecture, the presentation of the interface is separated from the control of the interface components. The presentation is further divided into input and output. The view module is used to present the output. The controller is used to define the input. The model component contains the application algorithms. This architecture reduces the need for communication among the modules because the model and view components can communicate directly if there are changes in the output. The controller and model components communicate to handle input. The SmallTalk programming language provides this type of architecture (18).

Toolkits

Researchers have developed several different types of tools for constructing graphical user interfaces. The toolkit approach is widely used today. Toolkits contain code that can be called from application programs to handle input and output. The pieces of code in these toolkit libraries are called widgets, controls, or glyphs. Toolkits have several advantages for developers and users of the applications. It is reported that tools for GUI development reduce development time considerably (19).

The developers, however, must learn to use the toolkit—what calls to make and what parameters to pass. Tools for interface development provide the programmer with predefined interactive components (user interface toolkits) or help the programmer create interactive components (user interface development systems). End users will see the same type of controls or widgets and will know how to use them, assuming that the same toolkits are used. Examples of visible controls provided by toolkits are text fields, menus, buttons, and scrolling lists. Other controls provided by toolkits are used to position objects on the display.

Toolkits can be open or closed (15). Closed toolkits do not provide mechanisms to programmers for defining new interactive objects. Therefore, all interactive objects have been

thoroughly tested. All applications using a closed toolkit contain the same (but limited) set of interactive objects. An open toolkit supports creation of new interactive objects that are easily inserted in the library. Programmers can create new interactive objects from scratch or modify existing interactive objects. Application programs communicate with these controls through procedural calls.

Object-oriented toolkits allow developers to use inheritance properties and classes to define interactive objects. An object-oriented style is a good fit for interactive techniques because by nature, it is event-driven and asynchronous. MacApp (19) is an example of an object-oriented, application interface framework. This framework implements the Macintosh user interface standard, including menus and windows. What is missing is the contents of those menus and windows. Rather than calling a series of procedures to deal with the user interface controls, the programmer, using an application framework, defines the objects that pertain to the application and installs them in the framework. The base functionality of the interface is there, and the programmer merely installs the specifics that customize the interface to this particular application. Schmucker (19) notes that by using such a tool the amount of code a programmer produces and the development time are reduced by a factor of 4 or 5.

User interface generators (7,20) were developed in research laboratories but have never been widely used in industry. A user interface generator uses a specification of the interface and produces the actual interface for the application. One reason for developing user interface generators was to solve the problem of having to write several interfaces for applications that run on more than one platform. A user interface generator can be separated into two components, the front end, which parses the specification text, and the back end that produces the platform-dependent interface. By substituting a different back end, the user interface generator can produce a user interface for the same application on a different platform. However, learning the specification language for user interface generators is often as difficult as simply writing the interface. Moreover, specifications for direct manipulation interfaces are difficult to produce.

Myers (21) developed the Peridot system in which user interfaces are created by demonstration. His goal was to eliminate work for the interface designer and support creating the look and behavior of the interface by demonstration. Peridot uses direction manipulation and makes some inferences so that the designer does not have to demonstrate the entire user interface.

Communication

The controls supplied by toolkits display data from the application program or are used to input data to the application program. These controls must communicate with the application program and with the window management system. A toolkit control interacts with the application code by a *callback*. The developer defines a routine to be used when the user interacts with an object in the interface. For example, the programmer defines a routine to be executed when the user clicks a button. The programmer defines this button using the code in the toolkit and provides the name of the routine that should be called when the user clicks the button.

Traditionally, application programs handled input and output in a sequential fashion. With graphical user interfaces, the user is free to select any object visible on the display and to apply any allowable action on a selected object. Sequential program design cannot handle this type of interaction. User interactions are handled as events. The code that handles the user interface is a loop that cycles until an event happens. Device drivers for the various devices sense events, such as keyboards, pointing devices, and keypads. These events are placed in an event queue. The user interface loop examines the event queues for the various input and output devices. If an event queue is not empty, the first event in that queue is removed and the code for that particular interface control is called. The application code is invoked by the callback specified in the control. The code that handles an event is called an event handler. The application program also generates events when messages or results must be presented to the user.

