State of the Art in User Interface Software Tools

Brad A. Myers

February 1992
CMU-CS-92-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This will appear as a chapter in
Advances in Human-Computer Interaction, Volume 4,
H. Rex Hartson and Deborah Hix, editors.
Norwood, NJ: Ablex Publishing, 1992.

This is a revision of technical report CMU-CS-88-107,
Tools for Creating User Interfaces: An Introduction and Survey.

Abstract

User Interface Software is inherently difficult to design and implement. This paper discusses some important reasons why user interface software is more difficult than other types of software, and then surveys the approaches that have been taken to address this problem. Many kinds of tools have been designed to help create user interface software, and these can be classified by the styles of interfaces they create, and the techniques used by the user interface designer to create the software. This paper discusses this taxonomy and a number of tools, both research and commercial, in each category. Finally, current research directions and open issues in the field are discussed.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 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.

Keywords: User Interface Management Systems, UIMS, Toolkits, Interface Builders, User Interfaces, Software Engineering.

1. Introduction

The *user interface* 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*. Experience and studies have shown that creating the software for user interfaces is a very expensive and difficult task. The easy-to-use ‘‘Direct Manipulation’’ interfaces popular with most modern systems are among the most difficult to implement. In these interfaces, the objects of interest are visible on the screen and the user can operate on them by direct reference and rapid, reversible, incremental actions [Shneiderman 83]. Examples include most programs running on systems with *windowing systems* (that divide the screen into separate rectangular areas—see section 4), such as Macintosh, Microsoft Windows, Presentation Manager, SunView, X, etc.

This paper discusses why user interface software is so difficult to create, and what kinds of tools have been created to help with this process. There have been many previous surveys on this topic [Myers 89a, Hartson 89], but since this is a fast-changing field, a new one seemed in order. In addition, 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. For example, there are over 100 commercial graphical user interface builders, and many new research tools are reported every year at conferences like the annual ACM User Interface Software and Technology Symposium (UIST) and the ACM SIGCHI conference. There are also probably at least three PhD theses on user interface tools every year. Therefore, this paper tries to provide an overview of the most popular approaches, rather than an exhaustive survey.

User interface tools have been called various names over the years, with the most popular being *User Interface Management Systems (UIMS)*. 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*, and *Application Frameworks* have been used. This paper 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* toolkits.

Four different classes of people are involved with user interface software, and it is important to have different names for them to avoid confusion. The first is the person using the resulting program, who is called the *end user* or just *user*. The next person creates the user interface of the program, and is called the *user interface designer* or just *designer*. Working with the user interface designer will be the person who writes the software for the rest of the application. This person is called the *application programmer*. The designer may use special user interface tools which are specifically created 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.

2. Why User Interfaces Are Hard to Build

It is the conventional wisdom that user interface software is difficult and expensive to create. Many surveys have shown that the user interface portion of the software accounts for over half of the code and development time. For example, a new survey reports that over a wide class of program types, machine types and tools used, the percent of the design time, the implementation time, the maintenance time, and the code size devoted to the user interface was about 50% [Myers 92a]. In fact, there are a number of important reasons that user interface software will *inherently* be among the most difficult kinds of software to create, as described below.

2.1 Iterative design

It is notoriously difficult to *design* a user interface that will be easy to use, and there are no design strategies that will guarantee that the resulting user interface will be learnable, easy to use, and ‘‘user-friendly.’’ Consequently, the only reliable method for generating quality user interfaces is to test prototypes with actual end users and modify the design based on the users’ comments and performance [Gould 85, Swartout 82]. As reported by Sheil [Sheil 83] ‘‘complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so.’’ This methodology is called ‘‘iterative design’’ and has been used in the creation of some of the best user interfaces: the Xerox Star [Bewley 83], the Apple Lisa and Macintosh [Morgan 83], and the Olympic Messaging system [Boies 85]. A recent survey reported that 87% of the development projects used iterative design in some form [Myers 92a].

The need for iterative design means that the conventional software engineering approach to software design [Yourdon 89] is inadequate. It is impossible to fully specify the user interface of a system, then implement it, and later test it. Instead, the specification, the implementation, and the testing must be intertwined [Swartout 82]. Furthermore, it can be very difficult to schedule and manage user interface development when it is not clear how many iterations of re-implementation will be necessary.

2.2 Multiprocessing

As discussed below (section 4.2.2), almost all windowing systems queue ‘‘event’’ records to deliver the keyboard and mouse inputs from the user to the user interface software. Users expect to be able to abort and undo actions (for example, by typing control-C or Command-dot) and to freeze the output of the program, if it is scrolling by too fast (for example, using control-S). Also, if a window’s picture needs to be redrawn by the application, the window system notifies the application by adding a special event to the queue. Therefore, the user interface software must be structured so that it can accept input events at all times, even while executing commands. Consequently, any operations that may take a long time, (such as printing, searching, global replace, repaginating a document, or even repainting the screen) should be executed in a separate process.[1] Therefore, most programmers creating user interface software will encounter the well-known problems of multiple processes (synchronization, maintaining consistency

[1]Alternatively, the long jobs could poll for input events in their inner loop, and then check to see how to handle the input, but this is essentially a way to simulate multiple processing.

among multiple threads, deadlocks, etc.).

2.3 Real-time Programming

Most graphical, direct manipulation interfaces will have objects that are animated or which move around with the mouse. In order for this to be attractive to users, the objects must be redisplayed between 30 and 60 times per second without uneven pauses. Therefore, the programmer must ensure that any necessary processing to calculate the feedback can be guaranteed to finish in about 16 milleseconds. Many of the problems of real-time programming therefore also appear in user interface programming.

2.4 Robustness

When the user provides erroneous input to a program, the program should respond with a helpful error message, and allow the user to repair the error and continue. This means that programs should *never* crash or go into a debugger. Whereas a programmer might define the interface to an internal procedure to only work when passed a certain type of value, the user interface must always accept any possible input, and continue to operate. As a result, there is a tremendous requirement for robustness in user interface software.

To make the task even more difficult, user interfaces should allow the user to abort and undo any operation. Therefore, the programmer must implement all actions in a way that will allow them to be aborted while executing and un-done after completion.

2.5 Testability

A related problem is the difficulty of testing user interface software for correctness. Since systems may have hundreds of commands, and they may interact in many ways, it is difficult to exercise all possible paths through the program. Automated testing tools are rarely useful for direct manipulation systems, since they have difficulty pretending to be users. Techniques such as making a transcript of the mouse movements and button presses do not work if elements of the user interface are moved around or change type, since then the transcript will no longer work. Also, how would the testing program compare the output with the expected values, since the output is usually directed at the screen?

2.6 Modularization

One of the most important ways to make software easier to create in general is to appropriately modularize the different parts. The standard admonition in textbooks is that the user interface portion should be separated from the rest of the software, in part so that the user interface can be easily changed (for iterative design). Unfortunately, programmers find in practice that it is difficult or impossible to separate the user interface and application parts [Rosson 87], and changes to the user interface usually require reprogramming parts of the application also.

2.7 Complexity of the Tools

The existing tools for creating user interface software are notoriously difficult to use. Manuals for the tools often run to many volumes and contain hundreds of procedures (for example, the Macintosh ToolBox manuals [Apple 85] now fill six books and the X documentation is similar). In spite of this, many of the provided functions may not be flexible enough to achieve the desired effect. For example, in the Macintosh and Motif toolkits, it is easy to have a keyboard accelerator that will perform the same operation as a menu item, but very difficult to have a keyboard command do the same thing as an on-screen button. Some tools even require the programmer to learn an entirely new special-purpose programming language to create the user interface.

