USER INTERFACE MANAGEMENT SYSTEMS

Almost as long as there have been user interfaces, there have been special software systems and tools to help design and implement the user interface software. Many of these tools have demonstrated significant productivity gains for programmers, and they have become important commercial products. Others have proven less successful at supporting the kinds of user interfaces people want to build.

User-interface software is often large, complex and difficult to implement, debug, and modify. A 1992 study found that an average of 48% of the code of applications is devoted to the user interface and that about 50% of the implementation time is devoted to implementing the user interface portion (1), and the numbers are probably much higher today. As interfaces become easier to use, they become harder to create (2). Today, direct manipulation interfaces, also called graphical user interfaces (GUIs), are almost universal. These interfaces require that the programmer deal with elaborate graphics, multiple ways for giving the same command, multiple asynchronous input devices (usually a keyboard and a pointing device such as a mouse), a "mode free" interface where the user can give any command at virtually any time, and rapid "semantic feedback" where determining the appropriate response to user actions requires specialized information about the objects in the program. Tomorrow's user interfaces will provide speech and gesture recognition, three-dimensions (3-D), intelligent agents, and integrated multimedia, and will probably be even more difficult to create. Furthermore, because user interface design is so difficult, the only reliable way to get good interfaces is to iteratively redesign (and therefore reimplement) the interfaces after user-testing, which makes the implementation task even harder.

Fortunately, there has been significant progress in software tools to help with creating user interfaces; and today, virtually all user interface software is created using tools that make the implementation easier. For example, the MacApp system from Apple was reported to reduce development time by a factor of four or five (3). A study commissioned by NeXT claimed that the average application programmed using the NeXTStep environment wrote 83% fewer lines of code and took one-half the time compared to applications written using less advanced tools, and some applications were completed in one-tenth the time (4).

This article surveys user interface software tools, and it explains the different types and classifications. However, it is now impossible to discuss *all* user interface tools, since there are so many. A comprehensive list which is frequently updated is available through the World-Wide Web as http://www.cs.cmu.edu/~bam/toolnames.html. For example, there are over 100 commercial graphical user interface builders, and many new research tools are reported every year at conferences such as the annual ACM User-Interface Software and Technology Symposium (UIST) (see http://www.acm.org/uist/) and the ACM SIGCHI conference (see, for example, http://www.acm.org/sigchi/chi99). There are also about three PhD theses on user interface tools every year. Therefore, this article provides an overview of the most popular approaches, rather than an exhaustive survey.

DEFINITIONS

The user interface (UI) of a computer program is the part that handles the output to the display and the input from the person using the program. The rest of the program is called the *application* or the *application semantics.*

User interface tools have been called various names over the years, with the most popular being user-interface management systems (UIMS) (5). However, many people feel that the term UIMS should be used only for tools that handle the sequencing of operations (what happens after each event from the user), so other terms like Toolkits, user-interface development environments, interface builders, interface development tools, and application frameworks have been used. This article will try to define these terms more specifically and will use the general term "user interface tool" for all software aimed to help create user interfaces. Note that the word "tool" is being used to include what are called "toolkits," as well as higher-level tools, such as interface builders, that are *not* toolkits.

Four different classes of people are involved with user-interface software, and it is important to have different names for them to avoid confusion. The first is the person using the resulting program, who is called the *end-user* or just *user.* The next person creates the user interface of the program and is called the *user-interface designer* or just *designer.* Working with the user interface designer will be the person who writes the software for the rest of the application. This person is called the *application programmer.* The designer may use special user interface tools which are provided to help create user interfaces. These tools are created by the *tool creator.* Note that the designer will be a user of the software created by the tool creator, but we still do not use the term "user" here to avoid confusion with the end user. Although this classification discusses each role as a different person, in fact, there may be many people in each role or one person may perform multiple roles. The general term *programmer* is used for anyone who writes code, and it may be a designer, application programmer, or tool creator.

IMPORTANCE OF USER-INTERFACE TOOLS

There are many advantages to using user interface software tools. These can be classified into two main groups:

1. *The Quality of the Interfaces Might Be Higher.* This is because:
   - Designs can be rapidly prototyped and implemented, possibly even before the application code is written.
   - It is easier to incorporate changes discovered through user testing.
   - More effort can be expended on the tool than may be practical on any single user interface since the tool will be used with many different applications.
   - Different applications are more likely to have consistent user interfaces if they are created using the same user interface tool.
   - It will be easier for a variety of specialists to be involved in designing the user interface, rather than having the user interface created entirely by programmers. Graphic artists, cognitive psychologists, and human factors specialists may all be involved. In particular, professional user-interface designers, who may not be programmers, can be in charge of the overall design.
   - Undo, Help, and other features are more likely to be available in the interfaces since they might be supported by the tools.
2. *The User-Interface Code Might be Easier and More Economical to Create and Maintain.* This is because:
   - Interface specifications can be represented, validated, and evaluated more easily.
   - There will be less code to write, because much is supplied by the tools.
   - There will be better modularization due to the separation of the user-interface component from the application. This should allow the user interface to change without affecting the application, and a large class of changes to the application (such as changing the internal algorithms) should be possible without affecting the user interface.
   - The level of programming expertise of the interface designers and implementers can be lower, because the tools hide much of the complexities of the underlying system.
   - The reliability of the user interface will be higher, since the code for the user interface is created automatically from a higher-level specification.
   - It will be easier to port an application to different hardware and software environments since the device dependencies can be isolated in the user-interface tool.

Based on these goals for user-interface software tools, we can list a number of important functions that should be provided. This list can be used to evaluate the various tools to see how much they cover. Naturally, no tool will help with everything, and different user interface designers may put different emphasis on the different features.

In general, the tools might:

- help *design* the interface given a specification of the end-users' tasks
- help *implement* the interface given a specification of the design
- help *evaluate* the interface after it is designed and propose improvements, or at least provide information to allow the designer to evaluate the interface
- create easy-to-use interfaces
- allow the designer to rapidly investigate different designs
- allow nonprogrammers to design and implement user interfaces
- allow the end-user to customize the interface
- provide portability
- be easy to use themselves

This might be achieved by having the tools:

- automatically choose which user-interface styles, input devices, widgets, and so on, should be used
- help with screen layout and graphic design
- validate user inputs
- handle user errors
- handle aborting and undoing of operations
- provide appropriate feedback to show that inputs have been received
- provide help and prompts
- update the screen display when application data changes
- notify the application when the user modifies graphical objects
- handle field scrolling and editing
- help with the sequencing of operations
- insulate the application from all device dependencies and the underlying software and hardware systems
- provide customization facilities to end-users
- evaluate the graphic design and layout, usability, and learnability of the interface

OVERVIEW OF USER-INTERFACE SOFTWARE TOOLS

Since user-interface software is so difficult to create, it is not surprising that people have been working for a long time to create tools to help with it. Today, many of these tools and ideas have progressed from research into commercial systems, and their effectiveness has been amply demonstrated. Research systems also continue to evolve quickly, and the models that were popular five years ago have been made obsolete by more effective tools, changes in the computer market (e.g., the demise of OpenLook has taken with it a number of tools), and the emergence of new styles of user interfaces such as pen-based computing and multimedia.

Components of User-Interface Software

As shown in Fig. 1, user interface software may be divided into various layers: the windowing system, the toolkit, and

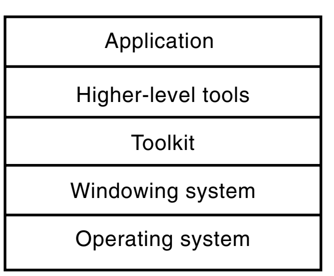

Figure 1. The components of user-interface software.

higher-level tools. Of course, many practical systems span multiple layers.

The windowing system supports the separation of the screen into different (usually rectangular) regions, called *windows*. The X system (6) divides the window functionality into two layers: the window system, which is the functional or programming interface; and the window manager, which is the user interface. Thus the "window system" provides procedures that allow the application to draw pictures on the screen and get input from the user, and the "window manager" allows the end user to move windows around and is responsible for displaying the title lines, borders, and icons around the windows. However, many people and systems use the name "window manager" to refer to both layers, since systems such as the Macintosh and Microsoft Windows do not separate them. This article will use the X terminology, and it will use the term "windowing system" when referring to both layers.

On top of the windowing system is the toolkit, which contains many commonly used widgets such as menus, buttons, scroll bars, and text input fields. On top of the toolkit might be higher-level tools, which help the designer use the toolkit widgets. The following sections discuss each of these components in more detail.

WINDOWING SYSTEMS

A windowing system is a software package that helps the user monitor and control different contexts by separating them physically onto different parts of one or more display screens. A survey of various windowing systems was published earlier (7). Although most of today's systems provide toolkits on top of the windowing systems, as will be explained below, toolkits generally only address the drawing of widgets such as buttons, menus, and scroll bars. Thus, when the programmer wants to draw application-specific parts of the interface and allow the user to manipulate these, the window system interface must be used directly. Therefore, the windowing system's programming interface has significant impact on most user-interface programmers.

The first windowing systems were implemented as part of a single program or system. For example, the EMACs text editor (8), and the Smalltalk (9) and DLISP (10) programming environments had their own windowing systems. Later systems implemented the windowing system as an integral part of the operating system, such as Sapphire for PERQs (11), SunView for Suns, and the Macintosh and Microsoft Windows systems. In order to allow different windowing systems to operate on the same operating system, some windowing systems, such as X and Sun's NeWS, operate as a separate process and use the operating system's interprocess communication mechanism to connect to applications.

Structure of Windowing Systems

A windowing system can be logically divided into two layers, each of which has two parts (see Fig. 2). The window system, or base layer, implements the basic functionality of the windowing system. The two parts of this layer handle the display of graphics in windows (the output model) and the access to the various input devices (the input model), which usually include a keyboard and a pointing device such as a mouse. The primary interface of the base layer is procedural, and is called the windowing system's application programmer interface (API).

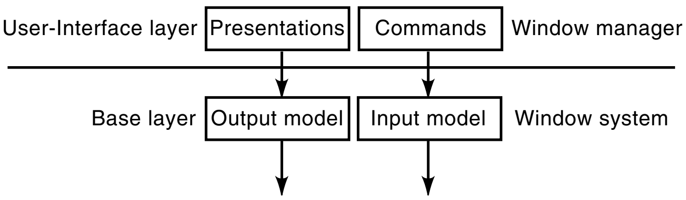

Figure 2. The windowing system can be divided into two layers, called the base or window system layer and the user-interface or window manager layer. Each of these can be divided into parts that handle output and input.

The other layer of windowing system is the window manager or user interface. This includes all aspects that are visible to the user. The two parts of the user-interface layer are the presentation, which is comprised of the pictures that the window manager displays, and the commands, which are how the user manipulates the windows and their contents.

