# Putting Your Best Face Forward: Designing an Effective User Interface

Maria R. Capucciati

Maria R. Capucciati has been working with developers of Windows-based systems since 1985. She specializes in user interface design and standards.

In the Microsoft® Windows™ operating system, a great deal of control is in the hands of developers. Nearly unlimited object manipulation is possible. There is a vast potential for building dazzling, elegant, useful applications—and equally vast potential for mass interface confusion. The results of poor interface design can range from grudging acceptance by users to a significant drop in your product's sales.

Microsoft has been slow to set interface design guidelines. Until the publication of The Windows[1] Interface: An Application Design Guide (Microsoft Press, 1992), IBM's SAA/CUA guidelines were the standard for Windows-based applications. The Windows Interface is an excellent starting point for designing user interfaces for Windows 3.1. It expands on the familiar CUA standards. This book specifies right up front that it does not require conformity to its guidelines, which are classified as recommended, optional, or suggested.

To say that conformity to the recommended guidelines is not required is akin to saying that all car manufacturers are free to place the steering wheel, brake, and accelerator anywhere in the car they want. Then again, just because an application conforms to the guidelines doesn't mean it's well designed. Your application interface should be easy to read, easy to look at, and easy to use in behavior and arrangement. To produce a functional and visually pleasing interface, you need an understanding of the business requirements of your application and a good sense of aesthetics. The guidelines in The Windows Interface are only part of the story.

## Don't Follow Me, I'm Lost

Before I get into some methods for producing good design, let's look at what not to do. I've changed some details to protect the guilty, but these are actual screens designed by experienced developers.

Figure 1 shows a screen designed to gather information on an individual for coverage under an insurance plan. Your eyes bounce all over this screen—the arrangement is abysmal and nothing aligns. Which of these data elements are related? (Has this developer ever heard of group boxes?) Are any controls dependent upon the others being selected? Which field is the most important? Are the numbers at the bottom editable? Whether they are or not, what do they mean? Actually, depending upon the answers to these questions, the screen in Figure 1 should be relatively easy to fix.

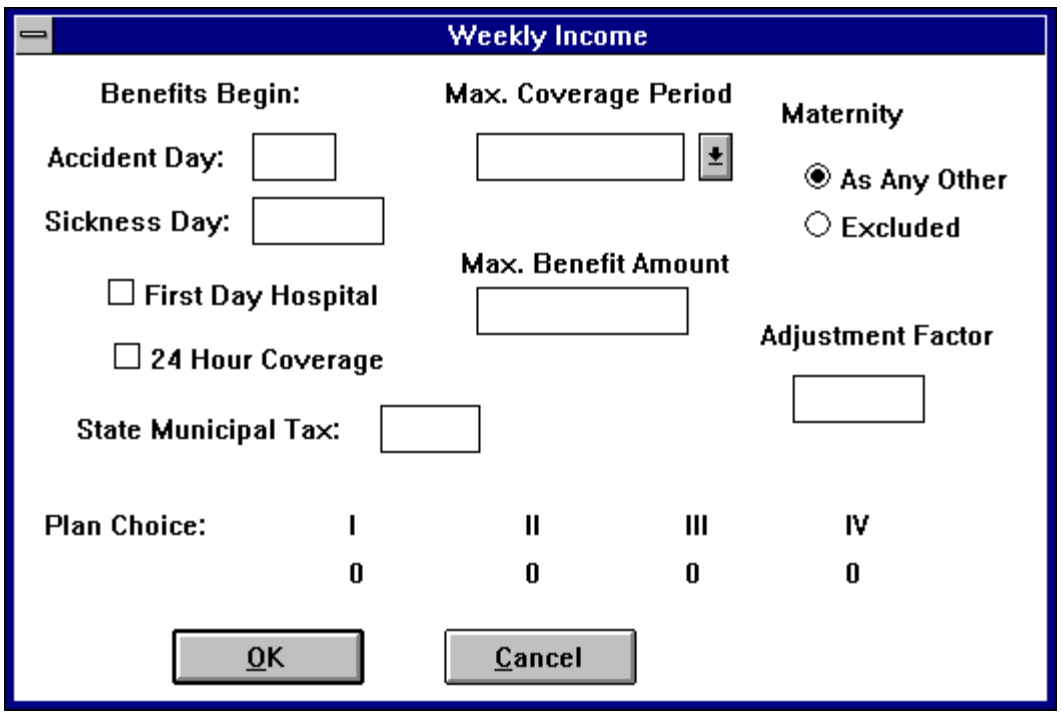


**Figure 1 Don't let this happen to your program**

Figure 2 is more subtle. The appearance of the screen isn't great, but the sequence of operations is worse. The screen lists an individual applicant's coverage under an insurance plan, including coverage for existing family members. The static text field at the bottom issues a useless instruction, since there are no selections for "new application" or "update application" anywhere on screen. I watched users click on the words NEW and UPDATE expecting something to happen. The developer combined two mutually exclusive conditions (New and Increase) into one check box (Application Status). The Family Members check boxes were dependent upon which Member Number the user selected from the list box. Application Status was enabled or disabled incorrectly depending upon the existence of Family Members. As if that weren't confusing enough, when the users clicked on the Application Status check box, the dialog box disappeared! When I spoke with the developer, I discovered that there were actually three, not two, possible statuses and they were all mutually exclusive. In this case, redoing the interface required revision of the application status code, not just shifting controls around on screen.

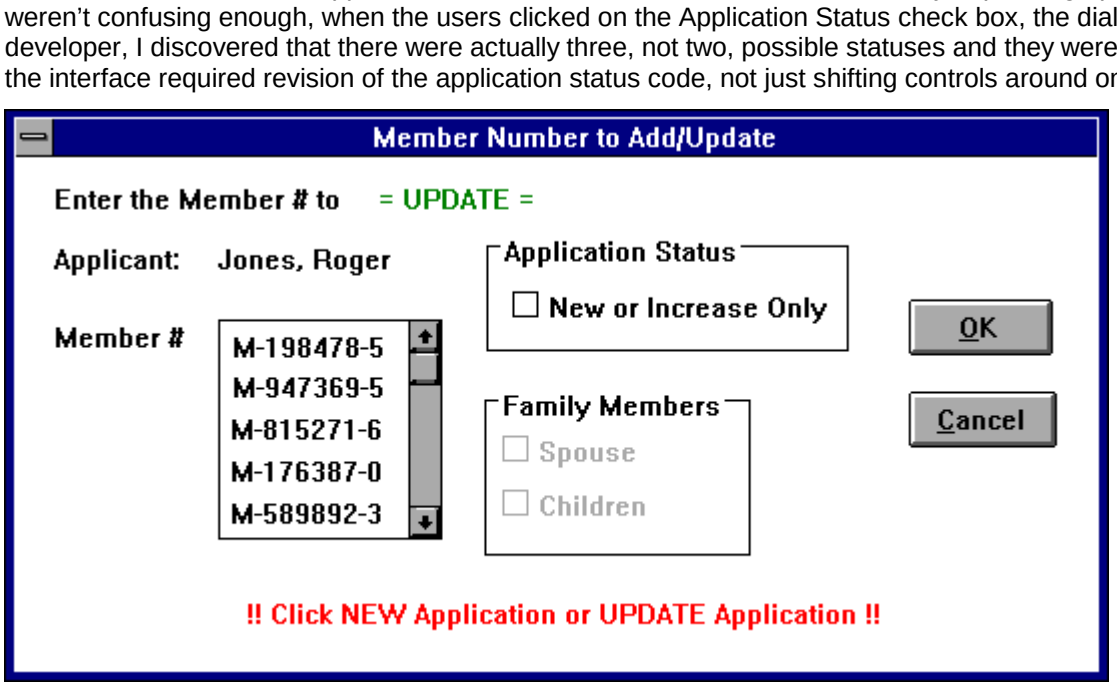


**Figure 2 Inscrutable instructions**

The author of the dialog in Figure 3 seems to have favored alignment over ease. In Western culture, we read left to right and top to bottom. This screen forces the user to fill in fields on the right and then travel to the left to perform an action. Plus, all the edit fields are exactly the same length. A cursory reading of the control labels will tell you that the edit fields should be varying lengths to accommodate the type of data being entered.

