Session SMF303

Best Practices for User Interfaces

Tamar E. Granor, Ph.D.

Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Fax: 215-635-2234
Email: tamar@tomorrowssolutionsllc.com

The user interface is your application's appearance to the world. Throwing it together as an afterthought is like putting on the first clothes you grab from your closet. In both cases, the result may or may not look good and may or may not be appropriate to the occasion. This session will start from the top and dig down into the mysteries of designing the interaction and interface of an application. After discussing why user interfaces matter, it will provide best practices for both the design and implementation stages. Although the code examples for this session use Visual FoxPro, the principles and ideas apply across development platforms, though web development raises some additional issues not discussed here.

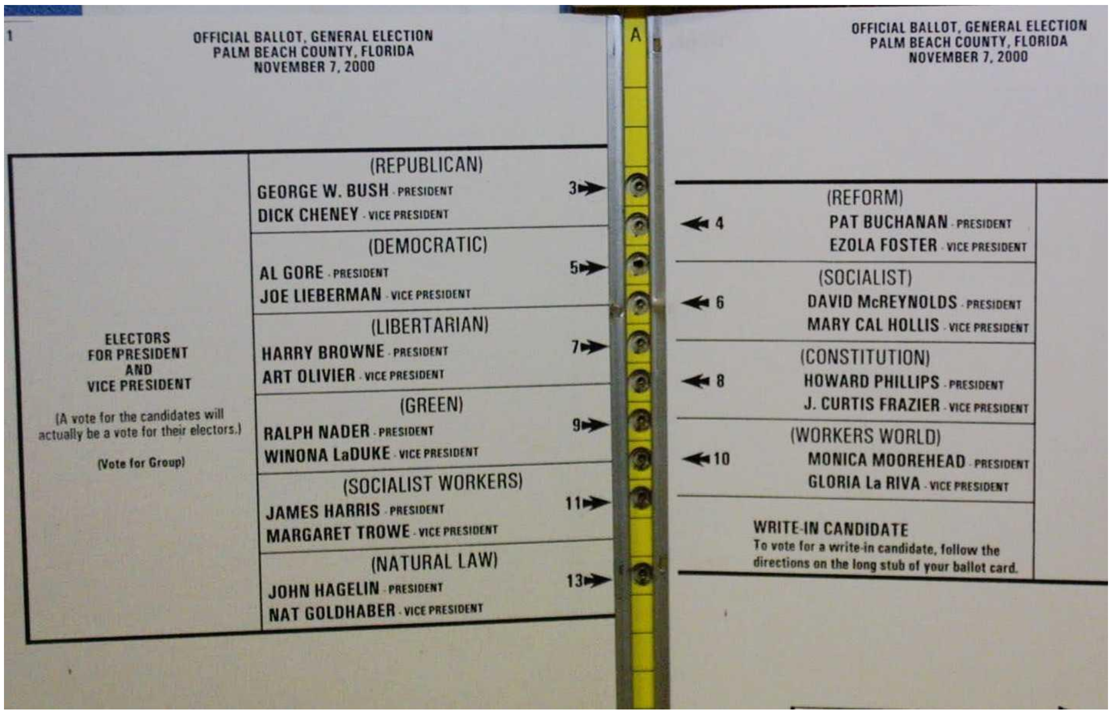
Why do user interfaces matter?

> *If it was hard to write, it should be hard to use.*
> *-- Old Programmer Wisdom*

The maxim above pretty much sums up the view of many software developers. They implement user interfaces that reflect the underlying architecture of the application and don’t really care how hard or easy the result is to use. Their response to user’s complaints is to tell them that they just need to learn how it works. They rail about the stupid users who just don’t get it.

Is it just the users? Does it really matter what the user interface of your applications looks like and how it works? Yes, interfaces matter. In fact, a bad user interface can have world-changing or life-threatening consequences.

In the United States’ Presidential election of 2000, Palm Beach County, Florida, used a ballot design (**Figure 1**) meant to make it easier to fit all the candidates onto a double-page. The design introduced confusion as to which hole to punch to vote for Al Gore. While normally such confusion wouldn’t have mattered, in this case, the Florida vote was extremely close and the roughly 20,000 affected votes were more than enough to swing Florida for Bush and change the overall results of the election. (For a user interface oriented look at the butterfly ballot, see http://www.asktog.com/columns/042ButterflyBallot.html.)


1

OFFICIAL BALLOT, GENERAL ELECTION
PALM BEACH COUNTY, FLORIDA
NOVEMBER 7, 2000

ELECTORS
FOR PRESIDENT
AND
VICE PRESIDENT

(A vote for the candidates will actually be a vote for their electors.)

(Vote for Group)

(REPUBLICAN)
GEORGE W. BUSH - PRESIDENT
DICK CHENEY - VICE PRESIDENT
3➨

(DEMOCRATIC)
AL GORE - PRESIDENT
JOE LIEBERMAN - VICE PRESIDENT
5➨

(LIBERTARIAN)
HARRY BROWNE - PRESIDENT
ART OLIVIER - VICE PRESIDENT
7➨

(GREEN)
RALPH NADER - PRESIDENT
WINONA LaDUKE - VICE PRESIDENT
9➨

(SOCIALIST WORKERS)
JAMES HARRIS - PRESIDENT
MARGARET TROWE - VICE PRESIDENT
11➨

(NATURAL LAW)
JOHN HAGELIN - PRESIDENT
NAT GOLDHABER - VICE PRESIDENT
13➨

A

OFFICIAL BALLOT, GENERAL ELECTION
PALM BEACH COUNTY, FLORIDA
NOVEMBER 7, 2000

◀ 4
(REFORM)
PAT BUCHANAN - PRESIDENT
EZOLA FOSTER - VICE PRESIDENT

◀ 6
(SOCIALIST)
DAVID McREYNOLDS - PRESIDENT
MARY CAL HOLLIS - VICE PRESIDENT

◀ 8
(CONSTITUTION)
HOWARD PHILLIPS - PRESIDENT
J. CURTIS FRAZIER - VICE PRESIDENT

◀ 10
(WORKERS WORLD)
MONICA MOOREHEAD - PRESIDENT
GLORIA La RIVA - VICE PRESIDENT

WRITE-IN CANDIDATE
To vote for a write-in candidate, follow the directions on the long stub of your ballot card.

Figure 1. The “butterfly ballot” used in Palm Beach County in the 2000 election. Although Gore/Lieberman is the second group shown on the left, you have to punch the third hole to choose that ticket. Many voters punched two holes because they were both next to Gore/Lieberman.

In the ensuing discussion, many people blamed the voters for their mistakes and the word “stupid” was thrown around. But when thousands of people make the same mistake, can they all be stupid?

Bad user interfaces can cost lives as well as elections. A study published in the Journal of the American Medical Association (http://jama.ama-assn.org/cgi/content/abstract/293/10/1197) reported on 22 different ways a particular medicine order entry system led to medication errors. Among the problems were default values that misled doctors as to appropriate dosages, having data for a patient spread out among multiple screens, fonts too small for clarity and failing to include the patient’s name on every screen for that patient. Doctors in the study group indicated that errors occurred at least weekly.

There are other well-known examples of badly-designed user interfaces leading to death or serious damage. For example, singer-songwriter John Denver died in a plane crash, at least in part due to the plane’s builder choosing to put an important safety switch behind the pilot. (http://www.asktog.com/columns/027InterfacesThatKill.html) In fact, almost every time an accident is ascribed to “human error,” it’s likely some facet of a user interface led directly or indirectly to the error.

As far as the customer is concerned, the interface is the product.
--Jef Raskin, Macintosh pioneer

Beyond safety issues, a good user interface is a strong marketing tool. When users find a product easy to work with and supportive of their goals, they become loyal, sometimes to the point of fanaticism. They not only use the product themselves, but spread the word. Apple’s Macintosh has benefited tremendously from this kind of loyalty.

What goes into building the kind of interface that makes users crazy about your product rather than driving them crazy? As a developer, how can you help the people who use the applications you create?

The answer has two aspects. The first task is to design an interaction paradigm that supports rather than hinders users. A great deal has been written on this subject (see the Resources section) and this document will take only a high-level look at the subject.

The second step is to make the right choices in creating an actual user interface to implement that design. This includes choosing the right control for each task; remembering that the computer belongs to the user, not to your application; supporting different work styles; keeping things consistent; and much more. The bulk of this document considers these nitty-gritty issues.

Designing Interaction

When most people sit down in front of a computer, whether at work or at home, their goal is to accomplish a particular task. Few of them care about how the data is stored, the internal structure of the menu, or how reporting is optimized. They simply want to write a letter or update a patient’s chart or order 14 dozen widgets or any of the myriad other things that software can do. The purpose of a user interface, then, is to support them in accomplishing their goals.

Software developers, on the other hand, are generally fascinated by the details of how things work. We’re mostly the kind of people who take things apart to see what’s inside and may or may not bother to put them back together. We get excited by whiz-bang cool techniques and love to show them off.

Not surprisingly, when software developers design interfaces, the result is a mismatch. We’re likely to design interfaces that reflect the architecture of the application rather than the tasks users want to perform with it. The result is software that’s frustrating for users and users who’d rather have their teeth pulled than use some applications.

The best solution to the problem is hard for management (and developers) to accept. That’s having interaction designed by experts in design rather than by developers. While hiring user interaction designers may be realistic for large companies and major applications, most of the people who want software written are unwilling to invest either the money or the time to bring in the pros. Thus most developers inherit the job of designing both the interaction and the interface, whether they want it or not.

Best Practice: Design First; Code Later

The first place most developers go wrong with user interfaces is by not looking at the big picture, the overall plan for interaction in the application. Interface design is concerned with specific controls, colors, screen layouts and so forth. Interaction design takes a much larger view, addressing questions of functionality, paradigm and metaphor. Alan Cooper describes it this way:

> *Look and Feel stuff is Interface Design. It's all very stylistic. It's the color that you paint your walls. Interaction Design is about the Architecture. It's what kind of building are we building. What functions does it support. What are the shapes of the rooms and the walls and ceilings. What is the infrastructure. What kind of elevators. What kind of cooling and heating. That's Interaction Design.*
> *[http://www.uidesign.net/Articles/Interviews/AnaudiencewithAlanCooper.html]*

Plan to design the user interaction before you start writing any code; once you start coding, though you may not realize it, you're locking in aspects of the way the application and its users will interact. More importantly, designing interaction that works for users may affect the overall architecture of the application.

The place to start is with users. As you collect the information needed to design the application, listen carefully to what users tell you about their goals and the ways they use (or plan to use) the application. Make sure to talk to actual users, not just their managers, who may be quite unaware of how their employees actually work.

Once you understand what different users hope to do with your application, create a small number of *personas*, fictitious characters who represent prototypical users. Personas have names, biographies, and goals. Find photos to assign to them as well. In short, do anything you can to make the personas seem like real people. Plan to have one persona for each major type of user of your application. Generally, you'll need three to seven. (See http://www.infotoday.com/online/jul03/head.shtml and Cooper's *The Inmates are Running the Asylum* for more on designing personas.)

Let the personas drive interface design. As you consider an approach to a problem, ask yourself if it would make sense to Bob or Mary Jane. Would David feel comfortable working with this form? Once you start designing for individuals rather than the abstract "users," you're more likely to really consider whether a particular design works. It's easy to say "users don't want to do x," no matter what x is, but when you ask whether Helen wants to do x, you have to actually think about it.

Like other design processes, interaction design is an iterative process. Once you think you have it right (on paper), you need to show it to users and let them react. Then, revise to address their concerns.

Designing Good Interfaces

Once you have an interaction plan, you still have to design the user interface that implements that plan. In Cooper's terms, you have the house; now you have to decide on the floor coverings, wall colors, furniture, lighting and so forth.

Basic principles

As you work on the design, there are a number of principles to keep in mind. (This list is adapted from the writings of Cooper, Norman, Johnson, Raskin, and others.)

Consistency

This is a simple principle that's overlooked with surprising frequency. If something works one way in one part of the UI, it should work the same way throughout. If you use a Close button to leave one data entry form, another data entry form shouldn't use an Exit button for the same purpose.

Object-orientation helps enforce this kind of consistency, as you can create objects and use them throughout an application. If every data entry form is to close with a Close button, create a data entry form class that already has the button on it.

Consistency is important in the other direction as well. A word should have a single meaning in an application. If you use "widget" for one concept in one part of the application, don't use it for a different concept in another part of the application.

Visibility

This principle says to let users see what they can do. It may sound obvious, but with both physical objects and software, there are a surprising number of examples of invisible options. **Figure 2** shows a small toolkit (a giveaway from some conference); opening it the first time is not obvious.

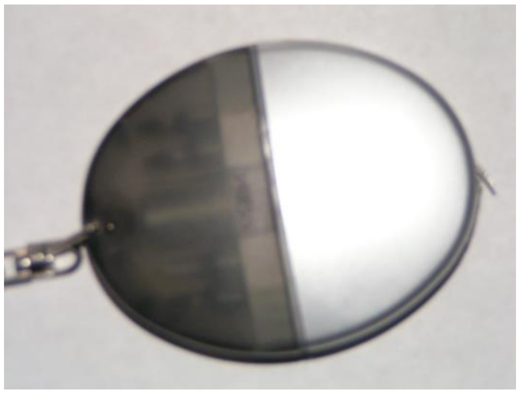


Figure 2. This tiny toolkit is opened by pressing in at the seam between the lid and the container. The only clue is the word "Press" in dark ink on the dark lid. (The word is present in this photo, but is unreadable.)

Visual FoxPro offers a good example of an invisible option. One of the most common questions posed by developers upgrading to VFP 7 or later is how to keep the Command Window from sitting on top of other windows. The solution is to uncheck the Dockable option on the Command Window's toolbar. While the reasoning makes sense once you know the solution, it's so far from the perceived problem as to be hidden.

The principle of visibility also means that a key combination or a function key shouldn't be the only way to trigger a particular action. It needs to appear on a menu or toolbar or somewhere in the visible interface as well.

Best Practice: Make every option visible.

Of course, invisibility can be useful in certain situations. An undocumented hot key offers a way to put a developer's back door into an application, but be prepared for users to stumble onto it occasionally without knowing how they got there.

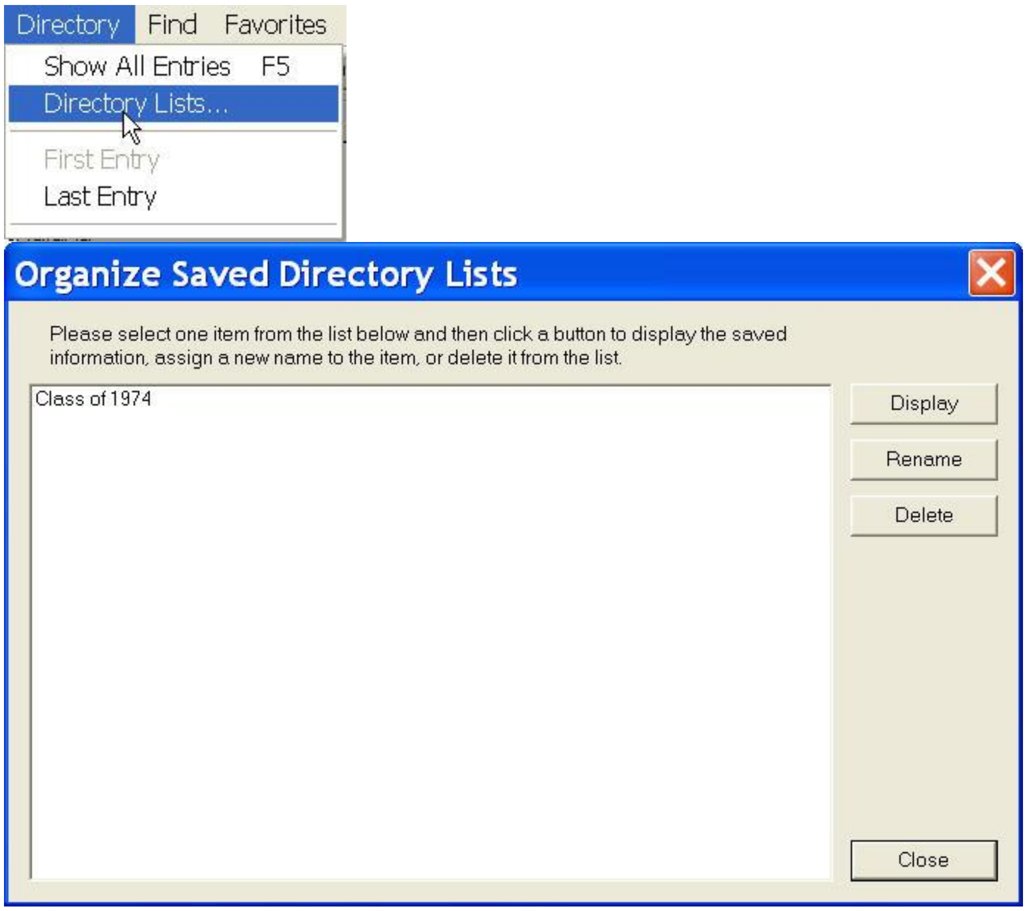
Feedback

The principle of feedback states that users should get some indication that their actions were noted and are being acted on appropriately. Feedback is so important to people that it's often explicitly added to devices. For example, people expect to feel something when they press a physical button; devices where the buttons don't actually move typically use a tone to indicate that the button press was received. In fact, even some devices where the buttons do actually move, like most telephones, provide auditory feedback.