Base Layer

The base layer is the procedural interface to the windowing system. In the 1970s and early 1980s, there were a large number of different windowing systems, each with a different procedural interface (at least one for each hardware platform). People writing software found this to be unacceptable because they wanted to be able to run their software on different platforms, but they would have to rewrite significant amounts of code to convert from one window system to another. The X windowing system (6) was created to solve this problem by providing a hardware-independent interface to windows. X has been quite successful at this, and it drove all other windowing systems out of the workstation hardware market. In the small computer market, the Macintosh runs its own window system, and IBM PC-class machines primarily run some version of Microsoft Windows.

Output Model. The output model is the set of procedures that an application can use to draw pictures on the screen. It is important that all output be directed through the window system so that the graphics primitives can be clipped to the window's borders. For example, if a program draws a line that would extend out of a window's borders, it must be clipped so that the contents of other, independent windows are not overwritten. These operations can be much quicker, but are very dangerous and therefore should seldom be used. Most modern computers provide graphics hardware that is optimized to work efficiently with the window system.

In early windowing systems, such as Smalltalk (9) and Sapphire (12), the primary output operation was BitBlt (also called "RasterOp"). These systems primarily supported monochrome screens (each pixel is either black or white). BitBlt takes a rectangle of pixels from one part of the screen and copies it to another part. Various boolean operations can be specified for combining the pixel values of the source and destination rectangles. For example, the source rectangle can simply replace the destination, or it might be XORed with the destination. BitBlt can be used to draw solid rectangles in either black or white, display text, scroll windows, and perform many other effects (9). The only additional drawing operation typically supported by these early systems was drawing straight lines.

Later windowing systems, such as the Macintosh and X, added a full set of drawing operations, such as filled and unfilled polygons, text, lines, arcs, and so on. These cannot be implemented using the BitBlt operator. With the growing popularity of color screens and nonrectangular primitives (such as rounded rectangles), the use of BitBlt has significantly decreased. It is primarily used now for scrolling and copying off-screen pictures onto the screen (e.g., to implement double-buffering).

A few windowing systems allow the full Postscript imaging model (13) to be used to create images on the screen. Postscript provides device-independent coordinate systems and arbitrary rotations and scaling for all objects, including text. Another advantage of using Postscript for the screen is that the same language can be used to print the windows on paper (since many printers accept Postscript). Sun created a version used in the NeWS windowing system, and then Adobe (the creator of Postscript) came out with an official version called "Display Postscript" which is used in the NeXT windowing system and is supplied as an extension to the X windowing system by a number of vendors, including DEC and IBM.

All of the standard output models only contain drawing operations for two-dimensional (2-D) objects. Two extensions to support 3-D objects are PEX and OpenGL. PEX (14) is an extension to the X windowing system that incorporates much of the PHIGS graphics standard. OpenGL (15) is based on the GL programming interface that has been used for many years on Silicon Graphics machines. OpenGL provides machine independence for 3-D since it is available for various X platforms (SGI, Sun, etc.) and is included as a standard part of Microsoft Windows NT.

As shown in Fig. 3, the earlier windowing systems assumed that a graphics package would be implemented using the windowing system. For example, the CORE graphics package was implemented on top of the SunView windowing system. All newer systems, including the Macintosh, X, NeWS, NeXT, and Microsoft Windows, have implemented a sophisticated graphics system as part of the windowing system.

Input Model. The early graphics standards, such as CORE and PHIGS, provided an input model that does not support the modern, direct manipulation style of interfaces. In those standards, the programmer calls a routine to request the value of a "virtual device" such as a "locator" (pointing device position), "string" (edited text string), "choice" (selection from a menu), or "pick" (selection of a graphical object). The program would then pause, waiting for the user to take action. This is clearly at odds with the direct manipulation "mode-free" style, where the user can decide whether to make a menu choice, select an object, or type something.

With the advent of modern windowing systems, a new model was provided: A stream of event records is sent to the window which is currently accepting input. The user can select which window is getting events using various commands, described below. Each event record typically contains the type and value of the event (e.g., which key was pressed), the window to which the event was directed, a timestamp, and the x and y coordinates of the mouse. The windowing system

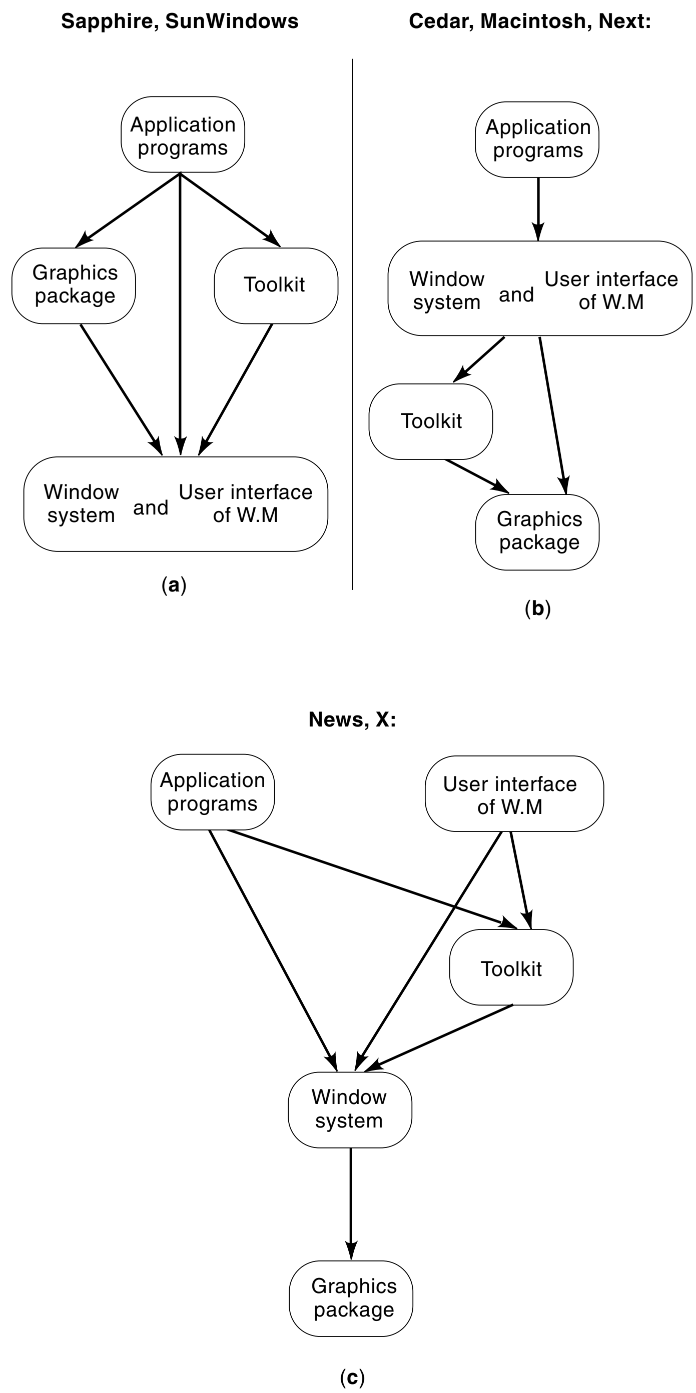

Figure 3. Various organizations that have been used by windowing systems. (a) Early systems tightly coupled the window manager and the window system, and they assumed that sophisticated graphics and toolkits would be built on top. (b) The next step in designs was to incorporate into the windowing system the graphics and toolkits, so that the window manager itself could have a more sophisticated look and feel, and so applications would be more consistent. (c) Other systems allow different window managers and different toolkits, while still embedding sophisticated graphics packages.

queues keyboard events, mouse button events, and mouse movement events together (along with other special events), and programs must dequeue the events and process them. It is somewhat surprising that although there has been substantial progress in the output model for windowing systems (from BitBlt to complex 2-D primitives to 3-D), input is still handled in essentially this same way today as in the original windowing systems, even though there are some well-known unsolved problems with this model:

- There is no provision for special stop-output (control-S) or abort (control-C, command-dot) events, so these will be queued with the other input events.
- The same event mechanism is used to pass special messages from the windowing system to the application. When a window gets larger or becomes uncovered, the application must usually be notified so it can adjust or redraw the picture in the window. Most window systems communicate this by enqueuing special events into the event stream, which the program must then handle.
- The application must always be willing to accept events in order to process aborts and redrawing requests. If not, then long operations cannot be aborted, and the screen may have blank areas while they are being processed.
- The model is device-dependent, since the event record has fixed fields for the expected incoming events. If a 3-D pointing device or one with more than the standard number of buttons is used instead of a mouse, then the standard event mechanism cannot handle it.
- Because the events are handled asynchronously, there are many race conditions that can cause programs to get out of synchronization with the window system. For example, in the X windowing system, if you press inside a window and release outside, under certain conditions the program will think that the mouse button is still depressed. Another example is that refresh requests from the windowing system specify a rectangle of the window that needs to be redrawn, but if the program is changing the contents of the window, the wrong area may be redrawn by the time the event is processed. This problem can occur when the window is scrolled.

Although these problems have been known for a long time, there has been little research on new input models [an exception is the Garnet interactors model (16)].

Communication. In the X windowing system and NeWS, all communication between applications and the window system uses interprocess communication through a network protocol. This means that the application program can be on a different computer from its windows. In all other windowing systems, operations are implemented by directly calling the window manager procedures or through special traps into the operating system. The primary advantage of the X mechanism is that it makes it easier for a person to utilize multiple machines with all their windows appearing on a single machine. Another advantage is that it is easier to provide interfaces for different programming languages: For example, the C interface (called xlib) and the Lisp interface (called CLX) send the appropriate messages through the network protocol. The primary disadvantage is efficiency, since each window request will typically be encoded, passed to the transport layer, and then decoded, even when the computation and windows are on the same machine.

User Interface Layer

The user interface of the windowing system allows the user to control the windows. In X, the user can easily switch user interfaces, by killing one window manager and starting another. Popular window managers under X include uwm (which has no title lines and borders), twm, mwm (the Motif window manager), and olwm (the OpenLook window manager). There is a standard protocol through which programs and the base layer communicate to the window manager, so that all programs continue to run without change when the window manager is switched. It is possible, for example, to run applications that use Motif widgets inside the windows controlled by the OpenLook window manager.

A discussion of the options for the user interfaces of window managers was previously published (7). Also, the video *All the Widgets* (17) has a 30 min segment showing many different forms of window manager user interfaces.

Some parts of the user interface of a windowing system, which is sometimes called its "look and feel," can apparently be copyrighted and patented. Which parts is a highly complex issue, and the status changes with decisions in various court cases (18).

