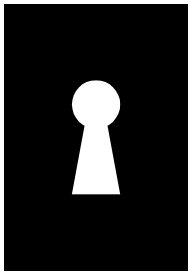

A Note to Reviewers

Thanks for agreeing to read and offer feedback on this draft material from *The Keyhole Problem*. I am interested in comments of every kind, but my primary concerns are (1) technical accuracy and completeness, (2) establishment of compelling arguments that back my case, and (3) a wide variety of clear, convincing, current examples.

I welcome suggestions for examples I should use in addition to or instead of the examples I already have. I am especially interested in examples that come from major companies (a *faux pas* by a major company is more interesting than one from a bit player), that are from companies I don't already include in the book (I want to spread the shame around), that are from more recent versions of the software I'm showing (current examples are better than old ones), and that demonstrate aspects of the problem I haven't considered. Again, however, I am interested in more than just examples. If my arguments are incomplete, misleading, incorrect, or unconvincing, I need to know about it.

There are a couple of conventions you should know about:

- Lines with change bars to the left (such as in this paragraph) indicate places where I have made modifications since the last posted version of the material. Unfortunately, the word processing software I use (FrameMaker) ignores graphics when generating change bars, and that leads to two problems. First, no change bars appear if a graphic is modified, but none of the text around it is. More significantly, if the text immediately preceding a graphic is changed and the text immediately following that graphic is also changed, the change bar will span the range between the two lines of text, thus suggesting that any intervening graphics have changed, even if they haven't.
- *Italicized paragraphs with red question marks in the margin (such as this one) are notes (typically questions) to reviewers and will not be part of the book. These are places where I am particularly looking for your input.*

?

Listbox Keyholes

A Listbox Keyhole arises when a listbox or a combobox (both are types of drop-down menu) refuses to display all its menu choices at once. Figure ??-1 shows an example of each type of control. Note how both examples include a vertical scroll bar. Each list has more choices available, but the only way to see them is to scroll. That's the essence of Listbox Keyhole: you are prevented from seeing all the list choices at once.

The difference between a listbox and a combobox is that a listbox requires that you select one of the entries it presents, while a combobox also allows you to type in an entry of your own choosing. As such, a combobox is a combination of a listbox and an edit control (hence its name), and it is subject to the keyhole pitfalls of both. I discuss its listbox-related keyhole pitfalls here, and I address edit control keyholes in Chapters [FSEC] and [LFSEC]. In this chapter, I refer only to listboxes, but my observations are equally valid for comboboxes.

Listbox Keyholes may well be encountered more frequently than any other keyhole type. They're widely imposed in both native and web applications, and the vast majority of them are gratuitous. The listbox in Figure ??-1, for example, shows only five of the 10 options available in my environment, and the

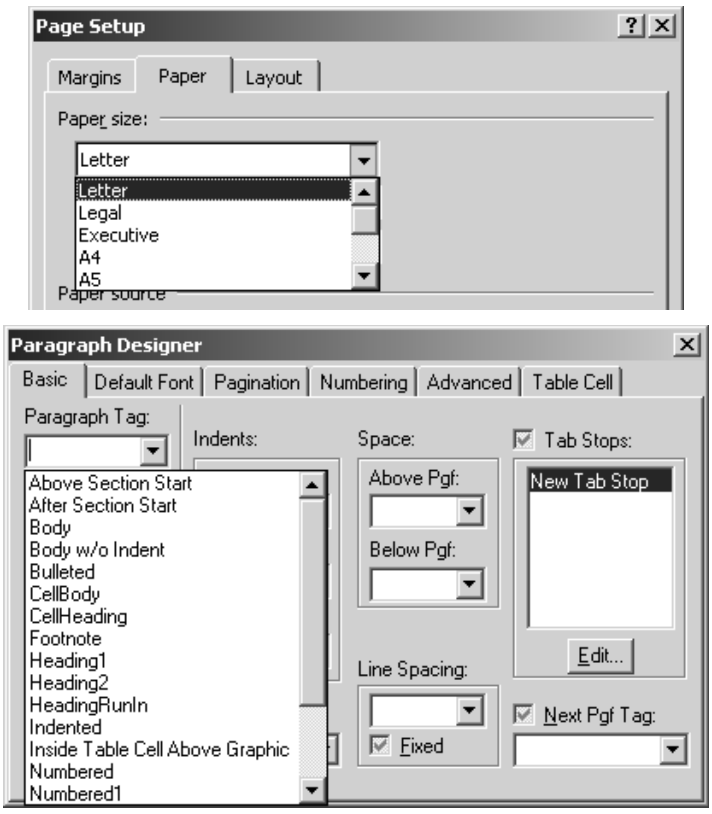

Figure ??-1. Top: Listbox control for selecting the paper size (from Microsoft Word 2002). Bottom: Combobox control for specifying the paragraph tag name (from Adobe FrameMaker 6). Both exhibit Listbox Keyholes.

combobox shows only 15 of 20, yet there is far more than enough screen space for each to display its full set of choices. Figure ??-2, which shows these controls in the context of the available vertical screen space, makes this quite clear.

