User Interface Software Tools

BRAD A. MYERS
Carnegie Mellon University

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 have become important commercial products. Others have proven less successful at supporting the kinds of user interfaces people want to build. This article discusses the different kinds of user interface software tools, and investigates why some approaches have worked and others have not. Many examples of commercial and research systems are included. Finally, current research directions and open issues in the field are discussed.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*user interface management systems*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms: Human Factors, Languages

Additional Key Words and Phrases: Interface builders, toolkits, user interface development environments, user interface software

1. INTRODUCTION

User interface software is often large, complex, and difficult to implement, debug, and modify. One 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 [Myers and Rosson 1992]. As interfaces become easier to use, they become harder to create [Myers 1994]. Today, direct-manipulation interfaces (also called "GUIs" for Graphical User Interfaces) are almost universal: one 1993

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, under contract F33615-90-C-1465, Arpa Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
This article is revised from an earlier version which appeared as: Brad A. Myers, "State of the Art in User Interface Software Tools." In *Advances in Human-Computer Interaction*, H. Rex Hartson and D. Hix, Eds. Vol. 4. Ablex, Norwood, N.J., pp. 110–150.
Author's address: Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: bam@cs.cmu.edu.

study found that 97% of all software development on Unix involved a GUI [X Business Group 1994, p. 80]. 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 locator or 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, 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 have iterative redesign (and therefore reimplementation) of the interfaces after user testing, which makes the implementation task even harder.

Fortunately, there has been significant progress in software tools to help create 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 has been reported to reduce development time by a factor of four or five [Schmucher 1986]. A study commissioned by NeXT claims 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 [Booz Allen and Hamilton 1992].

Furthermore, user interface tools are a major business. In the Unix market alone, over $133 million of tools were sold in 1993, which is about 50,000 licenses [X Business Group 1994]. This is a 64% increase over 1992. Forrester Research claims that the total market for UI software tools on all platforms, including "vertical tools" which include database and user interface construction tools, will be 130,000 developers generating $400 million in revenue. They estimate that this will double each year, growing to 700,000 developers and $1.2 billion by 1996 [DePalma and Woodring 1993].

Mark Hanner (private communication) from the Meta Group market research firm says that the user interface tool market is about to explode. Whereas the "first generation" of commercial tools was not fully graphical nor sufficiently powerful, this is no longer true for today's tools. Furthermore, prices for tools have dropped significantly, and fees for run-times have been mostly eliminated (so that designers do not have to pay the tool creator for products created using the tools). For the future, there is still a tremendous opportunity for good tools, especially in niche areas like multimedia, distributed systems, and geographical information systems.

This article surveys user interface software tools, and explains the different types and classifications. There have been many previous surveys on this topic, e.g., Hartson and Hix [1989] and Myers [1989], but since this is a fast-changing field, a new one seemed in order. Also, this article takes a broader approach and includes more components of user interface software, including windowing systems. However, it is now impossible to discuss *all* user interface tools, since there are so many.[1] 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 ACM User Interface Software and Technology Symposium (UIST) and the ACM SIGCHI conferences. 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.

2. 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) [Olsen 1992]. 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 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* tool*kits*.

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 may be a designer, application programmer, or tool creator.

3. 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 will be higher.** This is because:

[1]A partial list, which is frequently updated, is available through Mosaic or other world-wide-web interfaces as http:/ /www.cs.cmu.edu: 8001/afs/cs.cmu.edu/user/bam/www/toolnames.html.

—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.

—There can be multiple user interfaces for the same application.

—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.

(2) **The user interface code will 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 implementors 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 are 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 investigate different designs rapidly,
—allow nonprogrammers to design and implement user interfaces,
—allow the end-user to customize the interface,
—provide portability, and
—be easy to use themselves.

This might be achieved by having the tools:

—automatically choose which user interface styles, input devices, widgets, etc. 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 updates application data,
—deal with 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, and
—evaluate the graphic design and layout, usability, and learnability of the interface.

4. 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 will take with it a number of tools), and the emergence of new styles of user interfaces such as pen-based computing and multimedia.

4.1 Components of User Interface Software

As shown in Figure 1, user interface software may be divided into various layers: the windowing system, the toolkit, and higher-level tools. Of course, many practical systems span multiple layers.

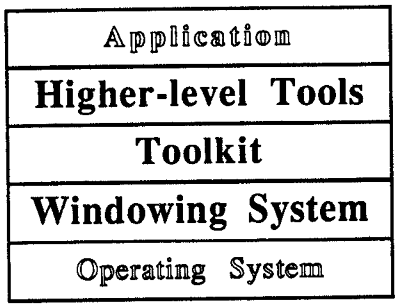


Fig. 1. The components of user interface software discussed in this article.

The *windowing system* supports the separation of the screen into different (usually rectangular) regions, called *windows*. The X system [Scheifler and Gettys 1986] 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 use the term "window*ing* 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.

5. 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 [Myers 1988a]. Although most of today's systems provide toolkits on top of the windowing systems, as will be explained below, generally toolkits address only 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 [Stallman 1979] and the Smalltalk [Tesler 1981] and DLISP [Teitelman 1979] 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 [Myers 1984], SunView for Sun, and the Macintosh, NeXT, 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.

5.1 Structure of Windowing Systems

A windowing system can be logically divided into two layers, each of which has two parts (see Figure 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 includes 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* or *program interface*.

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.

5.2 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 [Scheifle and Gettys 1986] was created to solve this problem by providing a hardware-independent interface to windows. X has been quite successful at this, and has driven virtually 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 Microsoft Windows or IBM's Presentation Manager (part of OS/2).

5.2.1 *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. Most windowing systems provide special escapes that allow programs to draw directly to the screen, without using the window system's clipping. These operations can be much quicker, but are very dangerous and therefore should seldom be used. Most modern computers provide graphics hardware that is specially optimized to work efficiently with the window system.

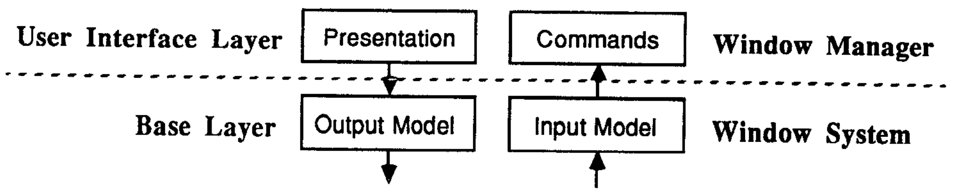


Fig. 2. The windowing system can be divided into two layers, called the *base* or *window system* layer, and the *user interface* or *window manager* layer. Each of these can be divided into parts that handle output and input.

In early windowing systems, such as Smalltalk [Tesler 1981], Blit [Pike 1983], and Sapphire [Myers 1986], the primary output operation was BitBlt (also called "RasterOp"). Primarily these systems 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 [Ingalls 1981]. The only additional drawing operation that was typically supported by these early systems was for 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, etc. 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 [Adobe Systems 1985] 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 objects. Two extensions to support 3D objects are PEX and OpenGL. PEX [Gaskins 1992] is an extension to the X windowing system that incorporates much of the PHIGS graphics standard. OpenGL (by Silicon Graphics, Inc.) is based on the GL programming interface that has been used for many years on Silicon Graphics machines. OpenGL provides machine independence for 3D since it is available for various X platforms (Silicon

Graphics, Inc., Sun, etc.) and will be included as a standard part of new versions of Microsoft Windows.

As shown in Figure 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.