Similarly, in applications, when the user does something, she needs to know that the application "heard" her. Feedback is needed at several levels. For example, when a user clicks a button, the first kind of feedback is the visual movement of the button. That confirms that she succeeded in clicking the button, rather than somewhere else.

The next type of feedback tells the user that the application understood the instruction implied by the button. The actual feedback varies with the situation. For a New or Add button, other controls on the form clear and the cursor is placed in the first field. For a Close button, the form closes. Often, the feedback for a control is obvious. Sometimes, though, you need to add feedback because there's no obvious choice or the action started by the user will take some time. In that situation, you need to use something like a message or a progress bar. (The key, of course, is to offer feedback that informs the user without impeding her.)

Feedback is also relevant when a button or menu option opens a form. The caption of the form should match the button's caption or menu item, so that the user knows the application did what she requested. **Figure 3** shows a menu and dialog from Search Party, a membership directory application that violates this rule.



Please select one item from the list below and then click a button to display the saved information, assign a new name to the item, or delete it from the list.

Class of 1974

Figure 3. In Search Party, an application for membership directories, choosing Directory Lists from the menu opens a dialog titled "Organize Saved Directory Lists." While a little thought makes it clear that this is the right item, the choice of a form title other than the menu option leads to a moment of uncertainty.

Simplicity

This may be the most difficult principle to apply because it's a judgment call. The goal is to make things as simple as possible, but no simpler. Clearly, making things more complex than they need to be is a bad idea (except in games or for Rube Goldberg devices). But why is making things too simple an equally bad idea?

Simplicity comes at a price; you have to trade something to get it, generally power, control or flexibility. For example, consider television remote controls. Some include a numeric keypad, on which you can type the channel number you want. Others omit the keypad, resulting in a simpler design, with fewer controls. However, on those remotes, you can only change one channel at a time, going up or down through the available channels.

Error-tolerance

Humans are imperfect, but most software expects them to be perfect. This principle asks you to throw away that expectation. Assume that users will make mistakes and make the consequences of mistakes as painless as possible.

The recycle bin is one attempt by Windows to save users from themselves. By keeping deleted files available until the user explicitly disposes of them, the user has one layer of protection. The Undo functionality available in many applications is another example.

On the other hand, the sequence of confirmations many applications require for dangerous actions doesn't really offer error-tolerance. It just offers the developer an excuse when the user moans over lost data. While confirmations seem like a good idea, the problem is that users come to expect them and respond automatically. Confirmations can be useful for extremely rare actions that are dangerous, but for normal activities such as deleting a record, they're just clutter.

The human capacity for error should also be considered when placing menu items and buttons. Users will land on the wrong menu item or click the wrong button. Keep destructive items away from common items. The shortcut menu for Windows Explorer includes a bad example. Delete, a dangerous action, is immediately adjacent to Rename, which for me, at least, is a common action. I often hit Delete when I want Rename; **Figure 4** shows how easy this is with the mouse. Note that the tip of the mouse pointer in the figure is right at the border between the Delete and Rename items; you only have to be a few pixels off to choose Delete rather than Rename. A better design for this menu would put Delete by itself with divider bars on both sides.

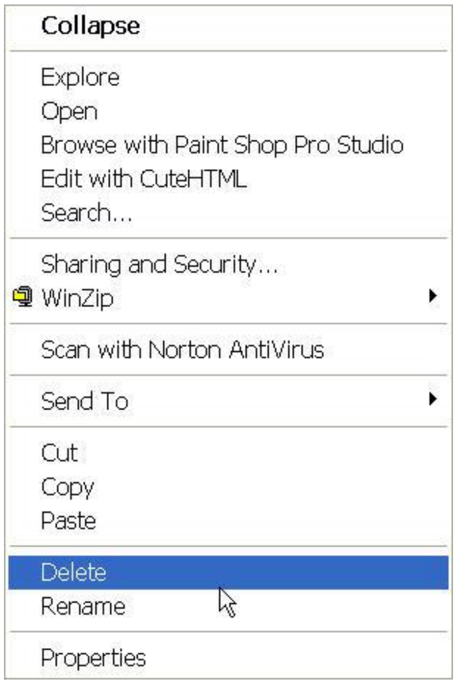


Figure 4. In Explorer's shortcut menu, it's easy to click Delete when you mean Rename. It would be safer to have Delete in a group by itself.

Best Practice: Expect users to make mistakes.

Accessibility

A significant number of people have one or more physical disabilities and, as the population ages, the percentage goes up. For the year 2003, the US Census Bureau estimates there were over 77 million Americans (about one-quarter of the US population) with a disability severe enough to impact their daily living. Failure to consider these people in designing user interfaces is a serious mistake. Fortunately, in most cases, ensuring that users with disabilities can work with your applications leads to good interface choices for all users.

The disabilities that affect user interfaces are primarily problems with vision, mobility or hearing, with vision the most common. Vision problems fall into three broad categories: no vision, limited vision and color blindness. Clearly, making an application accessible to users who cannot see is tricky, but most blind users have additional software that reads the screen to them. For those users, you just need to make sure that your interface provides the right kinds of information to the screen reader tools.

For users with limited vision and those who are color-blind, the first step is to respect the user's Windows settings. People who have vision impairments are likely to choose colors and fonts that enhance their vision. Your application should use these settings. In addition, it's a good idea to gives users a way to set the font size within your application.

Best Practice: Respect the user's Windows settings.

People with mobility issues may be unable to use a keyboard or a mouse, or may have problems using either with precision. (The most seriously affected may use only a blow stick for input and use their pointing devices with an on-screen keyboard.) For these users, it's essential that all options can be selected using either the keyboard or the mouse, so that whichever device a user can control suffices.

Best Practice: Make all options available using either the keyboard or the mouse.

Hearing problems pose less of an issue in user interface design, but make sure that nothing in your application requires the ability to hear. If you use sound as a signal, make sure there's an alternative visual signal as well.

Standards and guidelines

There are a variety of standards for user interface behavior. Among the best known are those published by Microsoft and Apple.

Follow existing standards and guidelines unless you have a compelling reason not to. It’s more acceptable to ignore standards in order to do something revolutionary than to ignore them in small ways.

The physical world is full of standards that make everyday life easier and safer. For example, the gas pedal of a car is on the right with the brake on the left; deviation from this standard would be dangerous. In North America, most light switches use up for “on” and down for “off”; in other parts of the world, the reverse is true. International travelers often find themselves fighting with the light switches until they adapt.

Although user interface standards may not always be the most logical (why *is* Ctrl+V the menu shortcut for Paste, anyway?), users already know them. Violating them will annoy your users. For example, early versions of WinZip used Ctrl+A as a menu shortcut for Add. For Windows users accustomed to Ctrl+A’s standard meaning of Select All, this was a surprise to say the least. The makers of WinZip obviously heard complaints because later versions use Ctrl+A for Select All and Shift+A for Add.

Does this mean you can’t ever do anything differently than existing applications? Of course not. If everyone did so, user interfaces would never change. If you have a new and better way to do something, go for it. But don’t ignore standards just because you personally prefer different behavior; your users won’t thank you.

Best Practice: Follow existing standards unless you have a compelling reason not to.

Putting the principles to work

With the basic principles in mind, you can get down to the nitty-gritty of actually designing and implementing a user interface. Some of the practical issues are direct reflections of the basic principles, while others require you to think about how the principles apply.

To demonstrate these ideas, I've created an application for a library that incorporates borrowing and returning books, managing members, looking things up in the catalog and maintaining the catalog. The application, which is a work in progress, is included in the session materials.

Application-wide issues

Some design choices apply to the user interface and the application as a whole. Make these decisions before you begin designing individual components.

Use task terminology

There are two aspects to using task terminology in an application. One is the whole point of view of the application, the way it looks at the world. The second is the choice of terms used to refer to the things the application deals with.

The first concern with task terminology is about the approach you take to presenting an application. From a developer's point of view, it's probably easiest to create one form for each table in an application and then put all those forms on the menu. When you do that, though, you're exposing the application's innards.

A good user interface instead presents the user with a set of tasks that reflect the process being modeled, not the model used. For example, Quicken is organized around different types of household accounting tasks; the main menu is shown in **Figure 5**.

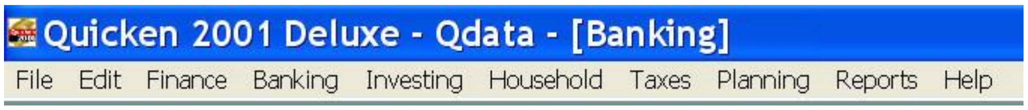


Figure 5. Quicken's main menu lists the types of household accounting tasks its users may want to perform, offering no indication of how the data for these tasks is actually stored.

> Note: Many user interface experts believe that the whole hierarchical directory structure used in today's operating systems is an example of exposing the implementation rather than considering the user's goals.

The second aspect of task terminology is the choice of actual words you use. Every field has its lingo, known as "words of art," including both terms unique to the field and common words used in a special way. "Assumption" means something entirely different to a mortgage banker than to a scientist. A musician, a mathematician and a psychologist each have different associations for the word "triangle."

Not only do different professions have their own languages, but individual companies may have their only way of referring to things or processes. Sometimes the choice of terms varies geographically. For example, in California, the final step in buying a house is called the "escrow closing," while in Pennsylvania, it's the "settlement."

It's important for an application to use the terminology users are familiar with and to use it in a consistent way. I learned this lesson the hard way many years ago. I wrote an application that essentially provided a front-end with search capability for a fixed set of data. The application allowed users to specify terms that must appear in the search result. On the main form of the application, I included a button that said "Set criteria"; clicking that button opened a form with a title of "Search Criteria." When I demonstrated the application for the client, he asked "What's a criteria?" I'd used a term I was comfortable with, but that was meaningless to the application's audience.

The best way to get the "words of art" right: don't guess; ask the users. Make a list of the terms they use for the objects and processes your application needs to deal with and use those terms throughout.

Best Practice: Work with the users to define all the terms that will be used and then use them consistently.

Use language well

Words of art aren't the only concern in an application. All use of language should be deliberate. In general, you want to follow many of the rules that apply for writing.

Check spelling carefully. Misspellings in your interface confuse users and lower their perceptions of your application.

Be grammatical. Although most aspects of a user interface don't use full sentences, do check for subject-verb agreement and other standard uses of the language. Use the right part of speech.

Use parallel form for items in lists, choosing the same grammatical form and tense for each item in the list. For example, if the caption for one option button in an option group is "Terms: Net 30 days," don't use "Terms are Net 60 days" for another button in the same group.

Avoid ambiguity. Not only did my "Set criteria" button use the wrong word, but it could be read two different ways, based on whether "set" was being used as a noun or a verb.

Use color wisely

The use of color in applications is somewhat controversial. No sooner did we reach the point where most users had color monitors than Windows came along with its recommendations for minimal color. What are the issues with respect to color?

Color is an easy way to make things stand out. Making one item red in a form full of black on white ensures that the user will notice that item. Color-coding is a very powerful mnemonic. The decision by computer manufacturers to color code the cords and connectors for assembling personal computers (**Figure 6**) has greatly increased the number of people who can put together their own machines. The decision by networks in the United States to represent Republicans with red and Democrats with blue on election maps has made the terms "red state" and "blue state" immediately recognizable.

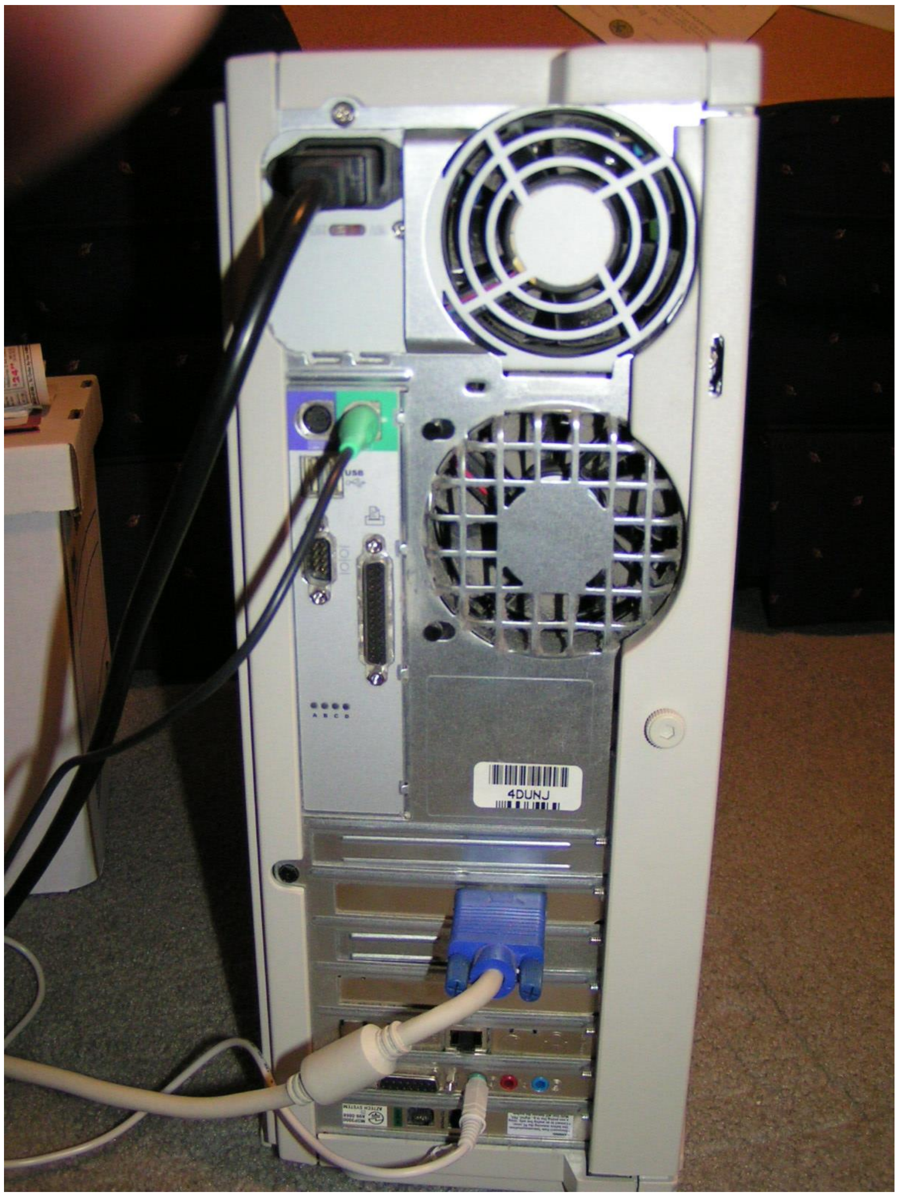
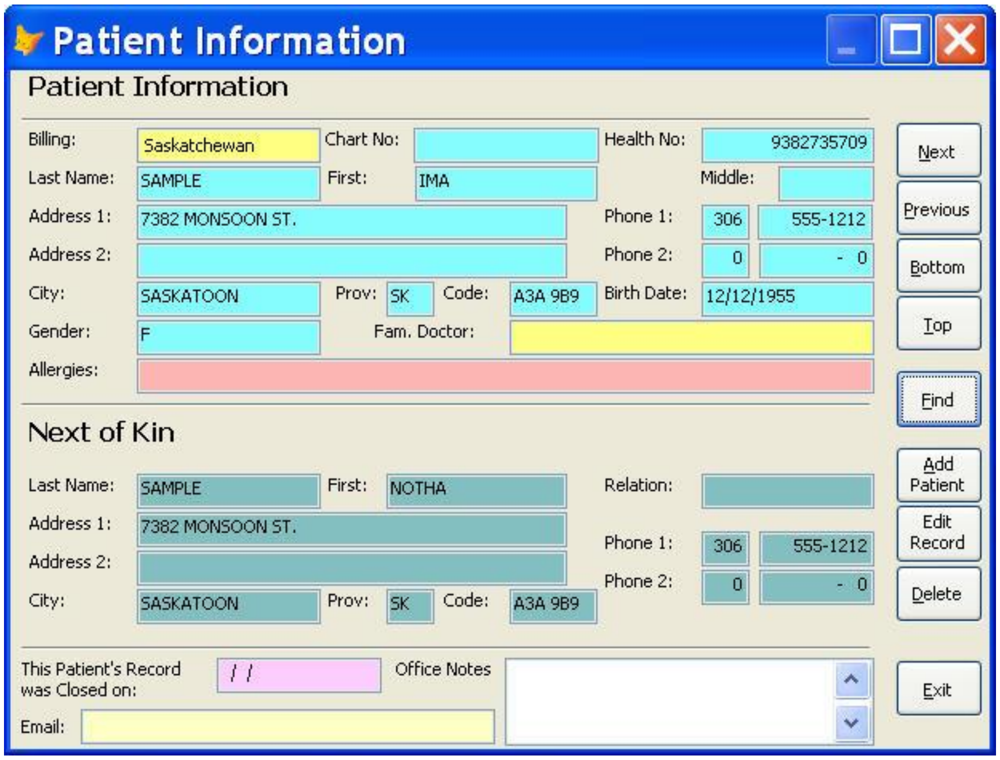


Figure 6. Computer manufacturers color-code connectors and cables to make it easier for people to assemble computers.