Presentation. The presentation of the windows defines how the screen looks. One very important aspect of the presentation of windows is whether they can overlap or not. Overlapping windows, sometimes called covered windows, allow one window to be partially or totally on top of another window. This is also sometimes called the *desktop metaphor,* since windows can cover each other like pieces of paper can cover each other on a desk. There are usually other aspects to the desktop metaphor, however, such as presenting file operations in a way that mimics office operations, as in the Star office workstation (19). The other alternative is called *tiled* windows, which means that windows are not allowed to cover each other. Obviously, a window manager that supports covered windows can also allow them to be side-by-side, but not vice versa. Therefore, a window manager is classified as "covered" if it allows windows to overlap. The tiled style was popular for a while and was used by Cedar (20), and early versions of the Star (19), Andrew (21), and Microsoft Windows. A study even suggested that using tiled windows was more efficient for users (22). However, today tiled windows are rarely seen on conventional window systems, because users generally prefer overlapping.

Modern "browsers" for the World-Wide Web, such as Mosaic, Netscape, and Microsoft's Internet Explorer provide a windowing environment inside the computer's main windowing system. Newer versions of browsers support frames containing multiple scrollable panes, which are a form of tiled window. In addition, if an application written in Java is downloaded (see the section entitled "Virtual Toolkits" below), it can create multiple, overlapping windows like conventional GUI applications.

Another important aspect of the presentation of windows is the use of icons. These are small pictures that represent windows (or sometimes files). They are used because there would otherwise be too many windows to conveniently fit on the screen and manage. Other aspects of the presentation include whether the window has a title line or not, what the background (where there are no windows) looks like, and whether the title and borders have control areas for performing window operations such as resize, iconify, etc.

Commands. Since computers typically have multiple windows and only one mouse and keyboard, there must be a way for the user to control which window is getting keyboard input. This window is called the *input* (or *keyboard*) *focus*. Another term is the *listener* since it is listening to the user's typing. Some systems called the focus the "active window" or "current window," but these are poor terms since in a multiprocessing system, many windows can be actively outputting information at the same time. Window managers provide various ways to specify and show which window is the listener. The most important options are:

- *Click-to-type,* which means that the user must click the mouse button in a window before typing to it. This is used by the Macintosh.
- *Move-to-type,* which means that the mouse only has to move over a window to allow typing to it. This is usually faster for the user, but may cause input to go to the wrong window if the user accidentally knocks the mouse.

Most X window managers (including the Motif and OpenLook window managers) allow the user to choose which method is desired. However, the choice can have significant impact on the user interface of applications. For example, because the Macintosh requires click-to-type, it can provide a single menubar at the top, and the commands can always operate on the focused window. With move-to-type, the user might have to pass through various windows (thus giving them the focus) on the way to the top of the screen. Therefore, Motif applications must have a menubar in each window so the commands will know which window to operate on.

All covered window systems allow the user to change which window is on top (not covered by other windows) and usually allow the user to send a window to the bottom (covered by all other windows). Other commands allow windows to be changed in size, moved, created, and destroyed.

TOOLKITS

A *toolkit* is a library of "widgets" that can be called by application programs. A *widget* is a graphical object that can be manipulated using a physical input device to input a certain type of value. Typically, widgets in toolkits include menus, buttons, scroll bars, text type-in fields, and so on. Figure 4 shows some examples of widgets. Creating an interface using a toolkit can only be done by programmers, because toolkits only have a procedural interface.

Using a toolkit has the advantage that the final UI will look and act similarly to other UIs created using the same toolkit, and each application does not have to rewrite the standard functions, such as menus. A problem with toolkits is that the styles of interaction are limited to those provided. For example, it is difficult to create a single slider that contains two indicators, which might be useful to input the upper and lower bounds of a range. In addition, the toolkits themselves are often expensive to create: "The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate" (23, p. 199). Another problem with toolkits is that they are often difficult to use since they may contain hundreds of procedures, and it is often not clear how to use the procedures to create a desired interface. For example, the documentation for the Macintosh Toolbox now is well over six books, of which about one-third is related to user interface programming.

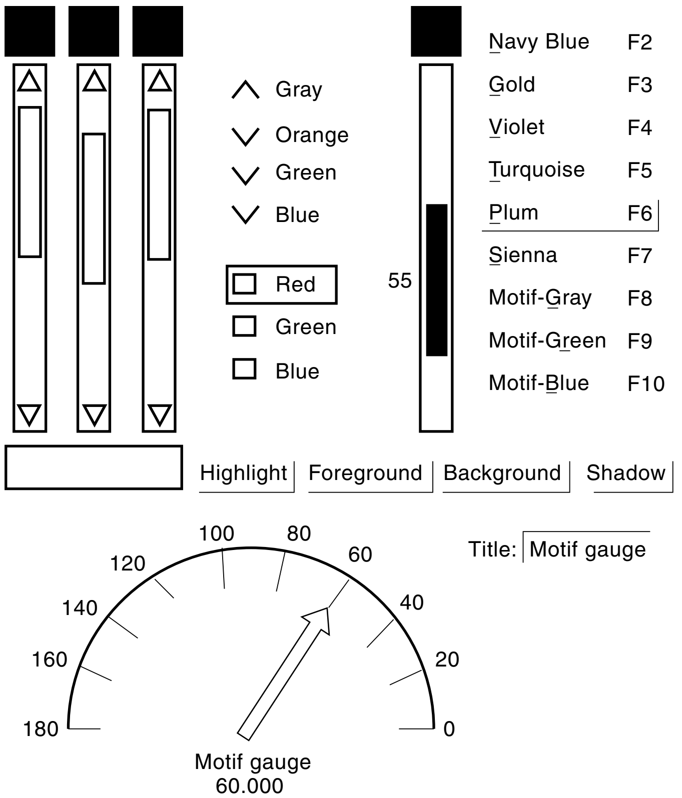

Figure 4. Some of the widgets with a Motif look-and-feel provided by the Garnet toolkit.

As with the graphics package, the toolkit can be implemented either using or being used by the windowing system (see Fig. 3). Early systems provided only minimal widgets (e.g., just a menu) and expected applications to provide others. In the Macintosh, the toolkit is at a low level, and the window manager user interface is built using it. The advantage of this is that the window manager can then use the same sophisticated toolkit routines for its user interface. When the X system was being developed, the developers could not agree on a single toolkit, so they left the toolkit to be on top of the windowing system. In X, programmers can use a variety of toolkits [for example, the Motif, OpenLook, InterViews (24), Amulet (25), or tcl/tk (26) toolkits can be used on top of X], but the window manager must usually implement its user interface from scratch.

Because the designers of X could not agree on a single look-and-feel, they created an intrinsics layer on which to build different widget sets, which they called Xt (27). This layer provides the common services, such as techniques for object-oriented programming and layout control. The widget set layer is the collection of widgets that is implemented using the intrinsics. Multiple widget sets with different looks and feels can be implemented on top of the same intrinsics layer [Fig. 5(a)], or else the same look-and-feel can be implemented on top of different intrinsics [Fig. 5(b)]. When Sun announced that it was phasing out OpenLook, the Motif widget set became the standard for X and Xt.

Toolkit Intrinsics

Toolkits come in two basic varieties. The most conventional is simply a collection of procedures that can be called by application programs. Examples of this style include the SunTools toolkit for the SunView windowing system and the Macintosh Toolbox (28). The other variety uses an object-oriented programming style which makes it easier for the designer to customize the interaction techniques. Examples include Smalltalk (9), Andrew (21), Garnet (29), InterViews (24), Xt (27), Amulet (25), and the Java toolkit AWT (30).

The advantages of using object-oriented intrinsics are that it is a natural way to think about widgets (the menus and buttons on the screen *seem* like objects), the widget objects can handle some of the chores that otherwise would be left to the programmer (such as refresh), and it is easier to create custom widgets (by subclassing an existing widget). The advantage of the older, procedural style is that it is easier to implement, no special object-oriented system is needed, and it is easier to interface to multiple programming languages. To implement the objects, the toolkit might invent its own object system, as was done with Xt, Andrew, Garnet, and Amulet, or it might use an existing object system, as was done in InterViews (24), which uses C++, NeXTStep from NeXT, which uses Objective-C, and AWT, which uses Java (30).

The usual way that object-oriented toolkits interface with application programs is through the use of *call-back procedures.* These are procedures defined by the application programmer that are called when a widget is operated by the end user. For example, the programmer might supply a procedure to be called when the user selects a menu item. Experience has shown that real interfaces often contain hundreds of call-backs, which makes the code harder to modify and maintain (1). In addition, different toolkits, even when implemented on the same intrinsics like Motif and OpenLook, have different call-back protocols. This means that the code for one toolkit is difficult to port to a different toolkit. Therefore, research is being directed at reducing the number of call-backs in user interface software (31).

Some research toolkits have added novel features to the toolkit intrinsics. For example, Garnet (29), Rendezvous (32), Amulet (25), and SubArctic (33) allow the objects to be connected using *constraints,* which are relationships that are declared once and then maintained automatically by the system. For example, the designer can specify that the color of a rectangle is constrained to be the value of a slider, and then the system will automatically update the color if the user moves the slider.

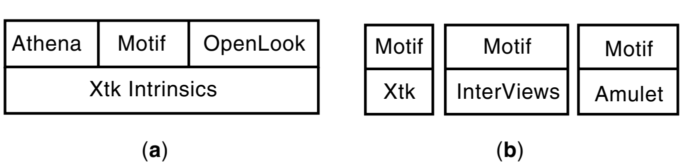

Figure 5. (a) At least three different widget sets that have different looks-and-feels have been implemented on top of the Xt intrinsics. (b) The Motif look-and-feel has been implemented on many different intrinsics.

Many toolkits include a related capability for handling graphical layouts in a declarative manner. Widgets can be specified to stay at the sides or center of a container. This is particularly important when the size of objects might change—for example, in systems that can run on multiple architectures. An early example of this was in InterViews (24), and layout managers are important parts of Motif and Java AWT.

Other important features include support for animation, video, and sound. For example, Amulet provides *animation constraints* (34) where any property of an object can be animated. Supporting video and sound in user interfaces has been studied, but the available tools are still very difficult to use.

Widget Set

Typically, the intrinsics layer is look-and-feel-independent, which means that the widgets built on top of it can have any desired appearance and behavior. However, a particular widget set must pick a look-and-feel. The video *All the Widgets* shows many examples of widgets that have been designed over the years (17). For example, it shows 35 different kinds of menus. Like window manager user interfaces, the widgets' look-and-feel can be copyrighted and patented (18).

As was mentioned above, different widget sets (with different looks and feels) can be implemented on top of the same intrinsics. In addition, the same look-and-feel can be implemented on top of different intrinsics. For example, there are Motif look-and-feel widgets on top of the Xt, InterViews, and Amulet intrinsics [Fig. 5(b)]. Although they all look and operate the same (so would be indistinguishable to the end-user), they are implemented quite differently and have completely different procedural interfaces for the programmer.