5.2.2 *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. Using various commands, the user can select which window is getting events (described in Section 5.3). Each event record typically contains the type and value of the event (e.g., which key was pressed), the window that the event was directed to, a timestamp, and the x and y position of the mouse. The windowing system 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 2D primitives to 3D), input is still handled in essentially the 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 3D pointing device or one with more

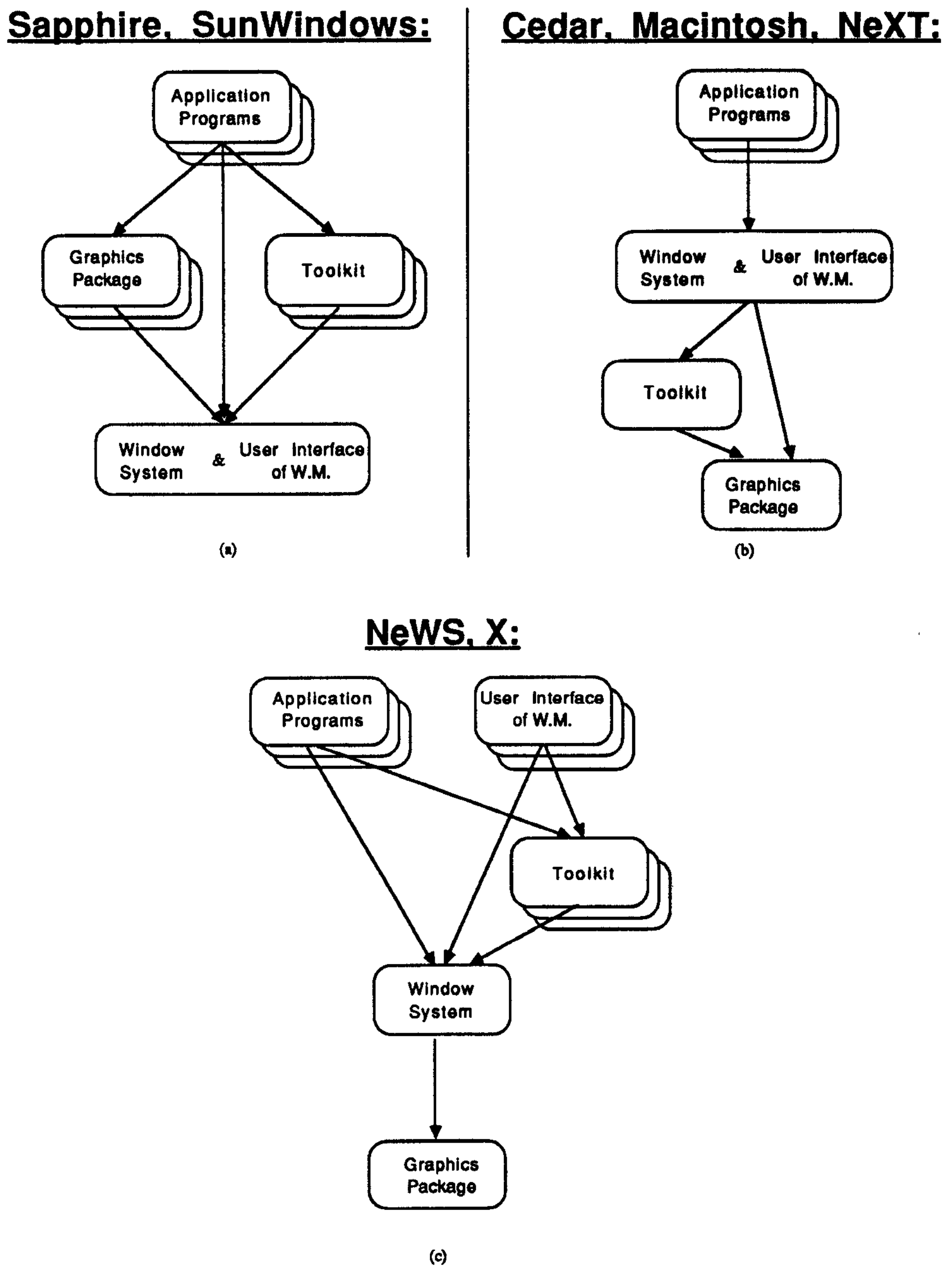


Fig. 3. Various organizations that have been used by windowing systems. Boxes with extra borders represent systems that can be replaced by users. Early systems (a) tightly coupled the window manager and the window system, and assumed that sophisticated graphics and toolkits would be built on top. The next step in designs (b) 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. Other systems (c) allow different window managers and different toolkits, while still embedding sophisticated graphics packages.

than the standard number of buttons was 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 [Myers 1990a]).

5.2.3 *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.

5.3 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 complete discussion of the options for the user interfaces of window managers was previously published [Myers 1988a]. Also, the video *All the Widgets* [Myers 1990b] has a 30-minute 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. Good references for more information are the "Legally Speaking" columns of *Communications of the ACM*, e.g., Samuelson [1993].

5.3.1 *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, as shown in Figure 4. 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.[2] The other alternative is called *tiled* windows, which means that windows are not allowed to cover each other. Figure 5 shows an example of tiled windows. 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.

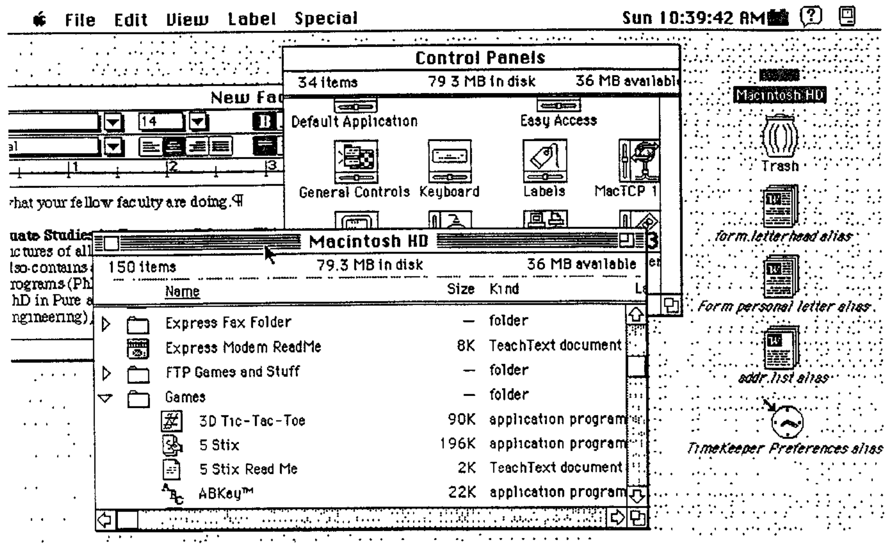
The tiled style was popular for awhile, and was used by Cedar [Swinehart et al. 1986], and early versions of the Star [Smith et al. 1982], Andrew [Palay et al. 1988], and Microsoft Windows. A study even suggested that using tiled windows was more efficient for users [Bly and Rosenberg 1986]. However, today tiled windows are rarely seen, because generally users prefer overlapping.

Another important aspect of the presentation of windows is the use of *icons* (also shown in Figures 4 and 5). These are small pictures that represent windows (or sometimes files). They are used because there would otherwise be too many windows to manage and fit conveniently on the screen. 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.

5.3.2 *Commands.* Since computers have multiple windows and typically 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.

[2]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 [Smith et al. 1982].


File Edit View Label Special
Sun 10:39:42 AM
Control Panels
34 items 79.3 MB in disk 36 MB available
Default Application
Easy Access
General Controls
Keyboard
Labels
MacTCP
Macintosh HD
Trash
form.letterhead alias
Form personal letter alias
addr.list alias
TimeKeeper Preferences alias
New Fa
that your fellow faculty are doing.¶

Macintosh HD

150 items 79.3 MB in disk 36 MB available

Name	Size	Kind
Express Fax Folder	—	folder
Express Modem ReadMe	8K	TeachText document
FTP Games and Stuff	—	folder
Games	—	folder
3D Tic-Tac-Toe	90K	application program
5 Stix	196K	application program
5 Stix Read Me	2K	TeachText document
ABKey™	22K	application program

Fig. 4. A screen from the Macintosh showing three windows covering each other, and some icons along the right margin.