Using the underlying graphics packages is also quite difficult. Since the human eye is quite sensitive to small differences, the graphic displays must essentially be perfect: a single pixel error in alignment will be visible. Most existing graphics packages provide no help with making the displays attractive.

2.8 Language Support

Most computer programming languages do not contain primitives for graphical input and output. Many, however, have input-output primitives that will read and write strings. For example, Pascal uses `readln` and `writeln`. Unfortunately, using these produces a very bad user interface, where the user is required to answer questions in a highly-modal style, and there are no facilities for undo or help. Therefore, the built-in input/output facilities must be ignored and special libraries must be used.

Many user interface tools have found other important language features lacking. For example, many people agree that user interface software should be ‘‘object-oriented’’ but most languages do not have an appropriate object system (see section 5.1). The creators of the Xtk toolkit [McCormack 88] (in which Motif and OpenLook are implemented), and the Andrew toolkit [Palay 88] decided that existing object systems (such as C++ and Objective-C) were unacceptable, and decided to implement their own special object-oriented facilities to C. Similarly, Apple added an object system to Pascal for MacApp [Wilson 90], and Garnet [Myers 90a] added a special object system to Lisp.

3. 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 the process. Today, many of these tools have progressed from research into commercial systems, and their effectiveness has been amply demonstrated. For example, the MacApp system from Apple has been reported to reduce development time by a factor of four or five [Schmucker 86]. Research systems also continue to evolve quickly, and the systems that were popular five years ago have been made obsolete by more effective tools and new styles of user interfaces.

3.1 Advantages of Using Tools

Naturally, the user interface tools attempt to overcome the problems in creating user interface software listed above. In particular, the claimed advantages of using tools include:

- **The quality of the interfaces should be higher.** This is because:
  - Designs can be rapidly prototyped and implemented, possibly even before the application code is written.
  - It is easier to incorporate changes discovered through user testing.
  - 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 likely to have more consistent user interfaces if they are created using the same user interface tool.
  - Conversely, some user interface tools make it easier to investigate different styles for a user interface, and thereby provide a unique ‘‘look-and-feel’’ for a program.
  - A UI tool should make it 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.
  - It will be easier to customize the interfaces to fit the needs of the users.
  - The ability to rapidly modify interfaces should allow system designers and salesmen to try different interfaces on products in front of customers and end users, and immediately incorporate their suggestions.
- **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 in standard libraries.
  - 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.
  - Training and learning costs for the user interface designers and implementers should be less, since the tools are easier to use than the underlying system.
  - The level of expertise of the interface designers and implementers can be lower, because the tools hide much of the complexities of the underlying system.
  - The reliability of the user interface should be higher, since the code for the user interface is created automatically from a higher level specification.
  - It should be easier to port an application to different hardware and software environments since the device dependencies are isolated in the user interface tool.

3.2 Goals for User Interface Software Tools

This section discusses some of the things that you might want a user interface tool to help with. The 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 goals.

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,
- create easy-to-use interfaces,
- allow the designer to rapidly investigate different designs,
- allow non-programmers to design and implement user interfaces,
- automatically evaluate the interface and propose improvements, or at least provide information to allow the designer to evaluate the interface,
- 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 user-specified 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, and
- insulate the application from all device dependencies and the underlying software and hardware systems.

3.3 Components of User Interface Software

As shown in Figure 1, user interface software may be divided into various components. Some systems layer the components differently, but most have a part that fills each of the roles shown.

The *windowing system* supports the separation of the screen into different (usually rectangular) regions, called *windows*. Some modern windowing systems divide the functionality into a *window system*, which provides the functional or programming interface, and a *window manager* which provides the user interface. This terminology comes from the X system [Scheifler 86]. However, many people and systems use the name ‘‘window manager’’ to refer to both the base layer and the user interface layer. This article will use the X terminology, and use the term ‘‘windowing system’’ when referring to both layers.

The ‘‘window system’’ part provides procedures that allow the application to draw pictures on the screen and get input from the user. 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. On top of

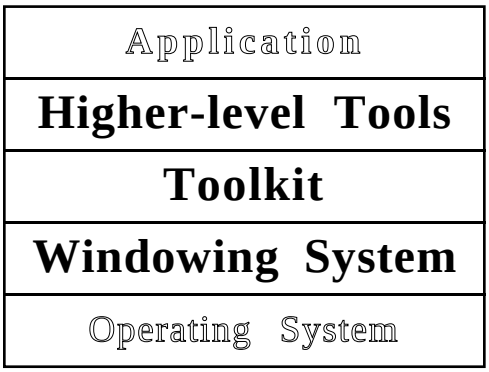

Figure 1: The components of user interface software discussed in this paper.

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.

4. 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 appears in [Myers 88a].

The first windowing systems were implemented as part of a single program or system. For example, the EMACs text editor [Stallman 79], and the Smalltalk [Tesler 81] and DLISP [Teitelman 79] 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 84], SunView for Suns, and the Macintosh and NeXT windowing 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 systems inter-process communication mechanism to connect to applications.

4.1 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*.

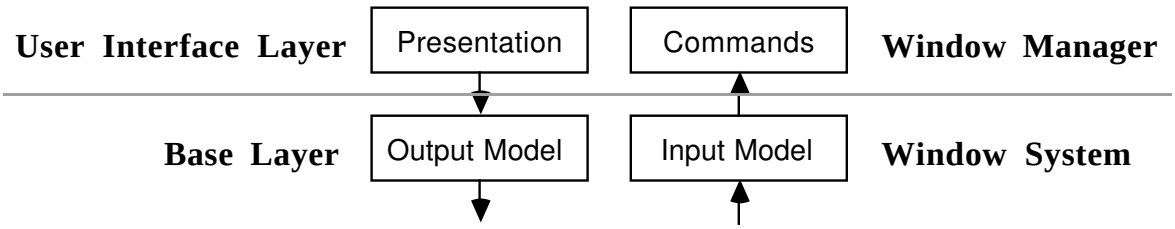

Figure 2: The windowing system can be divided into two layers, called the *base* or *window system* layer, and the *user interface* or *window manager* layer. Each of these can be divided into parts that handle output and input.

The other layer of windowing system is the *window manager* or *user interface*. This includes all aspects that are visible to the user. The two parts of the user interface layer are the *presentation*, which is comprised of the pictures that the window manager displays, and the *commands*, which are how the user manipulates the windows and their contents. Figure 3 shows windows which demonstrate different aspects of the presentation, including patterns or pictures for the area where there are no windows, title lines and borders for windows, icons, etc. Examples of the operations that may be provided for windows include moving them around on the screen and specifying their size.

4.2 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 [Scheifler 86] 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 (the notable exception is the NeXT system, and, of course, some older windowing systems, such as SunView, continue to be supported). In the small computer market, the Macintosh runs its own window system or X, and IBM PC-class machines primarily run Microsoft Windows or IBM's Presentation Manager.

4.2.1 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

Figure 3: An example screen running a covered window system. This is a Hewlett-Packard Vue screen, which is using Motif under X/11. (Courtesy of the User Interaction Design Group, Software Engineering Systems Division, Hewlett Packard, copyright 1991, used by permission.)

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 operations that are specially optimized to work efficiently with the window system.

In early windowing systems, such as Smalltalk [Tesler 81], Blit [Pike 83] and Sapphire [Myers 86], the primary output operation was BitBlt (also called ''RasterOp''). This takes a rectangle of pixels from one part of the screen and copies them to another part. Various boolean operations can be specified for combining the pixel values of the source and destination rectangles. For example, source rectangle can simply replace the destination, or it might be XORed with the destination. BitBlt can be used to fill a rectangle with a particular color, display text, scroll windows, and perform many other effects [Ingalls 81].