Problems

From a user's point of view, Listbox Keyholes are objectionable for a variety of reasons. For users who don't know in advance which menu option they want, Listbox Keyholes complicate the problem of making the correct selection, because they show only a subset of the available options. Sometimes, this subset can be confusing or misleading. For example, Figure ??-3 shows that only dollar signs are visible when one opens the currency selection listbox in

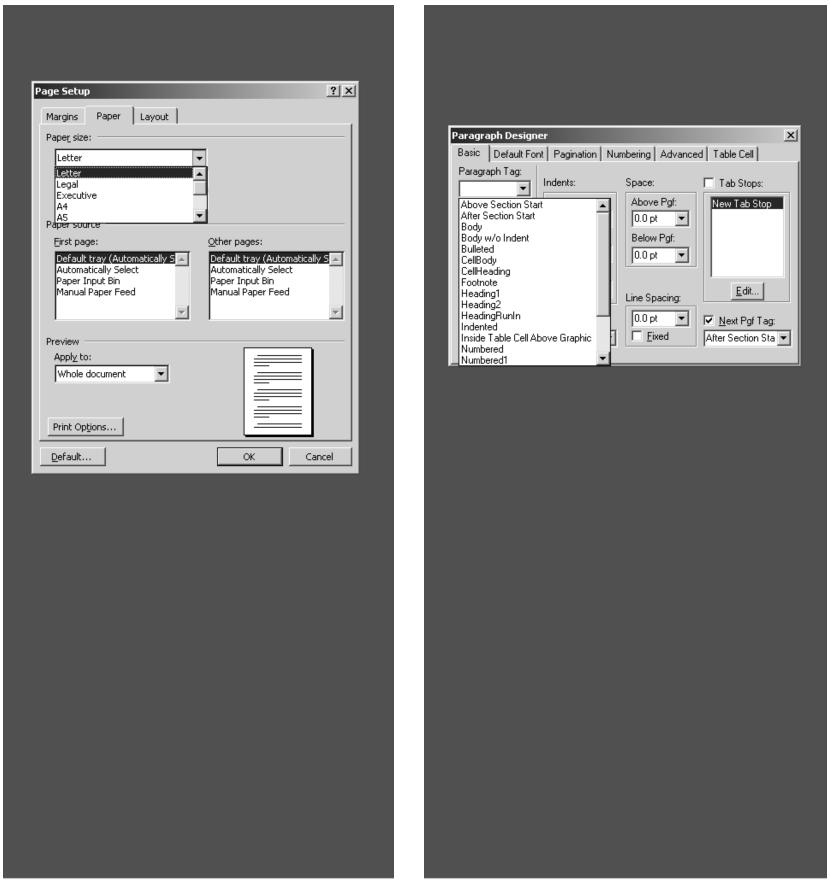

Figure ??-2. The listbox and combobox from Figure ??-1 in the context of the vertical screen space available to them.

Microsoft Excel 2002. Scrolling the listbox reveals additional dollar sign sym-

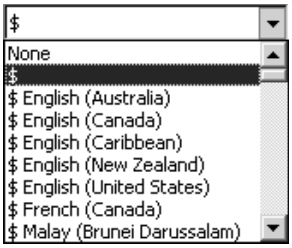

Figure ??-3. Listbox for selecting currency symbol from Microsoft's Excel 2002.

bols, but then the scrolling stops, suggesting that the dollar sign is the only currency symbol available. Certainly that's what it suggested to me when I encountered this listbox in my quest for a Euro symbol. The pause (which I suspect is caused by delayed loading of less commonly used symbols) is quite brief. Before one's astonishment has a chance to turn to anger, annoyance, disappointment or disbelief, the scrolling continues, soon revealing the plethora of additional currency symbols one expects from a program like Excel. With or without the scrolling pause, the initial listbox appearance is misleading. The pause just makes it worse. This is an example of how a keyhole, in addition to being problematic in its own right, can also exacerbate other user interface anomalies, in this case, nonlinear scrolling behavior.

Even when scrolling the contents of a listbox is clear and straightforward, it can be difficult for users to remember whether the best choice is currently displayed or a better choice has already scrolled past. That slows down the process of making a selection and increases the likelihood that an incorrect or suboptimal selection will be made.