—*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 menu bar 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 menu bar 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 to send a window to the bottom (covered by all other windows). Other commands allow windows to be changed size, moved, created, and destroyed.

6. TOOLKITS

A *toolkit* is a library of "widgets" that can be called by application programs. A *widget* is a way of 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, etc. Figure 6 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

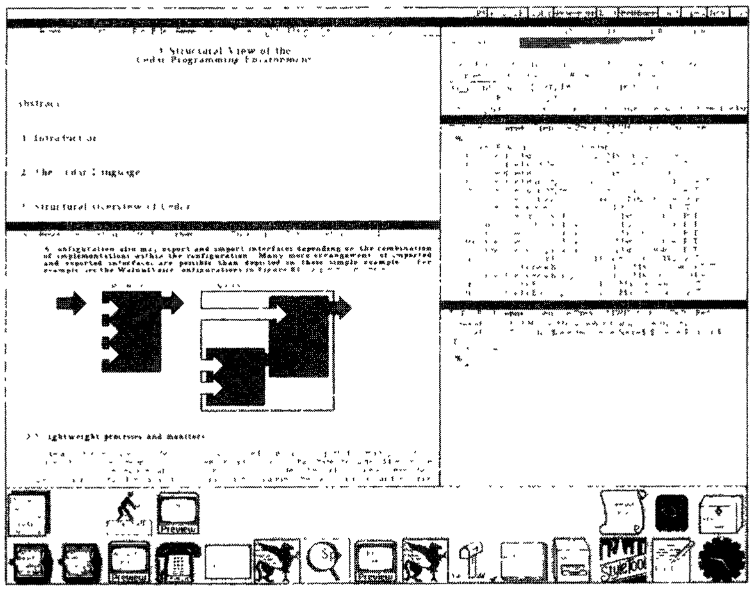


Fig. 5. A screen from the Cedar windowing system. Windows are "tiled" into 2 columns. There is a row of icons along the bottom. Each window has a fixed menu of commands below the title line. Reprinted from Swinehart et al., *ACM Trans. Program. Lang. Syst. 8*, 4 (Oct.), 452.

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. Additionally, the toolkits themselves are often expensive to create: "The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate" [Cardelli and Pike 1985, 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 covers six books, of which about 1/3 is related to user interface programming.

As with the graphics package, the toolkit can be implemented either using or being used by the windowing system (see Figure 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 xt [McCormack and Asente 1988], InterViews [Linton et al. 1989], Garnet [Myers et al. 1990], or tk [Ousterhout 1991] toolkits can be

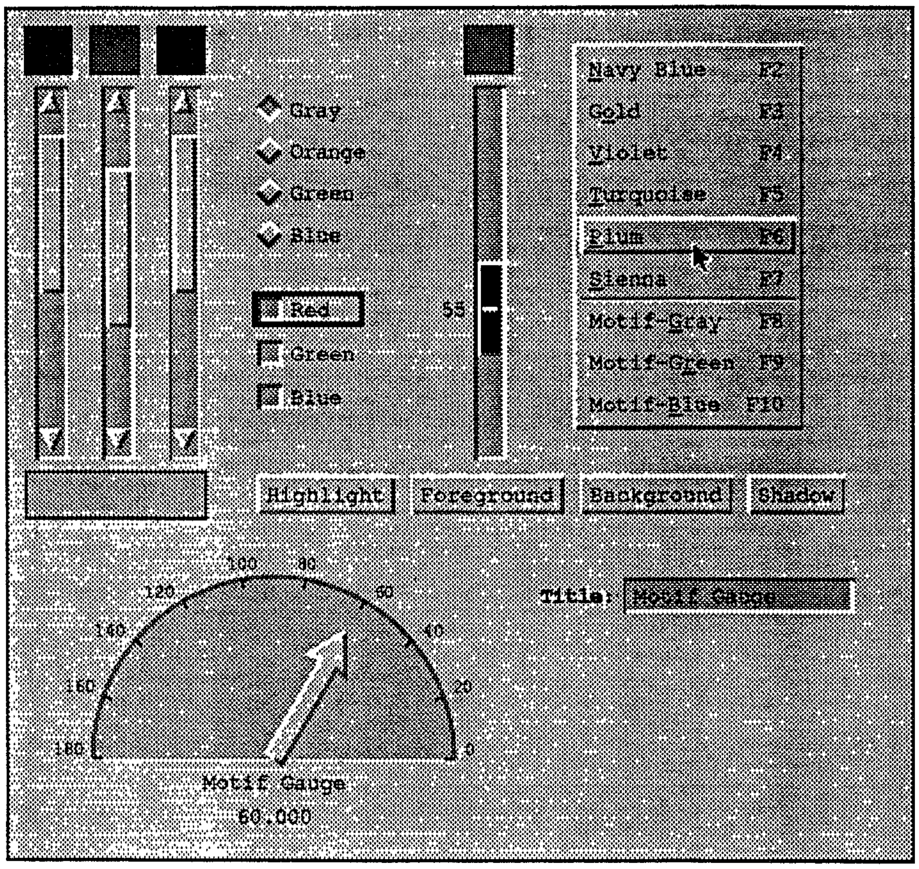


Fig. 6. Some of the widgets with a Motif look and feel provided by the Garnet toolkit.

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 [McCormack and Asente 1988]. 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 (Figure 7(a)), or else the same look-and-feel can be implemented on top of different intrinsics (Figure 7(b)). Recently, Sun announced that it was phasing out OpenLook, which means that X and xt will be standardized on the Motif widget set.

6.1 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 Sun Microsystems SunView

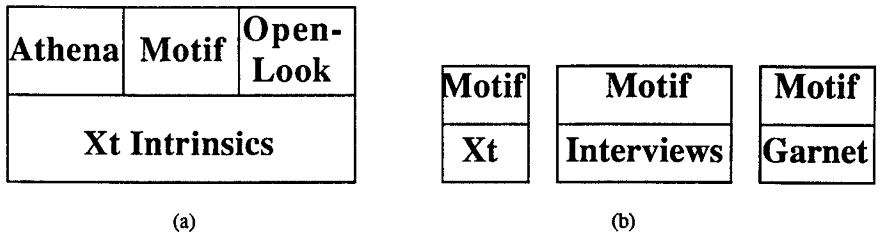


Fig. 7. (a) At least three different widget sets that have different looks and feels have been implemented on top of the xt intrinsics. (b) The Motif look and feel has been implemented on at least three different intrinsics.

windowing system and the Macintosh Toolbox [Apple Computer 1985]. The other variety uses an object-oriented programming style which makes it easier for the designer to customize the interaction techniques. Examples include Smalltalk, Andrew, Garnet, InterViews, and Xt.

The advantages of using object-oriented intrinsics are (1) that it is a natural way to think about widgets (the menus and buttons on the screen *seem* like objects), (2) the widget objects can handle some of the chores that otherwise would be left to the programmer (such as refresh), and (3) 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, and Garnet, or it might use an existing object system, as was done in InterViews [Linton et al. 1989] which uses C++, NeXTStep (NeXT, Inc.) which uses Objective-C, and Rendezvous [Hill et al. 1993] which uses CLOS (the standard Common Lisp Object System).

