Session SKL02

Designing User Interfaces that Work

By Tamar E. Granor, Ph.D.
Editor, *FoxPro Advisor*
tamar_granor@compuserve.com

Overview

Every mechanical or electronic device we use has a user interface - toasters, cars, televisions. But we barely notice the interface of most of the things we use because they're so well designed and we're so used to them that it's second nature. When we do notice (say, programming a VCR), we get annoyed. How often do you need to open the manual that came with your toaster oven?

User interfaces for software should be the same way - you shouldn't notice them most of the time. When you do notice the user interface, it's usually because it's in your way.

What makes the user interfaces of everyday objects so usable? This session looks at the user interface of real world objects that work and shows how to apply the same ideas in application development. It also discusses some of the ways that database applications differ from other computer applications and the impact those differences have on interface design. Finally, we'll look at the actual components of a user interface and discuss appropriate uses for them.

What's "intuitive" anyway?

The interfaces for most devices are pretty straightforward. To turn your car to the right, you turn the (top of the) steering wheel to the right. To make a piece of toast, you put the bread on the rack, close the door, set the control the way you want it, and push a button (most of the time, you can omit the third step because the toaster oven remembers the last setting - more about that later). It's not necessarily that there are just a few steps, but each step makes sense as part of the whole. (Of course, simple tasks may have few steps.)

There's a lot of talk about making user interfaces "intuitive." The goal is to have the same kind of straightforward, "every part takes you toward your goal" sequences as with physical objects. Most computer games (at least the popular ones) have accomplished this goal to some extent. To move a card in Solitaire, you don't choose commands from the menu - you grab the card with the mouse and drag it to where it belongs, almost exactly what you'd do in real game of Solitaire.

But most applications don't let users follow their natural instincts that way. For years, to move some text, you had to highlight it (fair enough - there has to be some way to indicate which text you're interested in), then choose Edit|Cut (either from the menu or via a toolbar), move the pointer to the destination (either with the mouse or keyboard) and finally choose Edit|Paste (again either from the menu or a toolbar). That's about two more steps than most users thought it should take. Many of today's applications support drag-and-drop techniques for copying, mapping the task to the user's mental model of it. (In fact, more and more, you can even drag-and-drop between applications.)

So, it seems that what "intuitive" means for interface design is that it's like what the user is used to. In fact, this leads us to two main points - consistency and familiarity.

Consistency is essential

Some of the devices we use, such as cars, are complex and have many controls. But, for the most part, once you learn how to use it once, your knowledge applies across the board. With cars, the most important interface components - the steering wheel, the pedals and the gearshift - are nearly the same in every car. (There are, of course, two major flavors here - automatic transmission and manual, but within each type, the controls are nearly the same.) Whether the gearshift is on the steering column or on the floor, it works pretty much the same way.

When it doesn't work the same way, it's a problem. Have you ever tried putting a Volkswagen into reverse? The process is different (or was the last time I drove a VW) than on pretty much any other manual transmission car.

Of course, not everything is the same from car to car. Most drivers have had the experience of trying to turn on the lights of a strange car and starting the windshield wipers instead or vice versa. That's the point at which we curse the unknown designer because he didn't put the controls in the same place in every car.

The key point here is consistency. If pressing the right-hand pedal in one car makes the car move, it's handy if the same action in another car moves that car. In fact, from a safety perspective, having the gas and brake pedals in the same position in every car is a major concern. Imagine if GM cars had the brake on the left and the gas on the right, but Fords were the opposite.

In user interfaces too, consistency is the key to making users feel secure. Put the same menu items in the same places in each application and make sure they do the same thing. Then, the user brings old knowledge to a new application.

Consistency is important within applications, too. If 90% of the dialogs in an app have the OK button on the left and the Cancel button on the right, and the other 10% reverse it, there's a pretty good chance that users are going to get it wrong most of the time in that 10%.

Most users develop "muscle memory" which allows them to work without having to consciously process each step. User interface muscle memory includes memorized hotkeys and menu sequences as well as knowing exactly where to click the mouse to make certain things happen. Anything in the interface which fights against that memory is doomed to fail - imagine a Windows application that used CTRL-Z for anything but Undo!