Look and Feel

The look and feel of a user interface refer to the presentation and the behavior of the controls. Consistency in look and feel is thought (7) to be beneficial to users of the applications. User interface toolkits help to provide this consistency and a common look and feel to all applications developed with the same toolkit. Look and feel have been defined at the platform or operating systems level.

Standards have been agreed upon to physically represent common interactive objects and for the behavior of those objects. For example, buttons on the Windows 95 platform are gray and have a three-dimensional look. Motif, OpenLook, and X Windows have slightly different variations in look and feel. The descriptions of look and feel for the different platforms are in published references but are also implemented in the toolkits for each of the platforms.

DEVELOPMENT PROCESS

The design and implementation of the graphical user interface cannot be separated from the process of designing and implementing the application. The architecture of the application dictates what can be displayed at what time on the interface and how the user can interact with the various objects. An iterative process of design and implementation (see Fig. 5), in which user feedback is gathered at appropriate points, is now accepted by usability professionals as the best process for ensuring usable applications. This differs from the traditional waterfall method in which a sequential process for collecting requirements and developing specifications was followed by implementation (22). One potential problem in this method is limiting the number of iterations. The software team needs to have a way of evaluating progress and determining when the design or implementation is good enough to proceed. This threshold should be determined prior to the start of design. These threshold values are often termed *usability requirements*. Some specification of the amount of time and number of errors typical end users could make in basic tasks and still perform their job effectively is used as a baseline. Several iterations of any particular step in the process are reasonable. Returning to a previous step and iteration is a more costly activity, but determining usability problems

during the requirements/design and implementation stage is still much less costly than discovering them just prior to shipping.

DESIGNING THE GRAPHICAL USER INTERFACE

Design of graphical user interfaces is a complex process. It involves a multidisciplinary team working together to produce an efficient, functional, and usable product. The terms user-centered and usability engineering are used to describe the need to produce interfaces easy for users to learn and efficient for them to use. Many references exist on how to design usable interfaces and how to incorporate usability evaluations into the development process (23,24).

In addition, usability guidelines have been developed for interface designers to use in constructing and evaluating their interfaces. One early set of guidelines was developed by Smith and Mosier (25). This is a huge set of guidelines, and following them is not a trivial task. Indeed, some of the guidelines are conflicting. Others depend heavily on the circumstances under which the software application is being used: novice or expert users, noisy or quiet environment, room for error or life-critical situations, etc.

Current window management systems have their own set of guidelines to deal with the look and feel of a particular system. For example, Windows (26) outlines the controls for the user interface, including behavior and presentation. Many corporations have another set of guidelines developed on top of a particular window management system to give a unique corporate look and feel. In addition, researchers in human factors are learning more about users' perceptions and cognitive loads in dealing with graphical user interfaces. As new types of input and output devices, new applications, and new interfaces are developed, empirical studies are used to evaluate the new against the old. The human–computer interaction and human factors literature reports these results.

REPRESENTATIONS OF DESIGN

There is a need to represent the design of a user interface. First, there is a need to communicate this design among the members of the team. Everyone needs to understand the design, make contributions to the design, and implement the design. Furthermore, there is a need to evaluate the design. Often this evaluation includes user testing. The section on evaluation discusses this in more detail. Some sort of design representation is needed to get feedback about the design.

State transition diagrams are commonly used to represent sequential interactions (27). State transition diagrams use nodes to represent various states that the interface can have. Arcs connect various nodes and represent the transition to that state, based on some input. There is one special state, called the start state, and one or more end states. A conceptual design can be represented in this fashion for a sequential interactive style. Although state transition diagrams are useful for designers and can be used to communicate among team members, this representation is not suitable for obtaining feedback from users. This sequential representation is also not suitable for asynchronous interactions in GUIs. Jacobs (28) developed a specification language for user interfaces using a set of state transition diagrams. This representation allows specifying GUIs but is only useful for communication among team members.