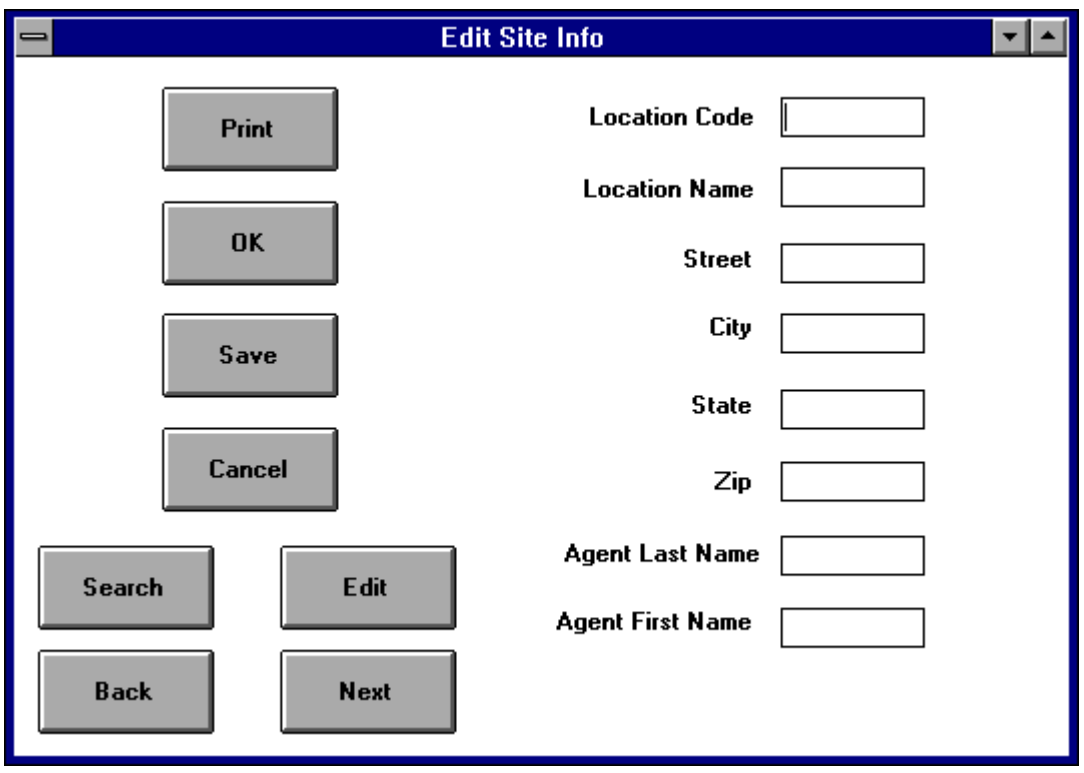


**Figure 3 Alignment über alles**

Using a metaphor familiar to the culture you are designing for can often help you build a sensible interface. Here, the developer might try using an address book or a Rolodex arrangement. Several of the command buttons are redundant (OK and Save? Edit?) and they are so oversized they take up nearly half the dialog box. A huge amount of unnecessary white space is included in Figure 3.

The developer of the screen in Figure 4 fell in love with cascading menus. I've even seen screens where cascades are falling off both sides of the menu! If there are really this many possible choices, presenting them in a dialog box is often a better choice—don't force the user into a dexterity contest with the mouse cursor.

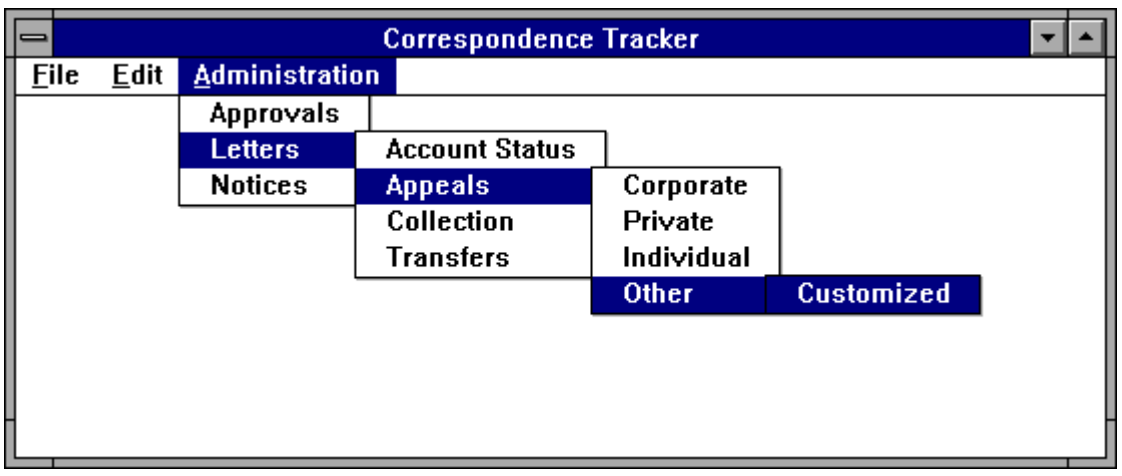


**Figure 4 Cascade confusion**

Figure 5 is crowded and awkward. There are too many group boxes and the command buttons look swollen and clunky. Also, the buttons are all different sizes. This was intended to give the user easy access to screens where he or she could view or update the product information. This could be accomplished with pull-down menus or by titling the command buttons themselves (Optional Services..., Annual Earnings..., and so on). Having four buttons perform the same function is excessive.

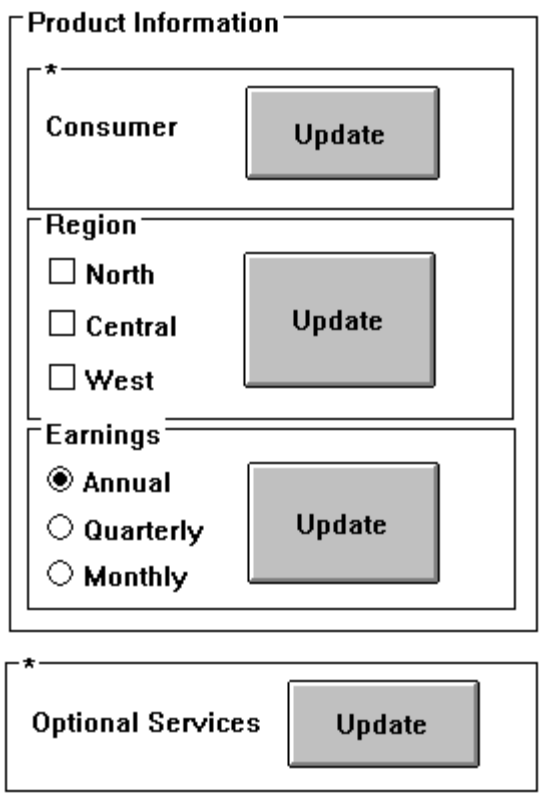


**Figure 5 Bulky buttons**

Lest we think that Microsoft designs only perfect screens, let's look at a couple of dialog boxes in Microsoft Excel version 4.0 whose layout could be improved. In their present state, I find them unclear and bothersome to use.