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 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 contain often hundreds of call-backs, which makes the code harder to modify and maintain [Myers and Rosson 1992]. Additionally, different toolkits, even when implemented on the same intrinsics like Motif and OpenLook, have different call-back protocols. This means that 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 [Myers 1991a].

Some research toolkits have added novel things to the toolkit intrinsics. For example, Garnet [Myers et al. 1990], Rendezvous [Hill 1993], and Bramble [Gleicher 1993] 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 rectangle if the user moves the slider.

6.2 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 [Myers 1990b]. For example, it shows 35 different kinds of menus. Like window manager user interfaces, the widgets' look-and-feel can be copyrighted and patented [Samuelson 1993].

As was mentioned above, different widget sets (with different looks and feels) can be implemented on top of the same intrinsics. Also, 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 Garnet intrinsics (Figure 7(b)). Although all look and operate the same (so would be indistinguishable to the user), they are implemented quite differently, and have completely different procedural interfaces for the programmer.

6.3 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 [Pausch et al. 1992] (which contains a toolkit and an interface builder), is specifically designed to be easy to learn and is aimed at classroom instruction. Garnet [Myers et al. 1990] provides high-level support for graphical, direct-manipulation interfaces, and includes a toolkit, interface builder, and other high-level tools. Rendezvous [Hill et al. 1993] is designed to make it easier to create applications that support multiple users on multiple machines operating synchronously. Whereas most toolkits only provide 2D interaction techniques, the Brown Animation Generation System [Zeleznik et al. 1991] and Silicon Graphics' Inventor toolkit [Strauss and Carey 1992; Wernecke 1994] provide preprogrammed 3D widgets and a framework for creating others. The Ttoolkit [Guimaraes et al. 1992] provides built-in primitives for controlling the timing of an interface, which is important for supporting multimedia, such as video. Special support for animations has been added to Artkit, including motion blur, timing, and curved trajectories [Hudson and Stasko 1993].

Tk [Ousterhout 1991] is a popular toolkit for the X window system because it uses an interpretive language called tcl which makes it possible to change the user interface dynamically. Tcl also supports the Unix style of programming where many small programs are glued together.

7. 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, etc. Although there are fewer windowing systems and toolkits than there were five years ago, people are still finding that they must do a lot of work to convert their software from Motif to OpenLook 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. A recent report [Chimera 1993] compares a number of virtual toolkits.

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 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 by Visix Software Inc. and Open Interface by Neuron Data provide libraries of widgets that look like those on the various platforms. 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 provide functions that only appear in all toolkits. Many of the virtual toolkits that take the second approach, for example 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 library.

All of the toolkits that work on multiple platforms can be considered virtual toolkits of the second type. For example, SUIT [Pausch et al. 1992] works on X, Macintosh, and Windows, and Garnet [Myers et al. 1990] works on X and the Macintosh. 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.

8. 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.

8.1 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 includes usually 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* 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 that have tried, such as MIKE [Olsen and Halversen 1988], 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 creating cognitive models automatically from high-level descriptions of the user interface. For example, USAGE creates an NGOMSL cognitive model from a UIDE user interface specification [Byrne et al. 1994].

8.2 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. As shown in Figure 8, 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 next. 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.

8.2.1 *Language Based.* With most of the older user interface tools, the designer specified 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, etc. The language is usually used to specify the syntax of the user interface, i.e., the legal sequences of input and output actions. This is sometimes called the "dialog." Green [1986] provides an extensive comparison of grammars, state transition diagrams, and event languages, and Olsen [1992] surveys various UIMS techniques.

8.2.1.1 *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 [1968] implemented a simple tool using finite-state machines which handled textual input. This was

Specification Format	Examples	Section
Language Based		8.2.1
State Transition Networks	VAPS	8.2.1.1
Context-Free Grammars	YACC LEX Syngraph	8.2.1.2
Event Languages	ALGAE Sassafras HyperTalk	8.2.1.3
Declarative Languages	Cousin Open Dialog Motif UIL	8.2.1.4
Constraint Languages	Thinglab C32	8.2.1.5
Screen Scrapers	Easel	8.2.1.6
Database Interfaces	Oracle	8.2.1.7
Visual Programming	LabView Prograph Visual Basic	8.2.1.8
Application Frameworks	MacApp Unidraw	8.2.2
Model-Based Generation	MIKE UIDE ITS Humanoid	8.2.3
Interactive Graphical Specification		8.2.4
Prototypers	Bricklin's Demo Director HyperCard	8.2.4.1
Cards	Menulay HyperCard	8.2.4.2
Interface Builders	DialogEditor NeXT Interface Builder Prototyper UIMX	8.2.4.3
Data Visualization Tools	DataViews	8.2.4.4
Graphical Editors	Peridot Lapidary Marquise	8.2.4.5

Fig. 8. Ways to specify the user interface, some tools that use the technique, and the section of this article that discusses the technique.

apparently the first user interface tool. Many of the assumptions and techniques used in modern systems were present in Newman's: 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 dialog 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. Additionally, 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 [1986] invented a new formalism, which is a combination of state diagrams with a form of event languages (see Section 8.2.1.3). There can be multiple diagrams active at the same time, and flow of control transfers from one to another in a coroutine fashion. The system can create various forms of direct-manipulation interfaces. VAPS is a commercial system by Virtual Prototypes Inc. 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. Transition networks have been thoroughly researched, but have not proven particularly successful or useful either as a research or commercial approach.

8.2.1.2 *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 BNF. Examples of grammar-based systems are Syngraph [Olsen and Dempsey 1983] 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 have been mostly abandoned for user interfaces by researchers and commercial developers.

8.2.1.3 *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.

The ALGAE system [Flecchia and Bergeron 1987] uses an event language which is an extension of Pascal. The user interface is programmed as a set of small event handlers, which ALGAE compiles into conventional code. Sassafras [Hill 1986], uses a similar idea, but with an entirely different syntax. Sassafras also adds local variables called "flags" to help specify the flow of control. As described in Section 8.2.4.2, the HyperTalk language that is part of HyperCard for the Apple Macintosh can also be considered an event language.

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, 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 and similar tools shows that this approach is appropriate for small to medium-size programs.

8.2.1.4 *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 [Hayes et al. 1985] and HP/Apollo's Open-Dialogue [Schulert et al. 1985] 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 in Section 8.2.3.

Another type of declarative language is the layout description languages that come with many toolkits. 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 changing dynamically the items in a menu based on the application mode or context. However, these languages are now proving successful as intermediate languages describing the layout of widgets (such as UIL) that are generated by interactive tools. They were also an important intermediate research step on the way to today's model-based approaches (Section 8.2.3).

8.2.1.5 *Constraint Languages.* A number of user interface tools allow the programmer to use constraints to define the user interface [Borning and Duisberg 1986]. Early constraint systems include Sketchpad [Sutherland 1963] which pioneered the use of graphical constraints in a drawing editor, and Thinglab [Borning 1981] which used constraints for graphical simulation. Subsequently, Thinglab was extended to aid in the generation of user interfaces [Borning and Duisberg 1986].

Section 6.1 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 [Myers et al. 1990]). Additionally, some systems have provided higher-level interfaces to constraints. Graphical Thinglab [Borning 1986] allows the designer to create constraints by wiring icons together, and NoPump [Wilde and Lewis 1990], C32 [Myers 1991b], and Penguims [Hudson 1993] allow constraints to be defined using spreadsheet-like interfaces.

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, etc. However, a disadvantage with constraints is that they require a sophisticated run-time system to solve them efficiently. Another problem is that they can be hard to debug when specified incorrectly since it can be difficult to trace the causes and consequences of values changing. 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 constraints.

8.2.1.6 *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 designer-written program 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. The leading program of this type is Easel, which also contains an interface builder for laying out the widgets.