Familiarity breeds comfort, not contempt

The controls in a car that move around from model to model and manufacturer to manufacturer are non-critical items, but even most of them only appear in a few different places. If I need to find the windshield wiper control for a car, I look first on the left hand side of the steering column. If it's not there, I try the extreme left of the dashboard. Chances are I'll find it in one place or the other. If not, I'll be forced to read the manual and I'll be annoyed at whoever designed the car.

Note that there's nothing intuitive about the positions of the controls in a car. It's their familiarity that makes the interface so easy to use.

Similarly, there's nothing particularly intuitive about CTRL-Z for "undo" and CTRL-A for "Select All" is only slightly intuitive. But almost every application I use includes them, so they've become familiar. I expect every Windows application I install to use them and so do the people who use the applications I write.

It's worth noting that, while consistency and familiarity are important for business applications, in some arenas (games, multimedia exploration, and so forth), it's lack of these attributes that makes the applications interesting and fun. (More specifically, it's that you can't look at the application and immediately know how to solve the problems it presents.)

The role of standards

There are probably things you really dislike about the Windows interface (whether you're using the old Win3.1 interface or the new Win95 version). I know there are things I dislike (bordering on hatred in a few cases).

Does this mean you should leave those things out in your applications? Of course not, unless there's a specific reason your app should work differently than the rest of the world. (There's a problem with this guideline, of course, in that it doesn't offer the opportunity for new standards to develop. Finding the balance between applying existing standards and developing better ones is extremely tricky.)

For example, as noted above, CTRL-Z is the Windows standard shortcut for "Undo". You may think that CTRL-U is a better, more intuitive choice, but using it means that your users have to remember that your application works differently than the rest of the applications they use.

On the other hand, if you're designing an application for disabled users who may be typing with a mouth stick or have only one hand for typing, it would make sense to use function keys as shortcuts rather than CTRL-key or ALT-key combinations. Even in that case, though, the first step is to find out whether there are established standards for the use of function keys in applications for the disabled; if not, establish your own and use them in every application you write. (In fact, the Win95 interface includes an option to make the modifier keys - CTRL, ALT and Shift - "sticky" so that they can be used in sequence with other keys instead of simultaneously.)

Standards exist to minimize the cognitive load on users and to provide both consistency and familiarity. Don't deviate from the published standards unless you do it knowingly and for a good reason. (Then, of course, document the difference and the reason.)

The "P" stands for "Personal"

I have touch-tone phone service. I'd be annoyed if I bought a new phone, plugged it in and found it only used the older, slower, pulse dialing. If I paid for cable TV and bought a cable-ready television that wouldn't pick up, say, ESPN because the manufacturer thinks televised sports are stupid, I'd be angry.

The point is that I'm the consumer and I've already made my choices about phone service and TV. It's not up to the appliance I use to implement that service to decide whether I should get what I asked for.

For applications, the key is to remember that the "P" in "PC" stands for "personal." It's the user's computer, not yours and your software is a guest. It needs to respect the user's choices. In Windows, this means especially respecting the selections users make in the Control Panel.

I'm asked pretty regularly how to set up an application for specific colors rather than using the Windows color scheme. My response is always to ask, "why do you want to do that?" The user chose those colors for some reason and our applications have no business ignoring the boss.

Whenever possible, draw colors, date and time settings, fonts, currency settings and so forth from the Windows settings. (It's easier than ever to do this in VFP5.)

Try to Remember

My toaster oven and dishwasher have a great feature that would make most interfaces work better for users. Once I find the right darkness setting for my toast, it stays set - I'll probably never have to touch that control again. My dishwasher remembers the last cycle I used. Chances are pretty good I'll want the same one next time, too. My TV and every radio I own share this feature, too - they remember what I last tuned in to and when I turn them on, they start on that station. The designers of these objects remembered that I'm the customer and, therefore, I'm always right. They made the objects do the hard part of remembering what I want, so I can just start them up and get it.