Figure 6 (File Page Setup) is a great example of a dialog box that worked fine in Microsoft Excel 3.0 but suffers from overkill in version 4.0. It's very busy, with odd empty places. There are too many group boxes for the space, and all the controls and text are crammed on top of each other. Even though Microsoft Excel has many more options to consider in Page Setup than Microsoft Word, for example, it's still no excuse to jam most of them into a three-by-five dialog.

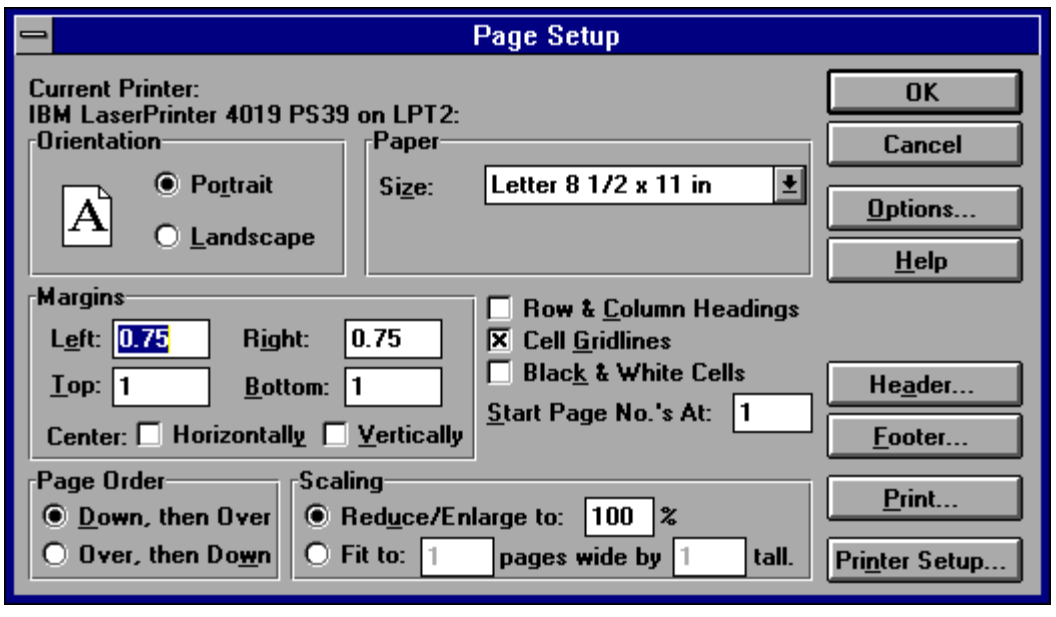


**Figure 6 Persnickety page setup**

Figure 7 (Formula Select Special) meanwhile could use some group boxes. Because it isn't visually clear which choices depend upon others, it appears to exhibit some strange behavior. It defaults to Notes selected and all the check boxes selected and grayed. This makes it appear as though Microsoft Excel will select the Numbers, Text, Logicals, and Errors within the Notes. The check boxes aren't really displaying any of the three possible states (on, off, and gray). And what about the other radio button, set to Direct Only? From its placement I would guess it has to do with Precedents and Dependents, but a group box would have clarified this. While this dialog box doesn't function incorrectly, the choices are not nearly as evident as they could be.

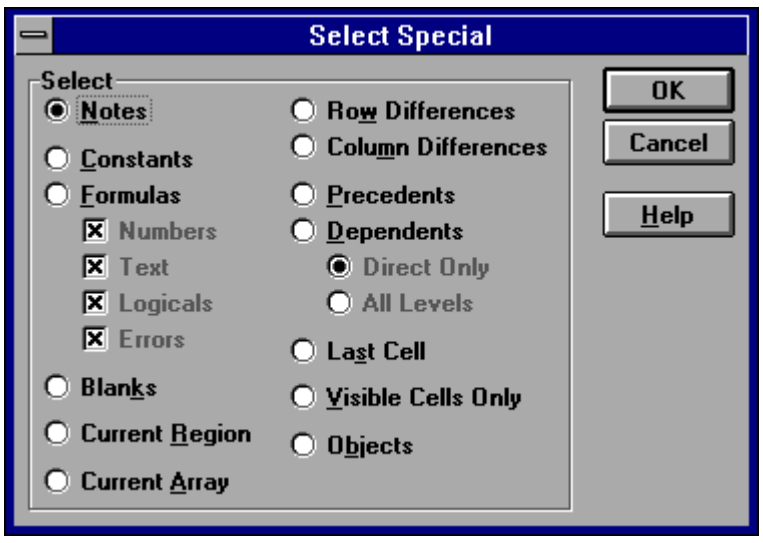


**Figure 7 Exasperating options**

Figure 8 shows actual error message boxes taken from an application I worked with some time ago. They may make some twisted sense to someone, but they make the best case I've seen for checking the clarity of messages you author with someone else—hopefully potential users.


**Figure 8 Time to call the help desk**

In the time I've been working with developers of Windows-based applications, I've seen some pretty amazing and esoteric uses for the various controls. Figure 9 suffers from an overkill of radio buttons. Because radio buttons are such an easy, visible way to present all the available choices to the user, developers often use them for data that would be better off in another form. Displaying the fifty states this way takes up an inordinate amount of room. (Of course, if you do opt for this method, be sure your list is nonalphabetical and has no perceivable order!) I wouldn't expect anyone to remember all the state abbreviations, so I'd choose a drop-down combo box for this field. Also, people do forget that check boxes are not mutually exclusive. The gender data field would be better if radio buttons were used.