For all users, even those who are already familiar with the listbox and who know which choice they want, Listbox Keyholes act like a speed bump. If the desired choice is not visible in the keyhole, the user cannot select it until the list has been scrolled to the appropriate location. This need to scroll yields two disadvantages. First, it inserts an extra task that the user must perform. Instead of simply selecting something, the user must first scroll and *then* select. This takes longer and requires more effort. Second, it interrupts the user's train of thought. The user wants to focus on the list entry to be selected, but if scrolling is necessary, she must suddenly think about the activity of scrolling, thus shifting her focus from the real task at hand. Admittedly, these are not large speed bumps,

but each inserts a small amount of friction into the process of users getting their work done, and, over time, small amounts of friction add up. The frequency with which Listbox Keyholes are encountered guarantees that their cumulative friction will generate a fair amount of heat. More fundamentally, gratuitous Listbox Keyholes have no technical justification, so there is no excuse for introducing them in the first place. It's just bad interface design.

Expert users will point out that my description in the previous paragraph is not entirely accurate. It is true that most users use the mouse or arrow keys to scroll the list until their desired selection is visible, but it is also possible to access the desired entry semi-directly by typing the letter(s) with which it begins. I say "semi-directly," because, though different listbox implementations vary in their exact behavior, at least under Windows, standard behavior is that the keyboard can be used for direct access to only the first letter of each entry. Consider Figure ??-4, which shows what anecdotal evidence suggests may be the most reviled of all listboxes, at least in the United States: one for choosing a state. (A colleague of mine—from Wyoming—has been known to rant, "Whoever designed those must live in Alabama!")

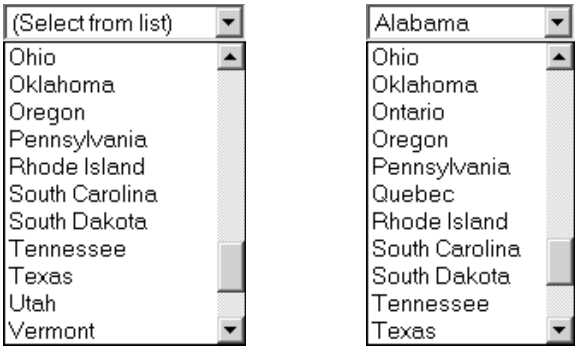

Figure ??-4. Listboxes for selecting one's state (left—from Dell's web site[1]) and one's state or province (right—from Channel Master's web site[2]).

I live in Oregon, and because I both fancy myself an expert user and tire of constantly scrolling state-selection listboxes to get to the "O" entries, I have accustomed myself to simply pressing O three times, once to jump to Ohio, twice more to advance over Oklahoma to Oregon. This generally works well, but, as Figure ??-4 makes clear, some state-selection listboxes include the Canadian provinces (thus making them state-or-province-selection listboxes), and when that happens, I often end up specifying Ontario instead of Oregon,

because in a listbox that includes both states and provinces in alphabetical order, Ontario is the third entry, and Oregon is the *fourth*. This kind of error cannot be directly attributed to a Listbox Keyhole, but the existence of the keyhole motivates my use of the keyboard shortcut in the first place, so it's reasonable to view the keyhole as a contributing factor.

One of the interesting aspects of Listbox Keyholes is that, as far as I can tell, there's neither rhyme nor reason to the sizes of the keyholes. As Figure ??-1 shows, Microsoft Word 2002's page setup dialog shows five options, while Adobe FrameMaker 6's paragraph designer shows 15. Such random sizes invariably mean that some keyholes will be *almost* big enough to show all the options, thus leading to "Off-by-One" Listbox Keyholes: listboxes that display every entry except one. Probability theory may predict that such keyholes should be rare, but my experience has been that when probability theory confronts Murphy's Law, the odds favor Murphy. Off-by-One Listbox Keyholes are not uncommon. Figure ??-5 shows two examples.

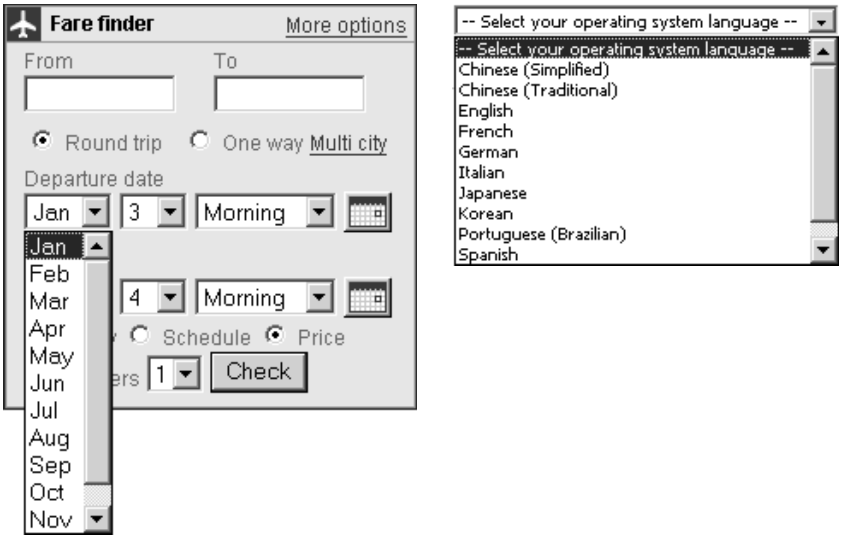

Figure ??-5. Off-by-One listbox keyholes from web sites for United Airlines (left[3]) and Microsoft (right[4]).

