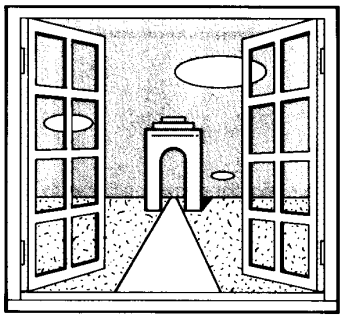


User-Interface Tools: Introduction and Survey

Brad A. Myers, *Carnegie Mellon University*

A good interface is vital, yet very hard to produce. User-interface tools seek to ease the burden. This article surveys the state of the art.

Creating good user interfaces for software is very difficult. There are no guidelines or techniques that guarantee the software will be easy to use, and software implementers have generally proven to be poor at providing interfaces that people like. Consequently, interface software must often be prototyped and modified repeatedly.

Interface software is inherently difficult to write because frequently it must control many devices, each of which may be sending streams of input events asynchronously. Also, interfaces typically have stringent performance requirements to ensure that there is no perceived lag between a user's actions and the system's response.

Therefore, there is great interest in developing tools to help design and implement interfaces. This article explains the advantages of user-interface tools, surveys the state of the art, and details the current systems' shortcomings that must be overcome.

The problem

Interface software is often large, complex, and difficult to debug and modify. An application's interface can account for a significant fraction of the code. Surveys of artificial-intelligence applications, for example, report that 40 to 50 percent of the code and runtime memory are devoted to interface aspects.[1]

As interfaces become easier to use, they become harder to create. The easy-to-use, direct-manipulation interfaces popular on many modern systems are among the most difficult to implement. These interfaces let the user operate directly on objects that are visible on the screen, performing rapid, reversible, incremental actions.[2]

Direct-manipulation interfaces are difficult to create because they often provide elaborate graphics, many ways to give the same command, many asynchronous input devices, a mode-free interface (the user can give any command at virtually any

0704-7459/89/0001/0015/$01.00 © 1989 IEEE

time), and rapid *semantic feedback*. With semantic feedback, the appropriate response to user actions is based on specialized information about the objects in the program. For example, in the Apple Macintosh Finder, icons highlight when another icon is dragged over them if they perform semantically meaningful operations on the icon being dragged.

Such interfaces are not only difficult to create, but there are no design strategies that guarantee the resulting interface will be easy to learn or easy to use. The only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments.[3]

This method, called iterative design, has been used to create some of today's best interfaces. For example, a mail system with a conventional textual command interface was tested with users and modified iteratively. In the final version, without any instruction, 76 percent of the commands that novices generated performed the expected operation, compared with 7 percent for the initial version.[4]

Advantages

Many tools have been created to make interfaces cheaper and easier to design and implement; some have been very successful. For example, Apple's MacApp has been reported to reduce development time by a factor of four or five.[5]

Using user-interface tools has several advantages in two main areas:

1. It results in better interfaces:

- Designs can be rapidly prototyped and implemented, possibly before the application code is written.
- It is easier to incorporate changes discovered through user testing because the interface is easier to modify.
- One application can have many interfaces.
- More effort can be expended on the user-interface tools than may be practical on any single interface because the tools will be used again and again.
- Different applications will have more consistent interfaces because they have been created with the same user-interface tools.
- It easier to investigate different styles for an interface, thereby providing a unique look and feel for a program.
- It is easier for many specialists to be involved in designing the interface, including graphic artists, cognitive psychologists, and human factors specialists. Professional interface designers, who may not be programmers, may be in charge of the overall design.

2. The interface code will be easier to create and more economical to maintain:

- The code will be better structured and more modular because it has been separated from the application. This lets the designer change the interface without affecting the application, and it lets the programmer change the application without affecting the interface.
- The code will be more reusable because the user-interface tools incorporate common parts.
- The reliability of the interface is higher because the code is created automatically from a higher level specification.
- Interface specifications can be represented, validated, and evaluated more easily.
- Device dependencies are isolated in the user-interface tool, so it is easier to port an application to different environments.

The only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments.

Definitions

User-interface tools come in two general forms: *user-interface toolkits* and *user-interface development systems.*

A user-interface toolkit is a library of interaction techniques, where an interaction technique is a way of using a physical input device (such as mouse, keyboard, tablet, or rotary knob) to input a value (such as command, number, percent, location, or name), along with the feedback that appears on the screen.

Examples of interaction techniques are menus, graphical scroll bars, and on-screen buttons operated with the mouse. A programmer uses a user-interface toolkit by writing code to invoke and organize the interaction techniques. Toolkits do not provide much support for the design of interfaces or for the specification of sequencing and dialogue control.

A user-interface development system is an integrated set of tools that help programmers create and manage many aspects of interfaces. These systems are usually called user-interface management systems,[6] but I prefer to call them UIDSs because "UIMS" is associated only with the runtime portion of the interface (rather than the part used at design time) or with systems that include an explicit dialogue-control component. UIDSs help both with designing and implementing the interface and so encompass a broader class of programs.

There are four classes of people involved with any UIDS. One is the designer of the UIDS, whom I call the UIDS creator. Another is the designer of an interface who will use a UIDS. This person may be a programmer or a graphic artist and is called the designer. A third person is the programmer who creates the application program that is connected to the interface created by the designer; I call this person the application programmer. The final person is the user.