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. A few windowing systems, such as NeWS, NeXT, and the latest versions of DECWindows, allow the full Postscript imaging system [Adobe 85] 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).

As shown in Figure 4, 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. As mentioned above, most newer systems, including the Macintosh, X, NeWS, and NeXT, have implemented the graphics system as *part* of the windowing system.

4.2.2 Input Model

It is somewhat surprising that, although there has been substantial progress in handling of the output in windowing systems, input is still handled in essentially the same way that the original windowing systems handled it: as a stream of event records sent to the window. 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.

The problems with this model are that:

- There is no provision for special abort or stop-output events (control-C, command-dot), so these will be queued with the other input events.
- The same event mechanism is used for special messages from the windowing system. When a window gets larger or becomes uncovered, the program 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 the event stream, which the program must handle.
- The application must always be willing to accept events in order to process aborts and redrawing requests, as discussed above in section 2.2.
- The model is device dependent, since the event record has fixed fields for the expected incoming events. If a 3-D pointing device or one with more than the standard number of buttons was used instead of a mouse, then the event mechanism could not 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 90b]).

4.3 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

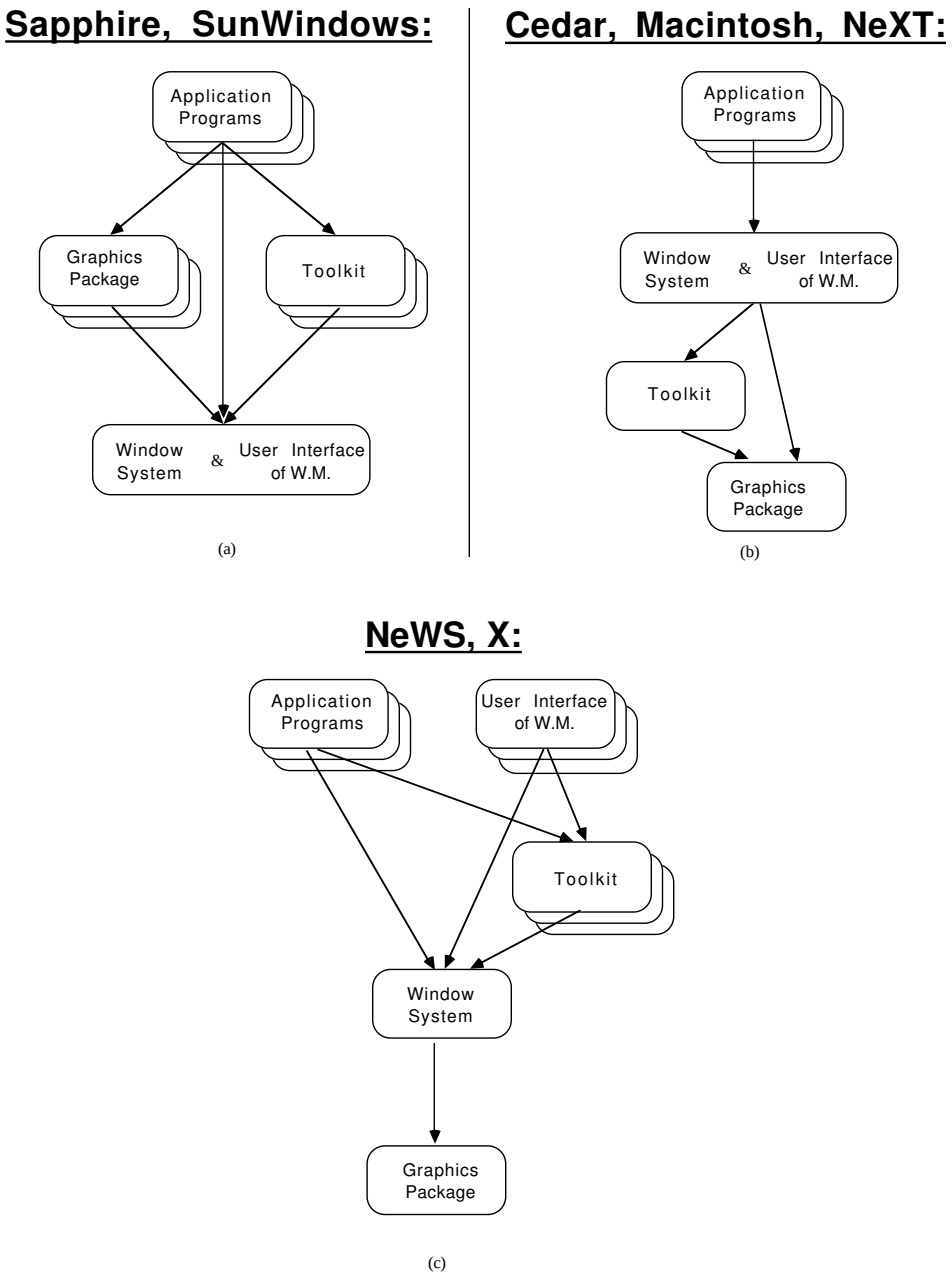

Figure 4: Various organizations that have been used by windowing systems. Boxes with extra borders represent systems that can be replaced by users. Early systems (a) tightly coupled the window manager and the window system, and assumed that sophisticated graphics and toolkits would be built on top. The 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. Most recently, (c) windowing systems have allowed different window managers and different toolkits, while still embedding sophisticated graphics packages.

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.

For a complete discussion of the options for the user interfaces of window managers, see [Myers 88a]. Also, the video *All the Widgets* [Myers 90c] has a 30 minute segment showing many different forms of window manager user interfaces.

The user interface of a windowing system, which is sometimes called its *look-and-feel*, can apparently be copyrighted and patented. This is the source of a number of the look-and-feel lawsuits [Samuelson 89].

4.3.1 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 3. 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. Although the tiled style was popular for awhile, it is now rarely seen, because users generally prefer overlapping.

Another important aspect of the presentation of windows is the use of *icons*. These are small pictures that represent windows (or sometimes files). They are used because there would otherwise be too many windows to conveniently fit on the screen and manage. When a window is not in use, it can be removed and replaced with its icon, and later conveniently retrieved when needed. Figure 6 shows examples of icons from various window managers. Some window managers allow applications to control parts of the icons for their own user interfaces. For example, Sapphire allows the icon to be a bitmap picture and to contain progress bars, and Motif allows the icon to contain arbitrary graphics, that might even be animated.

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.

[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 82].

Figure 5: A screen from the Cedar [Swinehart 86] 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.

4.3.2 Commands

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

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

Most X window managers (including the Motif and OpenLook window managers) allow the user to choose which method is desired.

All covered window systems allow the user to change which window is on top (not covered by other

Figure 6: Examples of icons from different systems: Sapphire [Myers 84], Macintosh, Star [Smith 82] and two styles of icons for the Motif Window Manager for X.

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.

5. Toolkits

A *toolkit* is a library of interaction techniques that can be called by application programs. An *interaction technique* is a way of using a physical input device to input a certain type of value. Interaction techniques are also often called *widgets* or *gadgets*. Typically, widgets in toolkits include menus, buttons, scroll bars, dialog boxes, etc. Figure 7 shows some examples of widgets. Toolkits can only be used by programmers, since they only have procedural interfaces.