Object orientation (29) is a natural representation for asynchronous events, and object-oriented programming languages are often used to implement GUIs. However, it is very difficult to use this same environment for designing the interaction because the flow of control in the user interface is actually distributed among many objects. It is necessary to synthesize the code from many objects to understand how the objects in the interface interact, given different input events. This type of representation is good for implementing the design but is inadequate as a communication vehicle among team members. It is also inadequate to communicate the design to end users.

The user action notation (UAN) (30) is another specification language created to describe the interaction of an interface. This type of design representation is useful for communicating the design to team members, assuming that team members take the time to learn the notation. Again, communicating the design to end users via this type of representation is not feasible.

Prototypes are also used to show the look and behavior of an interface or at least portions of an interface. A discussion of prototyping appears in the section on designing a user interface. This type of specification is useful for team member communication and also for communication with end users. Prototyping for interactive systems can be done with varying degrees of completeness (amount of actual system that is covered by the prototyping). Many portions of the application can be omitted from the prototype, depending on why the prototype is being built.

The problem of specifying how an interface looks and behaves has not yet been solved. Using any type of specification language is extremely tedious and impractical for large software applications. Using the demonstration or prototyping methodology is less precise than a written specification and leaves room for different interpretations. Again, it is often impractical to use a prototype to describe the entire interface. In many instances of software development, a combination of representation techniques is used for different reasons and different portions of the user interface. Hartson and Boehm–Davis (31) note that design representation is a major research issue. They conclude that no single representation technique adequately supports all instances of design representation and that research on criteria for selecting the best representation is also needed.

Conceptual Design

Design of the user interface is often described as *conceptual design* followed by *detailed design* (27). In conceptual design, the developer must decide on the actions and objects in the user interface and the interactions between objects. Detailed design involves screen layouts, the appearance of objects on the display, icon design, the wording of messages, navigation between screens in the interface, and so on.

The conceptual design or conceptual model for the user interface describes, at a high level, the actions and objects that will be presented to the end user. This design includes specifying which actions can be applied to which objects under which conditions and the results of doing so. The conceptual model represented in the user interface explains to the user

what is happening in the application software. The actions that the user takes are based on comprehending the application based on this representation.

Conceptual design includes considering the order in which a user will carry out tasks in the application. It is important to understand this so that the appropriate objects and actions can be presented to the user at the correct time. In the detailed design, the order of the information and the pieces of information that appear together are determined.

Metaphors are one way of explaining an application to an end user. The most familiar graphical user interface metaphor is the desktop. Graphical representations of objects (files, programs, trash cans) are displayed to the user and are supposed to function like those objects on the user's physical desktop. Files can be opened and read. New files can be created. Files in which the user is no longer interested can be thrown away in the trash container. Emptying the trash container implies that the user can no longer retrieve those files. Clocks, in and out boxes, calendars, and appointment books are often found on these electronic desktops as well. Because users know how the physical objects behave, they use this knowledge to understand how the electronic objects behave. Problems occur when the physical and electronic objects behave differently. Users can eject a disk from the floppy drive on a Macintosh by dragging the icon for that file to the trash containers. This behavior is contrary to the way that trash containers actually behave, and therefore, users had to learn this behavior. In selecting metaphors for the user interface, it is essential that the user is informed of such inconsistencies. Computer systems are very powerful and allow us to do many tasks not possible without this computational power. Many metaphors are incapable of representing the tasks that computer applications can perform. This first step in design is extremely important, and the selection of an overall representation should be made only after the designers have ensured that the representation is understood by the actual end users of the system. Design heuristics (32,33) can be used to evaluate alternative conceptual designs.

Dialog Design

A graphical user interface contains objects and actions that can be performed on those objects. Users must select those objects and appropriate actions and are then given information or feedback about what has happened. The details of selecting the objects and actions and the feedback that results must be designed. This is often termed *dialog design* (15).