To get a sense of how aggravating the absence of this behavior is, consider the last time you stayed in a hotel. Probably the TV started on the same station (most likely, the hotel's channel or main menu) every time you turned it on rather than remembering what you last watched. Annoying.

A few applications do remember things and it's made them popular with users. Quicken, for example, not only fills in the rest of the name as I type, but once I choose a vendor to pay, it assumes the transaction I'm entering is the same as the previous one for this vendor and fills in the check or register accordingly. When paying a bill that's the same every time, all I have to type is enough to uniquely identify the vendor. But changing the details once the entry appears is no harder than typing them all in.

Let your applications remember user customization (window positions and sizes, fonts, and so forth) and your users will be pleased. Let the users decide which things your app should remember and they'll probably be ecstatic. And, of course, always provide a way for the user who's made a mess of things to restore them to an acceptable state. (The Windows registry provides a good place to store application settings.)

The Document-Dialog Distinction

Applications use two kinds of forms: documents and dialogs. Documents contain the data the user is working on, whether it's text in a word processor, numbers and formulas in a spreadsheet, bullet points and text in a presentation program, or fields in a database. More and more, documents may contain components of different sorts. A single document may include text in Word, a graph created with Excel and data drawn from VFP.

Dialogs, on the other hand, are an extension of the menu system, allowing the user to specify the parameters for a task. Windows includes many native dialogs like Open, Save As and Print. In each case, the user specifies one or more options or values that let the application complete a task such as opening a file, saving and naming a document or printing a document.

Users work with documents and dialogs differently. Documents generally call for continuous entry of data - you want to type text as fast as you can or enter a series of numbers into a spreadsheet. As much as possible, when working in a document, you want to keep entering your data without interruption.

Dialogs, on the other hand, constitute an interruption in the first place. Dialogs only appear when something goes wrong or you've asked to do something.

In the Windows interface, dialogs and documents have distinct appearances. In the default color scheme, documents have a white background while dialogs are gray.

Perhaps the most important distinction, though, is that documents don't generally include controls. Dialogs, on the other hand, make extensive use of controls like dropdowns, checkboxes, spinners, lists, and in the Win95 interface, ActiveX controls like TreeViews and ListViews.

Why documents shouldn't include controls

When FoxPro 2.0 first gave us the ability to include fancy interface controls in our forms, we all jumped to create fancy looking forms which incorporated as many gizmos as we could. It didn't take long, though, to learn that lots of users, especially data entry clerks, hated them. Why? Because all those controls meant the user couldn't just look at the original document (as a touch typist normally does) and type the information - she had to notice when she got to a check box or dropdown and take special action there.

For heads-down data entry, anything other than textboxes is a distraction that slows the user down. By extension, when the user's job is more to enter data than to manipulate it, controls are likely to be in the way. When the user's job is to slice and dice the data, controls may help.

When designing an interface, consider using distinct forms for data entry and data lookup. The data entry form should be a document containing textboxes. Any other controls (like a Save button) should be out of the tab order and available by hot key as well as by mouse. In a data entry situation, the user should never have to remove her hands from the keyboard.

Forms for lookup and analysis should be dialogs (though the form that displays data once it's been looked up may be a document). In this situation, the user doesn't expect to just type at full speed and is likely to welcome the additional guidance of controls. They can limit the user to acceptable choices and make the task of finding data or manipulating it much simpler.

(It *is* possible to design a form that can handle both data entry and lookup comfortably, but it must be assembled carefully. Buttons outside the tab order and copious use of hotkeys aid in the effort.)

The key observation is that data entry forms for databases are documents, not dialogs.

What's different about databases?

The document-dialog distinction is easy to grasp when you look at applications like word processors or spreadsheets. The document you're writing or the spreadsheet you're preparing is the document. It's in front of you all the time and any other form that appears is in aid of preparing that document and, thus, is a dialog. In these applications, there's usually a one-to-one mapping between a document and the physical file where it's stored (though a complex spreadsheet model may involve multiple files).