Although I describe each role as if it was filled by a different person, there may be many people in each role and one person may perform many roles.

The box on p. 22 summarizes the commercial and experimental UIDSs that are described below. User-interface tools can also be classified by how they communicate with the application at runtime, as the box on p. 19 describes.

User-interface toolkits

Most window systems and UIDSs come with a toolkit that contains routines application programs can use. These typically include several types of menus, scroll bars, and so on. There are two kinds of toolkits. The most conventional is a collection of procedures that can be called by application programs. An example is the Macintosh Toolbox. The other kind uses an object-oriented programming style with

inheritance, which makes it easier for the designer to customize the interaction techniques. An example is the X.11 Toolkit for the X Window System manager.

Two toolkits, Grow and Coral, add constraints to object-oriented toolkits. Constraints let the designer specify relationships among objects and have the relationships maintained by the system. For example, the designer can specify that a line is connected to two rectangles, and the system will automatically move the line whenever either rectangle is moved. With all toolkits, the designer writes programs in a conventional programming language to control the interface.

The disadvantages of using toolkits are that they provide limited interaction styles and are often expensive to create and difficult to use. A toolkit typically includes hundreds of procedures that implement many interaction techniques. It is often not clear how to use the procedures to create a desired interface.

User-interface development systems

The problems with toolkits have led to the creation of UIDSs. In fact, Apple created the MacApp UIDS after it found that people were having difficulty using the Macintosh Toolbox.

UIDSs help the designer combine and sequence interaction techniques. Some UIDSs help the designer create toolkits; others help the designer lay out and use predefined toolkit items.

Function. A comprehensive UIDS handles all aspects of the interface, which includes all visible parts of the display and all aspects of the dialogue between the user and the application. The UIDS should

- handle the mouse and other input devices,
- validate user inputs,
- handle user errors,
- process user-specified aborts and undos,
- provide appropriate feedback to show that input has been received,
- provide help and prompts,
- update the display when application data changes,
- notify the application when the user updates application data,
- handle field scrolling and editing,
- insulate the application from screen-management functions,
- automatically evaluate the interface and propose improvements, or at least provide information to help the designer evaluate the interface, and
- let the user customize the interface.

Components. To do this, UIDSs may contain

- a toolkit,
- a dialogue-control component that handles event sequencing and interaction techniques,
- a programming framework that helps guide and structure the interface code and application semantics,
- a mouse-based layout editor to specify the location of graphical elements, and
- an analysis component that may either evaluate the interface automatically based on rules and guidelines or save information such as keystrokes for later evaluation by the designer.

UIDSs help the designer combine and sequence interaction techniques. Some UIDSs help designers create toolkits; others help designers use toolkit items.

One important way to classify UIDSs is by how they let the designer specify the interface. As the box on p. 22 shows, some UIDSs use special-purpose languages, some use graphical specifications, and others generate the interface automatically from an application's function specification. Of course, some UIDSs use different techniques to specify different parts of the interface. I have classified such systems according to their predominant or most interesting feature.

Language-based

In most UIDSs, the designer specifies the interface with a special-purpose language. This language may have many forms: menu networks, state-transition diagrams, context-free grammars, event languages, declarative languages, or object-oriented languages. In most systems, the designer uses the language to specify the interface's syntax — the legal sequence of input and output actions. Green[7] has written an extensive comparison of grammars, state-transition diagrams, and event languages.

Menu networks. One of the simplest forms of UIDS supports a menu hierarchy or network. When you select an option on one menu, another menu appears. The Tiger UIDS supports a sophisticated menu network that supports skipping levels and aborting. Many hypertext systems, such as Apple's Hypercard program, could also be considered UIDSs that manage menu networks.

State-transition diagrams. Because much of what an interface does involves handling a sequence of input events, it is natural to think of using a state-transition network to code the interface.

A transition network is a set of states. Arcs out of each state are labeled with the input token that will cause a transition to the state at the other end of the arc. In addition to input tokens, the arcs in some systems are labeled with application procedures to be called and output to be displayed.

Apparently, the first UIDS was implemented by Newman in 1968. It used finite-state machines to implement a simple UIDS that handled textual input only. Many of the assumptions made and techniques used in modern systems were present in Newman's: different languages for defining the interface and the semantics, table-driven syntax analysis, and device independence.

A proponent of state-transition diagrams is Jacob, who claims that they are better than context-free grammars because they show an explicit time sequence. Figure 1 shows a diagram created with Jacob's State-Diagram Interpreter system. Because the arcs can have recursive calls to other diagrams, they are called recursive-transition networks.

Rapid/USE is a state-transition system that is similar to Jacob's except that it has

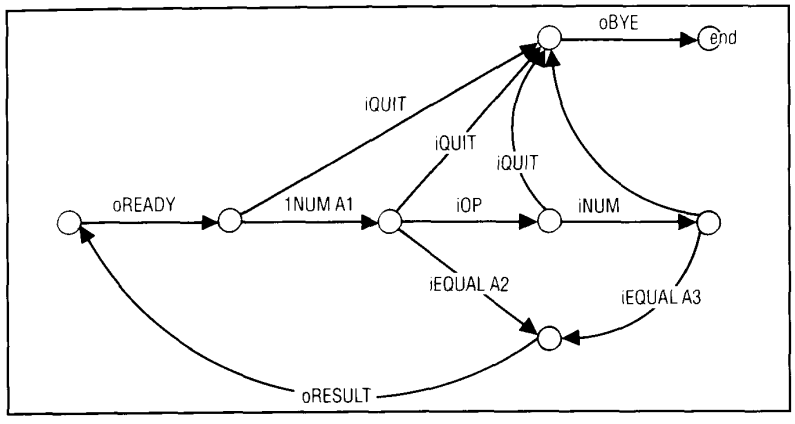


Figure 1. A state-transition diagram from Jacob's State-Diagram Interpreter of a simple desk calculator.

more powerful output primitives. Rapid/USE is just a small part of a large software-engineering system.