Dialog design specifies the messages that the user can give to objects in the application and how the objects respond to those messages. Some parts of the dialog are order-specific, that is, the user can select an action only after an object has been selected. By using toolkits that predefine standard interface objects on the various platforms, designers already have some predefined dialogs. The way in which users select objects and actions is predefined to some extent. Selection by a pointing device and selection by keyboard characters are two basic predefined dialogs.

Actions can include changing an object, deleting an object, copying an object, and undoing a previous action. Dialog designers need to consider which actions users can cancel or reverse, with what degree of difficulty, and how many actions can be reversed.

Selecting command objects can result in the display of new screens of information or the display of new objects and actions for the user. Moving between different screens of information in a graphical user interface is termed navigation. Navigation design should be presented to the users so that they understand when they will be moved to a different screen and how to get back to the previous screen if necessary.

APPEARANCE

Appearance of the graphical objects in the interface is also an important consideration. The overall screen design and the design of individual object representations has to be carefully done to aid the user in understanding how the application functions. Graphical designers are often employed by software development companies to work on the graphical user interface teams and produce high quality graphics. Horton (34), Galitz (35), and Tullis (36) discuss the design and perception of icons by users.

As with screen design, icons should first be designed in black and white. Color is added later for appeal but should not figure into the basic design. Icons should also be designed in sets, not individually. Although icons for different actions and concepts may look quite different, there should be a family resemblance to icons associated with the same application. As with other portions of the interface, icons need to be tested by users to ensure that users can recognize and differentiate between them.

Although there is much room for creativity in this area, certain basic icons have become associated with actions and objects in the desktop metaphor. Most current users of graphical user interfaces know what trash cans, file folders, scissors, and floppy disk icons represent. However, new computer applications are appearing and demand new metaphors and icons.

Screen Design

The design of screens to present information and to convey functionality to users is extremely important. Good screen design significantly reduces the amount of time it takes users to locate information. Quickly locating the pertinent information can be a critical issue in complex displays, such as air traffic control. Good screen design also helps prevent user errors in the input of information. Users can use context to help interpret information they are unsure of, assuming that information on input screens is organized into logical chunks. Screen design research did not originate with graphical user interfaces. Character-based interfaces also provide many challenges for screen design. However, graphical user interfaces rely much more on the visual information processing capabilities of users and present numerous new challenges to visual designers of user interfaces. Fortunately, research from perceptual psychology and studies of human factors have been used to develop some basic guidelines about screen design. The following are some of the issues to be considered in screen design (36):

- the amount of information to display
- how to group information

- the placement of the information in the display
- the best representation of the information displayed

Screen design details (37) include alignment of fields on screens, titles for screens, ordering of fields on screens, ordering of menus and presentation of menus, indicators of optional fields, and indicators of the format for input data. The studies and guidelines from alphanumeric interfaces are still appropriate for many questions about screen design in graphical user interfaces. Screen design issues for windowing systems include the amount of information to include in one window, providing feedback to the user about the window appearing in response to a user or application action, and arrangement of windows (35,37).

EVALUATION OF GRAPHICAL USER INTERFACES

Two types of evaluations are performed on graphical user interfaces; quality or assurance testing and usability testing. Quality testing consists of three basic steps (38): running a program with a set of inputs, observing the effects during execution, and examining output to determine correctness. In addition to actually executing code, other types of testing are done. During coding, teams of programmers conduct code walkthroughs looking for errors. Analysis is also done by programs to detect certain types of common errors. Units of code are tested dynamically in what is called *white box testing*. The internal code is traced during this type of testing to determine the paths used. Whole system testing is usually *black box testing*. In this type of testing, the inputs are given, and the outputs are observed with no attempt to understand the code execution.

There are several problems with attempting to completely test a graphical user interface. First, as objects and actions can be selected in any order and combination, all but the smallest application produces an extremely large number of possibilities. Secondly, writing specifications for what should happen in all cases is difficult. Therefore, it is difficult to know what should happen to compare it with what actually does happen. If specifications were produced for a user interface, then some of the testing of how the interface works could be done via proofs or it could be automated. However, much of what happens in a graphical user interface involves feedback to the user. Therefore, testing GUIs involves much more than just analyzing the final output. The actual behavior of the screen objects during execution must also be observed. Observing behavior does not lend itself well to automation.