Using a toolkit has the advantage that the final UI will look and act similarly to other UIs created using the same toolkit, and each application does not have to re-write the standard functions, such as menus. A

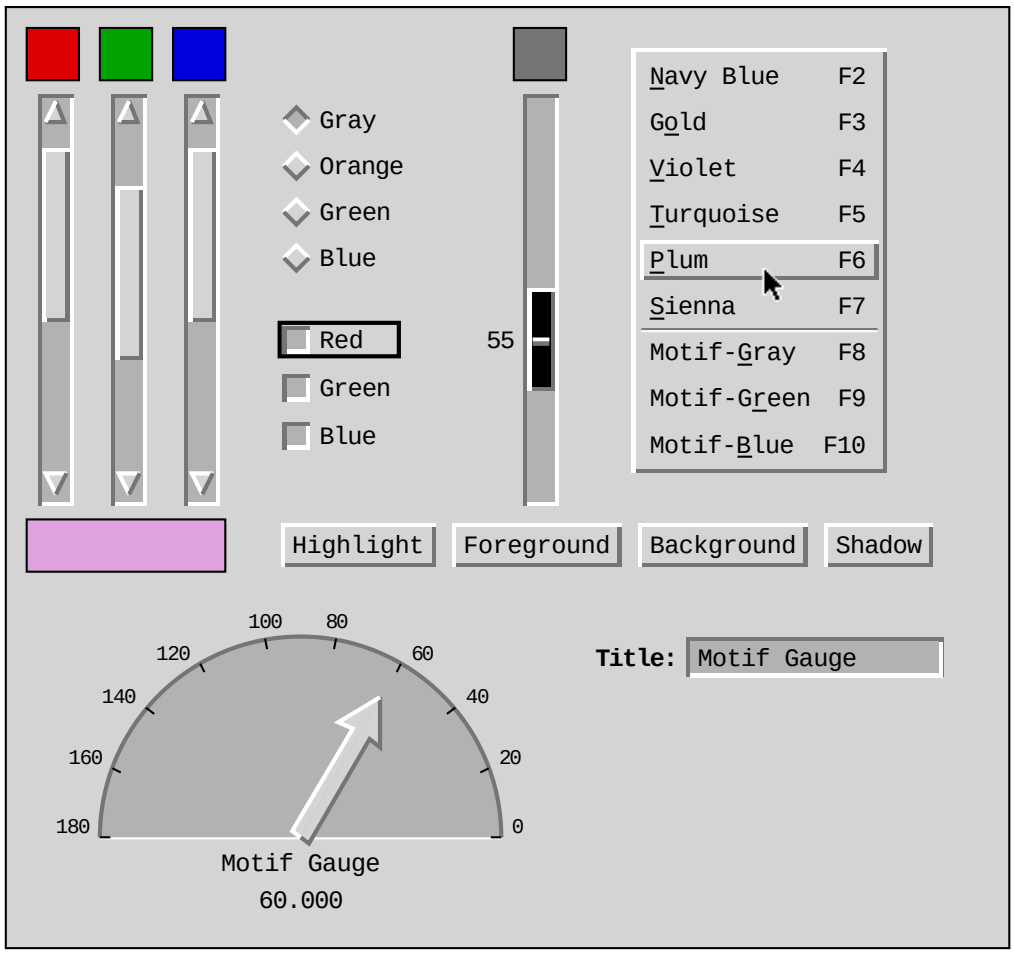

Figure 7: Some of the widgets with a Motif look-and-feel provided by the Garnet toolkit.

problem with toolkits is that the styles of interaction are limited to those provided. In addition, the toolkits themselves are often expensive to create: ‘‘The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate’’ [Cardelli 85, 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.

As with the graphics package, the toolkit can be implemented either using or being used by the windowing system (see Figure 4). 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 toolkit routines for its user interface. In other systems, such as X, the toolkit is on top of the windowing system. Here, it is easier to switch toolkits (for example, the Xtk [McCormack 88], InterViews [Linton 89], or Garnet [Myers 90a] toolkits can be used on top of X), but the window manager must usually implement its user interface from scratch.

In the terminology of the Xtk toolkit [McCormack 88] for the X Window System, a toolkit may be

divided into two layers. The *widget set* is the collection of widgets, and the *intrinsics* layer is how the widgets are implemented. The intrinsics provide the common services, such as techniques for object-oriented programming, etc. Multiple widget sets with different looks and feels can be implemented on top of the same intrinsics layer (Figure 8-a), or else the same look-and-feel can be implemented on top of different intrinsics (Figure 8-b).

(a) (b)

Figure 8: (a) At least three different widget sets that have different looks and feels have been implemented on top of the Xtk intrinsics. (b) The Motif look-and-feel has been implemented on at least three different intrinsics.

5.1 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 View windowing system, and the Macintosh Toolbox [Apple 85]. The other variety uses an object-oriented programming style which makes it easier for the designer to customize the interaction techniques. Examples include Smalltalk [Tesler 81], Andrew [Palay 88] and the Xtk toolkit [McCormack 88] for the X Window System.

The advantages of using object-oriented intrinsics are that it is a natural way to think about widgets (the menus and buttons on the screen *seem* like objects), the widget objects can handle some of the chores that otherwise would be left to the programmer (such as refresh), and it is easier to create custom widgets (by sub-classing 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 Xtk, Andrew and Garnet, or it might use an existing object system, as was done in InterViews [Linton 89] which uses C++, and NeXTStep [NeXT 91] which uses Objective-C.

The usual way that object-oriented toolkits interface with application programs is through the use of *call-back procedures*. These are procedures defined by the application programmer that are called when a widget is operated by the end user. For example, the programmer might supply a procedure to be called

when the user selects a menu item. Experience has shown that real interfaces often contain hundreds of call-backs, which makes the code harder to modify and maintain [Myers 92a]. In addition, 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 91a].

Some research toolkits have added novel things to the toolkit intrinsics. For example, GROW [Barth 86] and the Garnet toolkit [Myers 90a] 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 update the rectangle if the user moves the slider.

5.2 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 90c]. For example, it shows 35 different kinds of menus. Like window manager user interfaces, the widgets' look-and-feel can be copyrighted and patented.

As was mentioned above, different widget sets (with different looks and feels) can be implemented on top of the same intrinsics. In addition, the same look-and-feel can be implemented on top of different intrinsics. For example, there are Motif look-and-feel widgets on top of the xtk, InterViews and Garnet intrinsics (Figure 8-b). Although they all look and operate the same (so would be indistinguishable to the end user), they are implemented quite differently, and have completely different procedural interfaces for the programmer.

6. 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 are being 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. The code can then be linked to different actual toolkits, and will run without change. An example is the XVT system [XVT 91] which provides a C or C++ interface that hides the differences of the Motif, OpenLook, Macintosh, MS-Windows, and OS 2-PM toolkits (and also character terminals).[3]

[3]This can be distinguished from a toolkit that supports multiple looks-and-feels. For example, both the InterViews and Garnet toolkits provide a set of widgets with the Motif look-and-feel, and another set of widgets with a custom look-and-feel. To be called a ''Virtual Toolkit,'' a system must link to the native implementations of the underlying toolkits, rather than replacing them.

The interface to the virtual toolkit is still procedural, however, so the designer must be an experienced programmer. Typically, the virtual toolkits take a ‘‘least common denominator’’ approach, and only provide functions that appear in all toolkits. Therefore, a designer may not be able to achieve the desired result. In addition, the resulting interface may not be completely compliant with the look-and-feel of the target system, due to features of the target toolkit that do not appear in other toolkits.

7. Higher Level Tools

Since toolkits are hard to use, there is a tremendous interest in higher level tools that will make the user interface software production process easier.