Even in a programming environment like Visual FoxPro, the difference is fairly obvious. A program or a form or a report being edited is a document and these map to one or two files each (and the two files, SCX and SCT or FRX and FRT, can be seen as one table). When dialogs appear, they are an aid to editing.

For a database application, however, the mapping isn't as direct which makes it harder to see the distinction. What is a document in a database application? A single form may include data from many tables. Is the form the document, or is the data you're editing the document? This confusion is most apparent when you try to figure out what the items on the File menu mean for a database application.

The solution is to approach the problem from the user's point of view. The user doesn't care that the data is stored in ten neatly normalized tables or that a particular form involves four of those tables. The user approaches things from the perspective of the task to be accomplished like "enter an invoice" or "edit a student's information."

Once you look at things from the user's point of view, it becomes clear that, in fact, the form is the document and the fact that the data it manipulates may be stored in multiple files is simply a detail to be hidden. Following this thinking a little further makes it apparent that, in fact, the documents in an application often correspond to paper documents that users are used to dealing with, like invoices, transcripts and so forth. So properly applying the document model leads to increasing familiarity for the user.

Thinking in terms of documents

One of the places where this document-centric notion has the most application is on the menu. Following the Windows model of File-Edit-Help (there are actually some good reasons not to - see Alan Cooper's "About Face" for a revolutionary approach), we can now see that the items on the File menu definitely shouldn't apply to individual tables or records. What should they address?

There are two ways of looking at what the file menu should address. The first (proposed by Nancy Jacobsen in her June 1995 FoxPro Advisor article) sees the File menu as dealing primarily with containers. In a database application, the container is the database. Using this approach, File-Open lets you choose a database (for example, a company) to open and a separate Document menu provides access to the individual documents (invoices, sales orders, etc.) for that database.

An alternative approach is to view the File menu as addressing documents, so File-Open lets you choose a document (invoice, sales order, etc.) to edit. In neither case, however, is the user exposed to the physical structure of the database underneath.

Validation is an Interruption

A key feature of database applications is to ensure that the data that gets stored in the database is accurate. So our applications naturally put a lot of effort into validating the user's input. However, in a heads-down data entry situation, validation can interfere with data entry.

Most applications validate each field as it's entered. While this catches the error at the time it's made, it also constitutes an interruption for the user. Remember that it's called "heads-down" data entry because the user isn't looking at the screen – she is looking at the source document providing the data to be entered. In that setting, displaying an error message as soon as a field can't be validated disturbs the process of getting the work done. For heads-down data entry documents, it's better to validate data when the document is complete (typically, when the user asks to save) and then let the user make any necessary corrections.

For other documents, immediate validation may be more appropriate. In dialogs (discussed below), it's best to prevent the user from making errors as much as possible.

Optimize for the most likely

Microsoft Word normally opens up with a new, blank document (unless you open it by choosing a document). This is pretty handy behavior. After all, I can't do anything in Word without a document.

Recent versions of a number of other Microsoft products (including Access and PowerPoint) open with a dialog asking me what to do: create a new document or open an existing one. I find this behavior far less helpful than Word's default. It's no harder to open an existing document in Word once the application is running than to find what I want from PowerPoint's opening dialog and at least some of the time, Word is right and I do want to start a new document.

Too many applications fail to consider what the user is most likely to want to do and give that the shortest possible path. In database applications, this is especially relevant when deciding what status a form should have when it opens: should it be in "add mode" - ready for a new entry, in "edit mode" - ready to edit existing data, or in "view mode" – displaying existing data, but not allowing it to be edited.

Many applications choose View mode as the default and the user must press either an Add or an Edit button before he can do anything at all. In those applications, often, once a record has been saved, the user has to again choose whether to add or edit.

In most applications, it's likely that a particular form is used pretty much the same way most of the time. An order entry form is probably used for adding new orders far more often than for editing existing orders. On the other hand, an employee maintenance form is probably viewed or edited far more often than a new employee is added. (Of course, this may be different in particular situations.)