Capture and playback tools are sometimes used to create automatic test scripts for user interfaces. These are better suited to creating test scripts representative of expert user behavior because experts follow a more predictable path. Genetic algorithms are being investigated as a way to generate user events representative of novice behavior (39).

User interfaces are also evaluated to see how usable they are; that is, can the intended users of the system easily learn and use the interface to complete the necessary tasks? Originally, the users of computers were technically trained, and using a computer represented a large portion of their job responsibilities. Currently a large number of nontechnical users depend on computer applications to do their work. The usability of these applications can have a large effect on training costs, productivity costs, job satisfaction, and employee turnover (40). Usability evaluation during software development can often more than pay for itself in both money and time spent.

Much of the literature in human–computer interaction today concerns user-centered design and usability testing techniques (23,24,41,42). Most software developers now recognize that usability must be considered throughout the design and development of the software. The focus is on frequent checks with representatives of the intended user population to verify design decisions. Participatory design is a technique originating in Scandinavia (43) in which several representative users are on the software design team. They can work with the software developers and engineers during the development process to guide the design so that the software and interface are well suited to the users and their tasks.

Usability engineers use other techniques during software development to do usability evaluations. Prototypes are often developed and shown to representative users. The users are asked to complete some set of tasks with the prototype. Any problems that users have in doing the tasks along with their reactions to the prototype are used as the basis for changing the design.

Heuristic evaluations are also used by usability engineers to evaluate user interfaces with respect to a set of usability heuristics or principles known to cause user confusion if violated (44). Studies (45–48) have compared the effectiveness of doing heuristic evaluations and user testing and also noting differences in heuristic evaluations depending on the expertise level of the evaluators. In general, trained usability experts find more errors in performing heuristic evaluations than nonexperts. Although heuristic evaluations are relatively quick and inexpensive, they have the drawback that they only predict problems that users may have and they tend to find numerous potential problems. User testing finds the more severe usability problems.

User interfaces are sometimes evaluated with respect to published guidelines. However, guidelines are more suited for use in guiding interface design. Smith and Mosier (25) developed a set of 944 user interface guidelines. Many of these guidelines are based on principles from cognitive and perceptual psychology. Other guidelines have been developed through experimental studies comparing different techniques of interaction or different representations to determine which results in better user performance. These are general guidelines that apply to user interfaces on all platforms. There are also platform-dependent user interface guidelines. These guidelines determine the look and feel of user interfaces on different platforms. For example, these guidelines specify if the buttons in the user interface have a three-dimensional look, whether the background color is gray or white, and whether pop-up menus are used for certain types of functionality.

Formal evaluations of user interfaces, such as the GOMS model (Goals, Operators, Methods, Selection Rules) (49), can be used to compare different interactive techniques with respect to the number of keystrokes required to accomplish them. These formal evaluations can also be used to compare portions of the interface for consistency by comparing the mechanisms for carrying out tasks and subtasks. GOMS provides a means for coding the keystrokes and mental operations that a user must do to invoke a certain action on a cer-

tain object. This technique is limited in use due to the expertise needed and the amount of work involved to model a large portion of a user interface.

FUTURE OF GRAPHICAL USER INTERFACES

There are many new and interesting directions currently being developed in the field of user interfaces (50). The growing spread of the World Wide Web is now leading to the availability of information seeking applications and also to interactive applications. Using the World Wide Web as the delivery mechanism solves many of the problems of platform-dependent applications. The graphical user interface resides in the browser window. New programming languages, such as Java, are springing up, and new toolkits are being developed to allow programmers of these applications to use standard controls. The use of audio, video, and animation in these applications is adding to the complexity of programming these interfaces. Dynamically created Web pages or user interfaces can now be generated depending on the identity of the user. For example, the language that you see on the Web page can be different depending on the country from which you are requesting access.