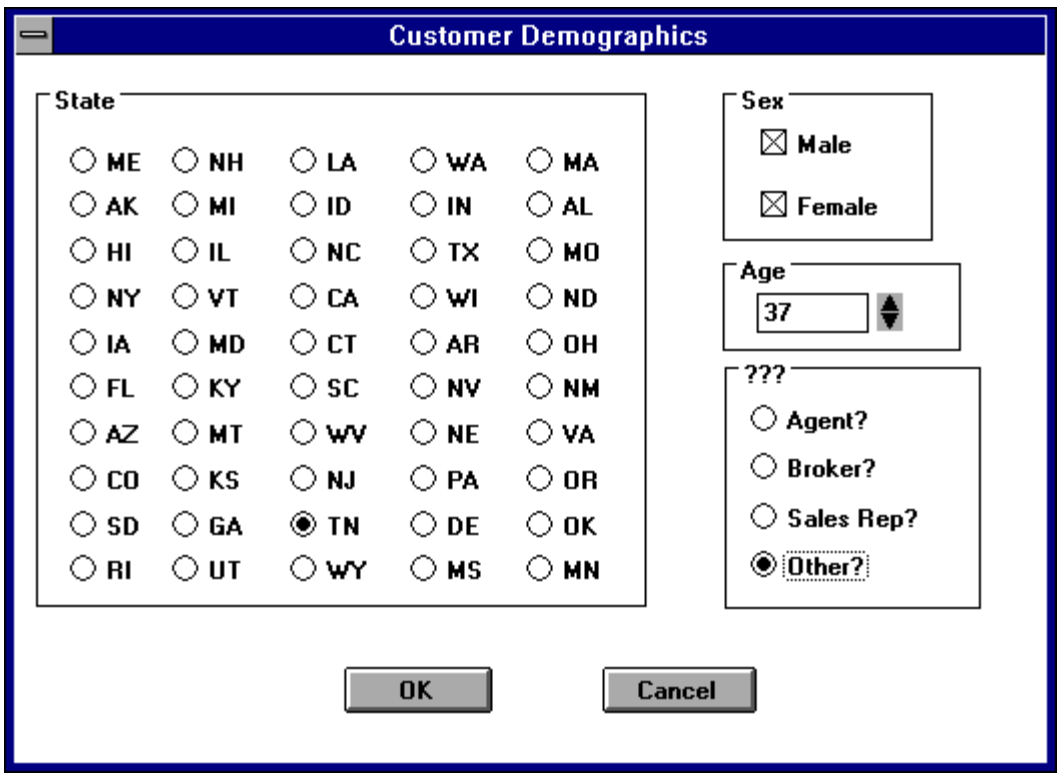


**Figure 9 States of disarray**

Controls couldn't be vaguer than the ones in Figure 10. The users of this application told me they could not type into the text box unless they clicked the command button, which was inactive unless the check box was checked. Besides violating all the standard uses for check boxes and command buttons, what about the button label? Info==>?

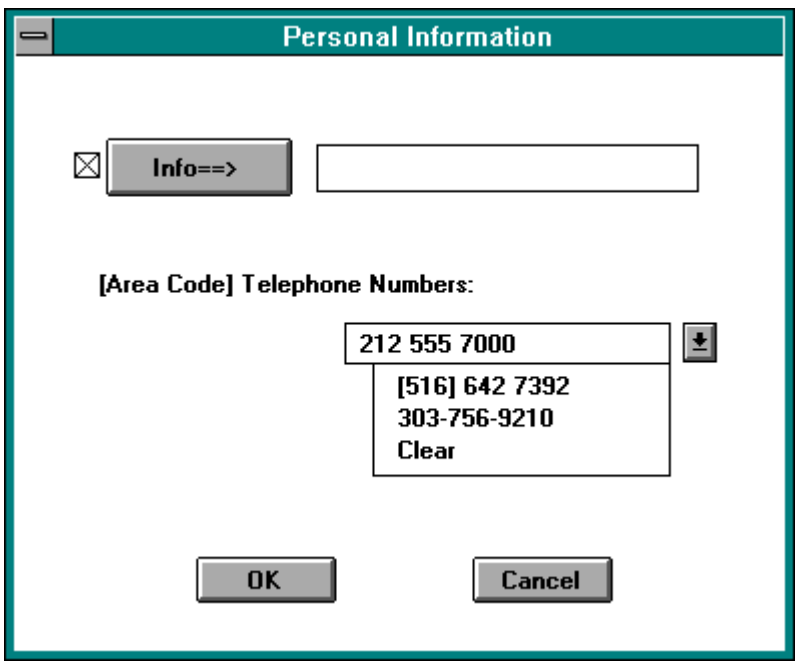


**Figure 10 Disordered dialog**

The drop-down combo box on this screen is an unmitigated disaster. There's clearly no control over the number format, and an action (Clear) has found its way into a list of selections. You don't have to get this exacting with control labels, either. This one states the obvious and is much too long.

Anyone familiar with the Font/DA Mover accessory on the Macintosh will recognize what Figure 11 is attempting. Building one set of fields from another is a good idea, but I could not convince this developer that the chevrons had to point in opposite directions.

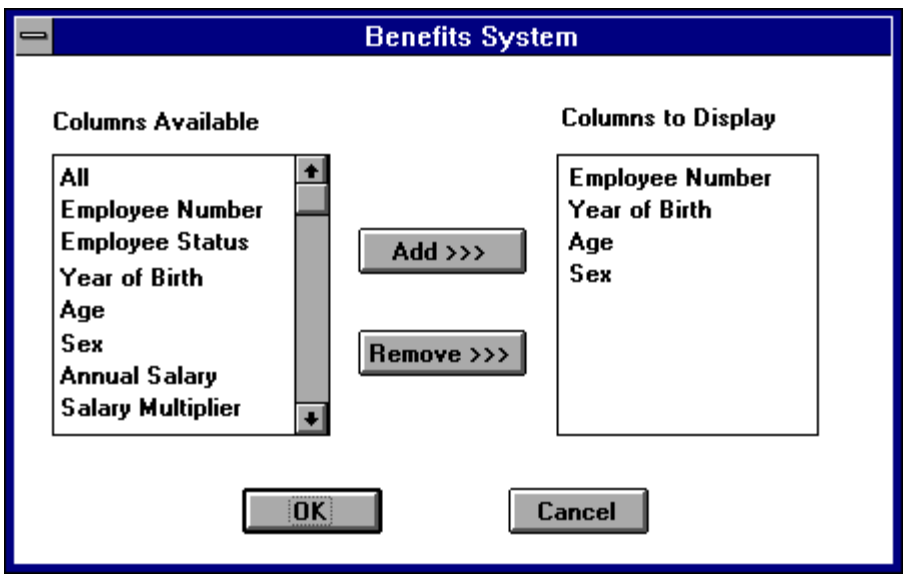


**Figure 11 Wrong way on remove**

Command buttons are often used for apparently conflicting actions. Figure 12 is another illustration of contentious buttons. What's the difference between Apply and Accept? Since there are no visible changes to apply, the button is superfluous anyway. Many times developers are only trying to provide command buttons for subtle differences in function, which ends up confusing the users even further. The actions should be obvious. This applies for bitmaps on command buttons as well. If users have to spend several minutes figuring out what the pictures mean, the purpose has been diminished.

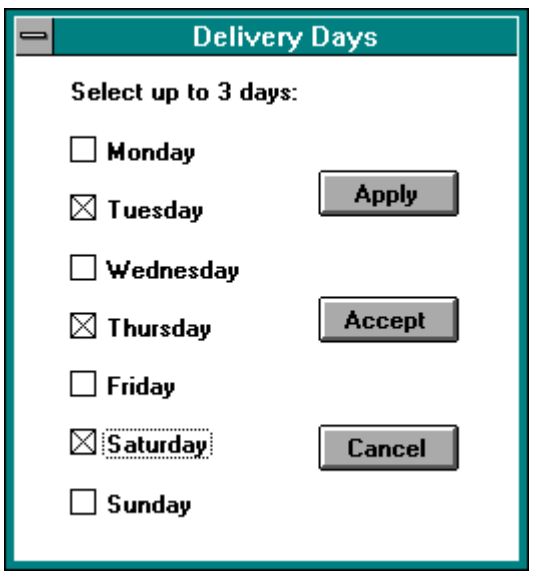


**Figure 12 Contentious controls**

Almost without exception, the developers who committed these design blunders made one fundamental mistake: they spent little or no time doing any form of analysis before beginning to code. If you're honest, you'll probably admit that this is not an uncommon scenario, even among experienced developers. They began with a business need, skeleton specs, and gave very little thought to interface design. When interface problems arose, the developers would find that some of them were structural in nature. Fixing them would have entailed nearly starting over and rethinking not just the code, but things like the database design. Quick-and-dirty is fine for temporary applications, but how temporary are they really? I've seen stopgap applications developed for corporate in-house use linger up to two years while everyone complains about them. Users won't purchase or use programs they perceive as unreliable or too difficult to use.