However, a significant portion of the population (about 10% of Western men) has some form of color blindness. People who are color blind can’t see the differences between some colors. What seems like good contrast to you may present little or no contrast to them. Subtle differences in color may be totally meaningless to them. Beyond those with color blindness, there’s a large group whose vision is weak. Contrast can be an issue for these users as well.

Color is also an emotional issue. People respond to color viscerally. This is apparent from the way we refer to color in the language (“I’m feeling blue”; “I see red”). An application painted in a color a user dislikes may evoke an unconscious negative reaction. Along the same lines, different colors have different meanings in various cultures. For example, in the West, white is the color of purity, used for wedding gowns, but in some Eastern cultures, white is the color of death.

Color can also be overused. When everything is colored and there's no pattern, forms look silly. **Figure 7** is a form from a vertical market application (with a few things changed to disguise it). There's no apparent meaning to the different colors used and the form contains far too many colors.



Patient Information

Billing: Saskatchewan
Chart No:
Health No: 9382735709
Last Name: SAMPLE
First: IMA
Middle:
Address 1: 7382 MONSOON ST.
Phone 1: 306 555-1212
Address 2:
Phone 2: 0 - 0
City: SASKATOON
Prov: SK
Code: A3A 9B9
Birth Date: 12/12/1955
Gender: F
Fam. Doctor:
Allergies:

Next of Kin

Last Name: SAMPLE
First: NOTHA
Relation:
Address 1: 7382 MONSOON ST.
Phone 1: 306 555-1212
Address 2:
Phone 2: 0 - 0
City: SASKATOON
Prov: SK
Code: A3A 9B9

This Patient's Record was Closed on: / /
Office Notes
Email:

Figure 7. The apparently random use of color on this form from a vertical market application leaves the user wondering what the meanings are.

So what's the solution? How can we use color to enhance an application without putting some users at a disadvantage or creating a garish mélange?

The first step is to follow the best practice noted earlier: Respect the user's Windows settings. Users have chosen their Windows theme for a reason. Sometimes it's just personal preference, and sometimes it's to ensure visibility. Whichever reason applies, developers shouldn't override those preferences.

The basic principle of consistency provides the next guide here. Use color consistently throughout your application. The application from which Figure 7 is drawn not only applies colors without meaning, but different forms in the application use different colors with no apparent reason.

Make colors meaningful. Use color to emphasize similarities and differences, not just to dress up your forms. Work with users as needed to discover the items that call for this type of emphasis.

In addition, consider this use of color an enhancement, not an essential part of your application. That is, whatever you're doing with color, make sure there's another way for users to gather the same information. While the plugs on the backs of computers are color-coded, they also have an icon or text to indicate what should be attached.

Use color sparingly. Too much color leads to overload for the users. One item in red draws your attention; 10 items in red are a distraction.

Finally, while emphasizing with color can be powerful, be aware that some users will assume that only the emphasized items are important and will ignore the rest.

Best Practice: Use color sparingly and meaningfully, but don't make it essential.

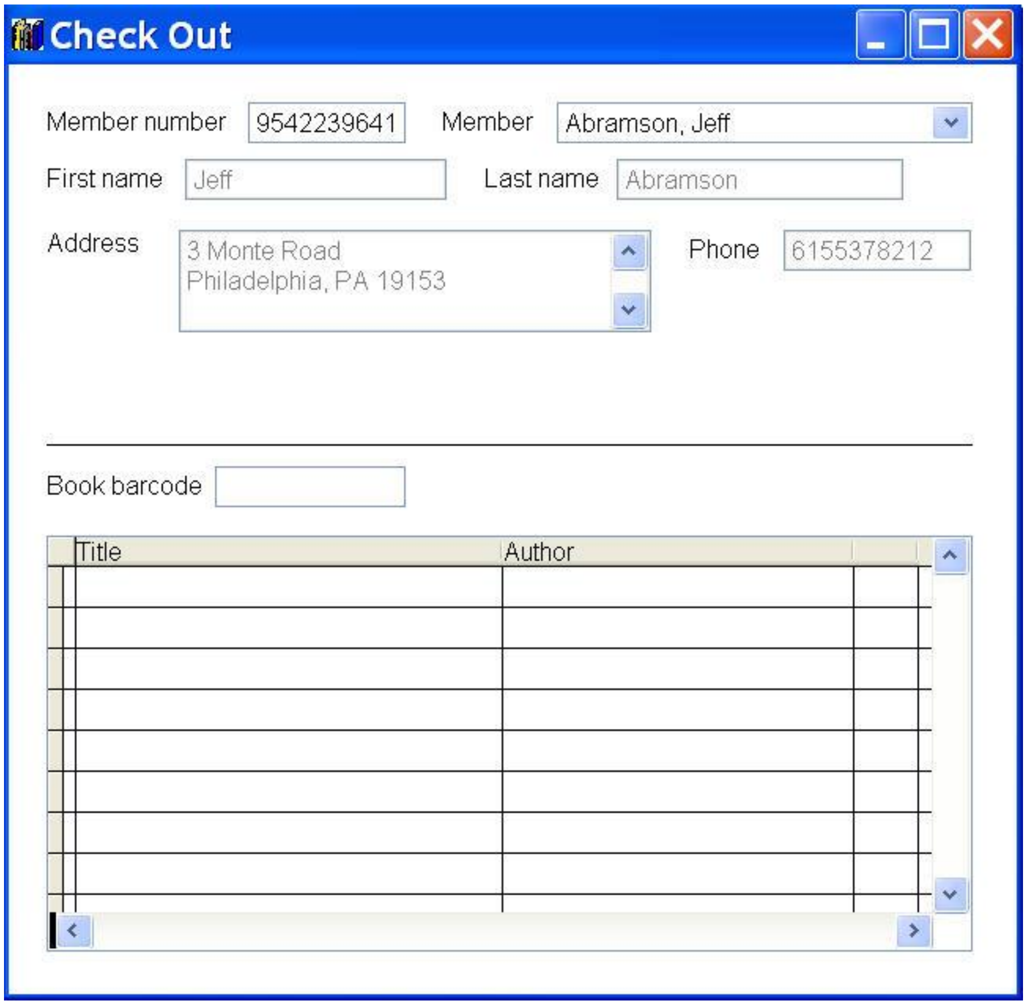
How can you follow all this advice in a VFP application?

- For the ColorSource property of forms, use either the default setting of 4-Windows Control Panel (3D colors) or 5-Windows Control Panel (Windows colors). Use the latter setting if you're treating forms in your application as documents rather than dialogs. (See "Distinguish documents and dialogs" later in these notes.)
- Leave the ColorSource property of your base class controls at the default of 4-Windows Control Panel (3D colors).
- If you put "wallpaper" on your application's main window, make sure it's either subdued, so it doesn't distract the user, or that the user can turn it off or change it.
- For situations where something out of the ordinary is needed, such as drawing attention to a particular control, draw the colors you use from the user's theme. You can use the GetSysColor API function to find the colors currently in use. The function is easy to use. The code in **Listing 1** shows the constant declarations and the function declaration. To use the function, just call it, passing the appropriate constant. Keep in mind that API functions are case-sensitive, so you must reference the function as GetSysColor with the embedded capital letters. For example, the CheckOut form in the example Library application has some controls that are there for reference only and always disabled. Because disabled controls are hard to read in a number of themes and color schemes, the form uses the AppWorkSpace color and the Window color for the disabled forecolor and disabled backcolor, respectively, as in **Figure 8**.

Listing 1. You can find out what colors the user has chosen in Windows using the GetSysColor API function.

```
#DEFINE COLOR_SCROLLBAR 0
#DEFINE COLOR_DESKTOP  1
#DEFINE COLOR_ACTIVECAPTION  2
#DEFINE COLOR_INACTIVECAPTION  3
#DEFINE COLOR_MENU  4
#DEFINE COLOR_WINDOW  5
#DEFINE COLOR_WINDOWFRAME  6
#DEFINE COLOR_MENUTEXT  7
#DEFINE COLOR_WINDOWTEXT  8
```

```
#DEFINE COLOR_CAPTIONTEXT  9
#DEFINE COLOR_ACTIVEBORDER  10
#DEFINE COLOR_INACTIVEBORDER  11
#DEFINE COLOR_APPWORKSPACE  12
#DEFINE COLOR_HIGHLIGHT  13
#DEFINE COLOR_HIGHLIGHTTEXT  14
#DEFINE COLOR_3DFACE  15
#DEFINE COLOR_3DSHADOW  16
#DEFINE COLOR_GRAYTEXT  17
#DEFINE COLOR_BTNTEXT  18
#DEFINE COLOR_INACTIVECAPTIONTEXT  19
#DEFINE COLOR_3DHIGHLIGHT  20
#DEFINE COLOR_3DDKSHADOW  21
#DEFINE COLOR_3DLIGHT  22
#DEFINE COLOR_INFOTEXT  23
#DEFINE COLOR_INFOBK  24

DECLARE INTEGER GetSysColor IN WIN32API INTEGER nElement
```



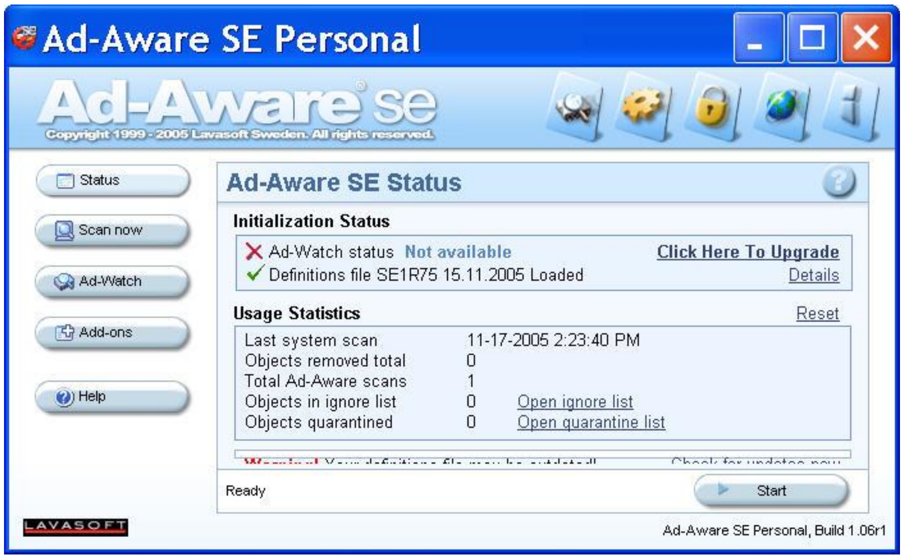
Member number: 9542239641
Member: Abramson, Jeff
First name: Jeff
Last name: Abramson
Address: 3 Monte Road
Philadelphia, PA 19153
Phone: 6155378212

Figure 8. This form draws colors from the current theme/scheme to make disabled controls more readable.

Use scalable fonts

Just as Windows' users can set colors, they can also make font-related choices. The most important from the application development perspective is the DPI setting that determines how fonts are drawn. (In earlier versions of Windows, the user could simply choose between "normal fonts" and "large fonts"; in Windows XP, users can make a custom setting as well.)

As long as you use scalable fonts, large fonts won't give you any trouble. But if you choose a non-scalable font (such as MS Sans Serif), captions and controls may be cut off for users with large fonts. **Figure 9** shows the main window of Lavasoft's Ad-Aware with Windows set to use large fonts; part of the information is cut off.



Ad-Aware SE Status

Initialization Status

✗ Ad-Watch status Not available Click Here To Upgrade
✓ Definitions file SE1R75 15.11.2005 Loaded Details

Usage Statistics Reset

Last system scan	11-17-2005 2:23:40 PM	
Objects removed total	0	
Total Ad-Aware scans	1	
Objects in ignore list	0	Open ignore list
Objects quarantined	0	Open quarantine list

Ready

Ad-Aware SE Personal, Build 1.06r1

Figure 9. Apparently, Ad-Aware uses a non-scalable font, as part of its display is cut off when Windows is set to use large fonts.

Following this advice in VFP is simple. Make sure the FontName setting for your base form and control classes specifies a scalable font.

Make your application easy to use with both the keyboard and the mouse

Watch a number of people work with a computer and you'll be surprised at the variations in what they do with the keyboard and what they do with mouse. Some type everything possible and resort to the mouse only occasionally. Others use the keyboard only for text input and do all selection and editing tasks with the mouse. Still others work in the middle ground between these alternatives.

Similarly, if an option is available through the menu, a menu shortcut, a toolbar option and a shortcut menu, different users will access it differently. In fact, the same user may access it differently at different times, depending on what she's currently doing.

Beyond the normal variations, some users find one type of device much easier to use. Sometimes, the issue is a permanent motor disability. Sometimes, it's a temporary disability such as a broken arm. Sometimes, the cause isn't the user, but the equipment. A temperamental keyboard or a switch from a desktop machine to a laptop may change a user's keyboard and mouse habits.

All of these issues lead to the best practice stated earlier: Make all options available using either the keyboard or the mouse.

This best practice isn't hard to implement in most cases:

- Give every menu item a hot key. Give commonly used menu items shortcuts, as well.
- Make sure that TabStop is True (the default) for every input control. It is okay to set TabStop to False for controls used only for display.
- If an item is on a toolbar, make sure that it's included on a menu somewhere, as well.
- Take advantage of the built-in menu items, such as those on the Windows menu.

The one item you don't need to worry about here is textual input using the mouse. If a user is totally unable to use a keyboard, you can assume he has access to an on-screen keyboard tool that can manipulated with a pointing device. (In fact, Windows includes one.)

The hardest issues on this front are situations where the user needs to point to some place on the screen. But there are generally solutions, even if they're a little awkward. For example, prior to VFP 8, the only way to select an item in the VFP Report Designer was with the mouse. In VFP 8 and later, you can tab through the controls in a report.

Remember so the user doesn't have to

Computers are great at remembering things; with the increased storage capacity of current machines, this is truer than ever. People aren't as good at remembering. Even when they're doing something they've done many times before, it's not unusual to skip a step or make a mistake. So one of your goals for the user interface should be to minimize what the user needs to remember.

Applications can remember all kinds of things—what the user did last, how a particular user likes things set up, the most common entries for a particular field, and much, much more. Making your application remember, though, takes advance planning.

Existing applications vary in their use of memory. In fact, even within an application, there can be variations. For example, when you open a document in Microsoft Word, it shows the same view you were last using. On the other hand, in other places, Word rapidly forgets. One that drives me crazy is the Cross-Reference dialog (**Figure 10**). When you choose a different type of reference to insert (using the Reference type dropdown), the content to be inserted always reverts to "Entire caption". Since I often insert several different types of references (figures, tables, etc.) into a document and never insert the entire caption, Word's insistence on this setting is frustrating.