Collaborative systems are another type of application in which the graphical user interface is extremely important. Users of computer-supported work cooperative (CSCW) systems need to interact with objects, such as documents and calendars, and also with each other. Interfaces for CSCW applications need to provide functionality for people interacting with each other and for interactions with documents and other objects. Interoperability is also a crucial aspect of CSCW systems. All users do not have the same platforms and the same capabilities but must be able to view the same information and interact.

Direction manipulation techniques are being employed in *virtual environments* or *virtual worlds* (51). Three-dimensional techniques are used to display worlds in which users can move around and interact with objects much the same as they interact with similar objects in the real world. Direct manipulative techniques are also being used in software interfaces for remote devices. Applications, such as telemedicine, are employing these techniques. The World Wide Web is also being used to allow remote users to control physical devices for experimentation.

In immersion virtual reality, the user wears a head-mounted display so that it appears as if the interface surrounds the user. A data glove is often used to interact in this environment. The user makes gestures while wearing the glove. These gestures are recognized and interpreted by the system into actions, such as moving to another area of the environment or manipulating an object in the environment. In nonimmersive virtual reality, the user interacts with a three-dimensional world displayed on the computer monitor. Interaction is accomplished by using a standard mouse or trackball input device. Users get a sense of being in the environment as they can change their view and zoom in on objects.

Collaborative virtual environments take these interfaces a step further and allow multiple users to interact simultaneously. Users select representations for themselves and move these representations around the virtual world, interacting with others in the world as well as with objects in the worlds. Researchers are currently experimenting with new forms of interactions and new application domains for virtual worlds.

Object-oriented application frameworks (52) are another direction being pursued to facilitate the development of different types of applications. These frameworks are extensions of the frameworks used by developers of graphical user interfaces but have been developed for complex domains, such as telecommunications and real-time avionics. The development of remote or distributed computing is facilitated in these new frameworks with the provision of code to deal with communication between remote and local objects.

Personal digital assistants (PDA) are small, mobile computer devices that are becoming popular. GUIs designed for PDAs are particularly challenging because of the reduced space available for the display. Users can use current PDAs to view Web pages, read e-mail, download files from their desktop machines, compose documents, and keep track of appointments and phone numbers.

New input and output devices are needed for interacting with the new application techniques and interfaces. Spoken language for input is improving in the past few years. Sensors of body movements are being used in virtual reality applications. Prototyping languages have been developed for spoken language interfaces. Work continues in multimodal interfaces that allow users access to the type or types of input most natural for the tasks being done. As new types of interactions are developed, toolkits for developing interfaces for applications using these interactive devices continue to be developed in the research labs.

Four problems are currently being addressed by user interface researchers. The first problem is bandwidth. New applications, such as collaborative virtual environments and digital libraries of multimedia data, require large amounts of bandwidth not currently available to everyone. Techniques for delivering high-quality services at reduced bandwidths must be pursued. A second problem is dealing with large quantities of information. Researchers are looking at visualization techniques to help users view, explore, and use large amounts of data. Accessibility is the third issue. Researchers are looking into techniques to allow all individuals, regardless of physical capabilities or limitations imposed by hardware and software capabilities, to have access to the same information. Human–computer interaction considerations are also being addressed. Researchers are exploring ways in which user interactions can be leveraged to provide more effective user experiences than technological advances alone.

BIBLIOGRAPHY

1. I. E. Sutherland, Sketchpad: A man-machine graphical communication system, *AFIPS Spring Joint Comput. Conf.,* 1963, pp. 329–346.
2. C. Smith et al., Designing the star user interface, *BYTE,* **7** (4): 242–282, 1982.
3. L. Bass and J. Coutaz, *Developing Software for the User Interface,* Reading, MA: Addison-Wesley, 1991.
4. B. Shneiderman, Direct manipulation: A step beyond programming languages, *IEEE Comput.,* **16** (8): 57–69, 1983.