7.1 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 usually includes a toolkit, but may also include additional software specifically for the tool. Since the run-time component is ‘‘managing’’ the user interface, the term *User Interface Management System* 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 88], discovered that there are very few metrics that can be applied by computers.

7.2 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 9, 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 specification, and others allow the interface to be specified graphically. Each of these types is discussed below. Of course, some tools use different techniques for specifying different parts of the user interface. These are classified by their predominant or most interesting feature.

7.2.1 Language Based

With most of the older user interface tools, the designer specified the user interface using 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 [Green 86] provides an extensive comparison of grammars, state transition diagrams, and event languages.

Specification Format	Examples	Section
Language Based		7.2.1
State Transition Networks	[Newman 68], [Jacob 86]	7.2.1.1
Context-Free Grammars	YACC, LEX, Syngraph [Olsen 83]	7.2.1.2
Event Languages	ALGAE [Flecchia 87], Sassafras [Hill 86], HyperTalk	7.2.1.3
Declarative Languages	Cousin [Hayes 85], Open Dialog [Schulert 85], Motif UIL	7.2.1.4
Constraint Languages	Thinglab [Freeman-Benson 90], C32 [Myers 91b]	7.2.1.5
Application Frameworks	MacApp [Wilson 90], Unidraw [Vlissides 90]	7.2.2
Automatic Generation	MIKE [Olsen 86], UIDE [Foley 89], ITS [Wiecha 90]	7.2.3
Direct Graphical Specification		7.2.4
Prototypers	Bricklin's Demo [Sage 90], Director [MacroMind 90], HyperCard	7.2.4.1
Cards	Menulay [Buxton 83], HyperCard	7.2.4.2
Interface Builders	DialogEditor [Cardelli 88], NeXT Interface Builder [NeXT 91], Prototyper [SmethersBarnes 90], UIMX [VisualEdge 90]	7.2.4.3
Graphical Editors	Peridot [Myers 88b], Lapidary [Myers 89b]	7.2.4.4

Figure 9: Ways to specify the user interface, some tools that use that technique, and the section of this paper that discusses the technique.

7.2.1.1 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 implemented a simple tool using finite state machines in 1968 [Newman 68] which handled textual input. This was apparently the first user interface tool. Many of the assumptions and techniques used in modern systems were present in Newman's: 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 a large amount of syntactic parsing is necessary or when 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. In addition, state diagrams cannot handle interfaces where the user can operate on multiple objects at the same time. Another problem is that they can be very confusing for large interfaces, since they get to be a ‘‘maze of wires’’ and off-page (or off-screen) arcs can be hard to follow.

Recognizing these problems, but still trying to retain the perspicuousness of state transition diagrams, Jacob [Jacob 86] has invented a new formalism, which is a combination of state diagrams with a form of event languages (see section 7.2.1.3). There can be multiple diagrams active at the same time, and flow of control transfers from one to another in a co-routine fashion. The system can create various forms of direct manipulation interfaces.

7.2.1.2 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 parsers built with YACC and LEX in Unix, and Syngraph [Olsen 83].

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.

7.2.1.3 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 87] 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 86], uses a similar idea, but with an entirely different syntax. This system also adds local variables called ‘‘flags’’ to help specify the flow of control. As described below in section 7.2.4.2, the HyperTalk language that is part of HyperCard for the Apple Macintosh, can also be considered an event language.

Most of the tools in this category are explicitly designed to handle multiple processes, which can be important in user interfaces. For example, research has shown that people can be more effective when operating multiple input devices at the same time [Buxton 86]. Also, it is often easier to program multiple interactions that are available at the same time (so the user can pick which one to use) using multiple processes. The disadvantage of these event languages 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 non-trivial size.

7.2.1.4 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 85] and HP/Apollo's Open-Dialogue [Schulert 85] 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 both can set and access.

Another type of declarative language is the layout description languages that come with many toolkits. For example, Motif's 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.

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, or drawing new graphical objects.

7.2.1.5 Constraint Languages

A number of user interface tools allow the programmer to use constraints to define the user interface [Borning 86a]. Early constraint systems include Sketchpad [Sutherland 63] which pioneered the use of graphical constraints in a drawing editor, and Thinglab [Borning 81] which used constraints for graphical simulation. Subsequently, Thinglab has been extended to aid in the generation of user interfaces [Borning 86a].

Section 5.1 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 90a]). In addition, some systems have provided higher-level interfaces to constraints. Graphical Thinglab [Borning 86b] allows the designer to create constraints by wiring icons together, and NoPump [Wilde 90] and C32 [Myers 91b] allow constraints to be defined using a spreadsheet-like interface.

The advantages 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 in error since it is 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 they have a great potential to simplify the programming task.

7.2.1.6 Summary

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 87]. Furthermore, it seems more natural to define the graphical part of a user interface using a graphical editor (see section 7.2.4). However, it is clear that for the forseeable 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 interfaces [Myers 92b].

7.2.2 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. They therefore created a software system that provides an overall *application framework* to guide programmers. This is called MacApp [Wilson 90] 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 90] 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. Unidraw has been used to create various draw and CAD programs, and a user interface editor [Vlissides 91]. Even more specialized are various graph programs, such as Edge [Newbery 88] and TGE [Karrer 90]. These provide a framework in which the designer can create programs that display their data as trees or graphs. The programmer typically specializes the node and arc classes, and specifies some of the commands, but the framework handles layout and the overall control.

All of these frameworks require the designer to write code, typically by creating application-specific sub-classes of the standard classes provided as part of the framework. Another class of systems that might be considered ‘‘frameworks’’ help create user interfaces that are composed of a series of cards, such as HyperCard from Apple. These systems are discussed in section 7.2.4.2 because their primary interface to the designer is graphical.

7.2.3 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 89], Jade [Vander Zanden 90], and Chisel [Singh 89], 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.

Another approach is to try to create a user interface based on a list of the application procedures. MIKE [Olsen 86] 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 7.2.4, to specify these changes.

UIDE (the User-Interface Design Environment) [Foley 89] requires that the semantics of the application be defined in a special-purpose, Pascal-like language, and therefore might be included with the language-based tools (section 7.2.1). It is placed here instead because the language is used to describe the functions that the application supports and not the desired interface. The description includes pre- and post-conditions of the operations, and the system uses these to reason about the operations, and to automatically generate an interface. One interesting part of this system is that the user interface designer can apply ‘‘transformations’’ to the interface. These change the interface in various ways. For example, one transformation changes the interface to have a currently selected object instead of requiring an object to be selected for each operation. Another transformation provides functions specialized for various types (‘‘delete-square’’ and ‘‘delete-triangle’’), rather than general-purpose functions (‘‘delete’’). UIDE applies the transformations and insures that the resulting interface remains consistent. Another feature of UIDE is that the pre- and post-conditions are used to automatically generate help [Sukaviriya 90].

Another rule-based system is ITS [Wiecha 90], which is being used to create the visitor information system for EXPO 1992 in Seville. Unlike the other systems described above, the designer in EXPO is expected to write some of the rules used to generate the interface, 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 generally not good enough. A further problem is that the designer must often use special-purpose specification languages which can be difficult to learn and use properly. The most promise currently seems to be for the generation of dialog-box and menu-based interfaces.

7.2.4 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 non-programmers. Therefore, psychologists, graphic designers and user interface specialists can more easily be involved in the user interface design process when these tools are used.

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

7.2.4.1 Prototyping Tools

The goal of *Prototyping tools* is to allow the designer to quickly create 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 many 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. Advertizements 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 here in a much more specific manner.