8.2.1.7 *Database Interfaces.* A very important class of commercial tools support form-based or GUI-based access to databases. Major database vendors such as Oracle 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), that support defining the interactive forms for accessing and entering data, fall into this category also.

8.2.1.8 *Visual Programming.* "Visual programs" use graphics and two-dimensional (or more) layout as part of the program specification [Myers 1990c]. Many different approaches to using visual programming to specify user interfaces have been investigated. Most systems that support state transition networks (Section 8.2.1.1) 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 (Section 8.2.4.3). Examples of visual programming systems for creating user interfaces include Labview by National Instruments which is specialized for controlling laboratory instruments, and Pro-Graph. 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 papers (e.g., Myers [1990c]) have analyzed the strengths and weaknesses of visual programming in detail.

Another popular language is Visual Basic from Microsoft. Although this is more of a structure editor for Basic combined with an interface builder, and therefore does not really count as a visual language, it does make the construction of user interface software easier. Microsoft is pushing Visual Basic as the extension language that people will use to customize and connect Windows-based applications.

8.2.1.9 *Summary of Language Approaches.* In summary, there have been many different types of languages that 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 [Olsen 1987]. This has been confirmed by market research [X Business Group 1994, p. 29]. Furthermore, it seems more natural to define the graphical part of a user interface using a graphical editor (see Section 8.2.4). 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, a new book is entirely devoted to investigating the languages for programming user intérfaces [Myers 1992a].

8.2.2 *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. Therefore, they created a software system that provides an overall *application framework* to guide programmers. This is called MacApp [Schmucker 1986; Wilson 1990] and uses the object-oriented language Object Pascal. Classes are provided for the important parts of an application, such as the main windows, the commands, etc., 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 has been very successful at simplifying the writing of Macintosh applications.

Unidraw [Vlissides and Linton 1990] uses a similar approach, 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 [Linton et al. 1989]. Unidraw has been used to create various drawing and CAD programs, and a user interface editor [Vlissides and Tang 1991]. The Garnet framework 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) [Myers et al. 1992b]. The ACE system from HP provides an interactive editor that allows some of the properties of objects to be specified, but most of the application-specific behavior must still be programmed [Johnson et al. 1993]. Even more specialized are various graph programs, such as Edge [Newbery 1988] and TGE [Karrer and Scacchi 1990]. These provide a framework in which the designer can create programs that display their data as trees or graphs. Typically, the programmer 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, monolythic 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 [Palay et al. 1988] 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 is 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 next generation of operating systems will use this approach extensively: it is the foundation for Microsoft's OLE and Apple's OpenDoc.

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.

Another class of systems that might be considered "frameworks" helps create user interfaces that are composed of a series of "cards," such as HyperCard from Apple. These systems are discussed in Section 8.2.4.2 because their primary interface to the designer is graphical.

8.2.3 *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, including Mickey [Olsen 1989], Jade [Vander Zanden and Myers 1990], Chisel [Singh and Green 1989], and DON [Kim and Foley 1993] have concentrated on creating menus and dialog boxes. Chisel and Jade allow 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, columness, symmetry, grouping, etc. Creating dialog 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 [Olsen 1986] 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, like those described in Section 8.2.4, to specify these changes.

UIDE (the User Interface Design Environment) [Sukaviriya et al. 1993] requires that the semantics of the application be defined in a special-purpose language, and therefore might be included with the language-based tools (Section 8.2.1). 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 postconditions of the operations, and the system uses these to reason about the operations, and to generate an interface automatically. 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 insures that the resulting interface remains consistent. Another feature of UIDE is that the pre- and postconditions are used to generate help automatically [Sukaviriya and Foley 1990]. One direction of current research is to make UIDE models easier to create by allowing users to demonstrate some parts of the interface [Frank and Foley 1993].

Another model-based system is HUMANOID [Szekely et al. 1993] 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 [Myers et al. 1990], 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 that integrates their approaches [Neches et al. 1993].

The ITS [Wiecha et al. 1990] 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 get 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.

While 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 not good enough. A further problem is that the specification languages can be quite hard to learn and use. Extensive current research is addressing the problems of expanding the range of what can be created automatically (to go beyond dialog boxes) and to make the model-based approach easier to use.

8.2.4 *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" (Section 8.2.1.8) since with direct graphical specification, the actual user interface (or a part of it) is drawn, rather than being generated indirectly by 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.

8.2.4.1 *Prototyping Tools.* The goal of *prototyping tools* is to allow the designer to mock up quickly 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 buzz-word. 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 quicker. 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, by Sage Software Inc. 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 like 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 Director, by MacroMedia, for the Macintosh 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 for the Macintosh is also often used as a prototyping tool.

The primary disadvantage of these prototyping tools is that they cannot create the actual code for the user interface. Therefore, the interfaces must be recoded after prototyping. There is also the risk that the programmers who implement the real user interface will ignore the prototype. Therefore, a new research tool is trying to provide a quick sketching interface and then convert the sketches into actual widgets [Landay and Myers 1995].

8.2.4.2 *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 were coded by hand.

An early example of this is Menulay [Buxton et al. 1983], which allows the designer to place text, graphical potentiometers, iconic pictures, and light buttons on the screen and see exactly what the user will see when the application is run. The designer does not need to be a programmer to use Menulay. Trillium [Henderson 1986], which is aimed at designing the user interface panels for photocopiers, is very similar to Menulay. One strong advantage that Trillium has over Menulay is that the cards can be executed immediately as they are designed since the specification is interpreted rather than compiled. Trillium also separates the behavior of interactions from the graphic presentation and allows the designer to change the graphics (while keeping the same behavior) without programming. One weakness is that it has little support for frame-to-frame transitions, since this rarely is necessary for photocopiers.

Probably, the most famous example of a card-based system is HyperCard from Apple. There are now many similar programs, such as GUIDE by Owl International, Inc., Spinnaker Plus by Spinnaker Software, and Tool Book by Asymetrix Corp. In all of these, the designer can easily create cards containing text fields, buttons, etc., along with various graphic decorations. The buttons can transfer 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 in Section 8.2.1.3, is really an event language, since the programmer writes short pieces of code that are executed when input events occur.

8.2.4.3 *Interface Builders*. An *interface builder* allows the designer to create dialog boxes, menus, and windows that are to be part of a larger user interface. These are also called *Interface Development Tools*. Interface builders allow the designer to select from a predefined library of widgets, and 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 subdialogs 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 [Stevens et al. 1983]. Other examples of research interface builders are DialogEditor [Cardelli 1988], vu [Singh and Green 1988], and Gilt [Myers 1991a]. There are literally hundreds of commercial interface builders. Just a few examples are the NeXT Interface Builder, Prototyper for the Macintosh by Smethers Barnes, WindowsMAKER for Microsoft Windows on the PC by Blue Sky Software Corp., UIMX for X Windows and Motif by Visual Edge Software Ltd., and devGuide from Sun Microsystems for OpenLook. 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, UIMX generates a UIL description. 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 dialog 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 dialog 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 UIMX, provide a C code interpreter.

8.2.4.4 *Data Visualization Tools.* An important commercial category of tools are 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 by V. I. Corp., SL-GMS by SL Corp., and VAPS by Virtual Prototypes Inc.

8.2.4.5 *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 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" [Myers 1992b]. 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 [Myers 1988b] 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 like those that might be found in toolkits. Experiments showed that Peridot can be used by nonprogrammers. Lapidary [Myers et al. 1989] 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 [Fisher et al. 1992] allows some dynamic, run-time properties of the objects to be demonstrated, such as how objects are created. The Marquise tool [Myers et al. 1993] allows the designer to demonstrate *when* various behaviors should happen, and supports palettes which control the behaviors. Research continues on making these ideas practical.