The problems with this approach are that the connections between the interface and application are made through a plethora of global variables, and all states must have explicit arcs for all possible erroneous input and all universal commands such as Help and Undo.

State-transition UIDSs are most useful when the interface must do a lot of syntactic parsing or has many modes (each state is really a mode). However, most highly interactive systems are largely mode-free, so the user has many choices at every point. This requires many arcs out of each state, so state-transition UIDSs have not been successful for these interfaces.

Another problem with state-transition networks is that they cannot handle interfaces that let the user operate on multiple objects concurrently (possibly using multiple input devices). Also, the diagrams get very confusing when used for large interfaces — they become a maze of wires, with arcs that are hard to follow as they go off the page or screen.

Jacob has invented a new formalism that recognizes these problems but tries to retain the clarity of state-transition diagrams. His new formalism combines state-transition diagrams with a form of event language. It lets multiple diagrams be active at the same time and transfers the control flow from one to another in a coroutine fashion. This Interaction Objects system can create some forms of direct-manipulation interfaces.

Context-free grammars. Most grammar-based systems are based on parser generators. For example, the designer might specify the interface syntax in Backus-Naur form. Grammar-based systems are good for textual command languages, but they have mostly failed for graphics programs, for reasons similar to those given for state-transition diagrams.

Syngraph generates interface programs in Pascal from a description written in a formal grammar using an extended Backus-Naur form. It handles prompting, echoing, and errors. It provides menus, textual input, and a few predefined interaction devices (locator, valuator, and pick) with some limited tracking.

Syngraph can deal with semantic error recovery, Cancel and Undo at the semantic level, and deciding what to select when multiple items are on the screen at the pick position. However, Syngraph does not provide semantic feedback or defaults because there is no way for application routines to affect the parsing.

Event languages. With event languages, input tokens are considered to be events that are sent immediately to event handlers. These handlers can cause output events, change the internal state of the system (which might enable other event handlers), and call application routines.

Algae uses an event language that is an extension of Pascal. The designer programs the interface as a set of small event handlers, which Algae compiles into conventional code.

Sassafras uses a similar idea but with an entirely different syntax. It also adds local variables called flags to help specify control flow. Sassafras is especially well-suited for interfaces that use multiple input devices concurrently (also called multithreaded dialogues). It can also support direct-manipulation interfaces because it promotes efficient, frequent communication between the interaction techniques and the application program.

Squeak, a textual language for programming mouse-based interfaces, exploits concurrency. Squeak's processes are similar to event handlers and the messages sent its processes are similar to events. Squeak supports many concurrently active input devices. The primitive input events are mouse-button transitions, keyboard key presses, incremental movements of the mouse or other devices, and clock time-outs. Squeak compiles the program into a sequential state machine. Although it provides a compact notation for specifying complex, time-dependent interfaces, Squeak is unfortunately a fairly difficult language to write in.

The UIDSs in this category are explicitly designed to handle multiple processes. Research has shown that people can be more effective when they operate multiple input devices concurrently.[8] It is also often easier to use multiple processes to program multiple interactions where the user can choose which interaction to use.

The disadvantage of event languages is that it is often very difficult to create correct code because the control flow is not localized. Small changes in one part of the program can affect many other parts. It is also often difficult for the designer to understand the code once it gets large.

Declarative languages. Declarative languages state what should happen rather than how to make it happen. The interfaces supported by declarative languages are usually form-based: The user types text into fields or selects options with menus or buttons. There are also often graphical output areas for use by applications. The application is connected to the interface through global variables that are set and accessed by both the application and interface. Cousin and Domain/Dialogue are systems in this category. Apollo has created a new UIDS based on Domain/Dialogue called Open Dialogue, which runs on the X Window System manager, so it is portable.

The advantage of declarative language-based UIDSs is that they free designers from worrying about the sequence of events, so they can concentrate on the in-

formation that is passed back and forth. The disadvantage is that they support only form-based interfaces — others must be hand-coded in the graphical areas provided to applications. They also provide only preprogrammed, fixed kinds of interactions. For example, these systems provide no support for such things as dragging graphical objects, rubberbanding lines, or drawing graphical objects.

Object-oriented languages. These systems are an important new class of UIDS. They provide an object-oriented framework in which the designer programs the interface. Typically, there are high-level classes that handle default behavior. The designer specializes these classes to deal with behavior specific to the interface, using the inheritance mechanism built into object-oriented languages.

MacApp is programmed in Object Pascal. GWUIMS uses object-oriented Lisp and provides a classification of interface operations and objects that fit into each class. Higgens adds a structured-data description that supports Undo and Redo and lets the UIDS automatically manage the recalculation and redisplay of objects intelligently.