Off-by-One keyholes have all the usual drawbacks of Listbox Keyholes in general, but they are especially irritating to users who need to access the one lone hidden list entry, because each access emphasizes the gratuitous nature of the restriction. This point was driven home to me last year when I used the

United Airlines web site to make some December air reservations. As Figure ??-5 shows, December is the one hidden month in the month-selection listbox. I had to scroll that listbox dozens of times! (In retrospect, I could have just hit the D key each time, but to be honest, it didn't occur to me. I guess I'm not the expert user I fancy myself to be.)

Perhaps more importantly, Off-by-One keyholes tend to convey an aura of poor or careless interface design. Users understand that scrolling listboxes are necessary from time to time, but a scrolling listbox that displays 11 of 12 months suggests slipshod design or implementation at some level. In fact, the operating-system-language-selection listbox in Figure ??-5 may be an example of this. There is no way for me to know why all but one language option is displayed, but I can observe that the first entry in the list is "Select your operating system language." Of course, this entry is not itself a language option. Did the designer or implementer of this listbox correctly count the number of language options and create a listbox of that size, simply forgetting that there was an additional non-language option at the top?

Another problem stemming from Listbox Keyholes is that because the sizes of the keyholes vary, inconsistencies arise in the number of displayed options for a particular kind of choice, even within a single piece of software. For example, Figure ??-6 shows how the simple act of asking users to choose a drive letter in Microsoft's Windows 2000 varies, depending on the part of the operating system that's being used. A fundamental usability guideline is that things that are the same should look the same, but the two drive-letter-selection listboxes in Figure ??-6 look quite different. A direct effect of this kind of inconsistency is increased user confusion. *Why* do the two listboxes look different? Is there some subtle difference in meaning? An indirect effect is a decrease in users' perception of the quality of the software—in its "fit and finish." If the two listboxes show and mean the same thing, having them look different is a sign of sloppiness or carelessness on the part of the people developing the software.

Most Listbox Keyholes arise due to vertical limitations, i.e., limitations on the length of the displayed list, but listboxes with horizontal keyholes exist, too. Figure ??-7 gives an example. One might argue that horizontal keyholes are not as bad as vertical keyholes, because entries are truncated rather than being completely hidden. Users can thus guess at what the truncated entries mean. Unfortunately, because most listboxes offer no way to scroll horizontally, users are often unable to verify that their guesses are correct. As such, horizontal Listbox Keyholes suffer not only from the usual Listbox Keyhole problem of mak-

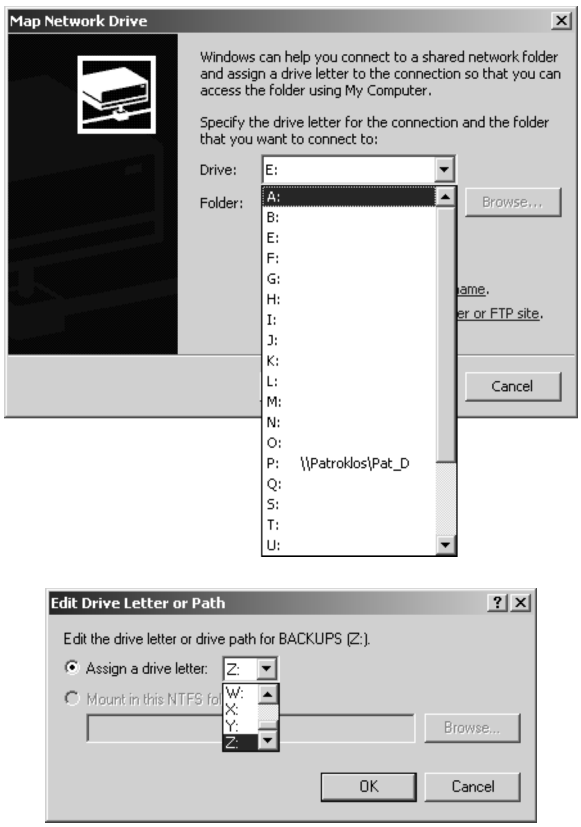

Figure ??-6. Inconsistent listboxes for selecting a drive letter in Microsoft Windows 2000. The top listbox arises when mapping a network drive. The bottom listbox is used when changing a drive letter through the Computer Management tool.

Figure ??-7. Horizontal Listbox Keyhole (from Hostway's web site[5]).

ing it unnecessarily difficult for users to choose the correct entry, they also make it difficult for users to know precisely what the entries are.