Probably the first prototyping tool was Dan Bricklin’s Demo [Sage 90]. 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 MacroMind Director [MacroMind 90] 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 re-coded after prototyping. There is also the risk that the programmers who implement the real user interface will ignore the prototype.

7.2.4.2 Cards

Many graphical programs are limited to user interfaces that can be presented as a sequence of mostly static pages, sometimes called ‘‘frames’’ or ‘‘cards.’’ 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 83], which allows the designer to place text, graphical potentiometers, iconic pictures, and light buttons on the screen and see exactly what the end user will see when the application is run. The designer does not need to be a programmer to use Menulay. Trillium [Henderson 86], 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 [OWL 91], Spinnaker Plus [Spinnaker 91], and Tool Book [Asymetrix 90]. 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. HyperCard provides a scripting language, called HyperTalk, to provide more functionality for buttons. As mentioned in section 7.2.1.3, HyperTalk is really an event language, since the programmer writes short pieces of code that are executed when input events occur.

7.2.4.3 Interface Builders

A related kind of tool allows the designer to create dialog boxes that are to be part of a larger user interface. These are often called *Interface Builders*. They allow the designer to select from a pre-defined library of widgets, and place them on the screen using a mouse. Other properties of the widgets can be set using dialog boxes. Usually, there is also some support for sequencing, such as bringing up sub-dialogs when a particular button is hit. Examples of research interface builders are DialogEditor [Cardelli 88], vu [Singh 88] and Gilt [Myers 91a]. There are literally hundreds of commercial interface builders. Just a few examples are the NeXT Interface Builder [NeXT 91], Prototyper for the Macintosh [SmethersBarnes 90], WindowsMAKER for Microsoft Windows on the PC [BlueSky 91], and UIMX for X Windows and Motif [VisualEdge 90].

Although Interface Builders make laying out the dialog boxes and menus easier, this is only part of the user interface design problem. 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.

7.2.4.4 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 end user's actions. Therefore, the designer must draw an *example* of the desired display, which will be modified at run-time. 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, all of these systems are research prototypes. However, some of the ideas are beginning to appear in commercial systems.

Peridot [Myers 88b] 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. Peridot can also be used by non-programmers. Lapidary [Myers 89b] extends the ideas of Peridot to allow general application objects to be drawn. For example, the designer can draw the nodes and arcs for a graph program. The DEMO system [Wolber 91] allows some dynamic, run-time properties of the objects to be demonstrated, such as when objects should be created.

7.3 Specialized Tools

For some application domains, there are customized tools that provide significant high-level support. These tend to be quite expensive, however. For example, in the aeronautics and real-time control areas, there are a number of high-level tools, including AutoCode [AutoCode 91] and InterMAPhics [Prior 91].

8. 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 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.

8.1 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 for your platform, 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, possibly 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.

8.2 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? Can new interaction techniques be added?
- **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 interface. Typically, the tools with the best ease of use are also the least functional, in terms of depth and breadth.
- **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.
- **Speed 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.
- **Price**. Some tools are provided free by research organizations, such as the Xtk toolkit from MIT 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 functionality.
- **Robustness and Support**. The free research tools are likely to have less support than commercial ones.

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. Also, there are a few formal studies of tools [Hix 89].

9. 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 [Marchionini 91]. Here, a few of the important ones are summarized.

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

Another approach is to allow the designer to specify the design in an appropriate notation, and then provide tools to convert that notation into interfaces. For example, the UAN [Hartson 90] is a notation for expressing the end user's actions and the system's responses.

Finally, much work is needed in ways for tools to help evaluate interface designs. Initial attempts, such as in MIKE [Olsen 88], 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.

9.2 Increased Breadth

We can expect the user interfaces of tomorrow to be different from the conventional window-and-mouse interfaces of today, and tools will have to change to support the new styles. For example, most tools today only deal with 2-dimensional objects, but there is already a demand to provide 3-D visualizations and animations. New input devices and techniques will probably replace the conventional mouse and menu styles. For example, gesture recognition is already appearing in commercial products, such as the notepad computers designed by GO [GO 91]. ''Virtual reality'' systems, where the computer creates an artificial world and allows the user to explore it (often using a DataGlove [Foley 87]), cannot be handled by any of today's tools.

A more immediate concern is supporting interfaces that can be moved from one natural language to another (like English to French). This is not simply a matter of translating the labels, however, since the

labels will be different sizes, so interfaces must be redesigned. Can the tools do this automatically? Also, some languages are entered from right to left or top to bottom (or a combination of these), and the input mechanisms should deal with this.

9.3 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. More effective mechanisms for users to customize existing applications and create new ones are needed. 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 end users, it will not be like programming in C, but rather at some higher level.

9.4 New models

It is well recognized that the tools in use today have some fundamental problems. For example, the input model used by windowing systems and toolkits is device-dependent and causes synchronization problems, as discussed in section 4.2.2. Another problem is that the object-oriented model of toolkits does not make it easy enough to customize widgets. Fundamentally different models might be useful for future user interface tools.

9.5 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 separate more difficult, due to the large number of call-back procedures required [Myers 92a]. Therefore, further research is needed into ways to better modularize the code, and how tools can support this.

9.6 Tools for the Tools

It is very difficult to create the tools described in this paper. 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 91c]. The Unidraw framework has also proven useful for creating interface builders [Vlissides 91]. However, more work is needed.

10. Conclusions

User interface software is inherently difficult to create, because it combines some of the most difficult aspects of programming, including multi-processing, real-time programming, the need for absolute robustness and frequent, iterative changes to the specification. As a consequence of this, 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, and chances are that today, whatever your project is, there is a tool that will help. Tools coming out of research labs are covering increasingly more of the user interface task, 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.

Acknowledgements

For help with this paper, I would like to thank Jim Foley, Brad Vander Zanden, John Turner, Robert Seacord, Bernita Myers, and the referees.

References

[Adobe 85] Adobe Systems, Inc.
Postscript Language Reference Manual.
Addison-Wesley, 1985.

[Apple 85] Apple Computer, Inc.
Inside Macintosh.
Addison-Wesley, 1985.

[Asymetrix 90] Asymetrix Corporation, 110 110th Ave. N.E., Suite 717, Bellevue, WA 98004. Phone (206) 462-0501.
ToolBook.
1990.

[AutoCode 91] Integrated Systems, 3260 Jay Street, Santa Clara, CA 94054. Phone (408) 980-1500.
AutoCode.
1991.

[Barth 86] Paul Barth.
An Object-Oriented Approach to Graphical Interfaces.
ACM Transactions on Graphics 5(2):142-172, April, 1986.

[Bewley 83] William L. Bewley, Teresa L. Roberts, David Schroit, William L. Verplank.
Human Factors Testing in the Design of Xerox's 8010 'Star' Office Workstation.
In *Human Factors in Computing Systems*, pages 72-77. Proceedings SIGCHI'83, Boston, MA, December, 1983.

[BlueSky 91] Blue Sky Software Corporation, 2375 East Tropicana Ave., Suite 320, Las Vegas, NV 89119. Phone (702) 465-6365.
WindowsMAKER.
1991.

[Boies 85] S.J. Boies, J.D. Gould, S. Levy, J.T. Richards, and J.W. Schoonard.
The 1984 Olympic Message System--A Case Study in System Design.
IBM Research Report RC-11138, 1985.

[Borning 81] Alan Borning.
The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory.
ACM Transactions on Programming Languages and Systems 3(4):353-387, October, 1981.

[Borning 86a] Alan Borning and Robert Duisberg.
Constraint-Based Tools for Building User Interfaces.
ACM Transactions on Graphics 5(4):345-374, October, 1986.