## Performing the Appropriate Analysis

One of my colleagues returned from a conference with a button that read "Know Thy Users—For They Are Not You." Fathoming the mind of a user is no mean accomplishment. In his book Information Anxiety (Doubleday, 1989), Richard Saul Wurman discusses the "inequity of perception"—even though no two people understand a concept in the same manner, we act as if everyone sees the world the way we do. You need to put aside this natural inclination and get to know your target market.

Who are your users? If you are writing an application in a corporate environment, you can find out easily—go interview them. Talk to them as much as they can stand seeing you. Live in their department for a while, if you can afford the time. The advantage of corporate users, besides their accessibility, is that you can virtually tailor your interface to their specific needs.

If you're a vendor, you'll probably handle the design phase of your development quite differently. ISVs tend to develop quietly (at least to their competitors)! Development is often driven by the marketing department, whose staff is constantly soliciting user feedback. Initial specs are written, and the product is prototyped and tested on a few select users. Development generally continues in an iterative mode until the product is ready for alpha testing. Vendors need to think about how the user will interact with the application, how it will operate and how to implement it. User feedback should be analyzed for trends and common threads rather than taken at face value. Sometimes the best feedback comes not from users, but from looking at other products and talking with people who have a similar vision for the application.

Whether you have corporate users or an independent customer group, there are questions whose answers will have bearing on your interface design.

What type of positions do the users occupy? Is it a homogeneous or heterogeneous group? Are you designing an executive information system for a board of directors, or an accounting system for the green eyeshades? I worked on an EIS once for a group of executives who wouldn't be caught dead using a keyboard (a menial task in their eyes). This was pre-Windows, so we opted for a simple iconographic interface with a touch screen. The project was so successful it expanded to nearly two hundred users across several departments.

Are they creative professionals: graphic designers, artists, architects, musicians? If your product is aimed at this segment, listen carefully. These are challenging users with an unerring sense of structure. Be prepared to learn from them.

Are they PC-literate? What environments are they familiar with? If you are writing an application for brand-new PC users, they will probably display more hostility than enthusiasm at first. Concentrate on making the interface so friendly and inviting they'll want to take it home.

How long have the users been doing their jobs? What's the tolerance level for a learning curve? Long-time employees usually know their stuff, but are often resistant to change. The learning curve for your application can become a surprising bone of contention.

What's their physical working environment? I implemented a Windows-based workstation for securities traders who were often on their feet, in constant motion, shouting on the telephone. They had to be able to read the market data from a distance of several feet. So the developers switched to uniformly large fonts in all applications. Workers at a manufacturing site can face similar circumstances. Consider the attention span the environment forces the users to adopt, and design accordingly.

How frequently will they use the application? If someone will be staring at your interface all day long, even the smallest design flaws quickly become major annoyances. If they'll enter the application, perform their task quickly and efficiently, and exit with minimum fuss, the design of your interface may be less critical.

Your next step is to understand what the industry, company, or department does. Summarize the business purpose of your application in one paragraph and keep it in mind as you are designing the interface. Does each control on the screen help accomplish that purpose? If not, you can probably dump it. Will your application have multiple purposes? If it does, it's easier to define and include them all up front than decide to implement major new functions after ninety percent of your code is written. Complexity isn't negative; it just has to be engineered carefully so the users are in control.

Once you have defined the primary purpose of your application, diagram how it's going to accomplish it. If the process already exists, find out how users do the tasks now. You may be able to automate a manual procedure, saving time and money. Group the steps into business functions or activities. Check out Figure 13 as an example. I work for an insurance company, and one of the first workflows I documented was the renewal procedure (when they decide how much to charge you for your group health insurance). As you peruse it, keep in mind that you're looking at part of one activity in the total process. There are manual calculations all over the place; one or two PC systems and three different mainframe systems are involved. There are three ways to renew a case; all involve unique data and formulas. In the first step, the underwriter is gathering material from a two-inch thick file folder and filling out a form to use for data entry. It's scary, but this type of scenario exists in corporations all over the country.

**Figure 13 Diagramming Workflow**

Determine New Billing Rates

| | SOURCE | INPUT | PROCESS | OUTPUT |
|---|---|---|---|---|
| 1 | Underwriter's case file | Current billed, tabular, and manual rates, census, plan of benefits, premium and claims info, exposure data, coverage | Underwriter calculates new tabular and manual rates (manually or on PC); completes renewal system input form | Renewal system input form |
| 2 | Renewal system input form | Data for APL program | Underwriter enters data into APL program on PC | Renewal rate level, renewal results, including new billing rates |

| | | | | |
|---|---|---|---|---|
| 3A | Conventional renewal | New billing rates, renewal results | Review for approval. If approved, make request to APL program. | APL program produces: policy exhibits, rate spec, policyholder letter |
| B | Alternate renewal B | New billing rates, renewal results, carryover, reconciliation report, claims incurred | Underwriter performs steps as for conventional renewal, then manually calculates B portion and develops formula | Policyholder letter with B rate spec via mainframe system |
| C | Alternate renewal C | New billing rates, renewal results | Manually calculate C portion of renewal | C rate spec |
| 4 | Underwriter | Rate spec (in all cases) | Input into mainframe system and print screens | Approved rates sent to mainframe system |
| 5 | Underwriter | Policy exhibits, rate spec, policyholder letter, memo, cost containment report | Control unit verifies new rates on mainframe system, manually gathers forms into renewal package | Verified rates, renewal package |

Streamlining the users' workflow will bring initial howls of protest, but simply automating an inefficient workflow is worse than leaving the manual procedure in place. Which activities can be combined or eliminated? How many manual steps can the application do instead? Is it your goal to replace several antiquated systems with a new one? What capabilities can you provide that were previously nonexistent or impossible under the circumstances? For example, do you want users to be able to share data over a network? Can the database be maintained on a server instead of the mainframe?

Think carefully about the sequence of the functions. You don't want to force users into unnecessary modes, but some steps of the workflow can be dependent upon others. Prototyping the application iteratively using a facility such as the Visual Basic™ programming system can be an excellent method for assuring that workflow translates properly to screen flow. User-requested changes to the prototype are far less painful than changes to the beta version.

## Basic Design Principles

We've all heard the acronym KISS—Keep It Simple, Stupid. I prefer to think that KISS stands for Keep It Simple and Straightforward. Once you know what fields or choices to include in your application, ask yourself again which are really necessary. Labels, static text, check boxes, group boxes, and option buttons often clutter the interface and take up twice the room mandated by the actual data. If a user can't sit before one of your screens and figure out what to do without asking a multitude of questions, your interface isn't simple enough.

Two easy ways to increase interface clarity are to promote alignment and consistency. When controls or fields do not align perfectly on the screens, users are often vaguely uncomfortable without being able to say why. I personally get seasick just looking at the dialog box in Figure 1. Lack of balance and harmony are annoying to the human eye. Also, try not to mix justification. Text is generally left-aligned (except for message box text, which is centered); numbers right-aligned.

Inter- and intra-screen consistency includes the size of controls, all labels, position, behavior, and style. Don't create a big command button on one screen and reproduce it in miniature on another. If you use three-dimensional controls, use them throughout your screens. If you label an edit field Customer Name on one screen, don't call the same field Client Name on subsequent screens. If data fields repeat on several screens and it's possible to give them the same physical placement on each screen, do so. Include a consistent display of information on the state of your program at all times. Help desk operators will love you if they're able to tell a confused user to read the status bar in the lower-left corner.