Specialized Toolkits

A number of toolkits have been developed to support specific kinds of applications or specific classes of programmers. For example, the SUIT system (35) (which contains a toolkit and an interface builder) is specifically designed to be easy to learn and is aimed at classroom instruction. Amulet (25) provides high-level support for graphical, direct manipulation interfaces, and it handles input as hierarchical *command objects,* making Undo easier to implement (36). Rendezvous (32), Visual Obliq (37), and GroupKit (38) are designed to make it easier to create applications that support multiple users on multiple machines operating synchronously. Whereas most toolkits provide only 2-D interaction techniques, the Brown 3-D toolkits (39) and Silicon Graphics' Inventor toolkit (40) provide preprogrammed 3-D widgets and a framework for creating others. Special support for animations has been added to Artkit (41) and Amulet (34). Tk (26) is a popular toolkit for the X window system (and also Windows) because it uses an interpretive language called tcl which makes it possible to dynamically change the user interface. Tcl also supports the Unix style of programming where many small programs are glued together.

VIRTUAL TOOLKITS

Although there are many small differences among the various toolkits, much remains the same. For example, all have some type of menu, button, scroll bar, text input field, and so on. Although there are fewer windowing systems and toolkits than there were 10 years ago, people are still finding it to be a lot of work to convert software from Motif to the Macintosh and to Microsoft Windows.

Therefore, a number of systems have been developed that try to hide the differences among the various toolkits, by providing virtual widgets which can be mapped into the widgets of each toolkit. Another name for these tools is *cross-platform development systems.* The programmer writes the code once using the virtual toolkit, and the code will run without change on different platforms and still look like it was designed with that platform's widgets. For example, the virtual toolkit might provide a single menu routine, which always has the same programmer interface but connects to a Motif menu, Macintosh menu, or a Windows menu depending on which machine the application is run on.

There are two styles of virtual toolkits. In one, the virtual toolkit links to the different actual toolkits on the host machine. For example, XVT (42) provides a C or C++ interface that links to the actual Motif, OpenLook, Macintosh, MS-Windows, and OS/2-PM toolkits (and also character terminals) and hides their differences. The second style of virtual toolkit *reimplements* the widgets in each style. For example, Galaxy (43), and Open Interface from NeuronData (44), and Amulet (25) provide libraries of widgets that look like those on the various platforms. Different versions of Java have used both forms. The advantage of the first style is that the user interface is more likely to be look-and-feel conformant (since it uses the real widgets). The disadvantages are that the virtual toolkit must still provide an interface to the graphical drawing primitives on the platforms. Furthermore, they tend to only provide functions that appear in all toolkits. Many of the virtual toolkits that take the second approach (e.g., Galaxy) provide a sophisticated graphics package and complete sets of widgets on all platforms. However, with the second approach, there must always be a large run-time library, since in addition to the built-in widgets that are native to the machine, there is the reimplementation of these same widgets in the virtual toolkit's library.

You might think that toolkits that work on multiple platforms should be considered virtual toolkits of the second type. For example, SUIT (35) and Garnet (29) work on X, Macintosh, and Windows. However, these use the same look-and-feel on all platforms (and therefore do not look the same as the other applications on that platform), so they are not classified as virtual toolkits.

The AWT toolkit that comes with the Java programming language (30) also can be classified as a virtual toolkit, since the programmer can write code once and it will operate on all platforms. Java programs can be run locally in a conventional fashion, or can be downloaded dynamically over the World-Wide Web into a browser such as Netscape.

HIGHER-LEVEL TOOLS

Since programming at the toolkit level is quite difficult, there is a tremendous interest in higher-level tools that will make the user-interface software production process easier. These are discussed next.

Phases

Many higher-level tools have components that operate at different times. The *design-time component* helps the user-interface designer design the user interface. For example, this might be a graphical editor which can lay out the interface, or a compiler to process a user interface specification language. The next phase is when the end-user is using the program. Here, the *run-time component* of the tool is used. This usually includes a toolkit, but may also include additional software specifically for the tool. Since the run-time component is "managing" the user interface, the term *User-Interface Management System* (UIMS) seems appropriate for tools with a significant run-time component.

There may also be an *after-run-time component* that helps with the evaluation and debugging of the user interface. Unfortunately, very few user interface tools have an after-run-time component. This is partially because tools [such as MIKE (45)] that have tried to use an after-run-time component have discovered that there are very few metrics that can be applied by computers. A new generation of tools are trying to evaluate how people will interact with interfaces by automatically creating cognitive models from high-level descriptions of the user interface. For example, the GLEAN system generates quantitative predictions of performance of a system from a GOMS model (46).

Specification Styles

High-level user interface tools come in a large variety of forms. One important way that they can be classified is by how the designer specifies what the interface should be. Some tools require the programmer to program in a special-purpose language, some provide an application framework to guide the programming, some automatically generate the interface from a high-level model or specification, and others allow the interface to be designed interactively. Each of these types is discussed below. Of course, some tools use different techniques for specifying different parts of the user interface. These are classified by their predominant or most interesting feature.

Language-Based Tools. With most of the older user interface tools, the designer specifies the user interface in a special-purpose language. This language can take many forms, including context-free grammars, state transition diagrams, declarative languages, event languages, and so on. The language is usually used to specify the syntax of the user interface—that is, the legal sequences of input and output actions. This is sometimes called the "dialogue." Green (47) provides an extensive comparison of grammars, state transition diagrams, and event languages, and Olsen (5) surveys various UIMS techniques.

State Transition Networks. Since many parts of user interfaces involve handling a sequence of input events, it is natural to think of using a state transition network to code the interface. A transition network consists of a set of states, with arcs out of each state labeled with the input tokens that will cause a transition to the state at the other end of the arc. In addition to input tokens, calls to application procedures and the output to display can also be put on the arcs in some systems. Newman implemented a simple tool using finite-state machines in 1968 (48) which handled textual input. This was apparently the first user interface tool. Many of the assumptions and techniques used in modern systems were present in Newman's tool: different languages for defining the user interface and the semantics (the semantic routines were coded in a normal programming language), a table-driven syntax analyzer, and device independence.

State diagram tools are most useful for creating user interfaces where the user interface has a large number of modes (each state is really a mode). For example, state diagrams are useful for describing the operation of low-level widgets (e.g., how a menu or scroll bar works) or the overall global flow of an application (e.g., this command will pop-up a dialogue box, from which you can get to these two dialog boxes, and then to this other window, etc.). However, most highly interactive systems attempt to be mostly "mode-free," which means that at each point the user has a wide variety of choices of what to do. This requires a large number of arcs out of each state, so state diagram tools have not been successful for these interfaces. In addition, state diagrams cannot handle interfaces where the user can operate on multiple objects at the same time. Another problem is that they can be very confusing for large interfaces, since they get to be a "maze of wires" and off-page (or off-screen) arcs can be hard to follow.

Recognizing these problems, but still trying to retain the perspicuousness of state transition diagrams, Jacob (49) invented a new formalism, which is a combination of state diagrams with a form of event languages. There can be multiple diagrams active at the same time, along with flow of control transfers from one to another in a co-routine fashion. The system can create various forms of direct manipulation interfaces. Visual applications builder (VAPS) is a commercial system that uses the state transition model, and it eliminates the maze-of-wires problem by providing a spreadsheet-like table in which the states, events, and actions are specified (50). Transition networks have been thoroughly researched, but have not proven particularly successful or useful as either a research or commercial approach.

Context-Free Grammars. Many grammar-based systems are based on parser generators used in compiler development. For example, the designer might specify the user interface syntax using some form of Backus–Naur form (BNF). Examples of grammar-based systems are Syngraph (51) and parsers built with YACC and LEX in Unix.

Grammar-based tools, like state diagram tools, are not appropriate for specifying highly interactive interfaces, since they are oriented to batch processing of strings with a complex syntactic structure. These systems are best for textual command languages, and they have been mostly abandoned for user interfaces by researchers and commercial developers.

Event Languages. With event languages, the input tokens are considered to be "events" that are sent to individual event handlers. Each handler will have a condition clause that determines what types of events it will handle, and when it is active. The body of the handler can cause output events, change the internal state of the system (which might enable other event handlers), or call application routines.

Sassafras (52) is an event language where the user interface is programmed as a set of small event handlers. The Elements-Events and Transitions (EET) language provides elaborate control over when the various event handlers are fired (53). In these earlier systems, the event handers were global. With more modern systems, the event handlers are specific to particular objects. For example, the HyperTalk language that is part of HyperCard for the Apple Macintosh can be considered an event language. Microsoft's Visual Basic also contains event-language features, since code is generally written as responses to events on objects.

The advantages of event languages are that they can handle multiple input devices active at the same time, and it is straightforward to support nonmodal interfaces, where the user can operate on any widget or object. The main disadvantage is that it can be very difficult to create correct code, especially as the system gets larger, since the flow of control is not localized and small changes in one part can affect many different pieces of the program. It is also typically difficult for the designer to understand the code once it reaches a nontrivial size. However, the success of HyperTalk, Visual Basic and similar tools shows that this approach is appropriate for small- to medium-size programs.

Declarative Languages. Another approach is to try to define a language that is declarative (stating what should happen) rather than procedural (how to make it happen). Cousin (54) and HP/Apollo's Open-Dialogue (55) both allow the designer to specify user interfaces in this manner. The user interfaces supported are basically forms where fields can be text which is typed by the user, or options selected using menus or buttons. There are also graphic output areas that the application can use in whatever manner desired. The application program is connected to the user interface through "variables" which can be set and accessed by both. As researchers have extended this idea to support more sophisticated interactions, the specification has grown into full application "models," and newer systems are described below.

The layout description languages that come with many toolkits are also a type of declarative language. For example, Motif's User Interface Language (UIL) allows the layout of widgets to be defined. Since the UIL is interpreted when an application starts, users can (in theory) edit the UIL code to customize the interface. UIL is not a complete language, however, in the sense that the designer must still write C code for many parts of the interface, including any areas containing dynamic graphics and any widgets that change.

The advantage of using declarative languages is that the user interface designer does not have to worry about the time sequence of events and can concentrate on the information that needs to be passed back and forth. The disadvantage is that only certain types of interfaces can be provided this way, and the rest must be programmed by hand in the "graphic areas" provided to application programs. The kinds of interactions available are preprogrammed and fixed. In particular, these systems provide no support for such things as dragging graphical objects, rubber-band lines, drawing new graphical objects, or even dynamically changing the items in a menu based on the application mode or context. However, these languages have been used as intermediate languages describing the layout of widgets (such as UIL) that are generated by interactive tools.

Constraint Languages. A number of user interface tools allow the programmer to use constraints to define the user interface (56). Early constraint systems include Sketchpad (57), which pioneered the use of graphical constraints in a drawing editor, and Thinglab (58), which used constraints for graphical simulation. Subsequently, Thinglab was extended to aid in the generation of user interfaces (56).

The discussion of toolkits above mentioned the use of constraints as part of the intrinsics of a toolkit. A number of research toolkits now supply constraints as an integral part of the object system [e.g., Garnet, Amulet, and SubArctic (33)]. In addition, some systems have provided higher-level interfaces to constraints. Graphical Thinglab (59) allows the designer to create constraints by wiring icons together, and NoPump (60) and C32 (61) allow constraints to be defined using a spreadsheet-like interface.