[Borning 86b] Alan Borning.
Defining Constraints Graphically.
In *Human Factors in Computing Systems*, pages 137-143. Proceedings SIGCHI'86, Boston, MA, April, 1986.

[Buxton 83] W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith.
Towards a Comprehensive User Interface Management System.
In *Computer Graphics*, 17(3), pages 35-42. Proceedings SIGGRAPH'83, Detroit, Mich, July, 1983.

[Buxton 86] William Buxton and Brad Myers.
A Study in Two-Handed Input.
In *Human Factors in Computing Systems*, pages 321-326. Proceedings SIGCHI'86, Boston, MA, April, 1986.

[Cardelli 85] Luca Cardelli and Rob Pike.
Squeak: A Language for Communicating with Mice.
In *Computer Graphics*, pages 199-204. Proceedings SIGGRAPH'85, San Francisco, CA, July, 1985.

[Cardelli 88] Luca Cardelli.
Building User Interfaces by Direct Manipulation.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 152-166. Proceedings UIST'88, Banff, Alberta, Canada, October, 1988.

[Flecchia 87] Mark A. Flecchia and R. Daniel Bergeron.
Specifying Complex Dialogs in ALGAE.
In *Human Factors in Computing Systems*, pages 229-234. CHI+GI'87, Toronto, Ont., Canada, April, 1987.

[Foley 87] James D. Foley.
Interfaces for Advanced Computing.
Scientific American :127-135, 1987.

[Foley 89] James Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray.
Defining Interfaces at a High Level of Abstraction.
IEEE Software 6(1):25-32, January, 1989.

[Freeman-Benson 90]
Bjorn N. Freeman-Benson, John Maloney, and Alan Borning.
An Incremental Constraint Solver.
Communications of the ACM 33(1):54-63, January, 1990.

[GO 91] GO Corporation. 950 Tower Lane, Suite 1400, Foster City, CA 94404. Phone (415) 345-7400.
PenPoint.
1991.

[Gould 85] J.D. Gould and C.H. Lewis.
Designing for Usability - Key Principles and What Designers Think.
Communications of the ACM 28(3):300-311, March, 1985.

[Green 86] Mark Green.
A Survey of Three Dialog Models.
ACM Transactions on Graphics 5(3):244-275, July, 1986.

[Hartson 89] H. Rex Hartson and Deborah Hix.
Human-Computer Interface Development: Concepts and Systems for Its Management.
Computing Surveys 21(1):5-92, March, 1989.

[Hartson 90] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix.
The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs.
ACM Transactions on Information Systems 8(3):181-203, July, 1990.

[Hayes 85] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner.
Design Alternatives for User Interface Management Systems Based on Experience with COUSIN.
In *Human Factors in Computing Systems*, pages 169-175. Proceedings SIGCHI'85, San Francisco, CA, April, 1985.

[Henderson 86] D. Austin Henderson, Jr.
The Trillium User Interface Design Environment.
In *Human Factors in Computing Systems*, pages 221-227. Proceedings SIGCHI'86, Boston, MA, April, 1986.

[Hill 86] Ralph D. Hill.
Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction — The Sassafras UIMS.
ACM Transactions on Graphics 5(3):179-210, July, 1986.

[Hix 89] Deborah Hix.
A Procedure for Evaluating Human-Computer Interface Development Tools.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 53-61. Proceedings UIST'89, Williamsburg, VA, November, 1989.

[Ingalls 81] Daniel H.H. Ingalls.
The Smalltalk Graphics Kernel.
Byte Magazine 6(8):168-194, August, 1981.

[Jacob 86] Robert J.K. Jacob.
A Specification Language for Direct Manipulation Interfaces.
ACM Transactions on Graphics 5(4):283-317, October, 1986.

[Karrer 90] Anthony Karrer and Walt Scacchi.
Requirements for an Extensible Object-Oriented Tree/Graph Editor.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 84-91. Proceedings UIST'90, Snowbird, Utah, October, 1990.

[Linton 89] Mark A. Linton, John M. Vlissides and Paul R. Calder.
Composing user interfaces with InterViews.
IEEE Computer 22(2):8-22, February, 1989.

[MacroMind 90] MacroMind, 410 Townsend Suite 408, San Francisco, CA 94107. Phone (415) 442-0200.
Director.
1990.

[Marchionini 91] Gary Marchionini and John Sibert, ed.
An Agendy for Human-Computer Interaction: Science and Engineering Serving Human Needs.
SIGCHI Bulletin 23(4):17-32, October, 1991.

[McCormack 88] Joel McCormack and Paul Asente.
An Overview of the X Toolkit.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 46-55. Proceedings UIST'88, Banff, Alberta, Canada, October, 1988.

[Morgan 83] C. Morgan, G. Williams, and P. Lemmons.
An Interview with Wayne Rosing, Bruce Daniels, and Larry Tesler.
Byte 8(2):90-114, February, 1983.

[Myers 84] Brad A. Myers.
The User Interface for Sapphire.
IEEE Computer Graphics and Applications 4(12):13-23, December, 1984.

[Myers 86] Brad A. Myers.
A Complete and Efficient Implementation of Covered Windows.
IEEE Computer 19(9):57-67, September, 1986.

[Myers 88a] Brad A. Myers.
A Taxonomy of User Interfaces for Window Managers.
IEEE Computer Graphics and Applications 8(5):65-84, Sept, 1988.

[Myers 88b] Brad A. Myers.
Creating User Interfaces by Demonstration.
Academic Press, Boston, 1988.

[Myers 89a] Brad A. Myers.
User Interface Tools: Introduction and Survey.
IEEE Software 6(1):15-23, January, 1989.

[Myers 89b] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Interactive Application Objects by Demonstration.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95-104. Proceedings UIST'89, Williamsburg, VA, November, 1989.

[Myers 90a] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal.
Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces.
IEEE Computer 23(11):71-85, November, 1990.

[Myers 90b] Brad A. Myers.
A New Model for Handling Input.
ACM Transactions on Information Systems 8(3):289-320, July, 1990.

[Myers 90c] Brad A. Myers.
All the Widgets.
SIGGRAPH Video Review 57, 1990.

[Myers 91a] Brad A. Myers.
Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 211-220. Proceedings UIST'91, Hilton Head, SC, November, 1991.

[Myers 91b] Brad A. Myers.
Graphical Techniques in a Spreadsheet for Specifying User Interfaces.
In *Human Factors in Computing Systems*, pages 243-249. Proceedings SIGCHI'91, New Orleans, LA, April, 1991.

[Myers 91c] Brad A. Myers and Brad Vander Zanden.
An Environment for Rapid Creation of Interactive Design Tools.
The Visual Computer; International Journal of Computer Graphics (6):(in press), November, 1991.

[Myers 92a] Brad A. Myers and Mary Beth Rosson.
Survey on User Interface Programming.
In *Human Factors in Computing Systems*, pages (in press). Proceedings SIGCHI'92, Monterrey, CA, May, 1992.

[Myers 92b] Brad A. Myers (editor).
Languages for Developing User Interfaces.
Jones and Bartlett, Boston, MA, 1992.

[Newbery 88] Frances J. Newbery.
An interface description language for graph editors.
In *1988 IEEE Workshop on Visual Languages*, pages 144-149. Pittsburgh, PA, October, 1988.
IEEE Computer Society Order Number 876.

[Newman 68] William M. Newman.
A System for Interactive Graphical Programming.
In *AFIPS Spring Joint Computer Conference*, pages 47-54. 1968.

[NeXT 91] NeXT, Inc. 900 Chesapeake Drive, Redwood City, CA 94063.
NeXTStep and the NeXT Interface Builder.
1991.