The behavior of most Windows controls is predefined, and for good reason. You should change their standard behavior only under rare and unusual circumstances.

Proper screen layout is essential to the overall effectiveness of your interface. Here are a number of methods to help you achieve it.

- Segregate data functionally. All the information the user needs to complete a particular activity should be on one screen.
- A cluttered and busy screen distracts from the task at hand. Balance this against the inconvenience of users having to go through multiple screens to find all the data they need.
- Place the most frequently used fields where they will be seen first. Remember that we read left to right, top to bottom. Carefully emphasize/deemphasize each control through its placement on screen.
- Group related controls (a perfect example is name/address/city/state/zip). The more fragmented the placement of controls, the less obvious their meaning. This also applies to menu items.
- Don't leave huge spaces between related objects–you force the eye to travel too far. Aim for even spacing—about one-eighth to one-quarter of an inch between related controls.
- Diagram the navigational sequence a user must follow to complete each task in your application. Putting it on paper helps you see the flow and the number of levels. Avoid sequences, for example, which take users through umpteen successive modal dialog boxes. Figure 14 shows five, which is a bit out of hand.
- Be familiar with basic principles of graphic design. Many knowledgeable, highly technical developers ignore this altogether and their screens show it. I can't count the number of times developers have said to me "But it's only cosmetic!" as though their application's functionality will somehow shine through no matter how sloppily it's presented. Some GUI application vendors take this aspect of their product so seriously they contract out the design of their user interface to professional designers. If this isn't feasible, read a book or article on the fundamentals of composition. Next time you pick up a magazine, study the advertisements. Consider the methods the graphic designer is using to attract you to the product. You want to attract people to your product also.

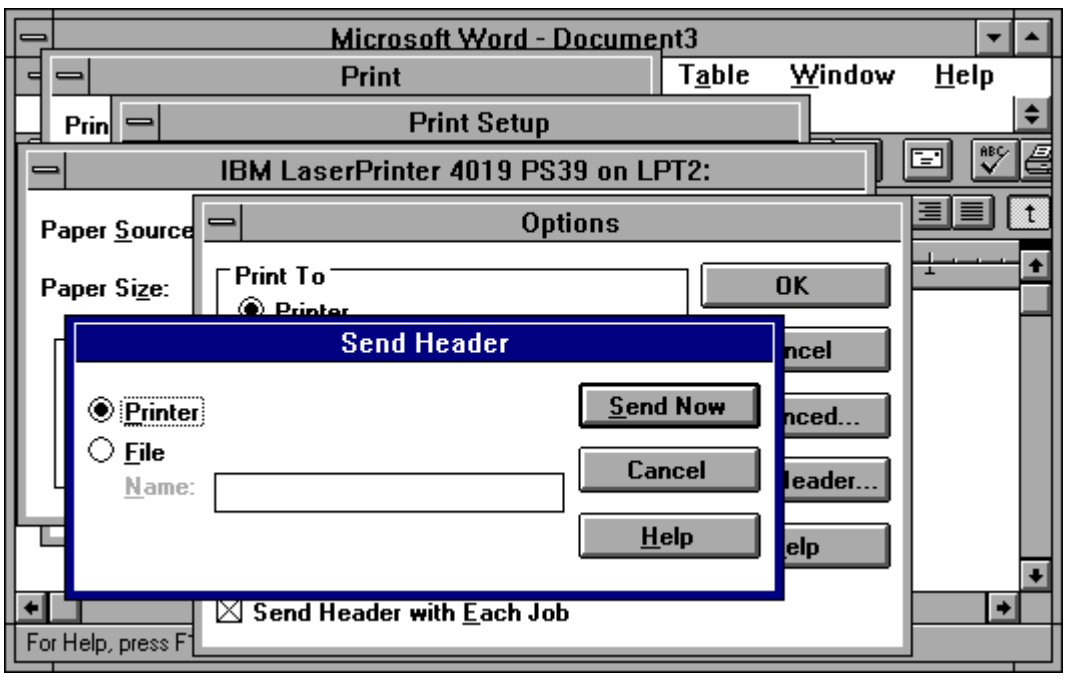


**Figure 14 Modal madness**

# Modeling the Application

Once you have defined all the application's functions, determine how many windows you wish to employ in presenting them to the users. We know that all Windows-based programs start in a main window, with sub-tasks represented in MDI windows or dialog boxes. Use MDI windows when the user needs to perform parallel tasks in multiple windows within the workspace (for example, exchanging data between documents). Because MDI windows can also be sizable and scrollable, they allow users unhindered access to all the application data. An ideal use for an MDI window is the presentation of dynamic data (for example, a stream of continuously updating news headlines), which would be inappropriate for a dialog box.

Dialog boxes are usually a fixed size, which makes them preferable for gathering discrete sets of information. The user can invoke a dialog and make selections or entries in any sequence.

Dialog boxes can be modal, semimodal (allowing limited access to the application window), or modeless. I recommend keeping all dialog boxes, including modal ones, movable. Few conditions short of complete system failure merit immovable dialogs. Modeless dialogs, while often the best and most flexible for end users, are rarely found—perhaps due to the additional coding effort required. One of the few true modeless dialog box I've seen in a commercial application is the grammar checker in Word for Windows.

To save time, you can use the common dialogs in COMMDLG.DLL. If your application will support the menu actions Open, Save As, Print, Print Setup, Color, Font, Find, and/or Replace, you'll save yourself considerable coding time.

The user interface must be the nucleus for every action the users are able to perform. User-initiated actions are triggered in Windows-based applications by menu items, command buttons, icons, and object embedding and linking (OLE). Functionality can be accessed multiple ways in an interface; for example, the Toolbars in Word and Microsoft Excel consist of command buttons that represent actions also found on the menu. The user can choose the means of access and modify the interface through a Preferences menu. If you provide only one way to perform a particular action, make sure it is immediately evident to the user. For example, many drawing and painting applications have no menu selections at all for tools or colors. This is acceptable if the application window comes up by default with a visible toolbar and color palette.

The arrangement of actions on the menus needs careful consideration. Use your workflow research to make a list of every possible action a user of your application can perform, then group related items under headings. The common menu actions (such as Save, Print, Exit, Cut, Copy, Paste) already have standard homes. Move them only under select circumstances. A good example of when to move a common action is illustrated in Figure 15. This application generated hard copies of several independent entities (a cost summary, a varying number of worksheets, and a sales proposal). The workflow indicated that the users would always print these items at the end of the session. The developer wisely chose to separate the Print function from the File menu, and positioned it at the end.