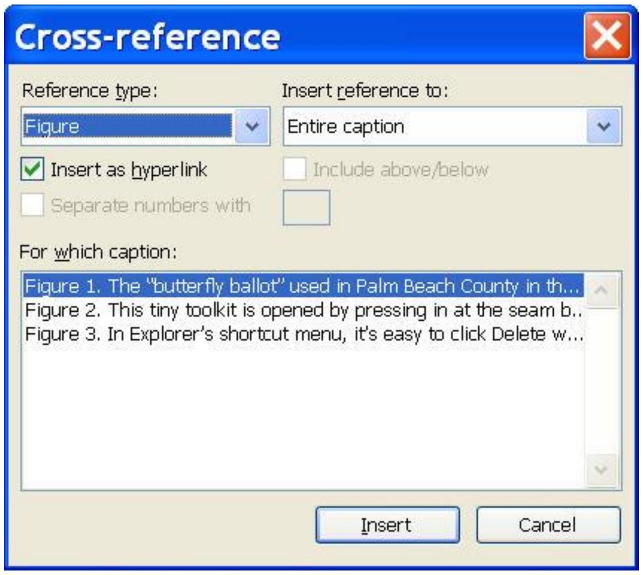


Figure 1. The "butterfly ballot" used in Palm Beach County in th...
Figure 2. This tiny toolkit is opened by pressing in at the seam b..
Figure 3. In Explorer's shortcut menu, it's easy to click Delete w...

Figure 10. When you change the Reference type in this dialog, the Insert reference to dropdown always reverts to Entire caption.

VFP's interface does quite a good job of remembering. Windows open where you left them and when you open a code window, the cursor is where you left it. Even better, VFP gives you control over this behavior through the SET RESOURCE setting. VFP's Debugger has another nice touch. You can rearrange the windows, add breakpoints, change settings and so on to your heart's content, but choose Window | Restore to Default from the Debugger menu and everything is put back to its "out of the box" state.

What can your application remember?

- Window positions and sizes
- Settings/preferences
- The record(s) last edited in various forms
- The most common selections and entries for various fields
- Most recently used items

How can your application remember? The answer varies with what there is to remember. The easiest thing to remember is the most common entries for various fields. VFP 9's auto-complete functionality lets you build memory into textboxes simply by setting a few properties.

To remember other items, you'll have to build functionality. Doug Hennig published an article in the January, 2000 issue of FoxTalk that showed how to store information such as window positions and the last edited record. It's easy to build this functionality into your base classes. Once you do so, you can store whatever information you want and restore it as needed. A refactored version of this capability is included with the session materials.

Provide Undo

One of the best practices discussed with the general principles is: Expect the users to make mistakes. One way to do so is to provide undo functionality. A well-designed undo capability can eliminate some of the generally ignored confirmation dialogs.

VFP provides a basic undo facility. As long as you include the Undo menu item (_med_undo) somewhere in the menu, you can undo typing. It works in textboxes and editboxes, as well as in various windows unlikely to be used in applications, such as memo windows, code windows, and so forth. However, in forms, it works only within a single control; as soon as you leave the control, you cannot undo typing there, even if you return focus to that control.

It's also not hard to write code to provide a basic undo for a form that restores all fields to the values stored in the table. Just call TableRevert() for each table and then refresh the form.

VFP's two-step deletion process also offers a type of undo capability, since it makes it possible to restore deleted records.

A more comprehensive undo facility, that tracks application actions, and provides a general way to return to an earlier state, would be an extremely powerful addition to any application.

Application Control and Menus

Most database applications use a menu bar as their primary control device. In most cases, that's a good choice, since users of business applications are familiar with this approach. In some situations, other approaches may make more sense. For example, a kiosk application for a public place is better served by an opening form (a switchboard form) that uses buttons to direct users to the main portions of the application. Even in a business application with a menu bar, such an opening form can be very helpful to new users. In that case, it would duplicate the menu choices rather than replacing them.

Beyond the menu bar or switchboard form, there are a number of tools available for application control. Toolbars and shortcut menus provide users additional ways to access options.

Organize the menu bar sensibly

The first issue in designing the main menu for an application is deciding what menu pads to use, that is, how you should organize the options of the application. Think about the user's tasks and goals. While the menu should probably include File, Edit and Help pads to match the user's expectations, don't make the other pads Forms and Reports. Use words that reflect the application domain. For example, in an application for managing a library, the pads might include Circulation, Reference and Collection. Each menu pad then includes the actions related to that aspect of a library, as in **Figure 11**.

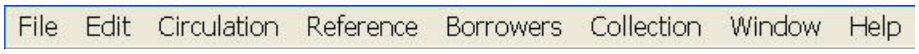


Figure 11. This main menu for a library application is organized around the main areas of responsibility for the librarians.

Best Practice: Organize the menu around the user's tasks.

Each menu pad, of course, leads to a menu popup, containing the items related to that pad. Within the popup, you have several choices as to organization. One option is to list the items in the order in which they're likely to be used; this makes the most sense for sequential processes. Another choice is put items in frequency order, with most often used items at the top.

Whichever organization you choose, use divider bars in popups to separate items into meaningful groups. The dividers make it easier to find the desired item, and harder to click on the wrong item by accident.

There's general agreement among usability researchers that cascading menus (where a menu item contains a submenu, which may in turn contain a submenu) are harder to use than flat menus. On the other hand, too many items in a single menu popup can be overwhelming. It's probably best to use no more than one level below the menu popup; that is, the popup contains items, which may contain submenus, but the submenus do not contain submenus. To avoid overloading the individual popups, consider using dialogs called from the menu. For example, choosing Format | Font… in VFP opens the Font dialog rather than showing a submenu containing Font Name, Font Size, and so forth.

It may be tempting to put some items in more than one place on the menu. When the same item appears more than once, users try to understand the difference between the two. Resist the temptation and view it instead as a sign that your menu organization needs revision. However, putting an item on the menu and a toolbar, or on the main menu and a shortcut menu is not a problem.

Provide hot keys and menu shortcuts

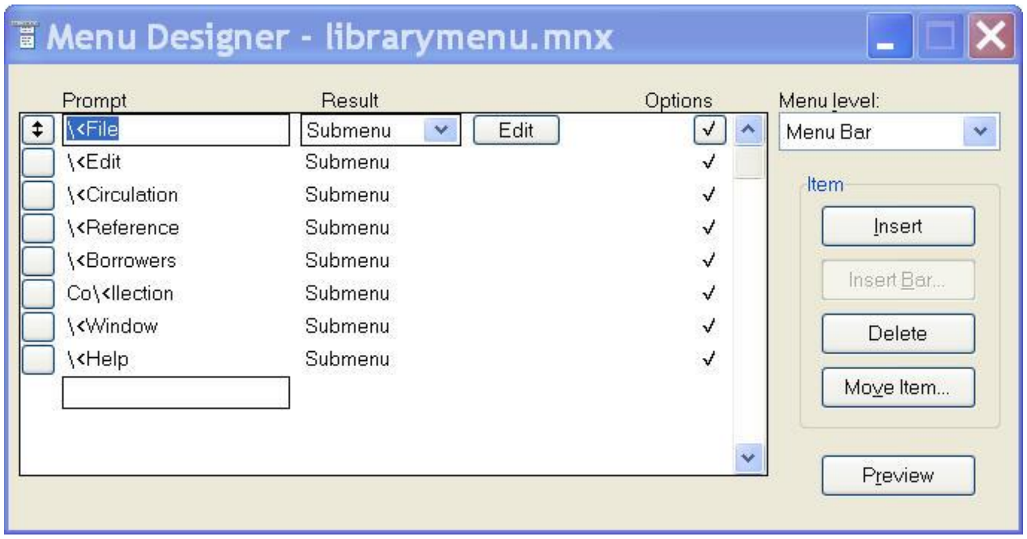
Windows menus offer two different mechanisms to simplify keyboard use: hot keys and shortcuts. Offer both in your applications to speed more advanced users.

Menu hot keys are the underlined keys that appear when the menu has focus (and always appear for menu pads in some versions of Windows). They allow users to navigate using the keyboard, without having to use the arrow keys. For example, C is the hot key for Copy on the Edit menu.

Menu shortcuts are the key combinations that appear at the end of menu items and allow users to choose the item without opening the menu. For example, Ctrl+C is the menu shortcut for Copy.

In VFP, both hot keys and shortcuts can be specified using the Menu Designer. To set a hot key, precede the chosen letter of the prompt with "\<" (omitting the quotes), as in **Figure 12**. To specify a shortcut, use the Prompt Options dialog (**Figure 13**) that's accessed through the

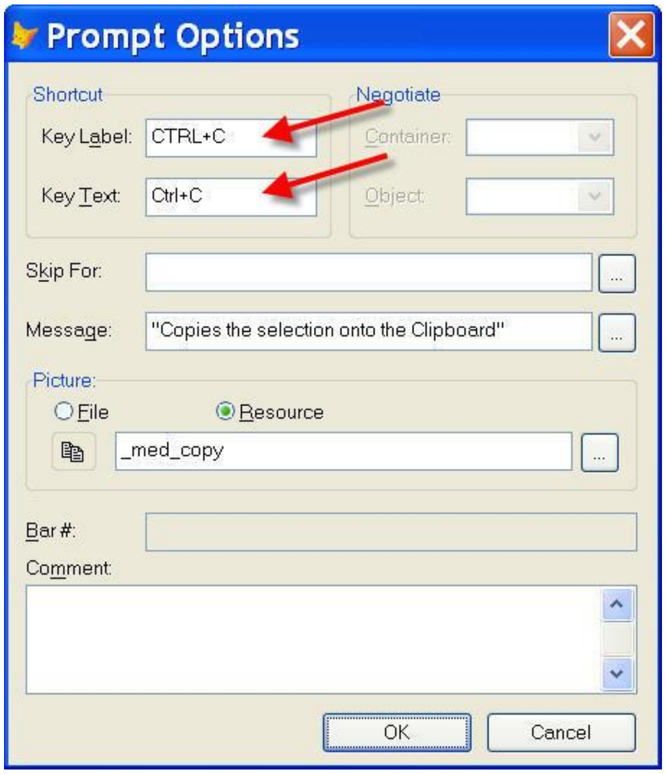
Options button for each item in the Menu Designer. Click into the Key Label textbox and then press the key combination you want to use. It’s traditional to use Alt-key combinations for menu pads and Ctrl-key combinations for menu bars. In the Key Text textbox, you can specify how the shortcut should be displayed on the menu. The shortcut appears at the right-hand side of the menu popup.


Menu Designer - librarymenu.mnx

Prompt	Result	Options
\<File	Submenu	✓
\<Edit	Submenu	✓
\<Circulation	Submenu	✓
\<Reference	Submenu	✓
\<Borrowers	Submenu	✓
Co\<llection	Submenu	✓
\<Window	Submenu	✓
\<Help	Submenu	✓

Menu level: Menu Bar

Figure 12. Use "\<" in the caption of an item to specify a hot key.



Key Label: CTRL+C
Key Text: Ctrl+C
Skip For:
Message: "Copies the selection onto the Clipboard"
Picture: [✓ Resource] _med_copy
Bar #:
Comment:

Figure 13. Use the Prompt Options dialog to specify a shortcut for a menu item. The Key Label textbox contains the name of the key; the Key Text textbox indicates how it will be displayed on the menu item.

Best Practice: Give every menu item a hot key.

Every item within a menu popup should have a unique hotkey. How do you choose the hot key for each menu item? This prioritized list is adapted from Johnson's *GUI Bloopers*:

- If there's a Windows standard hot key for an item, use it.
- Use the first letter of the prompt.
- Use the first letter of another word in the prompt (sticking to the meaningful words).
- Use a consonant from the prompt, preferably one that is pronounced rather than silent.
- Use the first letter in the prompt that's available.

Best Practice: Give frequently used menu items shortcuts.

You don't need a shortcut for every menu item, just for those users frequently access. In fact, it's better not to provide shortcuts for particularly destructive menu items. Here are some guidelines:

- Give each menu pad a shortcut, using Alt + a letter. If the pad is widely used in Windows menus, use the usual hotkey.

- Provide shortcuts for “standard” menu items such as File | New, Edit | Copy and so forth. Use the standard shortcuts for those items.
- Provide shortcuts for menu items users will need frequently. Use Ctrl-key combinations, making each combination unique across the application.

Clue users in about menu items’ behavior

Some menu items simply perform an action; for example, Edit | Paste pastes whatever is on the clipboard at the cursor position. Other items open a dialog to collect more information before acting; for example, VFP’s Program | Do… opens a dialog to choose the program to execute. When a menu item opens a dialog, put an ellipsis (three dots) at the end of the item’s text; this convention tells users that a form will open.

Including the ellipsis makes it safer for users to explore the application. When a menu item ends with an ellipsis, the user knows he can choose the item to see what appears without actually triggering an action.

Manage users’ access to menu items

Application actions can be divided into three groups for any user: those he can always use, those he can sometimes use, and those he can never use. Manage the menu to make the distinctions clear to users.

Disable items when they’re not available at the moment. In VFP, the SKIP FOR clause of the menu commands lets you handle such items. Use application properties or methods to track a particular user’s status. The built-in menu items, like Cut, Copy and Paste, handle enabling and disabling automatically, so you don’t have to worry about them.

Don’t show a user menu items he can never choose; make them disappear. If only administrative users can access the Human Resources module, don’t display it for other users. It’s both frustrating and tempting to them. While VFP’s Menu Designer can’t manage visibility of menu items, the public domain tool, GenMenuX, makes it easy to do so. (GenMenuX is included in the materials for this session.)

Best Practice: Don’t show a user menu items he can never use.

The exception to this rule is in trial versions of an application. There, you may choose to disable some options; in such cases, a message should indicate that this option is available in the full version of the product.

Use toolbars as mouse shortcuts

Toolbars are to mouse users what menu shortcuts are to keyboard users—a quick way to access the most frequently used items in an application. Keeping this rule in mind makes it easy to decide whether or not to use toolbars in an application and which items to put on toolbars. For the most part, it's the same items for which you provide menu shortcuts.

Everything on a toolbar should be available on the menu as well. Otherwise, it's only accessible to mouse users, not to those working from the keyboard.

Use separators to divide the items on a toolbar into logical groups. Separators serve the same purpose in a toolbar that lines do in a menu—they help users see the items and make it easier to land on the right one.

Use graphical buttons and checkboxes on toolbars and do not include a textual caption. Instead, give every item on the toolbar (except separators) a tooltip. Tooltips should be brief, one or two words. If you need a whole sentence or paragraph to explain what a toolbar button does, perhaps it's not really a good candidate for the toolbar.

Use the same icons on the toolbar as in the menu. That helps users learn what an icon represents.

Toolbars are not intended to contain input controls, like textboxes. While you can make them work, including them violates the basic idea that a toolbar is a set of menu shortcuts.

Best Practice: View toolbars as menu shortcuts for mouse users.

Large and small toolbars offer users more control

Toolbars can be very handy, but the traditional size of toolbar buttons can make it hard to interpret the icons and to hit the right button. Making toolbars larger uses valuable screen real estate. One good compromise is to offer both and let users decide.

Some applications that offer both large and small toolbars use the same icons on both. Others use more detailed icons on the large toolbar. **Figure 14** shows the small and large toolbar in WinZip;

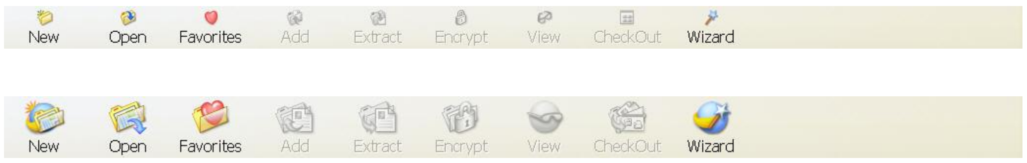


Figure 14. WinZip lets you choose whether to use small or large toolbars. The icons on the large toolbar have much more detail, yet are far more visible.

It's not terribly hard to provide large and small toolbars in VFP. The materials for this session include a toolbar class and classes for command buttons, checkboxes, and option buttons that can switch between large and small. The most difficult aspect is likely to be finding icons in both sizes, but some tools are available that let you resize graphics proportionally. (I've used SnagIt Editor for this task.)

Shortcut menus provide another alternative

Shortcut menus offer another way to make menu options available. These menus, also known as context menus or right-click menus, appear when the user right-clicks, presses the shortcut key on the keyboard or presses Shift+F10. Typically, shortcut menus contain just a few items chosen for their relevance to whatever the user is working on. **Figure 15** shows the shortcut menu available in VFP's code editing windows. It has more options than are typical.

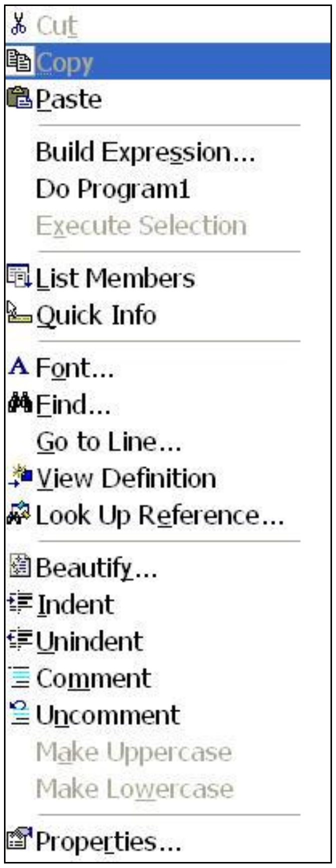


Figure 15. This shortcut menu, available in code editing windows in VFP, is longer than most.