These systems can handle highly interactive, direct-manipulation interfaces because there is a computational link between the input and the output that the application can modify to provide semantic processing. Although these systems make it much easier to create interfaces, they are programming environments and as such are inaccessible to nonprogrammers. Still, object-oriented toolkits and UIDSs show a lot of promise.

Graphical specification

Graphical UIDSs let you define the interface, at least partially, by placing objects on the screen with a mouse. The philosophy behind this approach is that, because the visual presentation of the interface is one of its most important aspects, a graphical tool is the most appropriate way to specify that presentation.

This technique is usually much easier for the designer to use. Some of these systems, including Menulay, Trillium, and Peridot, can be used by nonprogrammers. Three disadvantages of this technique are that the UIDS itself is more complicated to build, it supports the creation of a limited range of interfaces, and it forces the application to handle such things as help screens, aborting, and prompting.

One class of graphical UIDSs lets you place interaction techniques such as menus, buttons, and scroll bars on the screen. These systems usually let you specify additional areas for application-specific input and output graphics. The designer types in procedures to be called when the user executes each interaction technique. Cardelli's DialogEditor and the Interface Builder for the Macintosh are of this class.

Other graphical systems, including Menulay, Trillium, and Hypercard, organ-

Communication

User-interface tools can also be classified by how they communicate with the application programs at runtime. In application control (also called internal control), the application simply calls interface procedures when input is desired. This model is used by user-interface toolkits. In UIDS control (also called external control), the interface procedures call the application when the user gives a command. This model is used in most UIDSs because it lets the UIDS handle scheduling and sequencing.

External control can be further classified by how the UIDS communicates with the application. The most popular method is to use callback procedures, where the application passes to the UIDS the names of procedures to call. The application is therefore organized as a set of procedures that the interface calls. This is probably the most straightforward and efficient technique.

Another technique is to use shared memory, with the UIDS and application program each polling the data to check for changes or automatically being notified of changes. Other UIDSs use multiprocess message passing or event-handling mechanisms to communicate to applications. While these techniques may be better structured, they are less efficient than shared memory or direct procedure calling.

Finally, there may be mixed control, where either the UIDS or the application can be in charge.

An important communication issue is the bandwidth between the application and the interface. Early UIDS models[1] promoted a narrow connection, as shown in Figure A1, so the application would be more independent of the interface. This model, which provides coarse-grain control, is used by some UIDSs (such as Cousin). With coarse-grain control, the interface and application communicate rarely.

Unfortunately, semantic feedback requires frequent communication. It is difficult or impossible to provide such fine-grain control with this model. In direct-manipulation interfaces, the application and interface need to communicate frequently (up to 60 times a second), for example, to determine legal positions while an object is being dragged with the mouse. The amount of information passed each time is fairly small, however.

Proposed UIDS models[2] like the one shown Figure A2 have tried to provide this kind of feedback by sharing application data with the UIDS. Szekely[3] addresses how to enable this information sharing and still provide the advantages of modularization, but much more research is needed.

References

1. M. Green, "Report on Dialogue-Specification Tools," in *User-Interface Management Systems*, Gunther E. Pfaff, ed., Springer-Verlag, New York, 1985, pp. 9-20.
2. D.R. Olsen, Jr., "ACM SIGGraph Workshop on Software Tools for User-Interface Management," *SIGPlan Notices*, April 1987, pp. 71-147.
3. P. Szekely, "Modular Implementation of Presentations," *Proc. SIGCHI+GI 87*, ACM, New York, 1987, pp. 235-240.

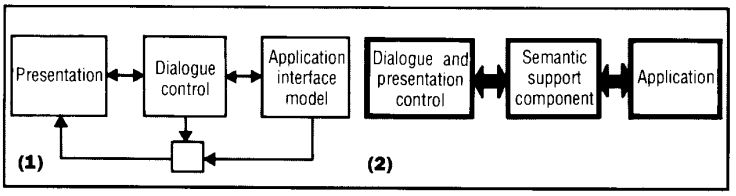


Figure A. Communication models. **(1)** The Seeheim model for UIDSs has a narrow channel between the application and the dialogue control; **(2)** a new model proposes sharing the application information to support semantic feedback.