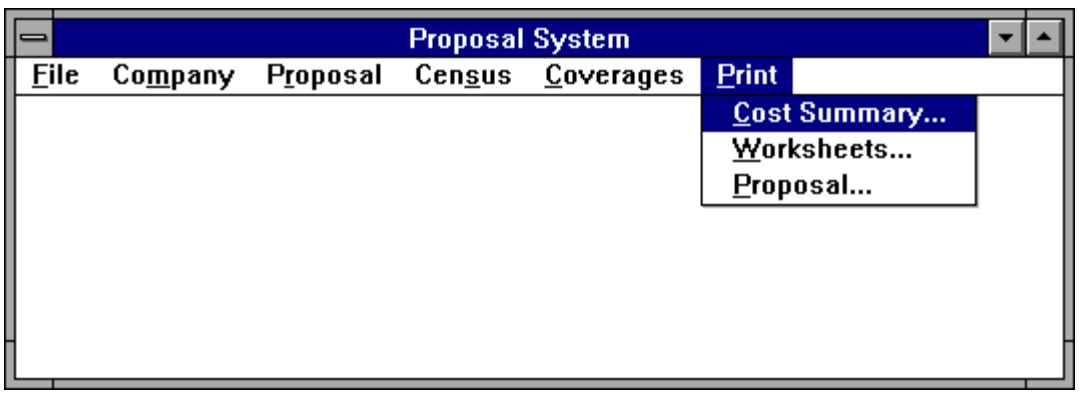


**Figure 15 I did it my way**

Arrange other menu items depending upon a number of factors: frequency of use, importance, the workflow sequence of the tasks, or an alphabetical or numerical order. Whatever the order of the menu items, it should make sense to your users. The menu item names need to be both concise and descriptive. Avoid action names that look redundant. Even if you know that Add and Insert perform two different actions, the users may have trouble distinguishing the functions. Change one to describe the action more accurately. A confusing menu and an improved version are shown in Figure 16. When all the actions refer to the same object (the record in this case), it's unnecessary to repeat it in each menu item.

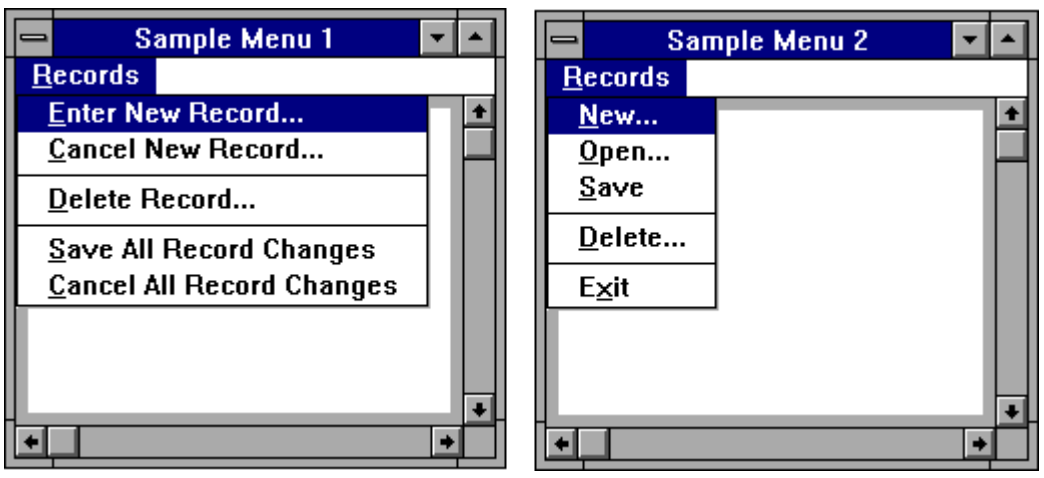


**Figure 16 Less is often more**

Command buttons are also used to initiate actions, most commonly in dialog boxes. When command buttons are present in main windows, they are usually on a control bar (such as the Toolbar or ribbon in Word). I've seen some developers use command buttons with text labels in the client area of an application window, on the theory that all actions should be immediately visible to the user. This is fine if there are a limited number of actions, but not everything on a menu can or should be presented as a button. Taking up half the client area with text-filled command buttons is clearly not a viable option. If you can represent an action on a button using a smaller graphical symbol, then place them on a control bar for accessibility. Otherwise, leave the actions on the menu.

As with menu items, ensure that command button names are short and descriptive. Whenever possible, make them action verbs. Arrange the buttons in evenly spaced, logical groups on the screen or control bar.

There are a number of creative ways to use icons to perform actions. For example, the Visual Basic Professional Toolkit contains a custom control called the animated button (see Figure 17). The animated button control can use any set of icons, bitmaps, or metafiles to create the appearance of motion. For example, an icon that represents an open book can "turn pages" when clicked with the mouse. You could use this kind of control in place of a command button labeled Next Page. Animated buttons could also find a place in button bars or on help screens. Use them with caution so your application doesn't turn into a cartoon.

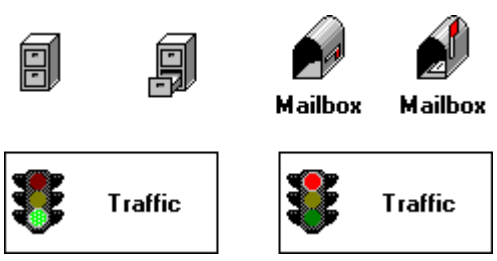

**Figure 17 Bitmaps for animated buttons**

The interface should always include immediate visual or auditory feedback. For example, I am writing this article using Word for Windows. The position and shape of the cursor (an I-beam) tell me I'm in text editing mode. The status bar at the bottom of my screen displays page number and section number. The ellipses on menu items and command buttons tell me a dialog box is coming.

Here are some ideas for error prevention:

- Disable choices that will do damage if the user selects them at a particular point in your application—gray them or hide them. It's simple and can save you enormous grief.
- Use list controls as often as is practical instead of edit controls so users can't make typing mistakes. If possible, create default selections.
- Force the users to confirm any potentially destructive action by putting up a message dialog asking them to verify their choice. (If you expect some expert users of your product, add an option to disable verification to keep them from going crazy.)
- Decide how important data accuracy is to your users and use validation accordingly. Do you validate the user entries after each keystroke, when the user moves to the next field, at the end of the screen, or when the user clicks OK? If you perform validation from field to field, the user may not notice the extra second or so lag time. Figure 18 illustrates field-specific validation. The application displayed an error message immediately after the user entered an invalid state abbreviation then hit the Tab key to move to the zip code field. If you wait until a screenful of fields has been filled, validation can take as long as twenty to thirty seconds depending on what you're validating against (say a database on a remote mainframe), which is endless to a user who wants to move on to the next function. Even a five or ten second wait seems long. Using a gauge control or a horizontal scroll bar to indicate progress or percentage complete can go a long way toward minimizing user impatience. If you know that the operation will take more than five seconds, use a progress indicator.

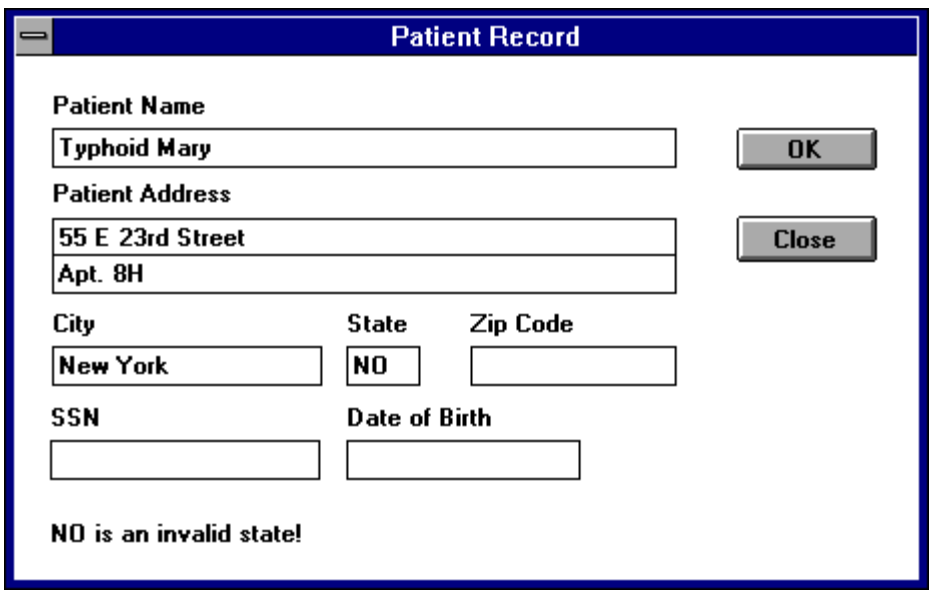


**Figure 18 Field-specific validation**

If errors occur, handle them as gracefully as possible with minimal disruption to the user's workflow. There are several ways to do this.