Well-designed shortcut menus can help users learn what an application can do. Some users right-click everywhere when learning an application to see what's available.

Be consistent in your use of shortcut menus. It's okay to omit them entirely, but if you choose to provide them, then do so uniformly. If one data entry form offers a shortcut menu, all forms should.

At the same time, shortcut menus should only contain items relevant to the current situation. Don't create a single shortcut menu to make available throughout your application; instead create separate shortcut menus for different situations.

Organize shortcut menus with most frequently used items at the top to make navigation as easy as possible. The exception to this rule is for dangerous items; make it hard to select these by accident or omit them from shortcut menus. In VFP, shortcut menus open with the first item selected. Be sure that accidentally choosing that item won't cause damage.

As in other menus, use lines to separate groups of items and to set off dangerous items.

Every item in a shortcut menu should appear on the main menu, as well. Use the same icon for the item everywhere it appears.

You can create shortcut menus with VFP's Menu Designer. Choose Shortcut from the New Menu dialog (**Figure 16**) that appears when you create a Menu.

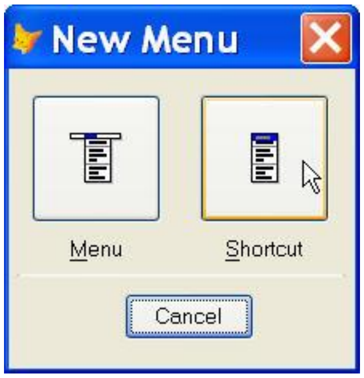


Figure 16. Choose Shortcut in this dialog to create a shortcut menu. VFP's menu generator program, GenMenu, generates a single popup that appears at the mouse position.

Shortcut menus are generally open by right-clicking on the relevant object. However, most keyboards also offer a key to open the shortcut menu; in addition, in most applications, Shift+F10 also opens the shortcut menu. It's easy to hook your shortcut menus into the RightClick method of the relevant controls. However, you need additional code to allow users to access shortcut menus from the keyboard. Checking for Shift+F10 in the KeyPress method catches both that key combination and the shortcut menu key.

Forms

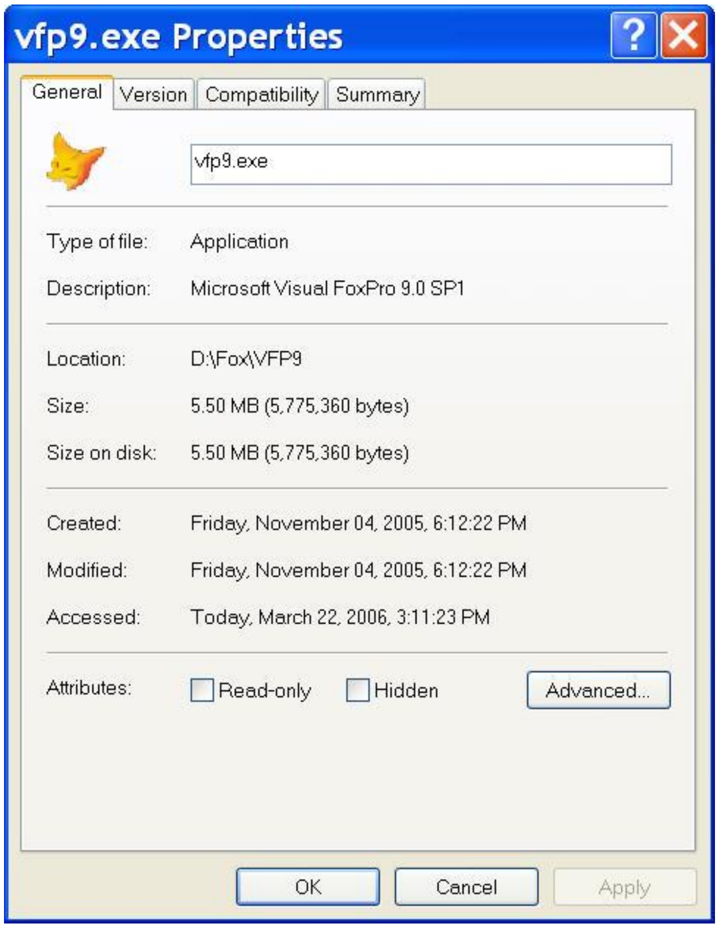
For most applications, forms are where users spend the most time—adding and modifying data, then examining and organizing it. So it's important for forms to reflect the user's way of thinking about the task at hand.

The main tasks for database applications generally involve entering new data, looking up and modifying existing data, and crunching data to get long-term results, trends, and so forth. The amount of time users spend on each of these tasks varies with the application; in some (like point-of-sale applications), the most common activity is entering new data, while in others (like a library system), it's looking up or modifying existing data. In addition, different users of the same application may spend much more time with one aspect than another. For example, a cashier using a POS system enters lots of new data, and does little else, but the store manager spends most of her time in the number-crunching portions of the application.

In designing the forms for an application, you need to consider how often and under what circumstances someone uses the form. A data entry form that needs to be filled in dozens of times a day should be streamlined to allow heads-down data entry (where the user doesn't even look at the screen). On the other hand, a form for performing month-end or year-end tasks should guide the user through the necessary steps.

Forms have title bars

With few exceptions, every form should have a title bar that identifies its purpose. The caption on the title bar should reflect the menu item or button used to open the form. (See the "Feedback" section earlier in this document.) You may want to enhance the title bar by including information about the specific item it's addressing. For example, when you choose Properties for a file in Windows Explorer, the Properties window includes the file name, as in **Figure 17**. To follow the Feedback principle, though, it's probably better to put the identifying information after the form name, so "Properties for vfp9.exe" is a better choice in Figure 17.



vfp9.exe Properties

vfp9.exe

Type of file:	Application
Description:	Microsoft Visual FoxPro 9.0 SP1
Location:	D:\Fox\VFP9
Size:	5.50 MB (5,775,360 bytes)
Size on disk:	5.50 MB (5,775,360 bytes)
Created:	Friday, November 04, 2005, 6:12:22 PM
Modified:	Friday, November 04, 2005, 6:12:22 PM
Accessed:	Today, March 22, 2006, 3:11:23 PM
Attributes:	☐ Read-only ☐ Hidden

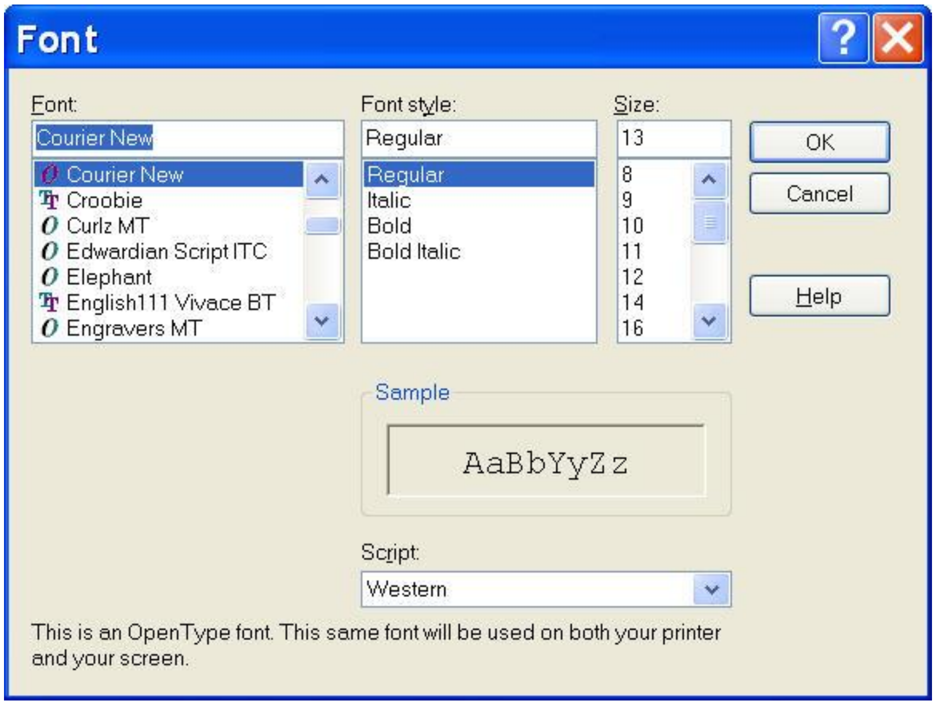
Figure 17. Choosing Properties for a file in Windows Explorer opens a window that includes the file name.

Best Practice: Give every form except the splash screen a title bar.

The principal exception to the title bar rule is a splash screen, displayed to provide rapid feedback on application start-up and to give the user something to look at while the application performs necessary start-up tasks. Splash screens typically omit title bars and include the name and version of the application, and other identifying information.

Distinguish dialogs and documents

Forms fall into two main categories, dialogs and documents. Dialogs let an application communicate with a user. Sometimes the communication is one way, as with the most basic form of the Windows message box, but often it's two-way, with the dialog collecting information needed to perform a particular task. For example, the Font dialog (VFP's version is shown in **Figure 18**) lets you specify font characteristics. When you choose OK, they're applied to a particular object. (This is another example where including identifying information would be useful; a title of "Font for xxx" would add clarity.) Dialogs are generally displayed briefly and then dismissed; they're almost always modal. Dialogs are an interruption.



This is an OpenType font. This same font will be used on both your printer and your screen.

Figure 18. The Font dialog lets you specify font settings for a particular item, though the VFP version of the dialog doesn't tell you what item.

Documents, also known as primary windows, are where the main action of an application takes place. In some applications, like Word and Excel, the term "document" is very comfortable. For database applications, describing data entry forms as documents may seem like a stretch, but in fact, when you view them as the creation and modification of records, the analogy makes sense. After all, Word is all about the creation and modification of letters, reports, labels and so forth. Excel is for creating and editing workbooks. Documents tend to be displayed for long periods of time and are never modal. Documents aren't an interruption; they're the main act.

Using the right window type helps users understand their options. Use documents for your application's main tasks and use dialogs for helper forms as needed. In VFP, though few developers do so, you can easily follow the Windows coloring guidelines for documents and dialogs using the ColorSource property. Leave dialog forms set to 4-Windows Control Panel (3D Colors) and set documents to 5-Windows Control Panel (Window Colors). For example, the CheckOut form for the Library application in Figure 8 is a document, so it uses Window colors.

Data entry forms are not modal

Back in the old days, when a user opened a form, she was expected to do what she needed with that form, close it and only then open another. FoxPro offered the possibility of opening multiple forms at once and letting a user work with multiple forms long before it was the most dominant paradigm. Nonetheless, some developers still use only modal forms in their applications. Don't.

Best Practice: Make data entry forms modeless.

Very few users work in an environment where the task they're performing can't be interrupted by another, more pressing, task. A billing clerk may get a phone call from the boss, wanting the details of last week's order from a big customer right now. Entry of accounts receivable may be interrupted by a customer asking for shipping information on a recent order. Even environments where we tend to think of the work as purely sequential don't necessarily have to be so; I've read about a point-of-sale application for supermarkets that allows the cashier to put one order on hold when there's a problem and continue with the next customer in line.

Even if a user isn't interrupted by another task, it's not unusual to have to look up information in order to perform the current task. While we can design applications to make the most common look-ups available from data entry forms, we certainly can't cover all the possibilities.

The bottom line here: A user should almost never have to cancel the work he's doing in order to go to another part of the application.

Keep dialogs to a minimum during processing

There are two kinds of dialogs in an application, the ones the user requests and the ones the application generates. The user requests things like an Options dialog that lets him establish the way the application operates or a Print dialog to choose a printer and specify print settings. For these windows, a dialog is usually a good choice.

Dialogs the application generates are an interruption; use them sparingly. Design your application so that users can work fluidly. Interrupt them only for important things.

What constitute important things? Here are some examples:

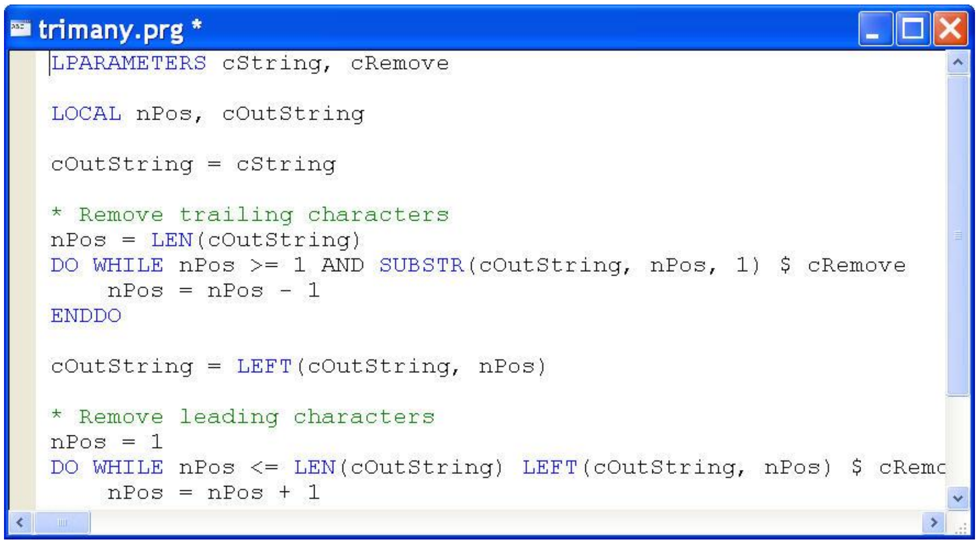
- Hardware problems the application can't work around or ignore;
- Confirmation of seriously destructive actions (like erasing all the data);
- Keeping the user informed of progress for long tasks;
- Letting the user select a file or folder.

What's just an interruption?

- Confirmation of normal action, like deleting a single record;
- Telling the user what was just done unless it's unusual (and even if it is, telling the user is only useful if she can do something about it);
- Telling the user about an error the program can simply fix.

In other words, alerts and confirmations should be used minimally, if at all. First, they're annoying because they're interruptions.

Second, they don't really accomplish anything. Most of what alerts do can be accomplished with other techniques. For example, VFP doesn't show you a "File Saved" dialog every time you save a program. Instead, the title bar of a code editing window contains an asterisk when there are unsaved changes, as in **Figure 19**. When you save, the asterisk disappears. This mechanism provides immediate feedback and eliminates the need for an alert.

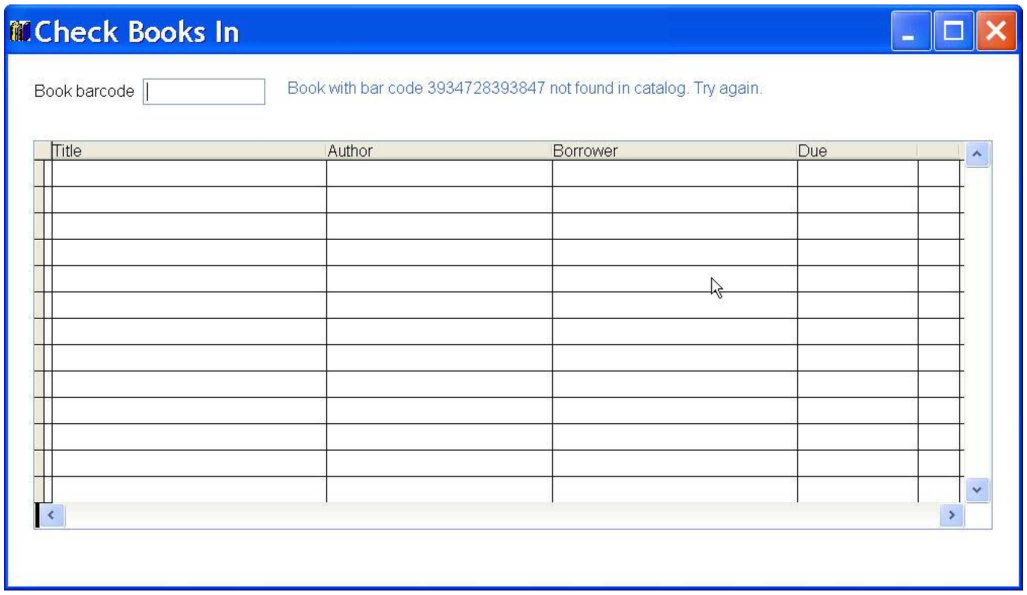


```
LPARAMETERS cString, cRemove

LOCAL nPos, cOutString

cOutString = cString

* Remove trailing characters
nPos = LEN(cOutString)
DO WHILE nPos >= 1 AND SUBSTR(cOutString, nPos, 1) $ cRemove
    nPos = nPos - 1
ENDDO

cOutString = LEFT(cOutString, nPos)

* Remove leading characters
nPos = 1
DO WHILE nPos <= LEN(cOutString) LEFT(cOutString, nPos) $ cRemc
    nPos = nPos + 1
```

Figure 19. Visual FoxPro indicates that code in an editing window has changed by adding an asterisk to the title bar. When you save, the asterisk disappears. This feedback avoids the need for an alert on saving the code.