[Olsen 83] Dan R. Olsen, Jr. and Elizabeth P. Dempsey.
Syngraph: A Graphical User Interface Generator.
In *Computer Graphics*, pages 43-50. Proceedings SIGGRAPH'83, Detroit, MI, July, 1983.

[Olsen 86] Dan R. Olsen, Jr.
Mike: The Menu Interaction Kontrol Environment.
ACM Transactions on Graphics 5(4):318-344, October, 1986.

[Olsen 87] Dan R. Olsen, Jr.
Larger Issues in User Interface Management.
Computer Graphics 21(2):134-137, April, 1987.

[Olsen 88] Dan R. Olsen, Jr and Bradley W. Halversen.
Interface Usage Measurements in a User Interface Management System.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102-108. Proceedings UIST'88, Banff, Alberta, Canada, October, 1988.

[Olsen 89] Dan R. Olsen, Jr.
A Programming Language Basis for User Interface Management.
In *Human Factors in Computing Systems*, pages 171-176. Proceedings SIGCHI'89, Austin, TX, April, 1989.

[OWL 91] Owl International, Inc. 2800 156th Avenue SE, Second Floor, Bellevue, WA 98007. Phone (206) 747-3203.
Guide.
1991.

[Palay 88] Andrew J. Palay, *et. al*.
The Andrew Toolkit - An Overview.
In *Proceedings Winter Usenix Technical Conference*, pages 9-21. Dallas, Tex, February, 1988.

[Pike 83] Rob Pike.
Graphics in Overlapping Bitmap Layers.
ACM Transactions on Graphics 2(2):135-160, April, 1983.
Also appears in *Computer Graphics: SIGGRAPH'83 Conference Proceedings*, Detroit, Mich. Vol. 17, no. 3, July 25-29, 1983. pp. 331-355.

[Prior 91] Prior Data Sciences, 240 Michael Cowpland Drive, Kanata, Ontario Canada, K2M 1P6, Phone (613) 591-7235.
InterMAPhics.
1991.

[Rosson 87] Mary Beth Rosson, Suzanne Maass, and Wendy A. Kellogg.
Designing for Designers: An Analysis of Design Practices in the Real World.
In *Human Factors in Computing Systems*, pages 137-142. CHI+GI'87, Toronto, Ont., Canada, April, 1987.

[Sage 90] Sage Software, Inc. 1700 NW 167th Place, Beaverton, OR 97006. Phone (503) 645-1150. A division of InterSolv.
Dan Bricklin's Demo II, Version 3.0.
1990.

[Samuelson 89] Pamela Sameuelson.
Why the Look and Feel of Software User Interfaces Should Not be Protected by Copyright Law.
Communications of the ACM 32(5):563-572, May, 1989.

[Scheifler 86] Robert W. Scheifler and Jim Gettys.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April, 1986.

[Schmucker 86] Kurt J. Schmucker.
MacApp: An Application Framework.
Byte 11(8):189-193, August, 1986.

[Schulert 85] Andrew J. Schulert, George T. Rogers, and James A. Hamilton.
ADM-A Dialogue Manager.
In *Human Factors in Computing Systems*, pages 177-183. Proceedings SIGCHI'85, San Francisco, CA, April, 1985.

[Sheil 83] Beau Sheil.
Power Tools for Programmers.
Datamation 29(2):131-144, February, 1983.

[Shneiderman 83] Ben Shneiderman.
Direct Manipulation: A Step Beyond Programming Languages.
IEEE Computer 16(8):57-69, August, 1983.

[Singh 88] Gurminder Singh and Mark Green.
Designing the Interface Designer's Interface.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 109-116. Proceedings UIST'88, Banff, Alberta, Canada, October, 1988.

[Singh 89] Gurminder Singh and Mark Green.
Chisel: A System for Creating Highly Interactive Screen Layouts.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 86-94. Proceedings UIST'89, Williamsburg, VA, November, 1989.

[SmethersBarnes 90]
SmethersBarnes, P.O. Box 639, Portland, Oregon 97207, Phone (503) 274-7179.
Prototyper 3.0.
1990.

[Smith 82] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem.
Designing the Star User Interface.
Byte 7(4):242-282, April, 1982.

[Spinnaker 91] Spinnaaker Software. 201 Broadway, Cambridge, MA 02139-1901. Phone (617) 494-1200.
Spinnaker PLUS.
1991.

[Stallman 79] Richard M. Stallman.
Emacs: The Extensible, Customizable, Self-Documenting Display Editor.
Technical Report 519, MIT Artificial Intelligence Lab, August, 1979.

[Sukaviriya 90] Piyawadee Sukaviriya and James D. Foley.
Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 152-166. Proceedings UIST'90, Snowbird, Utah, October, 1990.

[Sutherland 63] Ivan E. Sutherland.
SketchPad: A Man-Machine Graphical Communication System.
In *AFIPS Spring Joint Computer Conference*, pages 329-346. 1963.

[Swartout 82] W. Swartout and R. Balzer.
The Inevitable Intertwining of Specification and Implementation.
Communications of the ACM 25(7):438-440, July, 1982.

[Swinehart 86] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann.
A Structural View of the Cedar Programming Environment.
ACM Transactions on Programming Languages and Systems 8(4):419-490, October, 1986.

[Teitelman 79] Warren Teitelman.
A Display Oriented Programmer's Assistant.
International Journal of Man-Machine Studies 11:157-187, 1979.
Also Xerox PARC Technical Report CSL-77-3, Palo Alto, CA, March 8, 1977.

[Tesler 81] Larry Tesler.
The Smalltalk Environment.
Byte Magazine 6(8):90-147, August, 1981.

[Vander Zanden 90]
Brad Vander Zanden and Brad A. Myers.
Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces.
In *Human Factors in Computing Systems*, pages 27-34. Proceedings SIGCHI'90, Seattle, WA, April, 1990.

[VisualEdge 90] Visual Edge Software Ltd., 3870 Cote Vertu, Montreal, Quebec H4R 1V4. Phone (514) 332-6430.
UIMX.
1990.

[Vlissides 90] John M. Vlissides and Mark A. Linton.
Unidraw: A Framework for Building Domain-Specific Graphical Editors.
ACM Transactions on Information Systems 8(3):204-236, July, 1990.

[Vlissides 91] John M. Vlissides and Steven Tang.
A Unidraw-Based User Interface Builder.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 201-210. Proceedings UIST'91, Hilton Head, SC, November, 1991.

[Wiecha 90] Charles Wiecha, William Bennett, Stephen Boies, John Gould, and Sharon Greene.
ITS: A Tool for Rapidly Developing Interactive Applications.
ACM Transactions on Information Systems 8(3):204-236, July, 1990.

[Wilde 90] Nicholas Wilde and Clayton Lewis.
Spreadsheet-based Interactive Graphics: from Prototype to Tool.
In *Human Factors in Computing Systems*, pages 153-159. Proceedings SIGCHI'90, Seattle, WA, April, 1990.

[Wilson 90] David Wilson.
Programming with MacApp.
Addison-Wesley Publishing Company, Reading, MA, 1990.

[Wolber 91] David Wolber and Gene Fisher.
A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 221-230. Proceedings UIST'91, Hilton Head, SC, November, 1991.

[XVT 91] XVT Software, Inc., Box 18750 Boulder, CO 80308. Phone (303) 443-4223.
XVT.
1991.

[Yourdon 89] Edward Yourdon.
Modern Structured Analysis.
Prentice-Hall, Englewood Cliffs, N.J., 1989.