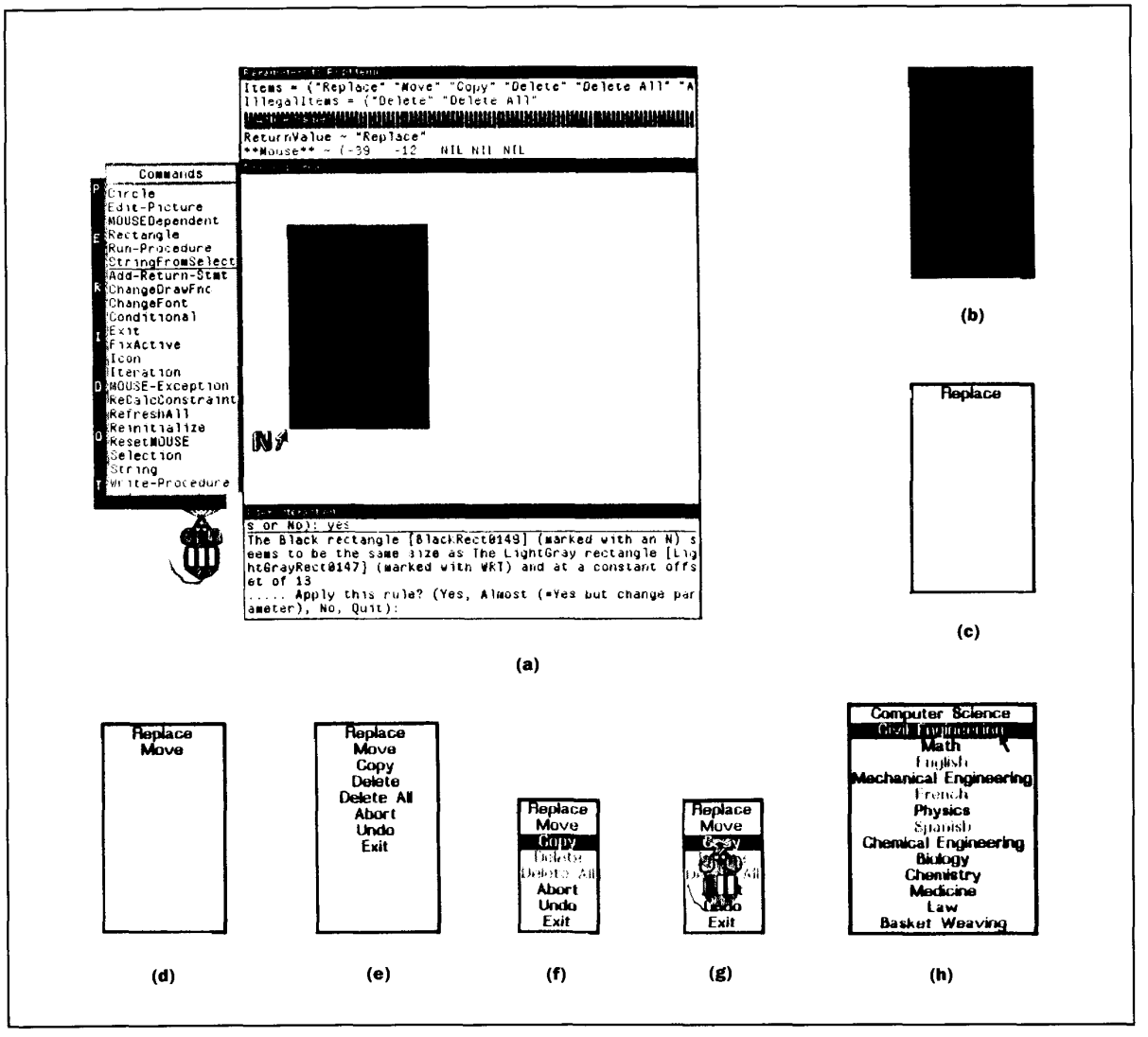


Figure 2. A sequence of snapshots during the creation of a pop-up menu with Peridot. **(a)** The Peridot windows and command menu. The upper left window contains the names of the parameters and active values used by PopMenu and examples for each, the center window contains the interface under construction, and the bottom window contains prompts and error messages. The designer has drawn a gray rectangle for the shadow and a black rectangle. Peridot infers that the black rectangle should be the same size as the gray rectangle but offset slightly. **(b)** The designer confirms the inference and Peridot adjusts the size of the black rectangle. **(c)** The designer draws a white rectangle and then enters text, which is the first element of the Items parameter. The designer drew these in approximately the correct position and Peridot adjusts them to be exact. **(d)** The designer places the second element of Items centered under the first. **(e)** Peridot infers that an interation is desired and displays the other elements similarly. **(f)** The designer modifies the size of the rectangle to the size of the text, and declares that the parameter IllegalItems controls which elements should be gray. **(g)** The Xored black rectangle should follow the mouse, so the designer moves the mouse icon over it with its left button down. **(h)** The menu is complete and can be used with a different set of parameters.

ize the interface as a network of mostly static pages or frames. Each page contains text, graphics, and interaction techniques, as well as commands that cause the system to erase the page and go to different pages. Usually, these systems require that you code the interaction techniques themselves in a conventional programming language.

Menulay lets the designer place text, potentiometers, icons, and buttons on the screen and see exactly what the user will see when the application runs. Each active item in the display is associated with a semantic routine that is invoked when the user selects that item with a pointing device. Like virtually all UIDSs, the semantic routines are written in a conventional programming language.

Menulay generates tables and code that are linked to its runtime support package, which executes the interface. Menulay generates its own interface and supports the concurrent operation of multiple input devices. However, its rigid table-driven structure limits the interaction between the semantic level and the interface, preventing all forms of semantic feedback.

Trillium, which is very similiar to Menu-

lay, supports the design of interface panels for photocopiers. One strong advantage Trillium has over Menulay is that it interprets rather than compiles the specification, so the frames can be executed as they are designed. Trillium also separates the interaction behavior from the graphical presentation so the designer can change the graphics without changing the behavior. However, Trillium provides little support for frame-to-frame transitions because it is rarely necessary for photocopiers.

The Macintosh Hypercard hypertext system could also fall in this UIDS class. It supports graphical specification (and programming in the Hypertalk language) of mostly static pages. Using the editor, the designer can define the text and graphics for the current page, and buttons that cause transitions to other pages.