5. J. Greenstein and L. Arnaut, Input Devices. In Martin Helander (ed.), *Handbook of Human–Computer Interaction,* Amsterdam: North Holland, 1988, pp. 495–536.
6. J. R. Brown and S. Cunningham, *Programming the User Interface: Principles and Examples,* New York: Wiley, 1989.
7. B. Shneiderman, *Designing the User Interface,* Reading, MA: Addison-Wesley, 1987.
8. S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction,* Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.
9. T. Roberts and T. Moran, Evaluation of Text Editors. In *Proc. Human Factors in Comput. Syst.,* 136–141, 1982.
10. B. Shneiderman and S. Margono, A study of file manipulation by novices using commands vs. direct manipulation, *Proc. 26th Annu. Tech. Symp. Washington D.C. Chapt. ACM,* Gaithersburg, MD: NBS, 1987.
11. D. B. Bobrow, S. Mittal, and M. J. Stefik, Expert Systems: Perils and Promise, *Commun. ACM,* 880–894, 1986.
12. B. A. Myers and M. Rosson, Survey on user interface programming, *Proc. CHI'92 Conf. Human Factors Comput. Syst.,* 1992 New York: ACM, pp. 195–202.
13. B. A. Myers, State of the Art in User Interface Software Tools. In H. R. Hartson and D. Hix (eds.), *Advances in Human–Computer Interaction,* Norwood, NJ: Ablex, 1998, Vol. 4, pp. 110–150.
14. B. A. Myers, State of the art in user interface software tools. In R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg (eds.), *Readings in Human–Computer Interaction: Toward the Year 2000,* San Francisco, CA: Morgan Kaufman, 1995.
15. J. Larson, *Interactive Software: Tools for Building Interactive User Interfaces,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
16. B. A. Myer, User-interface tools: Introduction and survey, *IEEE Softw.,* **6** (1): 15–23, 1989.
17. M. Green, Report on Dialogue Specification Tools. In Gunther E. Pfaff (ed.), User interface Management Systems, *Proc. Workshop User Interface Manage. Syst.,* Seeheim, FRG, November 1–3, 1983, Berlin: Springer-Verlag, pp. 9–20.
18. T. Kaehler and D. Patterson, A small taste of smalltalk, *BYTE,* **11** (8): 145–159, 1986.
19. K. J. Schmucker, MacApp, An Application Framework, *BYTE,* **11** (8): 189–194, 1986.
20. D. R. Oslsen, Jr. et al., Research directions for user interface software, *Behaviour Inf. Technol.,* **12** (2): 80–97, 1993.
21. B. A. Myers, *Creating User Interfaces by Demonstration,* San Diego, CA: Academic Press, 1988.
22. B. W. Boehm, A spiral model of software development and enhancement, *IEEE Comput.,* **21** (2): 61–72, 1988.
23. J. S. Dumas and J. C. Redish, *A Practical Guide to Usability Testing,* Norwood, NJ: Ablex, 1993.
24. J. Nielsen, *Usability Engineering,* San Diego, CA: Academic Press, 1993.
25. S. L. Smith and J. N. Mosier, *Design Guidelines for Designing User Interface Software,* Tech. Rep. MTR-100090, Bedford, MA: The MITRE Corporation, 1986.
26. *The Windows Interface Guidelines for Software Design,* Redmond, WA: Microsoft Press, 1995.
27. D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process,* New York: Wiley, 1993.
28. R. J. K. Jacob, A specification language for direct manipulation user interfaces. *ACM Trans. Graphics,* **5** (4): 283–317, 1986.
29. L. Sibert, W. D. Hurly, and T. W. Bleser, Design and implementation of an object-oriented user interface management system. In H. R. Hartson and D. Hix (eds.), *Advances in Human–Computer Interaction,* Norwood, NJ: Ablex, 1988, Vol. 2, pp. 175–213.