Sometimes it's more than difficult. Sometimes it's impossible. When a listbox contain multiple entries with a common prefix, but the listbox is too narrow to show more than the prefix, the listbox user is presented with choices that, other than their order in the list, are literally indistinguishable. This scenario commonly arises with listboxes that show network pathnames. Figure ??-8 shows an example.

Figure ??-8. [EXAMPLE TO BE ADDED]

Even when all listbox entries are unique, horizontal listbox keyholes can still render it impossible for users to determine what their choices are. That's because a (truncated) entry in a listbox may not be enough to fully identify the entry. Figure ??-9, for example, shows the history listbox for Windows 2000's Run dialog. The first entry in the listbox is a path whose visible portion is insuf-

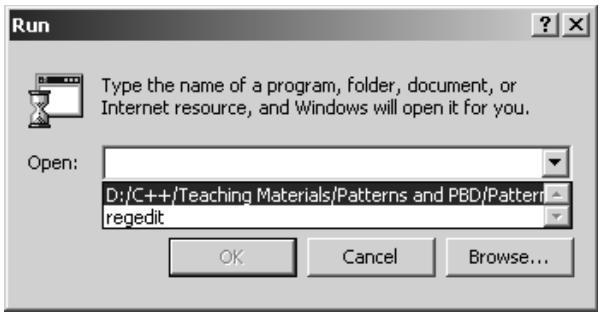

Figure ??-9. Horizontal Listbox Keyhole where the truncated entry is too short to fully identify the option it depicts.

ficient not only to identify the file corresponding to the entry, it's also too short to even identify the directory where that file is located.

Given the existence of both vertical and horizontal Listbox Keyholes, one can imagine listboxes that impose both. If it can be imagined, of course, it can usually be implemented—and it generally is. Figure ??-10 shows a listbox that's both too short and too narrow. I find this example particularly disturbing,

Figure ??-10. A listbox imposing both vertical and horizontal keyholes (from the web site for Microsoft's Windows Media Player 7[6]).

because I always worry that the entry "Burn your own a" is a truncated version of "Burn your own *author*." The only way to determine if this is the case (I'm happy to report that it is not) is to select that link entry and see where it leads, because, as is conventional, there is no way to perform horizontal scrolling of the listbox entries.

Listboxes often order their entries in a predictable way (e.g., alphabetically, as in Figures ??-1, ??-4, and ??-6), but there are many listboxes that do not. Figure ??-10 provides an example of this, but it's somewhat obscured by that listbox's horizontal keyhole, so consider instead Figure ??-11, which shows the listbox used to complete the phrase "I need help with..." at the web site for Norton Antivirus (NAV). When I visited that site, I was trying to find a way to disable a particular notification issued by NAV. Which of the listbox choices is most appropriate? Certainly "solving a software issue" is applicable, but that seems awfully general. Perhaps one of the hidden choices is something more specific, something like "resolving a configuration issue." Because of the keyhole, the only way to find out is to scroll the list. It turns out that the hidden choices are no better than "solving a software issue" for my problem, so having scrolled that choice out of view, I had to scroll the list back so that I could make

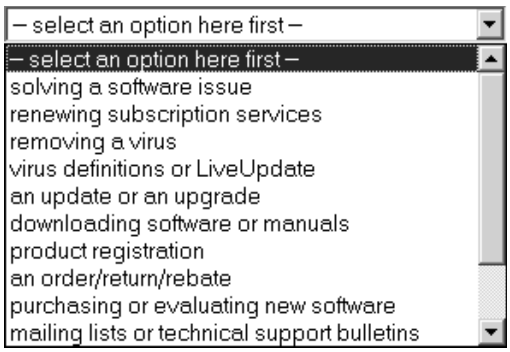

Figure ??-11. Listbox whose hidden entries are difficult to predict (from web site for Norton Antivirus[7]).

my selection. This kind of "scroll forward, scroll back" yo-yoing is a common (and annoying) side effect of Listbox Keyholes.

In sum, Listbox Keyholes give rise to the following problems:

- They slow users down by making them scroll to see all their available options.
- They interrupt users' trains of thought by forcing them to think about the act of scrolling when they really want to think about the act of selecting.
- They increase the likelihood that users will make an incorrect or suboptimal selection.
- They can convey to users an impression of poor interface design.
- They lead to application inconsistency, which confuses users and decreases perceived software quality.
- In the case of horizontal Listbox Keyholes, they make it more difficult for users to know what their options are.
- They often force users to scroll both forward and backwards for the same list, the forward scrolling to see all their options, the backward scrolling to allow them to make their selection.

Solutions

For each Listbox Keyhole, there are two straightforward ways to eliminate it. One is to have the listbox display all possible choices—literally to get rid of the keyhole. The other is to replace the listbox with some other control, i.e, to get

the data from the user in some other way (ideally one that doesn't introduce new keyholes). Both of these options are worth exploring, but before we do that, it's worth reflecting on the reasons why interface designers choose them in the first place.