Other applications use other techniques to avoid alerts. In Microsoft Word, when you save a file, a disk icon briefly appears in the status bar. Most web applications use an asterisk or other character to indicate required fields and then highlight missing items when the user attempts to save, rather than displaying a separate alert.

The Library application puts messages directly on forms rather than using alerts to tell the user about problems or provide additional information. **Figure 20** shows the library application's form for checking in books; when it receives an invalid barcode, it shows a message right on the form, rather than using a messagebox.

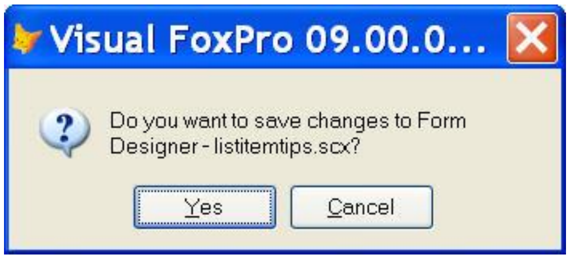


Book with bar code 3934728393847 not found in catalog. Try again.

Figure 20. If the barcode received isn't in the catalog, the library application's check-in form puts a message right on the form instead of popping up a messagebox.

Best Practice: Use techniques other than alerts to keep the user informed.

Confirmations seem like a good idea at first. After all, making sure the user wants to delete a record or leave a form without saving his changes seems like common courtesy. In fact, it's annoying. The user told your application what he wanted to do and you're second-guessing him. For example, by default, VFP prompts you to save changes when you run a form from the Form Designer (**Figure 21**); so many developers are annoyed by this that there's a checkbox to eliminate it in the Options dialog.



Do you want to save changes to Form Designer - listitemtips.scx?

Figure 21. This confirmation dialog appears when you click the Run button in the Form Designer. You can turn it off in the Options dialog.

More importantly, if an application always shows a confirmation in the same place, users answer it automatically. For example, Windows Explorer always asks for confirmation of file deletion; do you even look at that dialog or do you just hit Enter (for OK) automatically?

Reserve confirmation dialogs for situations where a mistake is dangerous and irreversible. For example, in VFP, the DELETE command (which marks a record for deletion-an easily reversible action) doesn't use a confirmation dialog, but the ZAP command (which destroys records permanently) does. Of course, omitting confirmation dialogs requires you to provide a comprehensive Undo facility.

Best Practice: Use confirmation dialogs only for devastating, irreversible actions.

Dialogs are modal

Although you can build modeless dialogs (like the Find dialog in most applications—VFP's version is shown in **Figure 22**), they're confusing. They don't follow the rules users have come to understand about the differences between documents and dialogs. Finishing your work with such a dialog is confusing, too; often, you have to click Cancel when what you really mean is "I'm done."

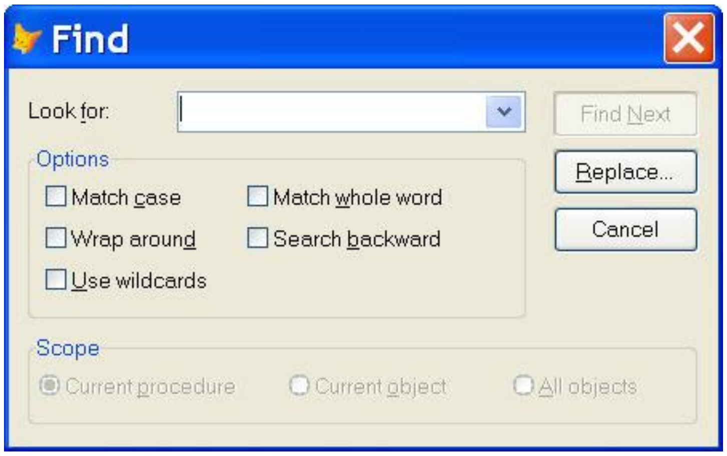


Figure 22. In most applications, the Find dialog is modeless, but how do you tell it when you're done searching? You have to click Cancel or the window's Close button.

Rather than puzzling users with a modeless dialog, when you need to show a set of controls on an ongoing basis while the user performs a task, use a toolbar or task pane instead.

Toolbars (see "Use toolbars as mouse shortcuts" earlier in this document) have been around for quite a while and most users know how to work with them. They're most useful in cases where the operations users need to perform mostly don't involve text entry and the controls involved don't require a lot of space. Word's mail merge toolbar (**Figure 23**) is a good example of this case.

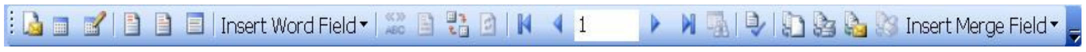


Figure 23. Word's mail merge toolbar incorporates all the controls related to creating and modifying mail merges, along with the controls to actually perform the merge into a single unobtrusive pane.

Of course, if a user finds the toolbar hard to work with, she can always convert it to a floating panel, like a modeless dialog, by undocking it, as in **Figure 24**.

FoxPro has offered the ability to create custom toolbars since VFP 3. Use the Class Designer to build them. Use SYS(2015) to give a toolbar control the same functionality as an item from the built-in menu. It's also possible to set up toolbar controls to enable and disable automatically, as menu items do; see my article in the July, 1997 FoxPro Advisor. The downloads for this session include a toolbar class that ties enabling and disabling of controls to their corresponding menu items.

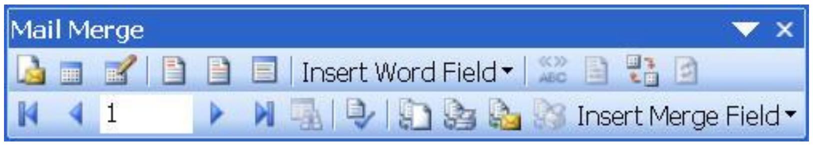


Figure 24. Toolbars can be undocked to provide a floating panel if the user prefers.

Task Panes are a newer entry in the user interface marketplace. They're similar to toolbars, but tend to be tall rather than wide. Typically, task panes get docked at the sides of the workspace, while toolbars are more likely to be docked at the top. Browsers may have been the first to introduce task panes (though theirs generally can't be undocked), using them for bookmarks and history listings. Recent versions of Office use them extensively for tasks that used to involve modal dialogs. **Figure 25** shows Word 2003's Styles and Formatting pane.

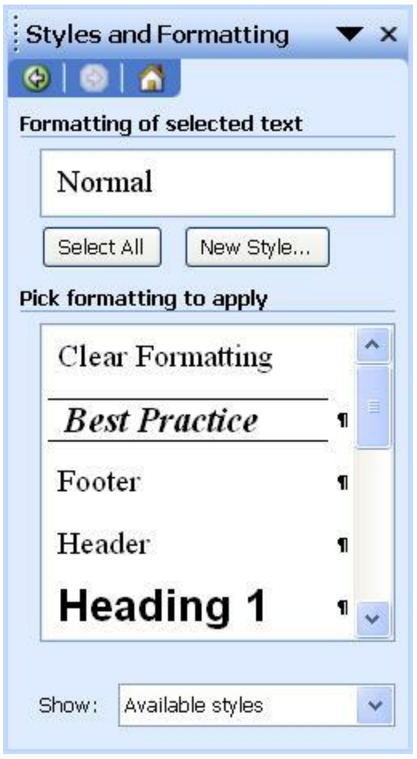


Figure 25. Task panes are a lot like toolbars, but better suited to larger quantities of text.

You can build task panes in VFP 9, using the ability to dock forms. However, forms can only be docked to the main VFP window, not to other top-level forms. The Library application demonstrates an alternative to docking for task panes; buttons to collapse and expand the window. The session materials contain a task pane form class and a form derived from it. **Figure 26** shows the expanded task pane, while **Figure 27** shows the form when collapsed.

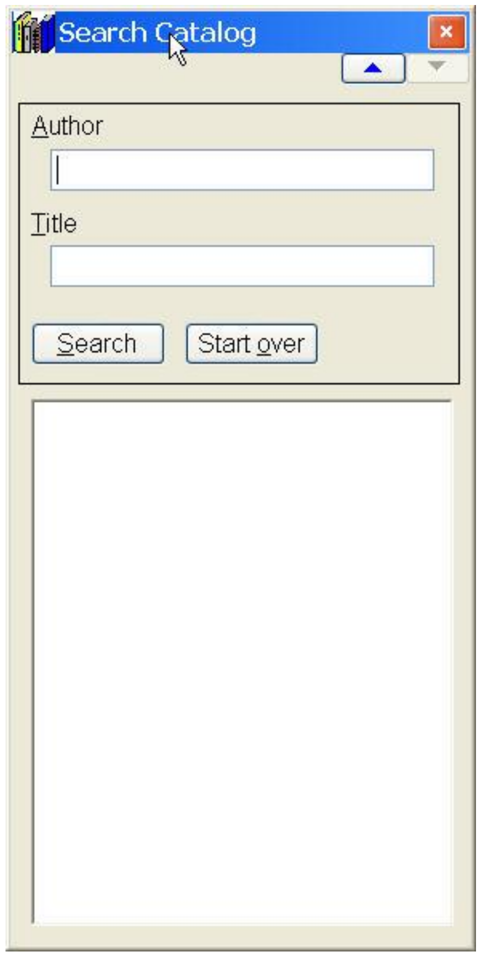


Figure 26. This form is a task pane. The buttons at the top allow it to be collapsed and expanded.

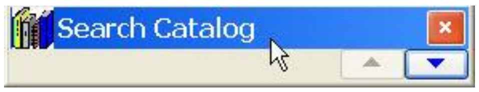


Figure 27. The collapsed task pane is easy to put out of the way.

Facilitate the most common action

In some applications, when you open a form, it shows a record that can be edited immediately. For example, when you double-click a Contact in Outlook, the form that appears is live—you can start making changes immediately.

In other applications, forms open in display mode and you have to click an Edit button to make changes. The Contacts application on Pocket PCs displays this behavior.

In yet another group of applications, forms open live, but empty. For example, when you open Microsoft Word, you're presented with a new, blank document and you can begin typing right away.

Which of these three approaches is best? It depends. The decision whether to open forms in edit mode or display mode, and whether to open on an existing record or a new record is based on the nature of the application and the users. To determine the best choice for a given application, consider the following questions:

- Are users more likely to add records or edit existing records? If the most common case is adding new records (as, for example, in a point-of-sale application), open forms ready for entry of new records. Do not require the user to click New as soon as he opens the form. If changing existing data is more common than adding data, open the form looking at existing records.
- Is there any way to know which record a user wants to edit? If so, open the form looking at that record. If not, then opening on a blank record may be a good choice.
- Are existing records more likely to be viewed or changed? If users work mostly with existing data, but look things up far more often than change it, open in display mode. If the more common action is editing, then prepare the form for editing when it opens.
- How much harm would accidental changes to the data cause? If the data is sensitive and even minor errors have the potential to do serious damage, open forms in display mode to avoid accidental changes.

What are the differences between the different modes? In display mode, data is shown, but making changes requires an action from the user—clicking an Edit button, for example. In edit mode, as soon as a record is displayed, the user can make changes. When editing is the normal action and errors aren't terribly risky, there's no reason to require an extra click or keystroke from the user on each record.

When a form opens on an existing record, adding a new record requires user action, such as clicking a New or Add button. When a form opens on a new record, editing an existing record requires a user action. Clearly, it's best to be prepared for what users do most often.

It may be tempting to make different choices for different forms or different users in an application. To some extent, that's alright. For example, opening some forms on existing records and others on new records isn't likely to confuse users, as long as the choices reflect what users actually need to do. However, opening some forms in display mode and others in edit mode probably will confuse users, unless you find a way to make the distinctions extremely clear.

It is possible to handle forms differently for different classes of users, for example, opening forms in edit mode on a new record for data entry clerks and in display mode on an existing record for managers. (There's no reason, ever, to open a form in display mode on a new record.) The difficult part of this approach is writing maintainable code that can handle the various cases.

Saving data

The flip side to how forms open is how records are saved. Most applications expect users to actively save their data by clicking a button or choosing a menu item (explicit save). For example, in Microsoft Word, you click the Save button or choose File | Save from the menu to save your work. A few applications save automatically and make abandoning an edited record the exception (automatic save). For example, in Outlook's Tasks view, you can click into a textbox to create a new task (**Figure 28**); when you navigate off that task, it's automatically saved into the list.

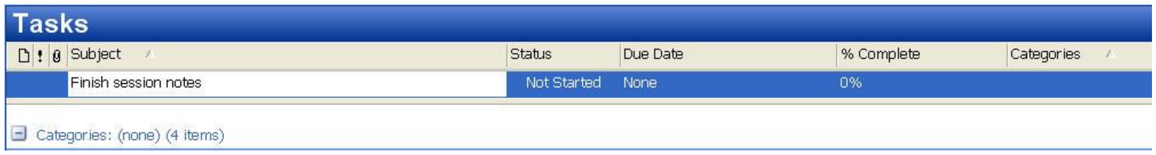


Figure 28. Outlook's Tasks view lets you add new items without explicitly saving them. Just create the item and navigate away from it.

Deciding which approach to take is tricky. In the physical world, people don't have to do anything to save data; you write on a piece of paper and it's there. To not save it, you have to throw the page away.

Software developed the explicit save model because storage space was expensive and using it wisely was more important than making things easy for users. Today, though, storage is cheap and the only harm in saving everything is difficulty in finding what you need. So the automatic save approach makes sense.

But, most users have years of experience with the explicit save model, and may find an application without a Save button confusing. One thing is clear; if you use an automatic save approach, you need to provide a comprehensive Undo facility that lets users undo mistakes.

When you use explicit save, users complete their editing and move on by clicking a Save button or choosing Save from the menu. If the application opens forms in display mode, the record returns to display mode. If the application opens forms with a new record, after saving, another new record should be presented; do not ask users to click Save followed by New in this situation. For edit mode forms that open on existing data, nothing changes; in this situation, it's a good idea to provide some visual feedback to distinguish records with outstanding changes from unchanged records.

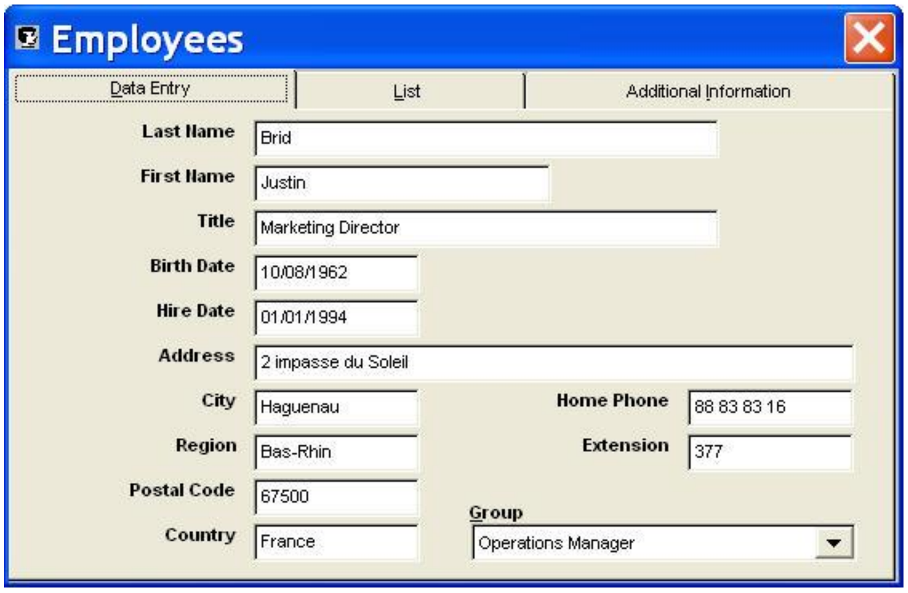
With automatic saves, data is saved when the user navigates to another record or closes the form. (The behavior is analogous to VFP's row-buffering.) No Save button or menu item is needed, but a New button is required to save the current record and add a new one. Automatic save is incompatible with forms that open in display mode.