8.3 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.,

$20,000 to $50,000). For example, in the aeronautics and real-time control areas, there are a number of high-level tools, including AutoCode by Integrated Systems and InterMAPhics by Prior Data Sciences.

9. TECHNOLOGY TRANSFER

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

10. 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.

10.1 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 dialog boxes and menus, an interface builder is very useful, and 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 dialog boxes and menus; and graphical editors will allow the graphical elements of the user interface to be drawn.

There is a big debate going on about the model-based and direct graphical specification approaches [Sukaviriya et al. 1994; Wiecha et al. 1989]. The model-based tools provide a top-down (or "application-out") approach where the functions are specified first, whereas the graphical tools provide a bottom-up (or "user-interface-in") approach where the user interface is designed before the functions. Furthermore, the automatic, model-based approaches seem to provide too little flexibility to the designer, whereas the graphical tools provide too much flexibility and not enough guidance. Some researchers are trying to create systems that combine the approaches to try to achieve the advantages of both [Frank and Foley 1993].

10.2 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 dialog 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?

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 interfaces.

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 tk from Berkeley and Garnet from CMU. Most personal computers and workstations today come with a free toolkit. Commercial higher-level tools can range from $200 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 [Myers and Rosson 1992], 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 *Open Systems Today* for Unix and *PC Magazine*. Market research firms are writing reports evaluating various tools [Depalma and Woodring 1993; X Business Group 1994]. Also, there are a few formal studies of tools [Hix 1989].

11. RESEARCH ISSUES

Although there are many user interface tools, there are plenty of areas in which further research is needed. A report prepared for an NSF study discusses future research ideas for user interface tools at length [Olsen et al. 1993]. Here, a few of the important ones are summarized.

11.1 New Programming Languages

The built-in input/output primitives in today's programming languages 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. 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. A new book discusses how programming languages can be improved to provide better support for user interface software [Myers 1992a].

11.2 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 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 [Landay and Myers 1995]. For example, an important first step in user interface design is task analysis, where the designer identifies the particular tasks that the 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 generate documentation and run-time help automatically.

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 [Hartson et al. 1990] is a notation for expressing the 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 [Olsen and Halversen 1988], have highlighted the need for better models and metrics against which to evaluate the user interfaces. Research in this area is continuing by cognitive psychologists and other user interface researchers (e.g., Byrne et al. [1994]).

11.3 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 two-dimensional objects, but there is already a demand to provide 3D visualizations and animations. 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" like 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" 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 like video [Nielsen 1993]. 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.

A more immediate concern is for 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 may include different number, date, and time formats, new input methods, redesigned layouts, different color schemes, and new icons [Russo and Boor 1993]. How can future tools help with this process?

11.4 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. More effective mechanisms for users to customize existing applications and create new ones are needed [Myers et al. 1992a]. 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 should be part of the underlying toolkit. Naturally, since this is aimed at users, it will not be like programming in C, but rather at some higher level.

The X Business Group predicts that there will be an increased use of tools by end-users, rather than professional software developers, which will present enormous opportunities and challenges to tool creators.

There are many levels at which users might want to modify these "malleable interfaces": simple changing of menus and properties, direct programming of new functions like in spreadsheets, or connecting together prebuilt components, as in the Andrew and OLE frameworks. Future UI tools should support changes at all of these levels.

11.5 Application and UI Separation

One of the fundamental goals of user interface tools is to allow the better modularization and separation of user interface code from application code. However, a recent survey reported that modern toolkits actually make this separation more difficult, due to the large number of call-back procedures required [Myers and Rosson 1992]. Therefore, further research is needed into ways to modularize the code better, and how tools can support this.

11.6 Tools for the Tools

It is very difficult to create the 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 is exploring mechanisms specifically designed to make high-level graphical tools easier to create [Myers and Vander Zanden 1992]. The Unidraw framework has also proven useful for creating interface builders [Vlissides and Tang 1991]. However, more work is needed.

12. CONCLUSIONS

The area of user interface tools is expanding rapidly. Five years ago, you would have been hard-pressed to find any successful commercial higher-level tools, but now there are over 100 different tools, and tools are turning into a billion dollar 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 near future.

REFERENCES

ADOBE SYSTEMS 1985. *Postscript Language Reference Manual.* Addison-Wesley, Reading, Mass.

APPLE COMPUTER 1985. *Inside Macintosh.* Addison-Wesley, Reading, Mass.

BLY, S. A. AND ROSENBERG, J. K. 1986. A comparison of tiled and overlapping windows. In *Human Factors in Computing Systems, Proceedings SIGCHI'86* (Boston, Mass., Apr.). ACM, New York, 101–106.

BOOZ ALLEN AND HAMILTON 1992. NeXTStep vs. other development environments. Booz Allen and Hamilton, Inc. Report available from NeXT, Inc.

BORNING, A. 1981. The programming language aspects of Thinglab. A constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct.), 353–387.

BORNING, A. AND DUISBERG, R. 1986. Constraint-based tools for building user interfaces. *ACM Trans. Graph. 5*, 4 (Oct.), 345–374.

BORNING, A. 1986. Defining constraints graphically. In *Human Factors in Computing Systems, Proceedings SIGCHI'86.* ACM, New York, 137–143.

BUXTON, W., LAMB, M. R., SHERMAN, D., AND SMITH, K. C. 1983. Towards a comprehensive user interface management system. *Comput. Graph. 17*, 3, 35–42.

BYRNE, M. D., WOOD, S. D., SUKAVIRIYA, P., FOLEY, J. D., AND KIERAS, D. E. 1994. Automating interface evaluation. In *Human Factors in Computing Systems, Proceedings SIGCHI'94.* ACM, New York, 232–237.

CARDELLI, L. 1988. Building user interfaces by direct manipulation. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88* (Banff, Alberta, Canada, Oct.). ACM, New York, 152–166.

CARDELLI, L. AND PIKE, R. 1985. Squeak: A language for communicating with mice. In *SIGGRAPH '85. Comput. Graph. 19,* 3 (July), 199–204.

CHIMERA, R. 1993. Evaluation of platform independent user interface builders. Tech. Rep. Working Paper 93-09, Human-Computer Interaction Laboratory, Univ. of Maryland.

DEPALMA, D. A. AND WOODRING, S. D. 1993. Client/server power tools futures. *Softw. Strat. Rep. 4*, 1 (Apr.), 2–13. This is available only from Forrester Research, Cambridge, Mass.

FISHER, G. L., BUSSE, D. E., AND WOLBER, D. A. 1992. Adding rule-based reasoning to a demonstrational interface builder. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'92* (Monterey, Ca., Nov.). ACM, New York, 89–97.

FLECCHIA, M. A. AND BERGERON, R. D. 1987. Specifying complex dialogs in ALGAE. In *Human Factors in Computing Systems, CHI + GI'87* (Toronto, Ont., Canada, Apr.). ACM, New York, 229–234.

FRANK, M. R. AND FOLEY, J. D. 1993. Model-based user interface design by example and by interview. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93* (Atlanta, Ga., Nov.). ACM, New York, 129–137.

GASKINS, T. 1992. *PEXlib Programming Manual.* O'Reilly and Associates, Inc., Sebastopol, Calif.

GLEICHER, M., 1993. A graphics toolkit based on differential constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93* (Atlanta, Ga., Nov.). ACM, New York, 109–120.

GREEN, M. 1986. A survey of three dialog models. *ACM Trans. Graph. 5*, 3 (July), 244–275.

GUIMARAES, N. M., CORREIA, N. M., AND CARMO, T. A. 1992. Programming time in multimedia user interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'92* (Monterey, Ca., Nov.). ACM, New York, 125–134.