Grins combines a grammar processor with a constraint-based, input-output linkage system to handle semantic feedback. It incorporates a graphics editor that lets the designer place interaction techniques (menus, icons, and text areas) with a mouse.

Peridot is very different from these systems because it lets the designer create the interaction techniques themselves. The designer manipulates primitives (rectangles, circles, text, and lines) to construct menus, scroll bars, sliders (graphical potentiometers), and buttons.

Peridot generalizes from the designer's actions to create parameterized, object-oriented procedures like those found in interaction technique toolkits. Figure 2 shows snapshots of Peridot during menu creation. The system is powerful enough to create its own interface and most of the interaction techniques in the Macintosh Toolbox.

Automatic creation

A new class of UIDSs tries to create the interface automatically from a specification of the application's semantic procedures and then lets the designer modify the interface to improve it. These systems try to address the difficulty people have using other types of UIDSs and their reluctance to use them.[9]

The Control-Panel Interface, shown in Figure 3, uses a procedure's parameter types to create a graphical interface using buttons for Boolean expressions, knobs or bars for integers, and so on. The designer can specify different displays, use the controls to change the values, and then execute the procedure.

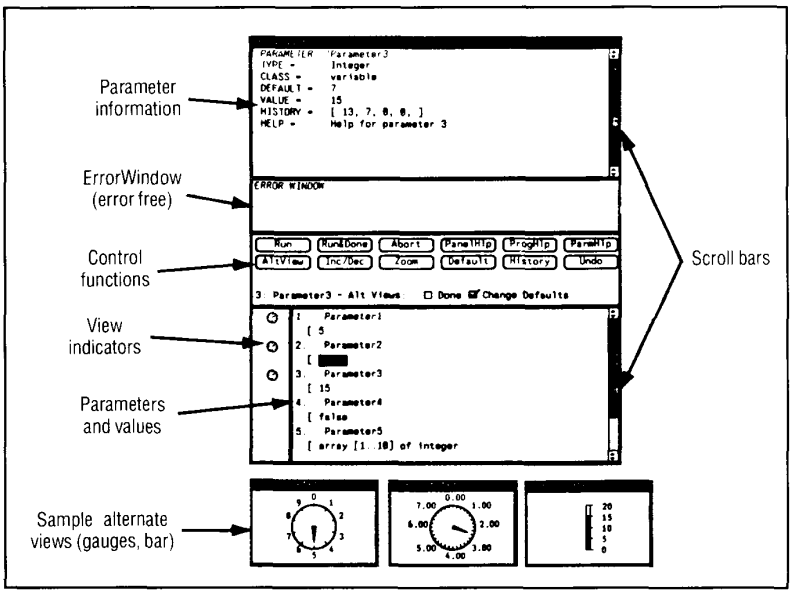


Figure 3. A Control-Panel Interface automatically created for a procedure. Three parameters are represented graphically in the bottom three windows.

Mike creates an interface based on a list of the application procedures. The initial default interface is menu-oriented and verbose, but the designer can change the menu structure, use icons for some commands, and even make some commands operate by direct manipulation. The designer specifies these changes with a graphical editor. Current research on Mike is aimed at adding automatic interface-evaluation capabilities.

IDL, which requires that the application's semantics be defined in a special-purpose, Pascal-like language, might be considered a language-based technique. I include it here instead because the language is used to describe the functions that the application supports, not the desired interface. The system automatically generates a menu-based interface from that description.

An interesting part of IDL is that the designer can change the interface in several ways by applying transformations to it. For example, one transformation changes the interface so that an object is selected before the operation rather than after. Another transformation provides functions specialized for various types (delete-square and delete-triangle) rather than general-purpose functions (delete). IDL applies the transformations and ensures that the resulting interface remains consistent.

Shortcomings

Although use of user-interface toolkits is rising, several problems with them and with UIDSs have limited their use:

- They are too difficult to use. Toolkits typically contain hundreds of procedures that interact subtly. Most UIDSs require that the designer learn a new special-purpose language. Designers who are programmers have proven to be very reluctant to learn a new language to specify the interface.[9] Because these languages are usually like programming languages, they cannot be used by designers who are not programmers. Graphical and automatic-creation UIDSs address this problem, but these systems are mostly experimental.
- The user-interface specifications are difficult to understand and edit. The languages used by many UIDSs to specify the interface are poorly structured in a software-engineering sense: They use global variables, nonlocal control flow, and explicit gotos. Some graphical languages, such as state-transition networks, can become an incomprehensible maze of wires as the interface becomes large. Consequently, it can be very difficult to understand and edit an interface specified with a UIDS. This is a serious problem for a technology intended to aid iterative design.
- They offer too little functionality. Most toolkits and UIDSs support only a small

Some user-interface tools

The commercial and experimental user-interface tools listed here are classified according to how they let the designer specify the interface. Each listing includes a source for more information.

User-interface toolkits. A library of interaction techniques (menus, graphical scroll bars, buttons, and so on) that can be called by the application. These do not provide much support for dialogue control.

- Macintosh Toolbox: *Inside Macintosh*, Addison-Wesley, Reading, Mass., 1985.
- X.11 Toolkit for the X Window Manager: J. McCormack and P. Asente, "An Overview of the X Toolkit," *Proc. ACM SIGGraph Symp. User-Interface Software*, ACM, New York, 1988, pp. 46-55.
- Grow: P.S. Barth, "An Object-Oriented Approach to Graphical Interfaces," *ACM Trans. Graphics*, April 1986, pp. 142-172.
- Coral: P. Szekely and B. Myers, "A User-Interface Toolkit Based on Graphical Objects and Constraints," *SIGPlan Notices*, Nov. 1988, pp. 36-45.