30. H. R. Hartson, A. C. Siochi, and D. Hix, The UAN: A user-oriented representation for direct manipulation interface designs, *ACM Trans. Inf. Syst.,* **8** (3): 181–203, 1990.
31. H. R. Hartson and D. Boehm-Davis, UI development processes and methodologies, *Behavior Inf. Technol.,* **12** (2): 98–114, 1993.
32. W. M. Newman and M. G. Lamming, *Interactive System Design,* Wokingham, England: Addison-Wesley, 1995.
33. D. J. Mayhew, *Principles and Guidelines in Software User Interface Design,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
34. W. Horton, *The Icon Book: Visual Symbols for Computer Systems and Documentation,* New York: Wiley, 1994.
35. W. O. Galitz, *It's Time to Clean Your Windows: Designing GUIS that Work,* New York: Wiley-QED Publ., 1994.
36. T. Tullis, Screen Design. In Martin Helander (ed.), *Handbook of Human-Computer Interaction,* Amsterdam: North Holland, 1988, pp. 377–411.
37. W. O. Galitz, *Handbook of Screen Format Design,* Wellesley, MA: QED Information Sciences, 1989.
38. E. F. Miller, Software testing technology: An overview. In C. R. Vick and C. V. Ramamoorthy (eds.), *Handbook of Softw. Eng.,* New York: Van Nostrand Reinhold, 1984, pp. 359–379.
39. D. Kasik and H. George, Toward automatic generation of novice user test scripts, *Proc. CHI'96 Conf. Human Factors Comp. Syst.,* New York: ACM 1996, pp. 244–251.
40. R. G. Bias and D. J. Mayhew (eds.), *Cost Justifying Usability,* London: Academic Press, 1994.
41. J. Preece, *Human Computer Interaction,* Wokingham, England: Addison-Wesley, 1994.
42. G. Lindgaard, *Usability Testing and System Evaluation,* London: Chapman & Hall Computing, 1994.
43. D. Schuler and A. Namioka (eds.), *Participatory Design: Principles and Practices,* Hillsdale, NJ: Lawrence Erlbaum, 1993.
44. J. Nielsen and R. L. Mack (eds.), *Usability Inspection Methods,* New York: Wiley, 1994.
45. R. Jeffries et al., User interface evaluation in the real world: A comparison of four techniques, *Proc. CHI'91 Conf. Human Factors Comput. Syst.,* New York: ACM 1991, pp. 119–124.
46. B. E. John and S. J. Marks, Tracking the effectiveness of usability evaluation methods, *Behavior Inf. Technol.,* **16** (4/5): 188–202, 1997.
47. T. S. Tullis, Is user interface design just common sense? In G. Salvendy and M. J. Smith (eds.), Human-computer interaction: Software and hardware interfaces, *Proc 5th Int. Conf. Human-Comput. Interaction,* (HCI International '93), Amsterdam, The Netherlands: Elsevier, 1993, pp. 9–14.
48. J. Nielsen and V. L. Phillips, Estimating the relative usability of two interfaces: Heuristics, formal, and empirical methods compared, *Proc. INTERCHI'93 Conf. Human Factors Comput. Syst.* New York: ACM 1993, pp. 214–221.
49. S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction,* Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.
50. R. J. K. Jacobs et al., UIST '007: Where will we be ten years from now? *UIST'97 Proc. ACM Symp. on User Interface Softw. Technol.,* New York: ACM 1997, pp. 115–118.
51. B. Shneiderman, *Designing the User Interface,* Reading, MA: Addison-Wesley, 1998.
52. M. E. Fayad and D. C. Schmidt, Object-oriented application frameworks, *Commun. ACM,* **40** (10): 32–38, 1997.

JEAN C. SCHOLTZ
National Institute of Standards and Technology (NIST)

GRAPHICAL USER INTERFACES. See SOFTWARE PROTOTYPING.

GRAPHICS, ANIMATION. See COMPUTER ANIMATION.

GRAPHICS, BUSINESS. See BUSINESS GRAPHICS.

GRAPHICS, COLOR. See COLOR GRAPHICS.

GRAPHICS HARDWARE. See RASTER GRAPHICS ARCHITECTURES.