The advantage of constraints is that they are a natural way to express many kinds of relationships that arise frequently in user interfaces—for example, that lines should stay attached to boxes, that labels should stay centered within boxes, and so on. A disadvantage with constraints is that they require a sophisticated run-time system to solve them efficiently. However, a growing number of research systems are using constraints, and it appears that modern constraint solvers and debugging techniques may solve these problems, so constraints have a great potential to simplify the programming task. As yet, there are no commercial user-interface tools using general-purpose constraint solvers.

Screen Scrapers. Some commercial tools are specialized to be "front-enders" or "screen scrapers" which provide a graphical user interface to old programs without changing the existing application code. They do this by providing an in-memory buffer that pretends to be the screen of an old character terminal such as might be attached to an IBM mainframe. When the mainframe application outputs to the buffer, a program the designer writes in a special programming language converts this into an update of a graphical widget. Similarly, when the user operates a widget, the script converts this into the appropriate edits of the character buffer. A leading program of this type has been Easel (62), which also contains an interface builder for laying out the widgets.

Database Interfaces. A very important class of commercial tools support form-based or GUI-based access to databases. Major database vendors such as Oracle (63) provide tools which allow designers to define the user interface for accessing and setting data. Often these tools include interactive form editors (which are essentially interface builders) and special database languages. Fourth-generation languages (4GLs), which support defining the interactive forms for accessing and entering data, also fall into this category.

Visual Programming. "Visual programs" use graphics and two (or more)-dimensional layout as part of the program specification (64). Many different approaches to using visual programming to specify user interfaces have been investigated. Most systems that support state transition networks use a visual representation. Another popular technique is to use dataflow languages. In these, icons represent processing steps, and the data flow along the connecting wires. The user interface is usually constructed directly by laying out prebuilt widgets, in the style of interface builders. Examples of visual programming systems for creating user interfaces include Labview (65), which is specialized for controlling laboratory instruments, and Prograph (66). Using a visual language seems to make it easier for novice programmers, but large programs still suffer from the familiar maze-of-wires problem. Other articles (64) have analyzed the strengths and weaknesses of visual programming in detail.

Another popular language is Visual Basic from Microsoft. However, this is more of a structure editor for Basic combined with an interface builder, and therefore it does not really count as a visual language.

Summary of Language Approaches. In summary, many different types of languages have been designed for specifying user interfaces. One problem with all of these is that they can only be used by professional programmers. Some programmers have objected to the requirement for learning a new language for programming just the user interface portion (67). This has been confirmed by market research (68, p. 29). Furthermore, it seems more natural to define the graphical part of a user interface using a graphical editor. However, it is clear that for the foreseeable future, much of the user interface will still need to be created by writing programs, so it is appropriate to continue investigations into the best language to use for this. Indeed, an entire book is devoted to investigating the languages for programming user interfaces (69).

Application Frameworks. After the Macintosh Toolbox had been available for a little while, Apple discovered that programmers had a difficult time figuring out how to call the various toolkit functions, and how to ensure that the resulting interface met the Apple guidelines. They therefore created a software system that provides an overall application framework to guide programmers. This was called MacApp (3) and used the object-oriented language Object Pascal. Classes are provided for the important parts of an application, such as the main windows, the commands, and so on, and the programmer specializes these classes to provide the application-specific details, such as what is actually drawn in the windows and which commands are provided. MacApp was very successful at simplifying the writing of Macintosh applications. Today, there are multiple frameworks to help build applications for most major platforms, including the Microsoft Foundation Classes for Windows and the CodeWarrior PowerPlant (70) for the Macintosh.

Unidraw (71) is a research framework, but it is more specialized for graphical editors. This means that it can provide even more support. Unidraw uses the C++ object-oriented language and is part of the InterViews system (24). Unidraw has been used to create various drawing and computer-aided design (CAD) programs and also to create a user interface editor (72). The Amulet framework (25) is also aimed at graphical applications, but due to its graphical data model, many of the built-in routines can be used without change (the programmer does not usually need to write methods for subclasses). Even more specialized are various graph programs, such as Edge (73) and TGE (74). These provide a framework in which the designer can create programs that display their data as trees or graphs. The programmer typically specializes the node and arc classes, and specifies some of the commands, but the framework handles layout and the overall control.

An emerging popular approach aims to replace today's large, monolithic applications with smaller components that attach together. For example, you might buy a separate text editor, ruler, paragraph formatter, spell checker, and drawing program and have them all work together seamlessly. This approach was invented by the Andrew environment (21), which provides an object-oriented document model that supports the embedding of different kinds of data inside other documents. These "insets" are unlike data that are cut and pasted in systems like the Macintosh because they bring along the programs that edit them, and therefore can always be edited in place. Furthermore, the container document does not need to know how to display or print the inset data since the original program that created it is always available. The designer creating a new inset writes subclasses that adhere to a standard protocol so the system knows how to pass input events to the appropriate editor. The approach is used by Microsoft OLE, Active Apple's OpenDoc, and JavaBeans.

All of these frameworks require the designer to write code, typically by creating application-specific subclasses of the standard classes provided as part of the framework.

Model-Based Automatic Generation. A problem with all of the language-based tools is that the designer must specify a great deal about the placement, format, and design of the user interfaces. To solve this problem, some tools use automatic generation so that the tool makes many of these choices from a much higher-level specification. Many of these tools, such as Mickey (75), Jade (76), and DON (77), have concentrated on creating menus and dialogue boxes. Jade allows the designer to use a graphical editor to edit the generated interface if it is not good enough. DON has the most sophisticated layout mechanisms and takes into account the desired window size, balance, columnness, symmetry, grouping, and so on. Creating dialogue boxes automatically has been very thoroughly researched, but there still are no commercial tools that do this.

Another approach is to try to create a user interface based on a list of the application procedures. MIKE (78) creates an initial interface that is menu-oriented and rather 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 uses a graphical editor to specify these changes.

The user-interface design environment (UIDE) (79) requires that the semantics of the application be defined in a special-purpose language, and therefore might be included with the language-based tools. It is placed here instead because the language is used to describe the functions that the application supports and not the desired interface. UIDE is classified as a "model-based" approach because the specification serves as a high-level, sophisticated model of the application semantics. In UIDE, the description includes pre- and post-conditions of the operations, and the system uses these to reason about the operations and to automatically generate an interface. One interesting part of this system is that the user-interface designer can apply "transformations" to the interface. These change the interface in various ways. For example, one transformation changes the interface to have a currently selected object instead of requiring an object to be selected for each operation. UIDE applies the transformations and ensures that the resulting interface remains consistent. Another feature of UIDE is that the pre- and post-conditions are used to automatically generate help (80).

Another model-based system is HUMANOID (81), which supports the modeling of the presentation, behavior, and dialogue of an interface. The HUMANOID modeling language includes abstraction, composition, recursion, iteration, and conditional constructs to support sophisticated interfaces. The HUMANOID system, which is built on top of the Garnet toolkit (29), provides a number of interactive modeling tools to help the designer specify the model. The developers of HUMANOID and UIDE are collaborating on a new combined model called MASTERMIND, which integrates their approaches (82).

The ITS (83) system also uses rules to generate an interface. ITS was used to create the visitor information system for the EXPO 1992 worlds fair in Seville, Spain. Unlike the other rule-based systems, the designer using ITS is expected to write many of the rules, rather than just writing a specification that the rules work on. In particular, the design philosophy of ITS is that all design decisions should be codified as rules so that they can be used by subsequent designers, which will hopefully mean that interface designs will become easier and better as more rules are entered. As a result, the designer should never use graphical editing to improve the design, since then the system cannot capture the reason that the generated design was not sufficient.

Although the idea of having the user interface generated automatically is appealing, this approach is still at the research level, because the user interfaces that are generated are generally not good enough. A further problem is that the specification languages can be quite hard to learn and use. Current research is addressing the problems of expanding the range of what can be created automatically (to go beyond dialogue boxes) and to make the model-based approach easier to use.

Direct Graphical Specification. The tools described next all allow the user interface to be defined, at least partially, by placing objects on the screen using a pointing device. This is motivated by the observation that the visual presentation of the user interface is of primary importance in graphical user interfaces, and a graphical tool seems to be the most appropriate way to specify the graphical appearance. Another advantage of this technique is that it is usually much easier for the designer to use. Many of these systems can be used by nonprogrammers. Therefore, psychologists, graphic designers, and user interface specialists can more easily be involved in the user interface design process when these tools are used.

These tools can be distinguished from those that use "visual programming" since with direct graphical specification, the actual user interface (or a part of it) is drawn, rather than being generated indirectly from a visual program. Thus, direct graphical specification tools have been called *direct manipulation programming* since the user is directly manipulating the user interface widgets and other elements.

The tools that support graphical specification can be classified into four categories: prototyping tools, those that support a sequence of cards, interface builders, and editors for application-specific graphics.

Prototyping Tools. The goal of prototyping tools is to allow the designer to quickly mock up some examples of what the screens in the program will look like. Often, these tools cannot be used to create the real user interface of the program; they just show how some aspects will look. This is the chief factor that distinguishes them from other high-level tools. Many parts of the interface may not be operable, and some of the things that look like widgets may just be static pictures. In most prototypers, no real toolkit widgets are used, which means that the designer has to draw simulations that look like the widgets that will appear in the interface. The normal use is that the designer would spend a few days or weeks trying out different designs with the tool, and then completely reimplement the final design in a separate system. Most prototyping tools can be used without programming, so they can, for example, be used by graphic designers.

Note that this use of the term “prototyping” is different from the general phrase “rapid prototyping,” which has become a marketing buzzword. Advertisements for just about all user interface tools claim that they support “rapid prototyping,” by which they mean that the tool helps create the user interface software more quickly. The term “prototyping” is being used in this article in a much more specific manner.

Probably the first prototyping tool was Dan Bricklin’s Demo program. This is a program for an IBM PC that allows the designer to create sample screens composed of characters and “character graphics” (where the fixed-size character cells can contain a graphic such as a horizontal, vertical or diagonal line). The designer can easily create the various screens for the application. It is also relatively easy to specify the actions (mouse or keyboard) that cause transitions from one screen to another. However, it is difficult to define other behaviors. In general, there may be some support for type-in fields and menus in prototyping tools, but there is little ability to process or test the results.

For graphical user interfaces, designers often use tools like Macromedia’s Director (84), which is actually an animation tool. The designer can draw example screens, and then specify that when the mouse is pressed in a particular place, an animation should start or a different screen should be displayed. Components of the picture can be reused in different screens, but again the ability to show behavior is limited. HyperCard and Visual Basic are also often used as prototyping tools. A research tool called SILK tries to provide a quick sketching interface and then convert the sketches into actual widgets (85).

The primary disadvantage of these prototyping tools is that sometimes the application must be re-coded in a “real” language before the application is delivered. There is also the risk that the programmers who implement the real user interface will ignore the prototype.