Figure out whether a form is primarily a data entry form, a data editing form or a viewing form. Then design it to optimize the more likely case. A data entry form should open in add mode and provide a way to switch to edit mode. A form most likely to be used for editing existing data should open in edit mode, with an easy way to choose the right record, and a way to add new records if necessary. Only a form most likely to be used for viewing existing data without editing or used for extremely sensitive data that must be protected at all costs should open in a read-only, viewing mode. (Another alternative is to open the form in the same mode it was last in or the one that has been used most frequently.)

Once data has been saved, the form should return to its original mode. When you save a new order, the form should be ready for the next order to be entered. When you finish editing an employee record, the form should be ready for you to choose another employee to edit.

It's likely that some forms in an application are data entry forms while others are edit forms. My experience is that the "workhorse" forms in an application, the ones where the user spends the most time, are usually data entry forms and that the maintenance forms that back up the data entry forms are likely to be edit forms. Does it create a problem for users that different forms open in a different state? Not if the state of a form is clear and if the right state is chosen for each form.

It's also possible that different users of the same application may use the same form differently. The data entry clerk primarily processes new data while his manager may spend a lot of time viewing the data which has been entered. It's not hard to design forms that can open in either state, based on a parameter or property, and then call them appropriately, based on the user or the menu item which opens the form or both.

What about modal forms?

One of the mantras of application development over the last few years has been "modeless, modeless, modeless." Users shouldn't have to leave one activity to work on another.

This view is correct. As developers, we certainly take advantage of the modeless nature of both Windows and Visual FoxPro. It's not unusual for me to have three or four applications running at the same time and to swap among them frequently. Development in VFP would be much harder if we couldn't browse a table, edit a form and keep the project manager open at once.

Applied to our applications, this means that the user should be able to examine open more than one document at the same time. If a clerk is entering an order and the boss calls to ask how many widgets were sold last month, the clerk shouldn't have to either abandon the order being entered or say to the boss "you'll have to wait while I finish this entry." He should be able to stop what he's doing, pull up the necessary information, then return to his task where he left it. (Of course, the boss ought to be able to get the information from his machine without calling the clerk, but that's a different issue.)

So, we've all redesigned our applications to make each document form non-modal and allow users to open several forms at once. In Visual FoxPro, this is a lot easier than in older versions and, of course, it's even simple to allow multiple copies of the same form to be opened.

Does this mean we shouldn't have any modal forms in our applications at all? No, it doesn't. There are some actions that really do need to be performed alone - reindexing data comes to mind. The forms that facilitate these activities should be modal to keep the user out of trouble.

But it's likely that the form that lets a user start such a process isn't really a document, designed for viewing or editing data. It's probably really a dialog, designed to collect some information from the user and then perform a task. In most applications, almost every actual document should be non-modal.

Using dialogs wisely

Modal dialogs may be the single most overused interface element in most applications. (Keep in mind that the ubiquitous Windows message box is a modal dialog.) Dialogs generally have three uses: to let you specify additional parameters when you make various menu choices; to confirm certain actions (like deleting a record); and to let you know that some task has been performed. These three uses for modal dialogs are of varying utility.

The first type, parameter specification, is often necessary (though Cooper rightly suggests that we could reduce these dialogs a lot by assuming the most likely choice and remembering what the user did last). These dialogs are normally requested by the user to let him configure processes.

Confirmation dialogs should be used only in the most extreme situations - when the action the user is about to undertake is unlikely to be the right choice and the action can't be undone. Most of the confirmation dialogs we encounter are simply annoyances in the way of getting things done. For example, when you run a form from the Form Designer in VFP, a "Do you want to save changes" dialog appears. Since you almost always do want to save those changes (after all, you just asked to run the form), you can actually turn off that dialog (in Tools-Options-Forms) and have the form automatically saved. But most confirmation dialogs can't be dispensed with that easily.

Some applications even prompt you multiple times to confirm the same action. If the action is "Reformat your hard drive," multiple confirmations may make sense. When the action is "Save your work," they're annoying.