The CheckOut form of the Library application (Figure 8) demonstrates a compromise between the two approaches. To complete this check-out and begin another, the user must choose Save, but closing the form automatically saves the data displayed.

Finding data

The third piece of basic form behavior is finding the data a user is interested in. In some cases, that information is available when a form opens, so the user doesn't have to do anything. Often, especially with forms that open in display mode, though, the user needs a way to get to the record he's interested in.

Many VFP applications use a pageframe in this situation, putting a grid on one page for the user to find the record of interest and then putting the data from a single record on the other pages. No doubt many developers use this approach because the sample TasTrade application that came with VFP through version 7 uses it; **Figure 29** shows the Employees form from TasTrade.



Last Name	Brid
First Name	Justin
Title	Marketing Director
Birth Date	10/08/1962
Hire Date	01/01/1994
Address	2 impasse du Soleil
City	Haguenau
Region	Bas-Rhin
Postal Code	67500
Country	France
Home Phone	88 83 83 16
Extension	377
Group	Operations Manager

Figure 29. The forms in the sample TasTrade application show a list of records on one page and the record data on the other pages. This is not a good way to help users find records.

While such a form is fairly easy to create, it's not an easy interface to use. There are better ways to give users access to their data. In addition, as the Employees form demonstrates, figuring out the best order for the pages isn't easy. (Why is the List page in between the two pages that contain the data for a single record?)

To determine the best approach, you need to consider the number of records involved, what data users are likely to already know, and whether you're using native VFP data or data stored on a SQL server.

When a server is involved, bringing large quantities of the data to the user for browsing or local searching is likely to be too slow. (This is one of the few ways that hardware and configuration issues still need to be considered in interface design.) In this situation, you need to get as much information as the user can offer and then let the user select among the matching records.

When the data set is small and local, a listbox or a grid configured to look like a listbox may be a good choice. However, putting the list on the same page as the data provides more fluid navigation than using separate pages.

A similar option is the ActiveX TreeView control; this is a particularly good choice when dealing with hierarchical data, where a user may need to navigate through a set of categories and then to a record within a category. Again, it's best to put the TreeView on the page with the detail data.

Yet another variation on providing a list is to make the items in the list links to the actual data. When the user clicks an item, the edit form for that item appears. This approach offers the space advantages of the pageframe, but is easier to navigate. The Contacts application on Pocket PCs uses this approach, as do most reports in Quicken.

When you use a list of any sort, you can make the user's task easier by offering quick navigation of the list with an alphabet. For example, in Intuit's QuickBooks Online Edition, various lists offer alphabet and number buttons, as in **Figure 30**; click a button to jump to that letter of the alphabet in the list. This approach can be used in client-server situations, as well, to limit the data loaded from the server.

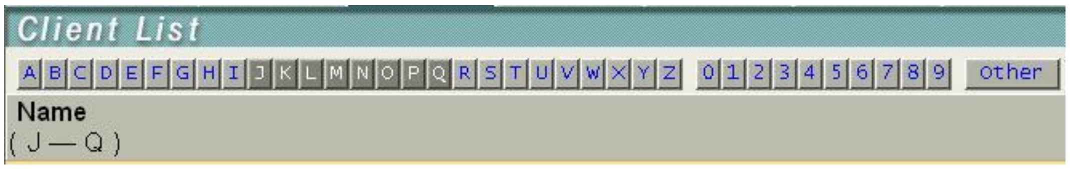

Client List
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 other
Name
(J — Q)

Figure 30. You can navigate the list of clients in QuickBooks Online Edition by clicking a letter of the alphabet. The darker letters show the portion of the alphabet currently displayed. (The actual client data here is omitted for privacy reasons.)

Intuit offers another alternative to asking users to scroll through a list, one they call QuickFill. The user starts typing and the application shows the first item that matches what the user has typed so far. As the user continues to type, the selection narrows down until either a match is found or it becomes apparent that the item is new. **Figure 31** shows an example.

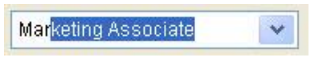


Figure 31. With Intuit's QuickFill, you start typing and the textbox portion of the combo is filled with the first matching item. The highlight shows the part that's being supplied by QuickFill.

QuickFill can be used with either a combobox or a textbox. Classes to provide QuickFill combos in your VFP applications are included in the session materials.

Controls

Once you've made your choices about overall form operation, you can design the actual forms for your application. There are a number of considerations in choosing and implementing controls.

Use the right control for the job

Modern applications have a variety of controls available; using the appropriate control for the task at hand makes things easier for your users. While some choices are simple (for example, using an editbox for unlimited text), some situations are a little trickier.

When the user needs to choose between two options or items, you can use a set of option buttons, a checkbox, or a dropdown list. Which is appropriate?

Use a checkbox only for items where a yes or no, on or off setting makes sense. So, for example, a checkbox that says "Case-sensitive" is clear. But a checkbox that says "Male," while technically correct, is confusing at best and insulting at worst.

Use a dropdown list if the number of items to choose from can change based on other considerations. The number of items in a dropdown list can be modified without redesigning the form and often without changing any code, but adding items to an option group calls for rearrangement on the form and a change to the source code.

Use an option group for mutually exclusive choices that won't change. For example, an option group makes sense for indicating sex, using buttons labeled "Male" and "Female."

Figure 32 shows three situations involving choosing between two options. In each case, a different control offers the best interface. In the first, where the goal is to toggle a reminder on or off, a checkbox is space-efficient and clear. In the second group, maleness isn't something that's toggled on and off, so the checkbox is confusing. With only two clear choices, the option group is effective; however, in situations where you may not know the answer, an enhanced combobox offering "unknown" may be more appropriate. For the final group in the form, the checkbox is clearly inadequate. While the option buttons serve the purpose now, if you later need to add "Fall" and "Spring" as choices, the form may have to be redesigned and code modified. In this case, the combobox provides a better long-term solution.

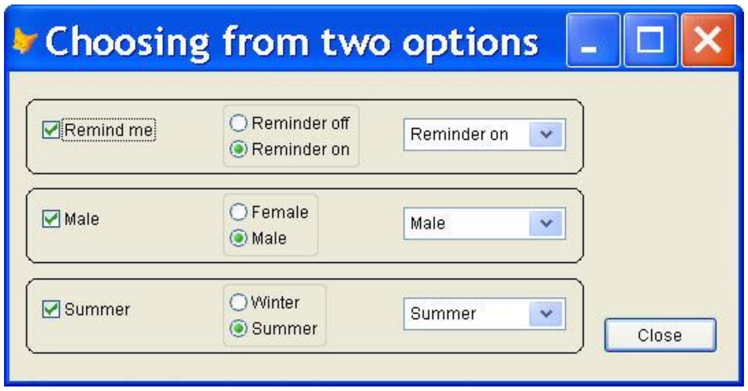


Figure 32. Helping a user choose between two options doesn't always call for the same control. Consider the meaning of the data and whether it's likely to always involve two choices when choosing between a checkbox, an option group and a combobox.

A related question is when to use option buttons and when to use a set of checkboxes. The answer here is straightforward. Use option buttons when the choices are mutually exclusive, that is, when only one item can be chosen. Use a set of checkboxes when multiple choices are permitted.

In some situations, you may be inclined to change the caption of a checkbox when the user checks it. Don't do it. The state of a checkbox indicates whether or not the statement next to it is true. Changing the caption leads to confusion.

Never use a single option button. To specify whether a setting is on or off, use a checkbox. A corollary to this rule is that there should always be a selected button in an option group.

Use buttons or double-clicks on listboxes (or grids set up to mimic listboxes) to trigger actions. While things (such as setting properties) may happen behind the scenes when other controls are used, users don't expect checking a checkbox or choosing an option button to open a form or run a report or delete a record. Controls that do so are confusing.

There is an exception to this rule. A graphical checkbox (especially in a toolbar) may be used to toggle the display of a tool. For example, VFP's standard toolbar includes graphical checkboxes for a number of tools, including the Command Window, the Data Session window and the Document View window. Figure 33 shows the toolbar when the Command Window and Data Session window are displayed. In the Library application, the Search Catalog button on the main toolbar is actually a checkbox; clicking it toggles the Search Catalog task pane on and off.

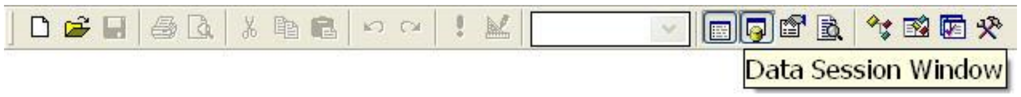


Figure 33. VFP's standard toolbar uses graphical checkboxes to toggle the status of several common tools, including the Data Session window.

Don’t overload comboboxes or listboxes. Combos are best used for relatively small lists. Remember that a user has to open the combo and scroll through the list to find the right item, so a combo with hundreds or thousands of items is very hard to work with. Because listboxes are always “open,” they can handle somewhat larger lists than combos, but thousands of entries are still too many. When using longer lists of items with these controls, make sure that incremental searching is turned on; you may want to implement QuickFill for combos, as well, as in Figure 31. For longer lists, consider the ActiveX TreeView control, which allows you to group data hierarchically.

Use textboxes for limited data entry; use editboxes (or the ActiveX RTF control) for unlimited data entry.

Think at least twice before you use a grid for data entry. It’s often better to use a read-only grid to display data, and give users a separate area with individual controls to enter new data or edit existing data. Users expect data-entry grids to behave like spreadsheets, but VFP’s grid isn’t a spreadsheet control, and may surprise both you and your users with its various behaviors.

Best Practice: Use the right control for the task at hand.

Use graphical buttons in toolbars

While buttons on forms can be either textual or graphical, buttons on toolbars should almost always be graphical. Text buttons on toolbars are generally too hard to read. Give each control on a toolbar a tooltip. Remember that tooltips should be brief; one word is best, but never use more than three words.

Keep tooltips to tools

Tooltips are intended to help users understand graphical controls; specifically, they’re meant to decipher icons. Don’t put tooltips on document forms. Use them only in toolbars and on actual tools, where you’re likely to have graphical controls. (Don’t confuse tooltips, which are used for controls, with itemtips, which provide expanded display of items in a list.)

Labels provide the first approach to making documents clear; in almost every case, the label for a control or its caption should give the user enough information to use the control. If you need to provide additional information, you have several options:

- Use the StatusBarText property to display clarification in the Status bar.
- Provide a “help” area on the form and change its text as focus moves.
- Provide context-sensitive help. (This is a good idea regardless of whether you implement one of the other approaches.)

The principle goal of each technique is to give the user additional information without cluttering up the screen.

Give the user access only to options that are available

There's not much more frustrating to a user than to choose an action only to be told that the action is unavailable. For example, in InnoSetup, a freeware application for building setup packages, you can test installation and uninstallation from within the development environment. The toolbar contains a graphical option group that toggles between installation and uninstallation and a separate button to run the chosen task; the relevant portion of the toolbar is shown in **Figure 34**. However, if you attempt to uninstall before installing, you get the error message shown in **Figure 35**. While installing before uninstalling seems obvious, when you're working on building a setup package, it's easy to forget to click the option button before clicking Run. It would be better if the Run button were disabled when it's not available or if the dialog that appears offered the option of installing instead.

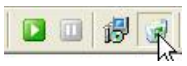
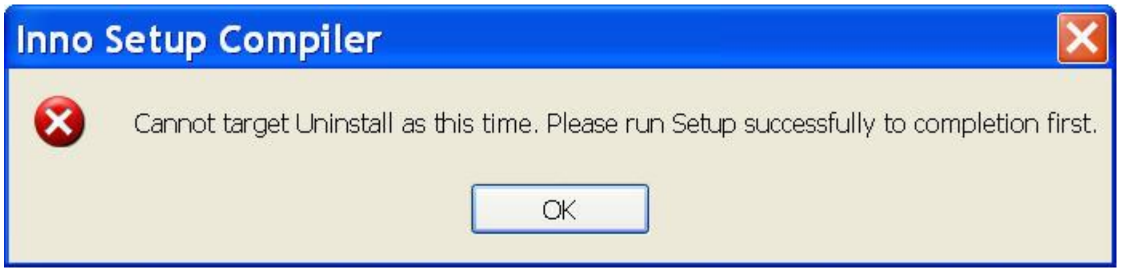


Figure 34. InnoSetup's toolbar uses option buttons to determine whether clicking the Run button attempts to install or uninstall the setup you're building.

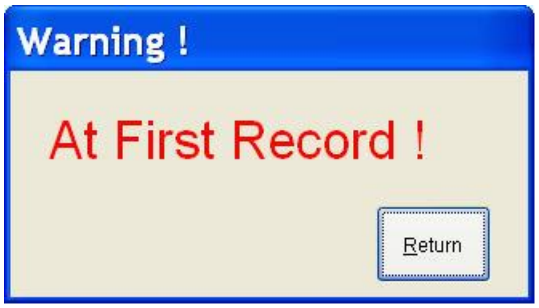


Inno Setup Compiler

Cannot target Uninstall as this time. Please run Setup successfully to completion first.

Figure 35. When you try to uninstall using InnoSetup's toolbar before installing, you get this error.

Properly enabling and disabling controls gives a user valuable feedback. For example, disabling a Next button when you're looking at the last record tells the user that you're at the end. One vertical market application I've seen fails to disable navigation buttons; when the user clicks Top or Previous on the first record, the warning in **Figure 36** appears.



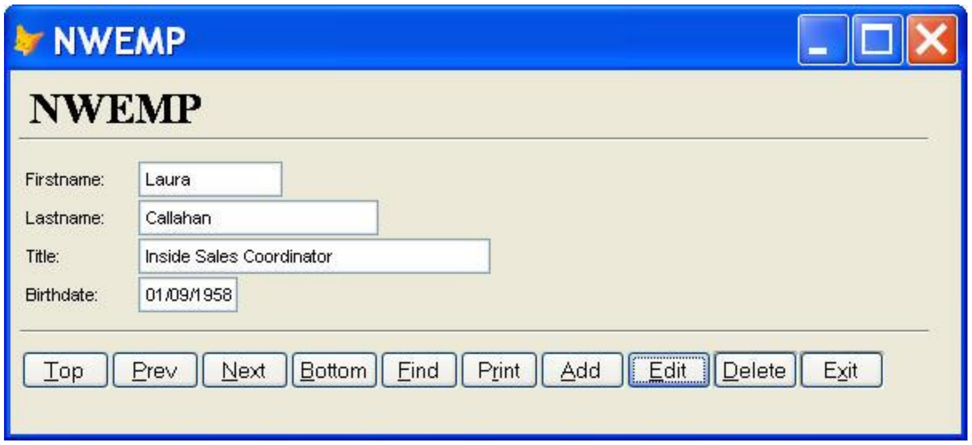
Warning !

At First Record !

Figure 36. In one vertical market application, this message appears when you click Previous or Top and you're already on the first record. Why aren't those buttons disabled in that situation?

Best Practice: If the user isn't permitted to use a particular control, disable it.

Some applications change the meaning of buttons based on the situation. For example, in forms built with the VFP Form Wizard, when you click the Edit button, the Add and Edit buttons change to Save and Revert. **Figure 37** shows the two states of the form. In this case, the change is fairly apparent, but often, such a change is confusing. (In fact, when you choose graphical buttons rather than text buttons in the wizard, the change of caption seems harder to deal with.)



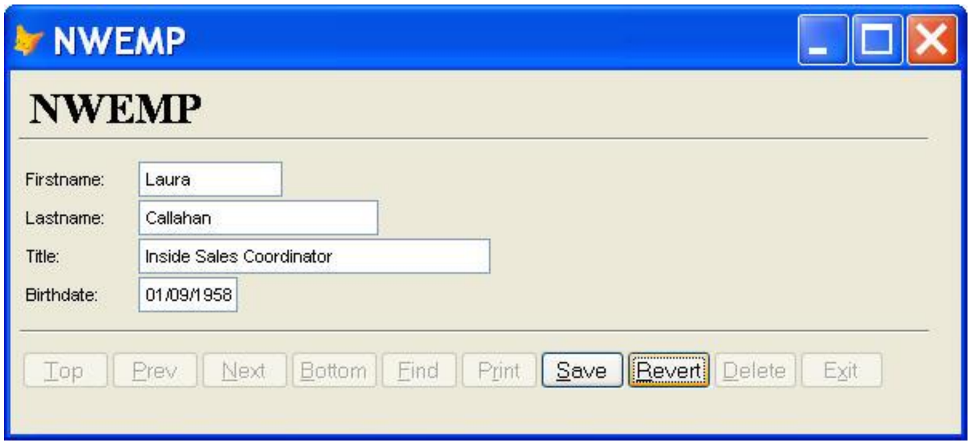


Figure 37. VFP's Form Wizard creates forms that change the captions of buttons based on the editing status.