User-interface development systems. An integrated set of tools that help programmers create and manage user interfaces.

Language-based. The designer specifies the interface's syntax in a special-purpose language. Applicable to a broad range of applications, but generally only usable by programmers. The special-purpose language may take several forms:

1. Menu networks:
- Tiger: D.J. Kasik, "A User-Interface Management System," *Computer Graphics*, July 1982, pp. 99-106.
- Hypertext: J. Conklin, "Hypertext: An Introduction and Survey," *Computer*, Sept. 1987, pp. 17-41.

2. State-transition diagrams:
- W.M. Newman, "A System for Interactive Graphical Programming," *Proc. Spring Joint Computer Conf.*, AFIPS, New York, 1968, pp. 47-54.
- State-diagram interpreter: R.J.K. Jacob, "A State-Transition Diagram Language for Visual Programming," *Computer*, Aug. 1985, pp. 51-59.
- Rapid/USE: A.I. Wasserman and D.T. Shewmake, "Rapid Prototyping of Interactive Information Systems," *SIGSoft Software Eng. Notes*, Dec. 1982, pp. 171-180.
- Interaction Objects: R.J.K. Jacob, "A Specification Language for Direct-Manipulation Interfaces," *ACM Trans. Graphics*, Oct. 1986, pp. 283-317.

3. Context-free grammars:
- Syngraph: D.R. Olsen, Jr. and E.P. Dempsey, "Syngraph: A Graphical User-Interface Generator," *Computer Graphics*, July 1983, pp. 43-50.

4. Event languages:
- Algae: M.A. Flecchia and R.D. Bergeron, "Specifying Complex Dialogs in Algae," *Proc. SIGCHI+GI 87*, ACM, New York, 1987, pp. 229-234.
- Sassafras: R.D. Hill, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction: The Sassafras UIMS," *ACM Trans. Graphics*, July 1986, pp. 179-210.
- Squeak: L. Cardelli and R. Pike, "Squeak: A Language for Communicating with Mice," *Computer Graphics*, July 1985, pp. 199-204.

5. Declarative languages:
- Cousin: P.J. Hayes, P.A. Szekely, and R.A. Lerner, "Design Alternatives for User-Interface Management Systems Based on Experience with Cousin," *Proc. SIGCHI 85*, ACM, New York, 1985, pp. 169-175.
- Domain/Dialogue: A.J. Schulert, G.T. Rogers, and J.A. Hamilton, "ADM: A Dialogue Manager," *Proc. SIGCHI 85*, ACM, New York, 1985, pp. 177-183.
- Open Dialogue: Apollo Computer, Inc. Chelmsford, Mass.

6. Object-oriented languages:
- MacApp: K.J. Schmucker, "MacApp: An Application Framework," *Byte*, Aug. 1986, pp. 189-193.
- GWUIMS: J.L. Sibert, W.D. Hurley, and T.W. Bleser, "An Object-Oriented User-Interface Management System," *Computer Graphics*, Aug. 1986, pp. 259-268.
- Higgens: S.E. Hudson and R. King, "A Generator of Direct-Manipulation Office Systems," *ACM Trans. Office Systems*, April 1986, pp. 132-163.

Graphical specification. These let you define the interface by placing objects on the screen with a mouse. While easier to use — some can be used by nonprogrammers — these systems are more difficult to build and only support the creation of a limited range of interfaces.

- DialogEditor: L. Cardelli, "Building User Interfaces by Direct Manipulation," Research Report 22, Digital Equipment Corp. Systems Research Center, Palo Alto, Calif., 1987.
- Interface Builder: ExperTelligence, Santa Barbara, Calif.
- Menulay: W. Buxton et al., "Toward a Comprehensive User-Interface Management System," *Computer Graphics*, July 1983, pp. 35-42.
- Trillium: D.A. Henderson, Jr., "The Trillium User-Interface Design Environment," *Proc. SIGCHI 86*, ACM, New York, 1986, pp. 221-227.
- Hypercard: Apple Computer, Cupertino, Calif.
- Grins: D.R. Olsen, Jr., E.P. Dempsey, and R. Rogge, "Input-Output Linkage in a User-Interface Management System," *Computer Graphics*, July 1985, pp. 225-234.
- Peridot: B.A. Myers, "Creating Interaction Techniques by Demonstration," *IEEE Computer Graphics and Applications*, Sept. 1987, pp. 51-60.

Automatic creation. These attempt to create an interface automatically from a specification of the application's semantics, thus overcoming designer resistance to UIDS technology.

- Control-Panel Interface: G.L. Fisher and K.I. Joy, "A Control-Panel Interface for Graphics and Image-Processing Applications," *Proc. SIGCHI+GI 87*, ACM, New York, 1987, pp. 285-290.
- Mike: D.R. Olsen, Jr., "Mike: The Menu Interaction Kontrol Environment," *ACM Trans. Graphics*, Oct. 1986, pp. 318-344.
- IDL: J. Foley, "Transformations on a Formal Specification of User-Computer Interfaces," *Computer Graphics*, April 1987, pp. 109-112.

part of the design task. While they are very good at handling menus and scroll bars, they can rarely be used to help control the display and manipulation of the application's data objects. Many of these systems make no attempt to handle an application's output. Few UIDSs can support direct-manipulation interfaces because these interfaces use semantic feedback, which many UIDSs do not support. Graphical and automatic-creation UIDSs are easier to use but have the most restricted functionality. Language-based systems are usually more general but still rarely support direct-manipulation interfaces.