The final type of common modal dialog, informational messages, should disappear from interfaces. If you just told the application to delete a record, why do you need to click "OK" to confirm "Record deleted." This is not to say that informational messages aren't called for - of course, they are. But using a modal dialog to tell the user that you just did what the user told you to do is likely to drive the user crazy. Use the status bar or a wait window that clears as soon as the user does something. (On the whole, the status bar is probably a better choice unless you need to be certain the user sees the message immediately.)

Use dialogs sparingly. Think of a dialog as an interruption and only interrupt users when it's really necessary.

The Right Control for the Job

So you've taken all the advice above. Your application design clearly distinguishes between documents and dialogs, using few of the latter. It remembers the user's choices, is internally consistent and follows published standards. Sooner or later, you need to actually design the documents and dialogs of your interface. How do you decide which of the myriad of available controls, both native and ActiveX, to use for each task?

The first decision is easy. Documents use very few different controls. They're composed primarily of textboxes with perhaps some editboxes, spinners and dropdown lists (all of which allow heads-down data entry, unlike many other controls). In documents, any other controls are kept outside of or at the end of the tab order, so the user can work without looking up.

What about in dialogs? The key to making the right choices is to consider the purpose of each control and the task at hand. Let's look at the controls available in VFP and see what they're meant for and where they are most useful.

Text input controls

Several controls are primarily for text input: textboxes, editboxes, spinners. These controls make it easy to enter text (with spinners limited to numeric text).

Textboxes and editboxes are for entering any alphanumeric text. Editboxes allow unlimited entry while textboxes are limited to whatever fits. The InputMask property lets you limit acceptable entries, while Format lets you modify the appearance of the text. Use textboxes for fields that are limited in size, like names, states and so forth, and editboxes for fields that are multi-line (like street addresses) or unlimited (like notes).

Spinners accept only numeric input and allow the user to edit the value by typing, clicking on spinner arrows or using the arrow keys. Use spinners for bounded numeric input, making sure to set upper and lower limits for both keyboard and spinner input. Consider using a tooltip or WhatsThis help to show the acceptable range of values.

Selection controls

A number of controls let the user choose among multiple alternatives: checkboxes, option buttons, comboboxes and listboxes. Each of these presents two or more alternatives and the user indicates the right choice.

Checkboxes let the user choose between exactly two states. They're most appropriate for items which are either on or off such as whether to include phone numbers on mailing labels. If you can state the choice the user is making in a simple phrase, a checkbox is appropriate. Groups of checkboxes can be used together to allow the user to set a number of related, but not mutually exclusive, states. For example, the View page of VFP's Tools-Options dialog includes checkboxes for Status bar, Clock, and other items that affect the visual appearance of VFP, but are not interconnected.

Option buttons (also known as radio buttons) present a set of alternatives from which exactly one is chosen at any time. Option button groups are useful for a fixed set of choices that will never change. For example, you might use option buttons to indicate output destination (Screen, File, Printer). Be careful not to use option buttons for sets that may grow. Option buttons may be appropriate for some two-state choices where a checkbox could be used. If you can't find a checkbox prompt that makes the two states clear, consider option buttons instead.

Both option buttons and checkboxes are available in a graphical format. In graphical form, these controls are used mostly on toolbars to let the user quickly change state. For example, Word's formatting toolbar includes checkboxes for bold, italic and underlined text and option buttons for text alignment. If you use graphical controls, be sure to choose icons that are clear and add tooltips to help users learn their meaning.

Comboboxes and listboxes let the user choose from a list of acceptable alternatives. Comboboxes may also allow the user to enter an item not on the list. Listboxes may permit multiple selection. Both controls include incremental search as a native feature - this allows the user to type characters and have the highlight move to the first matching entry.

There are two types of combos: dropdown lists and dropdown combos. Dropdown lists limit the user to the items provided. These can be used sparingly in documents since a user can make a choice without removing her hands from the keyboard. However, some characters the user types can be lost if they don't match any items. Therefore, dropdown lists should be used in documents only for items where the user knows the valid choices, such as state abbreviations.

Regardless of the number of items included in the list, a VFP combo shows only seven items when dropped open. This means that it should not used for long lists of similar items - the user won't be able to see enough to scroll intelligently.