From a user's point of view, listboxes offer two important features. First, they display all the valid choices the user is allowed to make. The eliminates the need for users to remember the options or to look them up. Second, they prevent users from making a choice that isn't allowed. A user might accidently make an incorrect or suboptimal choice, but listboxes make it impossible for a user to specify something that isn't a valid option.

From a programmer's point of view, this second feature is most appealing, because it reduces the need for input to be validated. If a listbox prevents a user from making an invalid selection, surely the selection the user makes must be valid! This pre-validation of user input is especially important for client-server systems (e.g., web-based systems), where the user interacts with client software on one machine and the user's input is processed by server software on a different machine. Interface elements like listboxes that can weed out invalid user inputs on the client machine are highly desirable, because they improve the scalability of the system by shifting some data validation tasks from client to server and by reducing client-server network traffic. Furthermore, client-side data validation provides users with faster feedback than is typically possible with server-side data validation, so interface controls like listboxes can lead to an improvement in an application's responsiveness.

Listboxes thus offer the following advantages:

- They document all valid options for users.
- They prevent users from entering an invalid selection.
- They reduce the need for user input to be validated.
- In client-server systems (including web-based systems), they improve the scalability of the system and facilitate increased application responsiveness.

Of course, keyholes don't help with any of these things. In fact, Listbox Keyholes interfere with some of these goals. For example, a listbox doesn't fully live up to its promise of showing users all their options if only some options are displayed at once. There is a case to be made for using listboxes, but the case for a listbox with a keyhole is weaker than the case for the same listbox without a keyhole.

Still, there are times when the best way to eliminate a Listbox Keyhole may be to eliminate the listbox. Consider again the state-selection listbox of Figure ??-4. There are certainly exceptions, but it seems reasonable to assume that the vast majority of users of most applications that need this information already know the two-letter abbreviation for their state, e.g., New Yorkers know to use NY and Californians know to use CA. That being the case, the primary challenge for most users is *specifying* the information, not *finding* it. A two-letter edit control would allow users to type in the abbreviation for the state they want to specify, and for most users, this would almost certainly take less time than manipulating a listbox. Furthermore, I suspect that for many applications, the error rate for such entry would be no higher than it would be for listbox-based state selection, because most people can type their state abbreviations almost instinctively. Speaking for myself, for example, I don't ever recall having mistyped OR into an edit control, but, as I said, I have been tricked by a listbox into incorrectly specifying that my home is in Ontario.

Replacement of listboxes with edit controls can help with two other problems, as well. One is what one might call the "what do they call it here?" problem. This difficulty arises when there are multiple common terms for the same thing. Remarking on the difficulty of finding the appropriate listbox entry for his country of residence, a colleague of mine who lives near London asks, "Is it Britain, Great Britain, England, the United Kingdom, or the UK?" A well-designed listbox might include all these entries, but many more will contain only one, leaving it to the user to figure out what the listbox author was thinking when the listbox was designed.

We saw a similar kind of uncertainly regarding the contents of state selection listboxes (is it truly just for states, or are provinces also included?), but even the 50 United States have more than one common textual depiction, e.g., is it "Oregon" or "OR"? This leads to opportunities for inconsistency, and where there's opportunity, there's achievement. This kind of inconsistency can be combined with the state-only or state-and-province uncertainty, and Figure ??-12 shows the result. Such inconsistency and uncertainty is the second issue that can be addressed when listboxes are replaced with edit controls.

In a client-server application, replacing a listbox with an edit control might seem to shift the burden for data validation from the client to the server, but this need not be the case. There are other ways to perform client-side data validation. For example, one could load an array-like data structure† with the set of valid values, then verify that the entered data was one of those values before

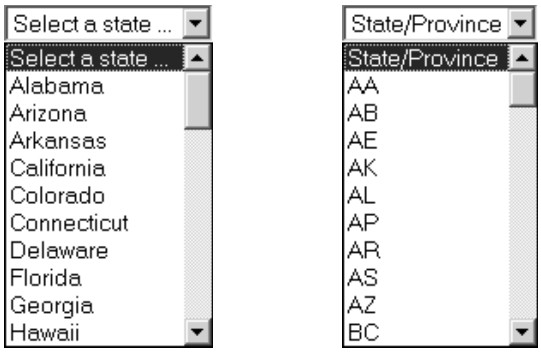

Figure ??-12. Inconsistent listboxes at Intuit's web store for TurboTax. The left listbox[8] is used to specify the state for which tax software is needed. The one on the right[9] is used to specify the state or province to which the software should be shipped.

shipping it off to the server. Such validation could be compiled into the client, could take the form of a scripting language like JavaScript, or could run as a client-side applet in environments like Java and .NET.