Decide what's critical. If a user tries to click on an uneditable field, the application might emit a simple system beep. If sound capabilities are available, you might try a more distinctive tone. If he or she tries to delete the database, put up a message dialog (hmm, how did he have access to delete that database, anyway?). The greater the error, the more dramatic your warning should be. Message dialogs exist in several types. Sometimes you only want to inform the user that an operation has completed normally.

Write the error messages in plain English. This can't be overstated. Look at Figure 8 again. They are actual error messages—I didn't make them up. If you were the user, would you have any idea what the problem was based on this garblespeak?

If possible, give the user something to do when an error occurs. Is there someone to call for help? Something in the application that can be entered/saved/reversed? Even a Help button might help. If you choose to provide such a button in a message dialog, give the help screen text some serious thought. Nothing is more frustrating than a help screen that tells you nothing you didn't already know.

Your application should offer keyboard equivalents for most activities the user can perform with the mouse. This may not always be practical—for example, the users of a CAD application will rarely touch the keyboard. There may be a business-driven need for one input device over another. Earlier I mentioned having worked on the development of a securities trading workstation. The competing traders at other firms could type a stock symbol, press Enter, and get the latest quote in about three seconds without removing hands from the keyboard. The original Windows-based applications we wrote required setting focus with the mouse, which cost an additional one or two seconds. This was enough to warrant the user's absolute refusal to use mice, and our development team had to build a keyboard-only interface.

Minimize the necessity to switch constantly between the mouse and the keyboard. Switching eats up time and can become annoying. If feasible, test the application using mouse only, keyboard only, and switching back and forth. Time the three approaches and observe at what point the tester's hand instinctively reaches for the mouse. If your application supports high volume data entry, well-thought-out keyboard equivalents are particularly crucial. Carefully consider hand movements, place the controls in a logical order and set the tabbing order correctly. Data entry operators have very exacting interface requirements. Speed is usually of the essence—their job performance may be judged by how much data they enter into the system daily. Generally they will not want to remove their hands from the keyboard and may request special accelerators to facilitate their tasks. (One application at my current firm required the capability for one-handed entry from the numeric keypad.) Deemphasize the mouse in such applications.

Don't forget to check your accelerators against the recommended list of key assignments. If you use lots of mnemonics, watch for conflicts. I've seen more than one commercial application where Alt-B referred to more than one control on the screen.

## Fonts and Color

Consistency is key to the effective use of fonts and color in your interface. Most commercial applications use 12-point System font for menus and 10-point System font in dialog boxes. These are fairly safe choices for most purposes. If System is too boring for you, any other sans serif font is easy to read (Arial® or Helv). The most practical serif font is Times New Roman®. Avoid Courier unless you deliberately want to look like a typewriter. The other fonts are lovely for word processing or desktop publishing purposes but don't really belong on Windows-based application screens. Avoid using all uppercase text in labels or any other text on your screens—it really is harder to read. The only exception is the OK command button—and please don't label it Ok. Also avoid mixing more than two fonts, point sizes and/or styles so your screens will have a cohesive look.

Color is a passionate issue among developers. No one can deny that color can add spice and life to your interface. I've heard more heated arguments on the use of color than almost anything else. Here are a few hints to cool things down:

Consider the hardware. Your Windows-based application may end up being run on just about any sort of monitor. Don't choose colors exclusive to a particular configuration, unless you know your application will be run on specific hardware. In fact, don't dismiss the possibility that a user will run your application with no color support at all.

Figure out a color scheme. If you use multiple colors, don't mix them indiscriminately. Nothing looks worse than a circus interface. Do you have good color sense? If you can't make everyday color decisions, ask an artist or a designer to review your color scheme. Use color as a highlight to get attention. If there's one field you want to the user to fill first, color it appropriately.

How long will users be sitting in front of your screens? If it's eight hours a day, this is not the place for screaming red text on a sun yellow

background. Use common sense and consideration. Go for soothing, cool colors—muted blues and greens work best. Check out the predefined Windows color schemes Arizona, Designer, Ocean, and The Blues for ideas. Variations that are also easy on the eyes are Rugby, Tweed, and Wingtips. Text must be readable at all times—black is the standard color, but blue, dark gray, and white can also work.

Give color significance on your screens. Perhaps you want to use blue for all the non-editable fields, green to indicate fields that will update dynamically, or red to indicate error conditions. If you choose to do this, ensure color consistency from screen to screen and make sure the users know what the various colors indicate. Don't use light gray for any text except to indicate an unavailable condition.

Remember that a certain percentage of the population is color blind. Don't let color be your only visual cue. Use an animated button, a sound package, or a message box.

Finally, color will not hide poor functionality. Users may ooh and aah over how pretty your screen looks, but if it doesn't work properly, all the colors in the palette won't help.

## A Word on Custom Controls

Specialized custom controls are popping up everywhere. If you decide to use them, it should only be after you have exhausted the existing possibilities. Custom controls can throw a monkey wrench into the consistency of your interface, and produce more complex maintenance issues. However, they sometimes are the only means to achieve unique visual effects and they afford a great opportunity for creativity.

If you choose a three-dimensional look for your interface (some good examples of this are Microsoft Excel and Quattro® Pro), ensure that it is consistent throughout. The Borland® Custom Controls Library DLL provides a complete set of elegant 3-D controls that could rival any widgets in Motif. The library also includes the Borland-style graphic push buttons with predefined bitmaps. Borland provides an API to assist you in writing your own custom controls and implementing multiple controls in a single DLL.

The Visual Basic Professional Toolkit includes a 3-D set of the standard controls, plus a number of other useful items such as spin buttons, graphs, and a spreadsheet-like control called a grid. The Professional Toolkit comes with a Control Development Guide for creating your own custom controls. Unlike the Borland custom controls, though, Visual Basic custom controls are VBX files and cannot currently be used in applications other than those developed with Visual Basic.

## Testing the Interface

Developers rarely design only for themselves, but most don't show anyone their application until it's almost complete. Don't code in a vacuum. Get a variety of opinions, particularly on the interface design. Start the testing process as soon as you have a working prototype, and tell the testers what isn't functional yet. When you hand them your application, don't explain too much about it—just sit back, let them tackle it, and watch them carefully. If possible, quantify the results. How long did it take them to enter the criteria to perform a search? What are the error rates? How many steps and how much time did they take to navigate through the entire system? Test the keyboard equivalents by unplugging the mouse before running the application. Ask what makes them uncomfortable, and if they get stuck, resist the urge to show them the way out. Instead, find out how and why they backed themselves into a corner. Don't skimp on testing or you'll regret it.

## Conclusion

As users become accustomed to well-designed programs, they will be less willing to settle for clumsy, awkward, ugly applications that force them into abstract patterns of thinking. Guidelines and standards for user interfaces will continue to evolve. I'd like to see an awakening of the imagination in developers of Windows-based applications. Too many still see interface design as an enemy!

---

1 For ease of reading, "Windows" refers to the Microsoft Windows operating system. "Windows" is a trademark that refers only to this Microsoft product.