- They are unavailable or not portable. Very few user-interface tools are available commercially. Those that are available typically do not work on very many systems. Many people would like to use the Macintosh Toolbox and MacApp, but they can't unless they are developing software to run on the Apple Macintosh. The X Window System manager, its toolkit, and systems that run under it (like Open Dialogue) offer the possibility of more

portable interface software.

- They do not support evaluation. Very few user-interface tools provide any support for evaluating the interface. More research into how the computer could do such evaluation is needed before such support is practical. Two areas where work is needed are how to let the computer perform the evaluation (most guidelines are too informal and general to be used by a computer) and how to save appropriate information efficiently (saving keystrokes generates too much data and works poorly for mouse-based interfaces).
- They are hard to build. While there is much research on how to make user-interface toolkits and UIDSs, many problems remain. Most useful tools have taken a lot of effort to produce.
- Designers are unwilling to give up control. Most user-interface tools enforce a particular interface style. The advantages are that applications will have similar interfaces and the designer does not have to create an interface style for every program. The disadvantage is that some designers are unwilling to give up this flexibility because they want their products to have a unique look and feel. This consideration might be important to differentiate their products from their competitors'. A few experimental UIDSs, such as Peridot, try to be somewhat style-independent, but this complicates the design task. It is probably impossible for a user-interface tool to be totally style-independent if it provides more support than a conventional programming language.
- They can impair the quality of the interface. User-interface tools often add a layer of interpretation and thus can slow execution significantly. These systems usually require that designers give up the ability to hand-tune some aspects of the interface, which may make it worse than one that was carefully crafted.
- It is difficult to separate the interface from the application. One survey of interface designers found that 50 percent "indicated that the user interface had not been considered distinct from the rest of the system during design; many seemed to have real difficulty in even imagining how such a separation might apply to the system they had designed, and a few made strong statements about the inadvisability or impossibility of making such a distinction."[10]

Although many of these user-interface tools have had only limited success and there are still many areas to research, the future for user-interface tools is bright. Toolkits are proving to be very popular and the recent surge of research in UIDSs has produced several new approaches. A few commercial UIDSs such as MacApp and Open Dialogue are being used widely.

It is certainly not now obvious which approaches are the most fruitful for further research and commercial development, but it is clear that there is a rising need and demand for these tools, so the pace of investigation and development is likely to accelerate. ❖

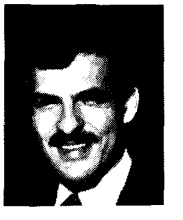
Acknowledgments

This research was sponsored by the Defense Dept.'s Advanced Research Projects Agency under order 4976, under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Div., Wright Patterson Air Force Base, Ohio. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of DARPA or the US government.

I thank Dario Giuse, Doug Bunting, Pedro Szekely, Randy Pausch, Tom Cobourn, Len Bass, Stephan Greene, the anonymous referees, and Bernita Myers for their help and support on this article.

References

1. D.G. Bobrow, S. Mittal, and M.J. Stefik, "Expert Systems: Perils and Promise," *Comm. ACM*, Sept. 1986, pp. 880-894.
2. B. Schneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, Aug. 1983, pp. 57-69.
3. W. Swartout and R. Balzer, "The Inevitable Intertwining of Specification and Implementation," *Comm. ACM*, July 1982, pp. 438-440.
4. M.D. Good et al., "Building a User-Derived Interface," *Comm. ACM*, Oct. 1984, pp. 1032-1043.
5. K.J. Shmucker, "MacApp: An Application Framework," *Byte*, Aug. 1986, pp. 189-193.
6. D.R. Olsen, Jr., "ACM SIGGraph Workshop on Software Tools for User-Interface Management," *Computer Graphics*, April 1987, pp. 71-147.
7. M. Green, "A Survey of Three Dialog Models," *ACM Trans. Graphics*, July 1986, pp. 244-275.
8. W. Buxton and B. Myers, "A Study in Two-Handed Input," *Proc. SIGCHI 86*, ACM, New York, 1986, pp. 321-326.
9. D.R. Olsen, Jr., "Larger Issues in User-Interface Management," *Computer Graphics*, April 1987, pp. 134-137.
10. M.B. Rosson, S. Maass, and W.A. Kellogg, "Designing for Designers: An Analysis of Design Practices in the Real World," *Proc. SIGCHI+GI 87*, ACM, New York, 1987, pp. 137-142.



Brad A. Myers is a research computer scientist developing a new UIDS at Carnegie Mellon University. His research interests include user-interface development systems, user interfaces, programming-by-example, visual programming, interaction techniques, window management, programming environments, debugging, and graphics.

Myers received a PhD in computer science from the University of Toronto. He received an MS and BS in computer science from the Massachusetts Institute of Technology. He is a member of SIGGraph, SIGCHI, ACM, and the IEEE Computer Society.

Address questions about this article to Myers at Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213-3890.