Cards. Many graphical programs are limited to user interfaces that can be presented as a sequence of mostly static pages, sometimes called “frames,” “cards,” or “forms.” Each page contains a set of widgets, some of which cause transfer to other pages. There is usually a fixed set of widgets to choose from, which have been coded by hand.

An early example of this is Menulay (86), which allows the designer to place text, graphical potentiometers, iconic pictures, and light buttons on the screen and see exactly what the end-user will see when the application is run. The designer does not need to be a programmer to use Menulay.

Probably the most famous example of a card-based system is HyperCard from Apple. There are many similar programs, such as GUIDE (87), and Tool Book (88). In all of these, the designer can easily create cards containing text fields, buttons, etc., along with various graphic decorations. The buttons cause transfers to other cards. These programs provide a scripting language to provide more flexibility for buttons. HyperCard’s scripting language is called HyperTalk and, as mentioned above, is really an event language, since the programmer writes short pieces of code that are executed when input events occur.

Interface Builders. An interface builder allows the designer to create dialogue boxes, menus and windows that are to be part of a larger user interface. These are also called *Interface Development Tools* (IDTs). Interface builders allow the designer to select from a predefined library of widgets and then place them on the screen using a mouse. Other properties of the widgets can be set using property sheets. Usually, there is also some support for sequencing, such as bringing up subdialogues when a particular button is hit. The Steamer project at BBN demonstrated many of the ideas later incorporated into interface builders and was probably the first object-oriented graphics system (89). Other examples of research interface builders are DialogEditor (90) and Gilt (31). There are literally hundreds of commercial interface builders. Just two examples are the NeXT interface builder and UIM/X for X (91). Visual Basic is essentially an interface builder coupled with an editor for an interpreted language. Many of the tools discussed above, such as the virtual toolkits, visual languages, and application frameworks, also contain interface builders.

Interface builders use the actual widgets from a toolkit, so they can be used to build parts of real applications. Most will generate C code templates that can be compiled along with the application code. Others generate a description of the interface in a language that can be read at run time. For example, UIM/X generates a UIL description. In Windows and the Macintosh, the Specifications are stored in *resource* files. It is usually important that the programmer not edit the output of the tools (such as the generated C code) or else the tool can no longer be used for later modifications.

Although interface builders make laying out the dialogue boxes and menus easier, this is only part of the user interface design problem. These tools provide little guidance toward creating good user interfaces, since they give designers significant freedom. Another problem is that for any kind of program that has a graphics area (such as drawing programs, CAD, visual language editors, etc.), interface builders do not help with the contents of the graphics pane. Also, they cannot handle widgets that change dynamically. For example, if the contents of a menu or the layout of a dialogue box changes based on program state, this must be programmed by writing code. To help with this part of the problem, some interface builders, like UIM/X (91), provide a C code interpreter, and Visual Basic has its own interpreted language.

Data Visualization Tools. An important commercial category of tools is that of dynamic data visualization systems. These tools, which tend to be quite expensive, emphasize the display of dynamically changing data on a computer and are used as front ends for simulations, process control, system monitoring, network management, and data analysis. The interface to the designer is usually quite similar to an interface builder, with a palette of gauges, graphers, knobs, and switches that can be placed interactively. However, these controls usually are not from a toolkit and are supplied by the tool. Example tools in this category include DataViews (92) and SL-GMS (93).

Editors for Application-Specific Graphics. When an application has custom graphics, it would be useful if the designer could draw pictures of what the graphics should look like rather than having to write code for this. The problem is that the graphic objects usually need to change at run time, based on the actual data and the end-user’s actions. Therefore, the designer can only draw an example of the desired display, which will be modified at run time, and so these tools are called "demonstrational programming" (94). This distinguishes these programs from the graphical tools of the previous three sections, where the full picture can be specified at design time. As a result of the generalization task of converting the example objects into parameterized prototypes that can change at run time, most of these systems are still in the research phase.

Peridot (95) allows new, custom widgets to be created. The primitives that the designer manipulates with the mouse are rectangles, circles, text, and lines. The system generalizes from the designer's actions to create parameterized, object-oriented procedures such as those that might be found in toolkits. Experiments showed that Peridot can be used by nonprogrammers. Lapidary (96) extends the ideas of Peridot to allow general application-specific objects to be drawn. For example, the designer can draw the nodes and arcs for a graph program. The DEMO system (97) allows some dynamic, run-time properties of the objects to be demonstrated, such as how objects are created. The Marquise tool (98) allows the designer to demonstrate *when* various behaviors should happen, and it supports palettes which control the behaviors. With Pavlov (99), the user can demonstrate how widgets should control a car's movement in a driving game. Research continues on making these ideas practical. Gamut (100) has the user give hints to help the system infer sophisticated behaviors for games-style applications.

Specialized Tools

For some application domains, there are customized tools that provide significant high-level support. These tend to be quite expensive, however (i.e., US$20,000 to US$50,000). For example, in the aeronautics and real-time control areas, there are a number of high-level tools, such as AutoCode (101) and InterMAPhics (102).

TECHNOLOGY TRANSFER

User interface tools are an area where research has had a tremendous impact on the current practice of software development (103). Of course, window managers and the resulting "GUI style" comes from the seminal research at the Stanford Research Institute, Xerox Palo Alto Research Center (PARC), and MIT in the 1970s. Interface builders and "card" programs like HyperCard were invented in research laboratories at BBN, the University of Toronto, Xerox PARC, and others. Now, interface builders are at least a US$100 million per year business and are widely used for commercial software development. Event languages, as widely used in HyperTalk and Visual Basic, were first investigated in research laboratories. The current generation of environments, such as OLE and Java Beans, are based on the component architecture which was developed in the Andrew environment from Carnegie Mellon University. Thus, whereas some early UIMS approaches such as transition networks and grammars may not have been successful, overall, the user interface tool research has changed the way that software is developed.

EVALUATING USER-INTERFACE TOOLS

There are clearly a large number of approaches to how tools work, and there are research and commercial tools that use each of the techniques. When faced with a particular programming task, the designer might ask which tool is the most appropriate. Different approaches are appropriate for different kinds of tasks, and orthogonally, there are some dimensions that are useful for evaluating all tools. An important point is that in today's market, there is probably a commercial higher-level tool appropriate for most tasks, so if you are programming directly at the window manager or even toolkit layer, there may be a tool that will save you much work.

Approaches

Using the commercial tools, if you are designing a command-line style interface, then a parser-generator like YACC and Lex is appropriate. If you are creating a graphical application, then you should definitely be using a toolkit appropriate to your platform. If there is an application framework available, it will probably be very helpful. For creating the dialogue boxes and menus, an interface builder is very useful and is generally easier to use than declarative languages like UIL. If your application is entirely (or mostly) pages of information with some fields for the user to fill in, then the card tools might be appropriate.

Among the approaches that are still in the research phase, constraints seem quite appropriate for specifying graphical relationships, automatic generation may be useful for dialogue boxes and menus, and graphical editors will allow the graphical elements of the user interface to be drawn.

Dimensions

There are many dimensions along which you might evaluate user interface tools. The importance given to these different factors will depend on the type of application to be created and the needs of the designers.

- *Depth.* How much of the user interface does the tool cover? For example, interface builders help with dialogue boxes, but do not help with creating interactive graphics. Does the tool help with the evaluation of the interfaces?
- *Breadth.* How many different user interface styles are supported, or is the resulting user interface limited to just one style, such as a sequence of cards? If this is a higher-level tool, does it cover all the widgets in the underlying toolkit? Can new interaction techniques and widgets be added if necessary?
- *Portability.* Will the resulting user interface run on multiple platforms, such as X, Macintosh, and Windows? Will it run on devices with different size displays, from wall-size to hand-held personal digital assistants?
- *Ease of Use of Tools.* How difficult are the tools to use? For toolkits and most language-based higher-level tools, highly trained professional programmers are needed. For some graphical tools, even inexperienced end-users can generate user interfaces. Also, since the designers are themselves users of the tools, the conventional user-interface principles can be used to evaluate the quality of the tools' own user interface.
- *Efficiency for Designers.* How fast can designers create user interfaces with the tool? This is often related to the quality of the user interface of the tool.

- *Quality of Resulting Interfaces.* Does the tool generate high-quality user interfaces? Does the tool help the designer evaluate and improve the quality? Many tools allow the designer to produce any interface desired, so they provide no specific help in improving the quality of the user interfaces.
- *Performance of Resulting Interface.* How fast does the resulting user interface operate? Some tools interpret the specifications at run time or provide many layers of software, which may make the resulting user interface too slow on some target machines. Another consideration is the space overhead since some tools require large libraries to be in memory at run time.
- *Price.* Some tools are provided free by research organizations, such as the SubArctic (33) from Georgia Tech and Amulet (25) from CMU. Most personal computers and workstations today come with a free toolkit. Commercial higher-level tools can range from $50 to $50,000, depending on their capabilities.
- *Robustness and Support.* In one study, users of many of the commercial tools complained about bugs even in the officially released version (1), so checking for robustness is important. Since many of the tools are quite hard to use, the level of training and support provided by the vendor might be important.

Naturally, there are tradeoffs among these criteria. Generally, tools that have the most power (depth and breadth) are more difficult to use. The tools that are easiest to use might be most efficient for the designer, but not if they cannot create the desired interfaces.

As tools become more widespread, reviews and evaluations of them are beginning to appear in magazines such as *PC Magazine.* Market research firms are writing reports evaluating various tools, and there are a few formal studies of tools (104).

RESEARCH ISSUES

Although there are many user interface tools, there are plenty of areas in which further research is needed. A report prepared for a National Science Foundation study discusses future research ideas for user interface tools at length (105). Here, a few of the important ones are summarized.

New Programming Languages

The built-in input/output primitives in today's programming languages, such as printf/scanf or cout/cin, support a textual question-and-answer style of user interface which is modal and well known to be poor. Most of today's tools use libraries and interactive programs which are separate from programming languages. However, many of the techniques, such as object-oriented programming, multiple-processing, and constraints, are best provided as part of the programming language. Even new languages, such as Java, make much of the user interface harder to program by leaving it in separate libraries. Furthermore, an integrated environment, where the graphical parts of an application can be specified graphically and the rest textually, would make the generation of applications much easier. How programming languages can be improved to better support user-interface software is the topic of a book (69).

Increased Depth

Many researchers are trying to create tools that will cover more of the user interface, such as application-specific graphics and behaviors. The challenge here is to allow flexibility to application developers while still providing a high level of support. Tools should also be able to support Help, Undo, and Aborting of operations.

Today's user interface tools mostly help with the generation of the code of the interface, and they assume that the fundamental user interface design is complete. What are also needed are tools to help with the generation, specification, and analysis of the design of the interface (85). For example, an important first step in user-interface design is task analysis, where the designer identifies the particular tasks that the end-user will need to perform. Research should be directed at creating tools to support these methods and techniques. These might eventually be integrated with the code generation tools, so that the information generated during early design can be fed into automatic generation tools, possibly to produce an interface directly from the early analyses. The information might also be used to automatically generate documentation and run-time help.