Frankly, the argument about client-side data validation is largely a red herring, anyway. In most client-server environments, servers have to perform some kind of data validation, regardless of how careful their clients are. There are three reasons for this. The first is to ensure consistency among data values. For example, if I specify that my state is Oregon and my zip code is 10603, there's a problem, because that zip code corresponds to a New York address. This particular example could be checked client-side, but, at least for web applications, it rarely is. If the mismatch is to be detected, it will be almost certainly occur on the server (possibly in the database).

A more significant reason why server-side data validation must take place is that for communication that takes place across an open network (e.g., the Internet), servers cannot assume that they are receiving data from a well-behaved client. Web servers, in particular, must be prepared to deal with malicious software masquerading as a web client that deliberately submits invalid data. For example, servers that expect to receive two-letter state abbreviations must be prepared to handle arbitrarily long strings of characters, because hackers will submit them in the hopes of causing a buffer overrun. (Such overruns are the

† I say "array-like," because there are good reasons not to use fixed-size arrays. For details, consult Chapter [RFS].

result of unchecked Restricted Field Size Keyholes, a topic I consider in Chapter [RFS].)

Finally, there's a mundane reason why server-side data validation is generally necessary for robust applications: client-side software may have bugs. For example, Figure ??-13 shows a listbox that allows a user to specify that the

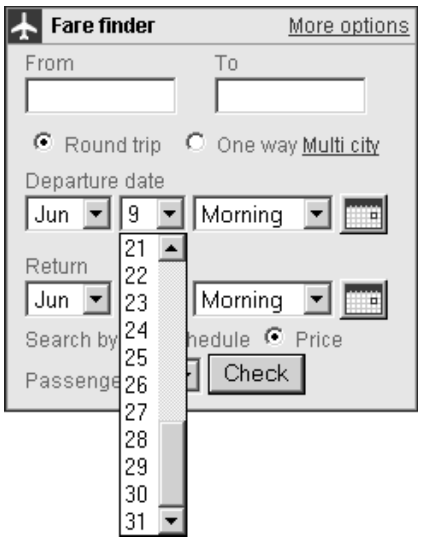

Figure ??-13. Day-selection listbox that allows specification of an invalid date (from United Airlines' web site[10]).

desired date is 31 June, a day that does not exist. (In the case of the date selection machinery in Figure ??-13, the date immediately shifts to 1 July if 31 June is chosen, so the problem appears to be handled on the client. It would be interesting to know whether that was the behavior upon initial deployment of the page....)

Simply put, while client-side data validation has its uses, it can't replace server-side validation, it can only complement it.

Getting back to the idea of replacing a listbox with an edit control, one issue you must confront if you go that route is that you lose the listbox's ability to show users all possible entries. This implies that you'll need to provide equivalent functionality in some other way. For example, you might offer a pop-up window that lists all valid options. Expedia exemplifies this in the way it allows users to specify dates—see Figure ??-14. Here, dates may be entered as free-

Figure ??-14. Pop-up window as an aid for entering the departure date (from Expedia's web site[11]).

form text into an edit control, but they may also be specified by clicking on a calendar that is available as a pop-up window.

There is no reason why a pop-up window must be used in conjunction with a conventional edit control. The Expedia approach is to optionally produce a window containing all possible values, thus letting users either type or select the data they wish to enter. This is really just a combobox with the edit control part physically decoupled from the entry-selection part. A listbox could similarly be broken onto two pieces: a pop-up window containing all possible choices and an edit-like control showing the current selection. However, the displayed value would be modifiable only by selecting something in the window. Assuming the window is resizable (as Chapters [FSW] and [LSW] argue it should be), such a decoupled listbox design would suffer from neither vertical nor horizontal gratuitous listbox keyholes. (Screen size constraints might still lead to keyholes, but such keyholes would not be gratuitous.)

But let us suppose that you've determined that a conventional-style listbox is truly the best interface control for the information you need to get. The challenge then becomes designing the listbox such that either no keyhole is imposed or, failing that, that any keyhole that is imposed is as large as possible.

We can divide listboxes into two categories. *Static listboxes* are those whose contents are fixed and known before runtime. State-selection listboxes (Figure ??-4), month selection listboxes (Figure ??-5), drive letter selection list-

boxes (Figure ??-6), and day selection listboxes for a particular month (Figure ??-13) are examples of static listboxes. *Dynamic listboxes* have entries that are determined at runtime, either because the listbox contents are read from an external source (e.g., a configuration file) at startup or because the contents of the listbox may change as the program runs. Figures ??-1 (paragraph type selection), ??-5 (language selection), and ??-10 and ??-11 (topic selection) are examples of dynamic listboxes.

?