HARTSON, H. R. AND HIX, D. 1989. Human-computer interface development: Concepts and systems for its management. *ACM Comput. Surv. 21*, 1 (Mar.), 5–92.

HARTSON, H. R., SIOCHI, A. C., AND HIX, D. 1990. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Trans. Inf. Syst. 8*, 3 (July), 181–203.

HAYES, P. J., SZEKELY, P. A., AND LERNER, R. A. 1985. Design alternatives for user interface management systems based on experience with COUSIN. In *Human Factors in Computing Systems, Proceedings SIGCHI'85* (San Francisco, Ca., Apr.). ACM, New York, 169–175.

HENDERSON, D. A., JR. 1986. The Trillium user interface design environment. In *Human Factors in Computing Systems, Proceedings SIGCHI'86.* ACM, New York, 221–227.

HILL, R. D. 1993. The Rendezvous constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93* (Atlanta, Ga., Nov.). ACM, New York, 225–234.

HILL, R. D. 1986. Supporting concurrency, communication and synchronization in human-computer interaction—The Sassafras UIMS. *ACM Trans. Graph. 5*, 3 (July), 179–210.

HILL, R. D., BRINCK, T., PATTERSON, J. F., ROHALL, S. L., AND WILNER, W. T. 1993. The Rendezvous language and architecture. *Commun. ACM 36*, 1 (Jan.), 62–67.

HIX, D. 1989. A procedure for evaluating human-computer interface development tools. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89* (Williamsburg, Va., Nov.). ACM, New York, 53–61.

HUDSON, S. E. 1993. User interface specification using an enhanced spreadsheet model. Tech. Rep. GIT-GVU-93-20, Georgia Tech Graphics, Visualization and Usability Center.

HUDSON, S. E. AND STASKO, J. T. 1993. Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93* (Atlanta, Ga., Nov.). ACM, New York, 57–67.

INGALLS, D. H. H. 1981. The Smalltalk graphics kernel. *Byte Mag. 6*, 8 (Aug.), 168–194.

JACOB, R. J. K. 1986. A specification language for direct manipulation interfaces. *ACM Trans. Graph. 5*, 4 (Oct.), 283–317.

JOHNSON, J. A., NARDI, B. A., ZARMER, C. L., AND MILLER, J. R. 1993. ACE: Building interactive graphical applications. *Commun. ACM 36*, 4 (Apr.), 41–55.

KARRER, A. AND SCACCHI, W. 1990. Requirements for an extensible object-oriented tree/graph editor. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90* (Snowbird, Utah, Oct.). ACM, New York, 84–91.

KIM, W. C. AND FOLEY, J. D. 1993. Providing high-level control and expert assistance in the user interface presentation design. In *Human Factors in Computing Systems, Proceedings INTERCHI'93* (Amsterdam, The Netherlands, Apr.). ACM, New York, 430–437.

LANDAY, J. A. AND MYERS, B. A. 1995. Interactive sketching for the early stages of user interface design. In *Human Factors in Computing Systems, Proceedings of SIGCHI '95*. ACM, New York.

LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. 1989. Composing user interfaces with InterViews. *IEEE Comput. 22*, 2 (Feb.), 8–22.

MCCORMACK, J. AND ASENTE, P. 1988. An overview of the X toolkit. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88* (Banff, Alberta, Canada, Oct.). ACM, New York, 46–55.

MYERS, B. A. 1994. Challenges of HCI design and implementation. *ACM Interactions 1*, 1.

MYERS, B. A. (ED.). 1992a. *Languages for Developing User Interfaces*. Jones and Bartlett, Boston.

MYERS, B. A. 1992b. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Comput. 25*, 8 (Aug.), 61–73.

MYERS, B. A. 1991a. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91*, (Hilton Head, SC, Nov.). ACM, New York, 211–220.

MYERS, B. A. 1991b. Graphical techniques in a spreadsheet for specifying user interfaces. In *Human Factors in Computing Systems, Proceedings SIGCHI'91* (New Orleans, La., Apr.). ACM, New York, 243–249.

MYERS, B. A. 1990a. A new model for handling input. *ACM Trans. Inf. Syst. 8*, 3 (July), 289–320.

MYERS, B. A. 1990b. All the Widgets. *SIGGRAPH Vid. Rev. 57*.

MYERS, B. A. 1990c. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput. 1*, 1 (Mar.), 97–123.

MYERS, B. A. 1989. User interface tools: Introduction and survey. *IEEE Softw. 6*, 1 (Jan.), 15–23.

MYERS, B. A. 1988a. A taxonomy of user interfaces for window managers. *IEEE Comput. Graph. Appl. 8*, 5 (Sept.), 65–84.

MYERS, B. A. 1988b. *Creating User Interfaces by Demonstration*. Academic Press, Boston.

MYERS, B. A. 1986. A complete and efficient implementation of covered windows. *IEEE Comput. 19*, 9 (Sept.), 57–67.

MYERS, B. A. 1984. The user interface for Sapphire. *IEEE Comput. Graph. Appl. 4*, 12 (Dec.), 13–23.

MYERS, B. A. AND ROSSON, M. B. 1992. Survey on user interface programming. In *Human Factors in Computing Systems, Proceedings SIGCHI'92* (Monterey, Ca., May). ACM, New York, 195–202.

MYERS, B. A. AND VANDER ZANDEN, B. 1992. Environment for rapid creation of interactive design tools. *Int. J. Comput. Graph. 8*, 2 (Feb.), 94–116.

MYERS, B. A., SMITH, D. C., AND HORN, B. 1992a. Report of the 'End-User Programming' working group. In *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, 343–366.

MYERS, B. A., GIUSE, D., AND VANDER ZANDEN, B. 1992b. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. *Sigplan Not. 27*, 10 (Oct.), 184–200.

MYERS, B. A., GIUSE, D. A., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D. S., PERVIN, E., MICKISH, A., AND MARCHAL, P. 1990. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Comput. 23*, 11 (Nov.), 71–85.

MYERS, B. A., MCDANIEL, R. G., AND KOSBIE, D. S. 1993. Marquise: Creating complete user interfaces by demonstration. In *Human Factors in Computing Systems, Proceedings INTER-CHI'93*, (Amsterdam, The Netherlands, Apr.). ACM, New York, 293–300.

MYERS, B. A., VANDER ZANDEN, B., DANNENBERG, R. B. 1989. Creating graphical interactive application objects by demonstration. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89* (Williamsburg, Va., Nov.). ACM, New York, 95–104.

NECHES, R., FOLEY, J., SZEKELY, P., SUKAVIRIYA, P., LUO, P., KOVACEVIC, S., AND HUDSON, S. 1993. Knowledgeable development environments using shared design models. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*. ACM, New York, 63–70.

NEWBERY, F. J. 1988. An interface description language for graph editors. In *1988 IEEE Workshop on Visual Languages*. IEEE Computer Society, Washington, D.C., 144–149.

NEWMAN, W. M. 1968. A system for interactive graphical programming. In *AFIPS Spring Joint Computer Conference*. AFIPS, Montvale, N.J., 47–54.

NIELSEN, J. 1993. Noncommand user interfaces. *Commun. ACM 36*, 4 (Apr.), 83–99.

OLSEN, D. R., JR. 1992. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufmann, San Mateo, Calif.

OLSEN, D. R., JR. 1989. A programming language basis for user interface management. In *Human Factors in Computing Systems, Proceedings SIGCHI'89* (Austin, Tex., Apr.). ACM, New York, 171–176.

OLSEN, D. R., JR. 1987. Larger issues in user interface management. *Comput. Graph. 21*, 2 (Apr.), 134–137.