In general, lists and combos shouldn't be used for large lists of items. No user wants to scroll through 50,000 choices. Consider getting some initial input from the user to narrow down the choices, then using a list or combo to present the smaller group. (In VFP 3.0 and earlier versions of FoxPro, combos and lists are extremely slow with more than a few hundred or a few thousand items anyway.)

Combos are space efficient, so use them when space is at a premium. They're handy on toolbars, too, where they both show you the current value and let you change it. Use lists when you have the space or when you need multiple selection.

Action controls

There's only one control whose purpose is to trigger an action, a push button. Pressing the button takes the action indicated by the button's caption.

Use buttons for common actions, such as looking up a value, accepting the entries in a dialog and canceling changes. Actions which are frequent throughout an application can go on a application-standard toolbar.

In documents, buttons can be in the way. At a minimum, put them at the end of the tab sequence with the most likely choice first. Better is to keep them outside the tab order entirely. Alternatively, use a toolbar and a set of hotkeys and keep buttons off the document.

Display controls

Some controls provide output only. These include labels, lines, shapes and images. Use these controls to make your forms more readable. Lines and shapes are particularly useful for organizing a form into logical groupings.

In VFP, these controls are "live" so you can attach actions to them. If you do so, make sure to provide some visual indication that they're active. (For example, clicking or double-clicking an image might open a window showing the image in detail. A tooltip saying "Click here to see the detail image" helps the user know there's an action available.)

Organizational controls

Grids and pageframes let you organize other controls. While both have their uses, it's easy to overuse them. In particular, both can be out of place in documents and should be used sparingly in such forms.

Grids provide a multi-record view of a table. In controlled settings, a grid can be used to allow data to be entered and edited, but use caution. A grid may not be appropriate for heads-down data entry since it can require different navigation keystrokes than other controls. In addition, a grid which needs to be scrolled for all important data to be seen may be confusing to some users.

Pageframes are an easy way to organize information. However, like many other controls, pageframes may be inappropriate in documents. (In fact, the ActiveX version of a pageframe which comes with VFP5 is called a "tabbed *dialog*" control.) In dialogs, the organized multi-page view provided by pageframes can make the complex clear. But in documents, pageframes require too much effort by the user if he really needs to visit multiple pages to enter a single record. A pageframe is appropriate where some information is rarely entered or edited. Put the fields that are used every time on the first page and fields that are rarely accessed on other pages. Do not expect a heads-down user to enter data on multiple pages for every record.

Toolbars, which also let you organize other controls, aren't really a part of documents or dialogs. They're an extension of the menu system and are discussed below.

ActiveX controls

Visual FoxPro allows you to use controls other than those built into the product. A large number of ActiveX controls come with VFP5 and others can be purchased separately.

Many ActiveX controls are improved versions of the built-in controls, such as enhanced pageframes or grids. Others provide special purpose interfaces, like calendar and communication controls

However, some ActiveX controls provide new interface capabilities. In particular, VFP5 provides the ListView and TreeView controls used throughout the Windows 95 interface. Use them to build dialogs like the Win95 dialogs. Don't use these controls in documents. Like the other selection controls, they require too many attention from the user.

Some of the ActiveX controls allow you to use Windows system dialogs (like Open, Save As and Print) in your applications. Since users know those dialogs from other applications, using them increases the consistency and familiarity of your applications.

Taking Action

The user interface isn't composed only of forms. Menus and toolbars let the user indicate actions. Menus include both the menu bars and context menus (which appear on right-click).

Menu bars generally provide many ways for the user to choose an action: the user can use the mouse to both navigate to and choose an item; the user can navigate and choose with the keyboard; mouse and keyboard navigation and choice can be mixed; finally, some items provide a shortcut (such as CTRL-C for copy) which the user can type without navigating the menu. Toolbars complete the picture by providing mouse shortcuts that can be accessed without navigation. Context menus provide an alternative form of shortcut which require a mouse initially, but can then be used with mouse or keyboard.

Always make every action available both by both keyboard and mouse. Users vary in whether they prefer keyboard or mouse and some users may have physical problems using one or the other.