I want to make two arguments here. First, I want to say that for static listboxes, their size (in number of lines needed to display all entries) can be determined at design time, so there is no reason for a keyhole. However, it is possible that the static size of the listbox would be too large to fit on some screens, e.g., a listbox holding all the states might not be displayable on a screen of 640x480, etc. If such a statically sized listbox is too big for the available screen space, do UI toolkits (e.g., MFC, Swing, Windows Forms, Web Forms, and whatever they use on Mac and Unix) and tools (e.g., VB, FrontPage, Dreamweaver) yield code that adapts gracefully? Similarly, does the code derived from such toolkits/tools also adapt gracefully when the font size is changed, e.g., if a user's browser increases the font size? If so, I can argue that designers should just design keyholeless listboxes in the static case and rely on the UI runtimes to handle things correctly if there isn't sufficient screen space too show the entire list. But if toolkits/tools don't yield code that adapts gracefully (e.g., if such code will try to display the bottom of a too-long listbox off-screen with no way to scroll), I'll probably just argue that static listboxes and dynamic listboxes have to be treated the same way.

Second, I want to argue that dynamic listboxes can avoid keyholes by determining their size at runtime. To make that happen, developers would have to write a function that, given the number of entries to go in the listbox, would return something equivalent to how long the listbox must be. I've heard that this function would be a pain to write (e.g, converting dialog units into screen units or something like that), but I assume that (1) it can be written without huge amounts of difficulty and (2) UI toolkits and tools truly do allow for the size of listboxes to be specified at runtime. Are these assumptions valid for MFC, Swing, Windows Forms, Web Forms, and other widely used UI toolkits as well as UI-generating tools like VB, FrontPage, and Dreamweaver? In general, is there some technical reason why the advice to calculate the correct size of a listbox at runtime is impractical?

I have a couple of additional points I have to add to this chapter. First, I need to say that if a keyhole can't be eliminated (e.g., because there isn't enough vertical space to show all the available options), it should be made as large as possible (i.e., uneliminatable keyholes should be as large as possible). Second, I want to mention that for listboxes with really long lists (e.g., hundreds of entries—like choosing an airport code at a travel site), a listbox is probably not the best UI choice, anyway .

I also need to say something about how to avoid horizontal keyholes in conventional fixed-width listboxes.

If any of the topics above affect listboxes and comboboxes differently, I need to know that.

How difficult are resizable listboxes to implement on major UI platforms, both for native and web applications?

Summary

Listboxes and comboboxes are important user interface components, but Listbox Keyholes decrease their effectiveness. Such keyholes slow users down, increase the likelihood of incorrect or suboptimal selections, increase the likelihood of interface inconsistency, and reduce the perceived overall quality of software that uses them. In general, Listbox Keyholes introduce friction into what should be a smooth process of choosing one from a set of possible options.

These keyholes can be eliminated by either replacing the listbox with a different user interface control or by removing the keyhole from the listbox. The basis of the latter strategy is deferring until runtime the decision on how many listbox entries to display at once, displaying all of them unless there is a compelling reason not to. This can be accomplished either automatically or by allowing users to manually resize listboxes.

Replacing static determination of the on-screen size of a listbox with dynamic determination may require creation of a nontrivial function to compute the appropriate size, but such a function need be written only once, and its cost can be amortized over the large number of Listbox Keyholes it eliminates or enlarges.

Listbox keyholes are very common, but they don't need to be. There are good reasons for getting rid of them, and doing so isn't terribly difficult. They

should be eliminated whenever possible, and in cases where elimination isn't practical, they should be made as large as possible.

URL References

1. https://ecomm.us.dell.com/dellstore/chkout1_ship.asp?customer_id=22&itemtype=ARB on 12 June 2003.
2. http://www.channelmaster.com/Pages/distributorlocator.htm on 23 July 2003.
3. http://www.ual.com/ on 6 June 2003.
4. http://www.microsoft.com/hardware/mouse/download.asp on 3 June 2003.
5. https://sitecontrol.hostway.com/pas/d/SiteControl/Manager/Public/OrderForm/Checkout/ExpressLogin.process on 14 June 2003.
6. http://windowsmedia.com/mg/gettingstarted/started.asp on 3 June 2003.
7. http://www.symantec.com/techsupp/nav/nav2003_pro_tasks.html on 9 June 2003.
8. http://www.shop.intuit.com/store/jhtml/freestate.jhtml;jsessionid=WT2R2VV3JLKPMCQIBMTRX1QKBAFSQF4K?pageLocation=TurboTax&priorityCode=5070000&productLine=TurboTax&catalogRefIds=sku12806&productId=prod11118&_requestid=89313 on 23 July 2003.
9. https://www.shop.intuit.com/share/jhtml/secure/customerinformation.jhtml;jsessionid=WT2R2VV3JLKPMCQIBMTRX1QKBAFSQF4K?priorityCode=5070000&_requestid=89456 on 23 July 2003.
10. http://www.ual.com/ on 9 June 2003.
11. http://www.expedia.com/ on 9 June 2003.