Another approach is to allow the designer to specify the design in an appropriate notation, and then provide tools to convert that notation into interfaces. For example, the UAN (106) is a notation for expressing the end-user's actions and the system's responses.

Finally, much work is needed in ways for tools to help evaluate interface designs. Initial attempts, such as in MIKE (45), have highlighted the need for better models and metrics against which to evaluate the user interfaces. Research in this area by cognitive psychologists and other user-interface researchers (46) is continuing.

Increased Breadth

We can expect the user interfaces of tomorrow to be different from the conventional window-and-mouse interfaces of today, and tools will have to change to support the new styles. For example, most tools today only deal with 2-D objects, but there is already a demand to provide 3-D visualizations and animations. Sound, video, and animations will increasingly be incorporated into user interfaces. New input devices and techniques will probably replace the conventional mouse and menu styles. For example, gesture and handwriting recognition are appearing in mass-market commercial products, such as notepad computers and "personal digital assistants" such as Apple's Newton (gesture recognition has actually been used since the 1970s in commercial CAD tools). "Virtual reality" systems, where the computer creates an artificial world and allows the user to explore it, cannot be handled by any of today's tools. In these "non-WIMP" (107) applications (WIMP stands for windows, icons, menus, and pointing devices), designers will also need better control over the timing of the interface, to support animations and various new media such as video. Although a few tools are directed at multiple-user applications, there are no direct graphical specification tools, and the current tools are limited in the styles of applications they support.

Another concern is supporting interfaces that can be moved from one natural language to another (like English to French). Internationalizing an interface is much more difficult than simply translating the text strings, and it includes different number, date, and time formats, new input methods, redesigned layouts, different color schemes, and new icons (108). How can future tools help with this process?

End-User Programming and Customization

One of the most successful computer programs of all time is the spreadsheet. The primary reason for its success is that end-users can program (by writing formulas and macros). However, end-user programming is rare in other applications and, where it exists, usually requires learning conventional programming. For example, AutoCAD provides Lisp for customization, and many Microsoft applications use Visual Basic. More effective mechanisms for users to customize existing applications and create new ones are needed (69). However, these should not be built into individual applications as is done today, since this means that the user must learn a different programming technique for each application. Instead, the facilities should be provided at the system level, and therefore they should be part of the underlying toolkit. Naturally, since this is aimed at end-users, it will not be like programming in C, but rather at some higher level.

Application and User-Interface Separation

One of the fundamental goals of user interface tools is to allow better modularization and separation of user-interface code from application code. However, a survey reported that conventional toolkits actually make this separation more difficult, due to the large number of call-back procedures required (1). Therefore, further research is needed into ways to better modularize the code, and how tools can support this.

Tools for the Tools

It is very difficult to create the kinds of tools described in this article. Each one takes an enormous effort. Therefore, work is needed in ways to make the tools themselves easier to create. For example, the Garnet toolkit explored mechanisms specifically designed to make high-level graphical tools easier to create (109). The Unidraw framework has also proven useful for creating interface builders (72). However, more work is needed.

CONCLUSION

The area of user interface tools is expanding rapidly. Ten years ago, you would have been hard-pressed to find any successful commercial higher-level tools, but now there are hundreds of different tools, and tools are turning into a billion-dollar-a-year business. Chances are that today, whatever your project is, there is a tool that will help. Tools that are coming out of research labs are covering increasingly more of the user interface task, are more effective at helping the designer, and are creating better user interfaces. As more companies and researchers are attracted to this area, we can expect the pace of innovation to continue to accelerate. There will be many exciting and useful new tools available in the future.

ACKNOWLEDGMENT

This article is revised from an earlier version which appeared as: Brad A. Myers, User interface software tools, *ACM Trans. Comput.–Hum. Interaction,* **2** (1): 64–103, 1995. © 1995 Association for Computing Machinery. Reprinted by permission.

BIBLIOGRAPHY

1. B. A. Myers and M. B. Rosson, Survey on user interface programming, *Proc. Hum. Factors Comput. Syst. (SIGCHI'92),* Monterey, CA, 1992, pp. 195–202.
2. B. A. Myers, Challenges of HCI design and implementation, *ACM Interact.,* **1** (1): 73–83, 1994.
3. D. Wilson, *Programming with MacApp,* Reading, MA: Addison-Wesley, 1990.
4. Booz Allen & Hamilton, Inc., *NeXTStep vs. Other Development Environments; Comparative Study,* Report available from NeXT Computer, Inc., 1992.
5. D. R. Olsen, Jr., *User Interface Management Systems: Models and Algorithms,* San Mateo, CA: Morgan Kaufmann, 1992.
6. R. W. Scheifler and J. Gettys, The X window system, *ACM Trans. Graphics,* **5** (2): 79–109, 1986.
7. B. A. Myers, A taxonomy of user interfaces for window managers, *IEEE Comput. Graphics Appl.,* **8** (5): 65–84, 1988.
8. R. M. Stallman, *Emacs: The Extensible, Customizable, Self-Documenting Display Editor,* Cambridge, MA: MIT Artificial Intelligence Lab, 1979, Technical Report Number 519.
9. L. Tesler, The Smalltalk environment, *Byte Mag.,* **6** (8): 90–147, 1981.
10. W. Teitelman, A display oriented programmer's assistant, *Int. J. Man-Mach. Stud.,* **11** (2): 157–187, 1979; also *Xerox PARC Tech. Rep.,* Palo Alto, CA, 1977, CSL-77-3.
11. B. A. Myers, The user interface for sapphire, *IEEE Comput. Graphics Appl.,* **4** (12): 13–23, 1984.
12. B. A. Myers, A complete and efficient implementation of covered windows, *IEEE Comput.,* **19** (9): 57–67, 1986.
13. Adobe Systems Inc., *Postscript Language Reference Manual,* Reading, MA: Addison-Wesley, 1985.
14. T. Gaskins, *PEXlib Programming Manual,* Sebastopol, CA: O'Reilly and Associates, 1992.
15. Silicon Graphics Inc., *Open-GL,* Mountain View, CA: Silicon Graphics Inc., 1993.
16. B. A. Myers, A new model for handling input, *ACM Trans. Inf. Syst.,* **8** (3): 289–320, 1990.
17. B. A. Myers, All the widgets, *SIGGRAPH Video Rev.,* **57**: 1990.
18. P. Samuelson, Legally speaking: The ups and downs of look and feel, *Comm. ACM,* **36** (4): 29–35, 1993.
19. D. C. Smith et al., Designing the Star user interface, *Byte,* **7** (4): 242–282, 1982.
20. D. Swinehart et al., A structural view of the Cedar programming environment, *ACM Trans. Programm. Lang. Syst.,* **8** (4): 419–490, 1986.
21. A. J. Palay et al., The Andrew toolkit—An overview, *Proc. Winter Usenix Tech. Conf.,* Dallas, TX, 1988, pp. 9–21.
22. S. A. Bly and J. K. Rosenberg, A comparison of tiled and overlapping windows, *Proc. Hum. Factors Comput. Syst., (SIGCHI '86),* Boston, 1986, pp. 101–106.
23. L. Cardelli and R. Pike, Squeak, A language for communicating with mice, *Proc. Comput.Graphics (SIGGRAPH '85),* Vol. 19, San Francisco, 1985, pp. 199–204.