Which items should get menu shortcuts or appear on toolbars? Those which will be used often enough to make the shortcuts or the position on the toolbar memorable. Cut, copy and paste are common actions in most applications, so they have Windows standard shortcuts and appear in the standard toolbar of most applications. On the other hand, although Arrange All appears in the Window menu of most applications, it doesn't have a shortcut because it's not that commonly used. Think about which actions in your application will be used frequently enough to justify keyboard or toolbar shortcuts.

Context menus should contain operations specific to the object they belong to. Organize the items in order of likelihood since navigating a context menu is a complex operation. The context menus in some application have Cut as the first choice – I find myself accidentally deleting things. Consider the order of items carefully to make it harder for the user to make mistakes. (Since context menus are a fairly new part of interfaces, standards for the items included and the order of the items are still in flux.)

Every item on a toolbar should have a tooltip. Users will quickly memorize the items they use, but will not learn the remaining items until they're needed. (On the other hand, controls in documents should never have tooltips. In dialogs, tooltips can be used sparingly.)

To make your users really happy, let them configure their toolbars, adding and removing items and organizing them for their convenience. This is a standard feature in commercial applications.

Keeping your design on track

Perhaps the most important thing you can do for your application's user interface is to document your standards. Make conscious decisions about how the interface will look and work, document those decisions and then don't allow variation unless there's a compelling reason to do so.

When you think you have a reason to use something other than your documented standards, you need to consider whether the problem is a weakness in the standard or something truly outside its scope. If you have a weakness in the standard, you may want to adhere to it anyway and plan to change it in a future version. If something is outside the scope of the standard, evaluate it, make a choice and add it to the standard.

Avoid the temptation to modify your standards in the middle of a project, even if you've learned about some wonderful new technique. Users prefer an application that works uniformly (even if it's not the newest whiz-bang thing) to one that works differently in different parts. If you can't make the change across the board, don't make it at all.

Summary

Like much of application design, user interfaces are a constantly changing field. While the tools for constructing interfaces and the final result change, the basic points of interface design remain the same:

- Consistency and familiarity ease the user's learning task.
- It's the user's computer; respect his choices.
- Remember how the user left things.
- Documents and dialogs are different and their appearance should reflect the difference.
- Design for the likely, but allow the unlikely.

Applying these ideas will result in interfaces that users like and feel comfortable using. Ignoring them is likely to result in unhappy users.

Acknowledgments

My thinking about user interfaces has been influenced by a number of people over the years. These notes owe a great deal to Alan Cooper and Nancy Jacobsen, in particular, though the ideas expressed here vary from their writings.

Ted Roche, Mac Rubel and Mark Wilden reviewed these notes and offered a number of suggestions.

Bibliography

Cooper, Alan, *About Face*, IDG Books, 1995, ISBN 1-56884-322-4. This is the book you *must* read about interface design. Some of Cooper's ideas are evolutionary while others are revolutionary. You'll never look at an interface the same way again.

Jacobsen, Nancy, *Designing Usable Data Entry Applications*, DataBased Advisor, July 1994, pg. 126. This article explains the document/dialog distinction and talks about designing documents to ease data entry.

Jacobsen, Nancy, *Making Sense of GUI Menus*, FoxPro Advisor, June 1995, pg. 44. This article picks up where the other one leaves off and discusses how database applications are different than other applications and how to organize menus that recognize the difference, yet still are consistent with GUI standards.

Norman, Donald A., *The Design of Everyday Things*, Doubleday Currency, 1988, ISBN 0-385-26774-6. This book argues that, in fact, many of the things we use aren't so well-designed and explains what's wrong with them. As with Cooper, you won't look at the objects in your life the same way after you've read this.

Isys Information Architects, *Interface Hall of Shame*, http://www.mindspring.com/~bchayes/mshame.htm. This web site contains dozens of examples of bad user interfaces, in each case explaining what's wrong. It's also linked to a Interface Hall of Fame, containing a few examples of particularly good interface design. Visiting this site is both fun and thought-provoking.