OLSEN, D. R., JR. 1986. Mike: The Menu Interaction Kontrol Environment. *ACM Trans. Graph. 5*, 4 (Oct.), 318–344.

OLSEN, D. R., JR. AND DEMPSEY, E. P. 1983. Syngraph: A graphical user interface generator. In *SIGGRAPH '83. Comput. Graph. 17*, 3 (July), 43–50.

OLSEN, D. R., JR. AND HALVERSEN, B. W. 1988. Interface usage measurements in a user interface management system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88* (Banff, Alberta, Canada, Oct.). ACM, New York, 102–108.

OLSEN, D. R., JR., FOLEY, J. D., HUDSON, S. E., MILLER, J. AND MYERS, B. 1993. Research directions for user interface software tools. *Behav. Inf. Tech. 12*, 2 (Mar.–Apr.), 80–97.

OUSTERHOUT, J. K. 1991. An X11 toolkit based on the Tcl language. In *Winter USENIX*. USENIX Assoc., Berkeley, Calif., 105–115.

PALEY, A. J., HANSEN, W., KAZAR, M., SHERMAN, M., WADLOW, M., NEUEUNDORFFER, T., STERN, Z., BADER, M., AND PETERS, T. 1988. The Andrew toolkit—An overview. In *Proceedings of Winter Usenix Technical Conference* (Dallas, Tex., Feb.). USENIX Assoc., Berkeley, Calif., 9–21.

PAUSCH, R., CONWAY, M., AND DELINE, R. 1992. Lesson learned from SUIT, the Simple User Interface Toolkit. *ACM Trans. Inf. Syst. 10*, 4 (Oct.), 320–344.

PIKE, R. 1983. Graphics in overlapping bitmap layers. *ACM Trans. Graph. 2*, 2 (Apr.), 135–160. Also in *Computer Graphics: SIGGRAPH'83 Conference Proceedings*, 1983. pp. 331–355.

RUSSO, P. AND BOOR, S. 1993. How fluent is your interface? Designing for international users. In *Human Factors in Computing Systems, Proceedings INTERCHI'93* (Amsterdam, Apr.). ACM, New York, 342–347.

SAMUELSON, P. 1993. The ups and downs of look and feel. *Commun. ACM 36*, 4 (Apr.), 29–35.

SCHEIFLER, R. W. AND GETTYS, J. 1986. The X Window System. *ACM Trans. Graph. 5*, 2 (Apr.), 79–109.

SCHMUCKER, K. J. 1986. MacApp: An application framework. *Byte 11*, 8 (Aug.), 189–193.

SCHULERT, A. J., ROGERS, G. T., AND HAMILTON, J. A. 1985. ADM-A Dialogue Manager. In *Human Factors in Computing Systems, Proceedings SIGCHI'85*. ACM, New York, 177–183.

SINGH, G. AND GREEN, M. 1989. Chisel: A system for creating highly interactive screen layouts. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89*. ACM, New York, 86–94.

SINGH, G. AND GREEN, M. 1988. Designing the interface designer's interface. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88* (Banff, Alberta, Canada, Oct.). ACM, New York, 109–116.

SMITH, D. C., IRBY, C., KIMBALL, R., VERPLANK, B., AND HARSLEM, E. 1982. Designing the Star user interface. *Byte 7*, 4 (Apr.), 242–282.

STALLMAN, R. M. 1979. Emacs: The extensible, customizable, self-documenting display editor. Tech. Rep. 519, MIT Artificial Intelligence Lab, Cambridge, Mass.

STEVENS, A., ROBERTS, B., AND STEAD, L. 1983. The use of a sophisticated graphics interface in computer-assisted instruction. *IEEE Comput. Graph. Appl. 3*, 2 (Mar./Apr.), 25–31.

STRAUSS, P. S. AND CAREY, R. 1992. An object-oriented 3D graphics toolkit. In *SIGGRAPH '92. Comput. Graph. 26*, 2, 341–349.

SUKAVIRIYA, P. AND FOLEY, J. D. 1990. Coupling a UI framework with automatic generation of context-sensitive animated help. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90* (Snowbird, Utah, Oct.). ACM, New York, 152–166.

SUKAVIRIYA, N., KOVACEVIC, S., FOLEY, J., MYERS, B., OLSEN, D., AND SCHNEIDER-HUFSCHMIDT, M. 1994. Model-based user interfaces: What is it and why should I care? In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'94*. ACM, New York, 133–135.

SUKAVIRIYA, P., FOLEY, J. D., AND GRIFFITH, T. 1993. A second generation user interface design environment: The model and the runtime architecture. In *Human Factors in Computing Systems, Proceedings INTERCHI'93* (Amsterdam, Apr.). ACM, New York, 375–382.

SUTHERLAND, I. E., 1963. SketchPad: A man-machine graphical communication system. In *AFIPS Spring Joint Computer Conference*. AFIPS, Montvale, N.J., 329–346.

SWINEHART, D., ZELLWEGER, P., BEACH, R., AND HAGMANN, R. 1986. A structural view of the Cedar programming environment. *ACM Trans. Program. Lang. Syst. 8*, 4 (Oct.), 419–490.

SZEKELY, P., LUO, P., AND NECHES, R. 1993. Beyond interface builders: Model-based interface tools. In *Human Factors in Computing Systems, Proceedings INTERCHI'93* (Amsterdam, Apr.). 383–390.

TEITELMAN, W. 1979. A display oriented programmer's assistant. *Int. J. Man Mach. Stud. 11*, 157–187. Also Xerox PARC Tech. Rep. CSL-77-3, Palo Alto, CA, March 8, 1977.

TESLER, L. 1981. The Smalltalk environment. *Byte Mag. 6*, 8 (Aug.), 90–147.

VANDER ZANDEN, B. AND MYERS, B. A. 1990. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Human Factors in Computing Systems, Proceedings SIGCHI'90*, (Seattle, Wash., Apr.). ACM, New York, 27–34.

VLISSIDES, J. M. AND LINTON, M. A. 1990. Unidraw: A framework for building domain-specific graphical editors. *ACM Trans. Inf. Syst. 8*, 3 (July), 204–236.

VLISSIDES, J. M. AND TANG, S. 1991. A Unidraw-based user interface builder. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91*. ACM, New York, 201–210.

WERNECKE, J. 1994. *The Inventor Mentor*. Addison-Wesley, Reading, Mass.

WIECHA, C., BENNETT, W., BOIES, S., GOULD, J., AND GREENE, S. 1990. ITS: A tool for rapidly developing interactive applications. *ACM Trans. Inf. Syst. 8*, 3 (July), 204–236.

WIECHA, C., BOIES, S., GREEN, M., HUDSON, S., AND MYERS, B. 1989. Direct manipulation of programming: How should we design interfaces? In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, *Proceedings UIST'89* (Williamsburg, Va., Nov.). ACM, New York, 124–126.

WILDE, N. AND LEWIS, C. 1990. Spreadsheet-based interactive graphics: From prototype to tool. In *Human Factors in Computing Systems, Proceedings SIGCHI'90*. ACM, New York, 153–159.

WILSON, D. 1990. *Programming with MacApp*. Addison-Wesley, Reading, Mass.

X BUSINESS GROUP. 1994. *Interface Development Technology*. X Business Group, Inc., Fremont, Calif.

ZELEZNIK, R. C., CONNER, D., WLOKA, M., ALIAGA, D., HUANG, N., HUBBARD, P., KNEP, B., KAUFMAN, H., HUGHES, J., AND VAN DAM, A. 1991. An object-oriented framework for the integration of interactive animation techniques. In *SIGGRAPH '91. Comput. Graph. 25*, 4 (July), 105–112.

Received August 1993; revised August 1994; accepted October 1994