24. M. A. Linton, J. M. Vlissides, and P. R. Calder, Composing user interfaces with InterViews, *IEEE Comput.,* **22** (2): 8–22, 1989.
25. B. A. Myers et al., The Amulet environment: New models for effective user interface software development, *IEEE Trans. Softw. Eng.,* **23**: 347–365, 1997.
26. J. K. Ousterhout, An X11 toolkit based on the Tcl language, *Proc. Winter Usenix Tech. Conf.,* 1991, pp. 105–115.
27. J. McCormack and P. Asente, An overview of the X toolkit, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '88),* Banff, Alberta, Can., 1988, pp. 46–55.
28. Apple Computer Inc., *Inside Macintosh,* Reading, MA: Addison-Wesley, 1985.
29. B. A. Myers et al., Garnet: Comprehensive support for graphical, highly-interactive user interfaces, *IEEE, Computer,* **23** (11): 71–85, 1990.
30. Sun Microsystems, *Java: Programming for the Internet,* 1998. http://java.sun.com/
31. B. A. Myers, Separating application code from toolkits: Eliminating the spaghetti of call-backs, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '91),* Hilton Head, SC, 1991, pp. 211–220.
32. R. D. Hill et al., The rendezvous architecture and language for constructing multiuser applications, *ACM Trans. Comput.-Hum. Interact.,* **1** (2): 81–125, 1994.
33. S. E. Hudson and I. Smith, Ultra-lightweight constraints, *Proc. ACM SIGGRAPH Symp. on User Interface Softw. Technol. (UIST '96),* Seattle, WA, 1996, pp. 147–155. [Online] Available http://www.cc.gatech.edu/gvu/ui/sub_arctic/
34. B. A. Myers et al., Easily adding animations to interfaces using constraints, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '96),* Seattle, WA, 1996, pp. 119–128. [Online] Available http://www.cs.cmu.edu/~amulet
35. R. Pausch, M. Conway, and R. DeLine, Lesson learned from SUIT, the Simple User Interface Toolkit. *ACM Trans. Inf. Syst.,* **10** (4): 320–344, 1992.
36. B. A. Myers and D. Kosbie, Reusable hierarchical command objects, *Proc. Hum. Factors Comput. Syst. (CHI '96),* Vancouver, BC, Can., 1996, pp. 260–267.
37. K. Bharat and M. H. Brown, Building distributed, multi-user applications by direct manipulation, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '94),* Marina del Rey, CA, 1994, pp. 71–81.
38. M. Roseman and S. Greenberg, Building real time groupware with GroupKit, A groupware toolkit. *ACM Trans. Comput. Hum. Interact.,* **3** (1): 66–106, 1996.
39. M. P. Stevens, R. C. Zeleznik, and J. F. Hughes, An architecture for an extensible 3D interface toolkit, *Proc. ACM SIGGRAPH Symp. User Interface Softw.Technol. (UIST '94),* Marina del Rey, CA, 1994, pp. 59–67.
40. J. Wernecke, *The Inventor Mentor,* Reading, MA: Addison-Wesley, 1994.
41. S. E. Hudson and J. T. Stasko, Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '93),* Atlanta, GA, 1993, pp. 57–67.
42. XVT Software, Inc., *XVT,* Boulder, CO: XVT Software, Inc., 1997.
43. Visix Software Inc., *Galaxy Application Environment,* Reston, VA: Visix Sotware Inc., 1997 (company dissolved in 1998).
44. NeuronData, *Open Interface,* Mountain View, CA: NeuronData, 1995.
45. D. R. Olsen, Jr. and B. W. Halversen, Interface usage measurements in a user interface management system, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '88),* Banff, Alberta, Can., 1988, pp. 102–108.
46. D. E. Kieras et al., GLEAN: A computer-based tool for rapid GOMS model usability evaluation of user interface designs, *Proc. 8th Annu. Symp. User Interface Softw. Technol. (UIST '95),* 1995, pp. 91–100.
47. M. Green, A survey of three dialog models, *ACM Trans. Graphics,* **5** (3): 244–275, 1986.
48. W. M. Newman, A system for interactive graphical programming, *AFIPS Spring Jt. Comput. Conf.,* **28**: 47–54, 1968.
49. R. J. K. Jacob, A specification language for direct manipulation interfaces, *ACM Trans. Graphics,* **5** (4): 283–317, 1986.
50. Virtual Prototypes Inc., *VAPS,* Montreal, Quebec, Can: Virtual Prototypes Inc., 1995.
51. D. R. Olsen, Jr. and E. P. Dempsey, Syngraph: A graphical user interface generator, *Proc. Comput. Graphics (SIGGRAPH '83),* Vol. 17, Detroit, MI, 1983, pp. 43–50.
52. R. D. Hill, Supporting concurrency, communication and synchronization in human-computer interaction—The Sassafras UIMS, *ACM Trans. Graphics,* **5** (3): 179–210, 1986.
53. M. R. Frank, Model-based user interface by demonstration and interview, Ph.D. thesis, Georgia Inst. of Technol., Comput. Sci. Dept., Atlanta, 1995.
54. P. J. Hayes, P. A. Szekely, and R. A. Lerner, Design alternatives for user interface management systems based on experience with COUSIN, *Proc. Hum. Factors Comput. Syst. (SIGCHI '85),* San Francisco, CA, 1985, pp. 169–175.
55. A. J. Schulert, G. T. Rogers, and J. A. Hamilton, ADM—A dialogue manager, *Proc. Hum. Factors Comut. Syst. (SIGCHI '85),* San Francisco, CA, 1985, pp. 177–183.
56. A. Borning and R. Duisberg, Constraint-based tools for building user interfaces, *ACM Trans. Graphics,* **5** (4): 345–374, 1986.
57. I. E. Sutherland, SketchPad: A man-machine graphical communication system, *AFIPS Spring Jt. Comput. Conf.,* **23**: 329–346, 1963.
58. A. Borning, The programming language aspects of Thinglab; a constraint-oriented simulation laboratory, *ACM Trans. Program. Lang. Syst.,* **3** (4): 353–387, 1981.
59. A. Borning, Defining constraints graphically, *Proc. Hum. Factors Comput. Syst. (SIGCHI '86),* Boston, 1986, pp. 137–143.
60. N. Wilde and C. Lewis, Spreadsheet-based interactive graphics: From prototype to tool, *Proc. Hum. Factors Comput. Syst. (SIGCHI '90),* Seattle, WA, 1990, pp. 153–159.
61. B. A. Myers, Graphical techniques in a spreadsheet for specifying user interfaces, *Proc. Hum. Factors Comput. Syst. (SIGCHI '91),* New Orleans, LA, 1991, pp. 243–249.
62. Easel, *Workbench,* Burlington, MA: Easel, 1993.
63. Oracle Corporation, *Oracle Tools,* Redwood Shores, CA: Oracle Corp., 1995.
64. B. A. Myers, Taxonomies of visual programming and program visualization, *J. Visual Lang. Comput.,* **1** (1): 97–123, 1990.
65. National Instruments, *LabVIEW,* Austin, TX: National Instruments, 1989.
66. Pictorius Inc., *Pictorius,* Halifax, Nova Scotia, Can.: Pictorius Inc., 1998, B3L 4G7. info@prograph.com
67. D. R. Olsen, Jr., Larger issues in user interface management, *Comput. Graphics,* **21** (2): 134–137, 1987.
68. I. X Business Group, *Interface Development Technology,* Fremont, CA: I. X Business Group, 1994.
69. B. A. Myers (ed.), *Languages for Developing User Interfaces,* Boston: Jones and Bartlett, 1992.

70. Metrowerks Inc. *PowerPlant for CodeWarrior,* Austin, TX: Metrowerks Inc., 1998. http://www.metrowerks.com/
71. J. M. Vlissides and M. A. Linton, Unidraw: A framework for building domain-specific graphical editors, *ACM Trans. Inf. Syst.,* **8** (3): 204–236, 1990.
72. J. M. Vlissides and S. Tang, A Unidraw-based user interface builder, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '91),* Hilton Head, SC, 1991, pp. 201–210.
73. F. J. Newbery, An interface description language for graph editors, *IEEE Comput. Soc. IEEE Workshop Visual Lang.,* Pittsburgh, PA, 1988, Order No. 876, pp. 144–149.
74. A. Karrer and W. Scacchi, Requirements for an extensible object-oriented tree/graph editor, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '90),* Snowbird, UT, 1990, pp. 84–91.
75. D. R. Olsen, Jr., A programming language basis for user interface management, *Proc. Hum. Factors Comput. Syst. (SIGCHI '89),* Austin, TX, 1989, pp. 171–176.
76. B. Vander Zanden and B. A. Myers, Automatic, look-and-feel independent dialog creation for graphical user interfaces, *Proc. Hum. Factors Comput. Syst. (SIGCHI '90),* Seattle, WA, 1990, pp. 27–34.
77. W. C. Kim and J. D. Foley, Providing high-level control and expert assistance in the user interface presentation design, *Proc. Hum. Factors Comput. Syst. (INTERCHI '93),* Amsterdam, The Netherlands, 1993, pp. 430–437.
78. D. R. Olsen, Jr., Mike: The menu interaction kontrol environment, *ACM Trans. Graphics,* **5** (4): 318–344, 1986.
79. P. Sukaviriya, J. D. Foley, and T. Griffith, A second generation user interface design environment: The model and the runtime architecture, *Proc. Hum. Factors Comput. Syst. (INTERCHI '93),* Amsterdam, The Netherlands, 1993, pp. 375–382.
80. P. Sukaviriya and J. D. Foley, Coupling A UI framework with automatic generation of context-sensitive animated help, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '90),* Snowbird, UT, 1990, pp. 152–166.
81. P. Szekely, P. Luo, and R. Neches, Beyond interface builders: Model-based interface tools, *Proc. Hum. Factors Comput. Syst. (INTERCHI '93),* Amsterdam, The Netherlands, 1993, pp. 383–390.
82. R. Neches et al., Knowledgable development environments using shared design models, *Proc. ACM SIGCHI, Int. Workshop Intell. User Interfaces,* Orlando, FL, 1993, pp. 63–70.
83. C. Wiecha et al., ITS: A tool for rapidly developing interactive applications, *ACM Trans. Inf. Syst.,* **8** (3): 204–236, 1990.
84. MacroMedia, *Director,* San Francisco, CA: MacroMedia, 1995.
85. J. Landay and B. A. Myers, Interactive sketching for the early stages of user interface design, *Proc. Hum. Factors Comput. Syst. (SIGCHI '95),* Denver, CO, 1995, pp. 43–50.
86. W. Buxton et al., Towards a comprehensive user interface management system, *Proc. Comput. Graphics (SIGGRAPH '83),* Vol. 17, Detroit, MI, 1983, pp. 35–42.
87. Owl International Inc., *Guide 2,* Bellevue, WA: Owl International Inc., 1991.
88. *ToolBook,* Bellevue, WA: Asymetrix Corp., Asymetrix Corporation, 1995.
89. A. Stevens, B. Roberts, and L. Stead, The use of a sophisticated graphics interface in computer-assisted instruction, *IEEE Comput. Graphics Appl.,* **3** (2): 25–31, 1983.
90. L. Cardelli, Building user interfaces by direct manipulation, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '88),* Banff, Alberta, Can., 1988, pp. 152–166.
91. Visual Edge Software Ltd., *UIM/X,* Montreal, Quebec, Can.: Visual Edge Software Ltd., 1990, H4R 1V4.
92. DataViews Corporation, *DataViews,* Northampton, MA: DataViews Corp., 1995.
93. SL Corp., SL-GMS, Corte Madera, CA: 1993.
94. B. A. Myers, Demonstrational interfaces: A step beyond direct manipulation, *IEEE Comput.,* **25** (8): 61–73, 1992.
95. B. A. Myers, *Creating User Interfaces by Demonstration,* Boston: Academic Press, 1988.
96. B. Vander Zanden and B. A. Myers, Demonstrational and constraint-based techniques for pictorially specifying application objects and behaviors, *ACM Trans. Comput.-Hum. Interact.,* **2** (4): 308–356, 1995.
97. G. L. Fisher, D. E. Busse, and D. A. Wolber, Adding Rule-Based Reasoning to a Demonstrational Interface Builder, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '92),* Monterey, CA, 1992, pp. 89–97.
98. B. A. Myers, R. G. McDaniel, and D. S. Kosbie, Marquise: Creating complete user interfaces by demonstration, *Proc. Hum. Factors Comput. Syst., (INTERCHI '93),* Amsterdam, The Netherlands, 1993, pp. 293–300.
99. D. Wolber, Pavlov: Programming by stimulus-response demonstration, *Proc. Hum. Factors Comput. Syst. (CHI '96),* Vancouver, BC, Can., 1996, pp. 252–259.
100. R. G. McDaniel and B. A. Myers, Building applications using only demonstration, *Int. Conf. Intell. User Interfaces,* San Francisco, CA, 1998, pp. 109–116.
101. Integrated Systems, *AutoCode,* Santa Clara, CA: Integrated Systems, 1991.
102. InterMAPhics, *Prior Data Sciences,* Kanata, Ontario, Can., InterMAPhics, 1991, K2M 1P6.
103. B. A. Myers, A brief history of human computer interaction technology, *ACM Interact.,* **5** (2): March, 1998, pp. 44–54.
104. D. Hix, A procedure for evaluating human-computer interface development tools, *Proc. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST '89),* Williamsburg, VA, 1989, pp. 53–61.
105. D. R. Olsen, Jr., et al., Research directions for user interface software tools, *Behav. Inf. Technol.,* **12** (2): 80–97, 1993.
106. 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.
107. J. Nielsen, Noncommand user interfaces, *Comm. ACM,* **36** (4): 83–99, 1993.
108. P. Russo and S. Boor, How fluent is your interface? Designing for international users, *Proc. Hum. Factors Comput. Syst. (INTERCHI '93),* Amsterdam, The Netherlands, 1993, pp. 342–347.
109. B. A. Myers and B. Vander Zanden, Environment for rapid creation of interactive design tools, *Visual Comput., Int. J. Comput. Graphics,* **8** (2): 94–116, 1992.

BRAD A. MYERS
Carnegie Mellon University

USER INTERFACES. See GRAPHICAL USER INTERFACES.
UTILITY PROGRAMS. See INPUT-OUTPUT PROGRAMS.
UV LASERS. See EXCIMER LASERS.
UWB RADAR. See RADAR EQUIPMENT.