In most situations, changing the caption and behavior of a control while the form is running is a bad idea. Users don't always notice the change. However, configuring a form when it opens to contain the appropriate controls for the situation is reasonable.

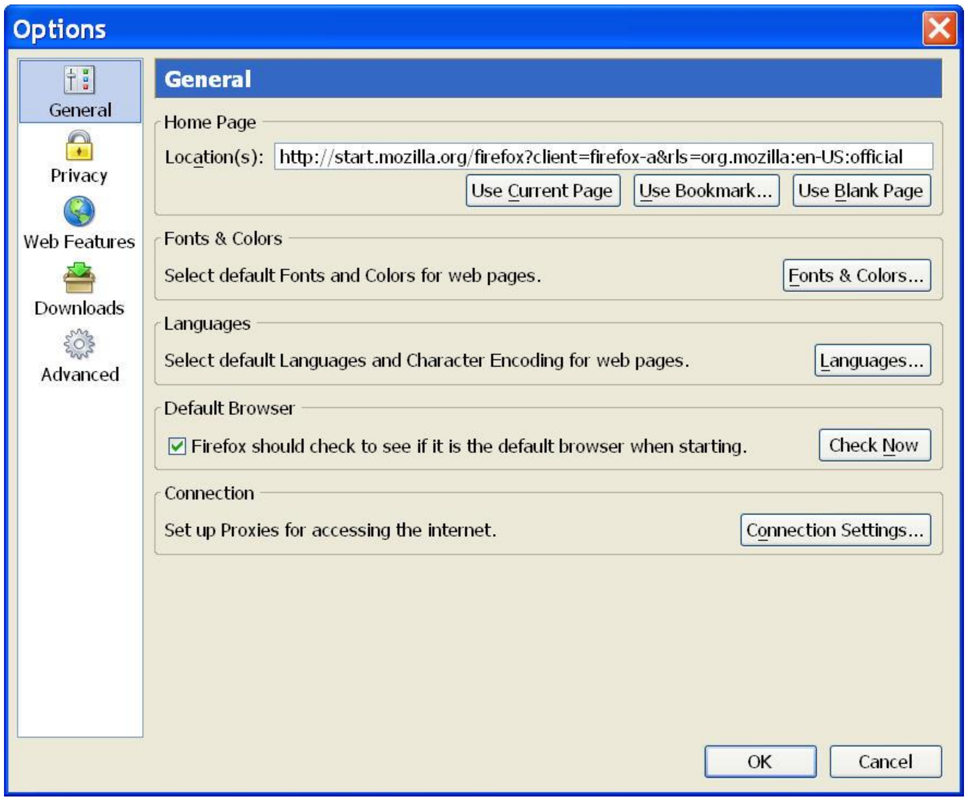
Another alternative to enabling and disabling controls is to have some controls appear or disappear based on the situation. One area where this approach is quite valuable is security; it's better for users not even to be aware of controls that provide access to data or portions of the application for which they don't have appropriate rights. Removing such controls from forms or making them invisible is a good choice. (Similarly, removing menu items for which the user doesn't have rights is a good idea; use GenMenuX.)

However, hiding controls that are simply not available due to application status is generally a bad idea. It leads to confusion; "I'm sure that checkbox was there yesterday." Disabling the control lets the user understand that the current situation makes it unavailable.

Make capitalization uniform

While it probably won't affect the usability of your application, random use of capital letters will affect user's perceptions of it. Establish a set of rules and follow them throughout. Capitalization is an issue for:

- Form captions—Most applications capitalize each word in form (and application) captions, treating them like titles in a book.
- Page captions—There's some variation among software companies as to whether to capitalize each word on a tab in a tabbed dialog. Microsoft capitalizes only the first word; other companies capitalize every word. Choose one practice and stick to it.
- Pseudo-page captions—Instead of a tabbed dialog, some applications combine a pane of "pages" with a pane showing the content from a single page. **Figure 38** shows the Options dialog from FireFox, which takes this approach. As with tabs, choose a capitalization rule for the name of each page and follow it throughout.



http://start.mozilla.org/firefox?client=firefox-a&rls=org.mozilla:en-US:official

Select default Fonts and Colors for web pages.

Select default Languages and Character Encoding for web pages.

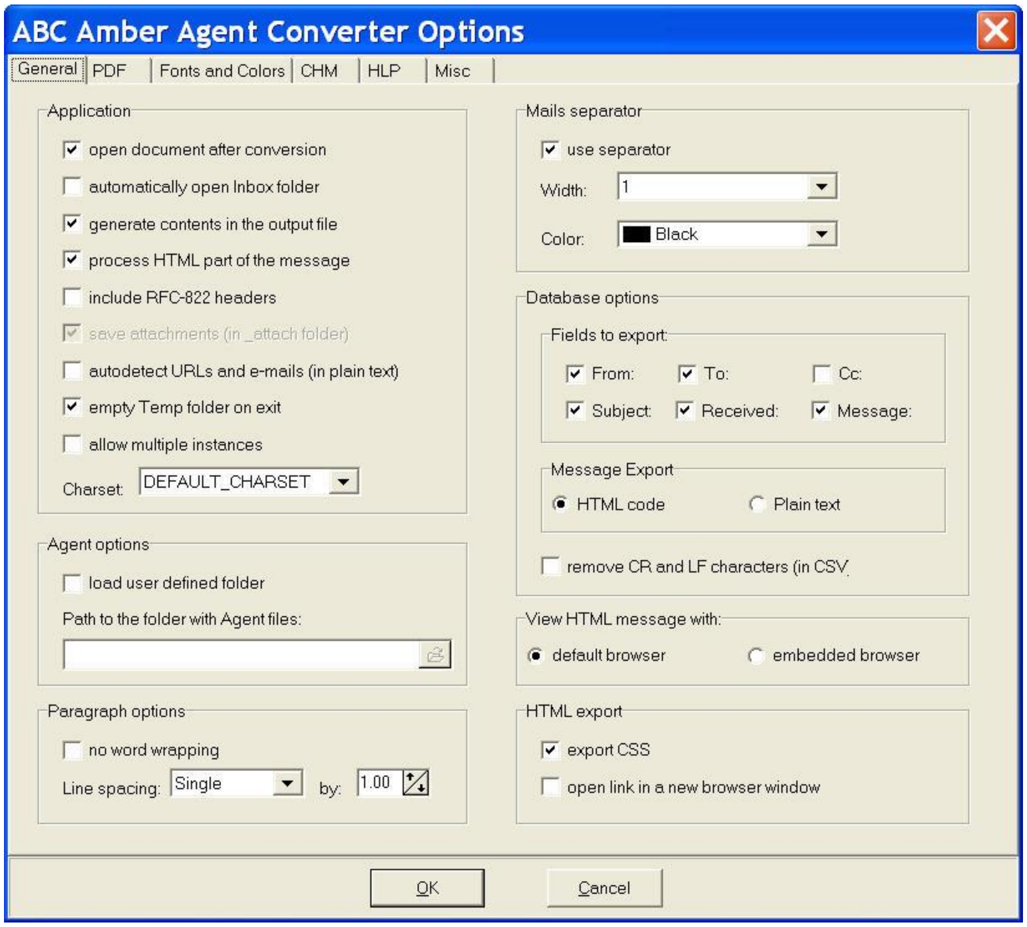
☑ Firefox should check to see if it is the default browser when starting.

Set up Proxies for accessing the internet.

Figure 38. The FireFox Options dialog uses two panes with the selection in the left pane determining the content of the right pane. The name of each pane uses “title case,” with each word capitalized.

- Control captions—Normally, the first word on a control is capitalized while others are not.
- Messages—Messages should use the normal capitalization rules for the underlying language.
- Tooltips—Microsoft uses “title case” for tooltips, capitalizing each word except for small helper words like “and.” Other vendors capitalize only the first word. Again, make a choice and stick to it.

Failing to follow capitalization rules can make forms look unattractive and unprofessional. The dialog in **Figure 39** starts some items with upper-case letters and others with lower-case letters; there appears to be no pattern.



ABC Amber Agent Converter Options

Application
- ☑ open document after conversion
- ☐ automatically open Inbox folder
- ☑ generate contents in the output file
- ☑ process HTML part of the message
- ☐ include RFC-822 headers
- ☑ save attachments (in _attach folder)
- ☐ autodetect URLs and e-mails (in plain text)
- ☑ empty Temp folder on exit
- ☐ allow multiple instances
- Charset: DEFAULT_CHARSET

Agent options
- ☐ load user defined folder
- Path to the folder with Agent files:

Paragraph options
- ☐ no word wrapping
- Line spacing: Single by: 1.00

Mails separator
- ☑ use separator
- Width: 1
- Color: ■ Black

Database options

Fields to export:
- ☑ From: ☑ To: ☐ Cc:
- ☑ Subject: ☑ Received: ☑ Message:

Message Export
- ● HTML code ○ Plain text

- ☐ remove CR and LF characters (in CSV

View HTML message with:
- ● default browser ○ embedded browser

HTML export
- ☑ export CSS
- ☐ open link in a new browser window

Figure 39. When capitalization is irregular, a form looks unfinished.

As you make your choice for each case, document it so that all members of your team and those who later maintain the application will know the rules.

Supply reasonable defaults

Most users do pretty much the same thing time after time. Design your application to assume that whatever that is, he's doing it again and provide default settings for controls that make it easy to do it again.

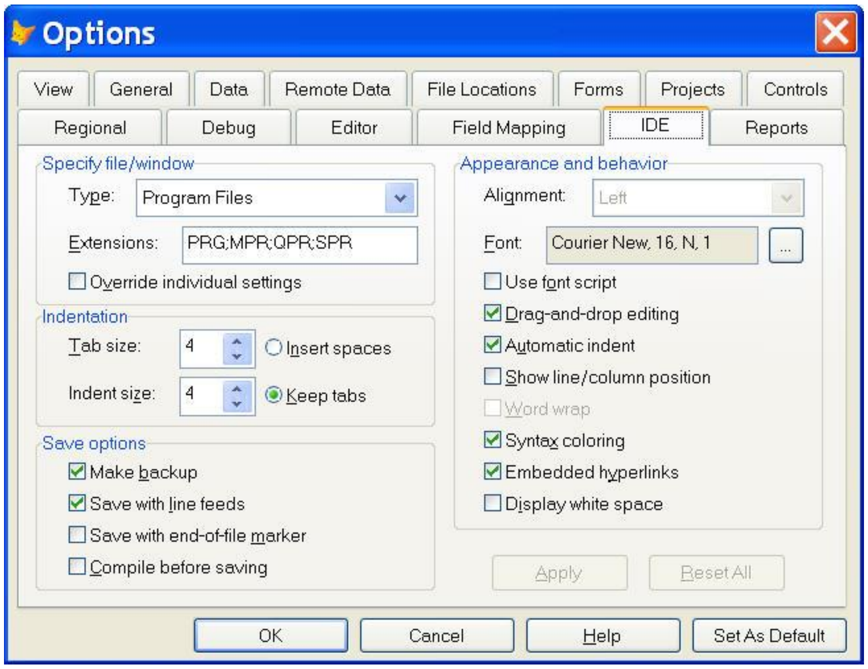
Supplying defaults can mean filling in certain fields. For example, in a medical office application, chances are that most patients live in the same state as the office; for a new record, you can pre-fill the state field. In this situation, it's easy for the user to get rid of the default and, if it's right, it's one less thing to type.

Defaults also apply to actions. For a dialog, figure out which button is most likely to be the user's choice and make it the default. For example, most Options dialogs have OK set as the default, so that simply pressing Enter exits the dialog, saving changes. In VFP, the Default property of command buttons lets you establish a default button.

Defaults are also related to remembering user's choices. Where appropriate, use a stored setting as a default.

Group related controls visually

You can help users understand what they need to do better by using lines and boxes (shapes) to show groups of related controls. When you use boxes, give each group a caption (using a label) describing that group's function. **Figure 40** shows the IDE page of VFP's Options dialog. Although there are more than 20 controls on this page, the grouping boxes make it easy to see which are related. **Figure 41** shows VFP's Import dialog; it uses a line to separate the controls for specifying the import source from those for the destination. In this dialog, grouping boxes would probably be overkill, but the line makes it easy to see which group is which.



Specify file/window
Type: Program Files
Extensions: PRG;MPR;QPR;SPR
☐ Override individual settings

Indentation
Tab size: 4
Indent size: 4
○ Insert spaces
● Keep tabs

Save options
☑ Make backup
☑ Save with line feeds
☐ Save with end-of-file marker
☐ Compile before saving

Appearance and behavior
Alignment: Left
Font: Courier New, 16, N, 1
☐ Use font script
☑ Drag-and-drop editing
☑ Automatic indent
☐ Show line/column position
☐ Word wrap
☑ Syntax coloring
☑ Embedded hyperlinks
☐ Display white space

Figure 40. VFP's Options dialog groups controls on each page using shapes. The caption for each group explains its role.

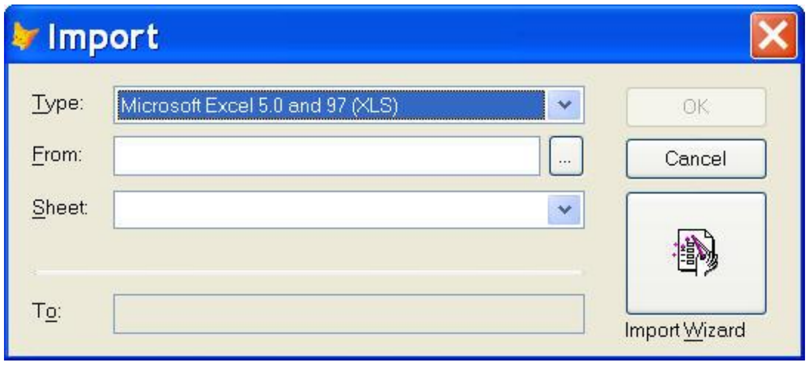


Figure 41.VFP's Import dialog uses a line to separate the input portion of the task from the output portion.

Set navigation order correctly

Some users will navigate your forms entirely with the keyboard. Those users must be able to press tab and go through the form in a logical order. The correct order varies from form to form, but for most forms, it's either across each row and then down to the next row, or down one column, then down the next column, and so forth. When controls are grouped visually, the tab order should go through an entire group before moving on to the next.

Include labels when setting tab order. There are two reasons to do so. One is illustrated in Figure 41 above. You can assign a hot key to a label, even though it can't get focus. When the user presses that key combination, focus lands on the next control after the label that can accept focus. So in Figure 41, pressing Alt+F sets focus on the textbox next to the From caption.

In addition, some screen reader tools used by people with visual disabilities use the tab order of forms to determine what to read in some cases. Such tools look for a label immediately before a textbox, editbox or spinner.

Better applications

Just as you generally look better when you plan your outfit and try the parts together before leaving the house, designing your application's interaction and user interface before writing code results in a better, more usable application. Considering the user's tasks and goals lets you design an application that will make sense to the intended users. Following established standards except when you're making revolutionary change will help users learn your application.

The downloads for this session include the examples presented here; in particular, the Library application (a work in progress) is included.

Resources

Much has been written on the topic of user interaction and user interfaces. If you want to learn more, the list here should get you started.

Books

Cooper, Alan. *About Face: The Essentials of User Interface Design*, IDG Books WorldWide, 1995. ISBN 1-56884-322-4.

Cooper, Alan. *The Inmates are Running the Asylum*, SAMS, 2004. ISBN 0-752-32614-0.

Johnson, Jeff. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kauffman, 2000. ISBN 1-55806-582-7.

Norman, Donald. *The Design of Everyday Things*, Basic Books, 1988. ISBN 0-465-06710-7.

Raskin, Jef. *The Humane Interface: New Directions for Designing Interactive Systems*, Addison-Wesley, 2000. ISBN 0-201-37937-6.

Websites covering UI design

http://www.asktog.com/

http://www.foruse.com

http://www.jnd.org/dn.pubs.html

http://www.useit.com/alertbox/

Websites covering UI patterns

http://www.cs.helsinki.fi/u/salaakso/patterns/

http://time-tripper.com/uipatterns/Introduction

Websites offering UI examples, good and bad

http://homepage.mac.com/bradster/iarchitect/shame.htm

http://homepage.mac.com/bradster/iarchitect/fame.htm

http://www.frankmahler.de/mshame/

http://www.rha.com/ui_hall_of_shame.htm

http://www.userinterfacehallofshame.com/

http://www.webpagesthatsuck.com/