The Keyhole Problem

Scott Meyers

July 2002 Draft

1 Introduction

Every day I work with my computer, and every day I come away with the feeling that I'm engaged in a constant battle with a subtle and unrelenting enemy. It's a bit like walking an uphill grade, always against a headwind, always taking care to prevent airborne grains of sand from getting in my eyes. By the end of the day, I'm worn down, much tireder than I should be given the distance I have covered.

I have come to realize that many of my struggles spring from a single underlying problem: I use my computer to interact with a big, wide, wonderful world, but the software I use imposes gratuitous restrictions on how much of that world I'm allowed to interact with at a time. It's as if my software is forcing me to look at the world through the keyhole of a door. Hence my name for this phenomenon: *The Keyhole Problem*.

The Keyhole Problem arises every time software artificially restricts something you want to see or something you want to express. If you want to see an image, but your image-viewing software artificially restricts how much of that image you can see at a time, that's the keyhole problem. If you want to specify a password of a particular length, but your software says it's too long, that's the keyhole problem. If you want to type in your U.S. telephone number, but your software refuses to let you punctuate it in the conventional manner with a dash between the three-digit prefix and the four-digit exchange, that's the keyhole problem.

You probably encounter the keyhole problem dozens of times each day, working around it so instinctively, you don't even realize you're doing it. Yet the need to develop and implement workarounds exacts a continuing toll in time and energy. Each keyhole injects friction into the tasks you are performing. Each is a mild irritant. Each causes a bit of frustration. Some are actually dangerous. Almost none are justifiable.

I have several goals in this paper:

- To identify and categorize the different types of keyholes I frequently encounter. To that end, I identify a dozen different types of keyholes divided into two general categories.
- To demonstrate the ubiquity of the keyhole problem. To do that, I include a large number of examples, many of which I expect you will recognize as representative of keyholes you encounter. Almost all of the examples are based on tasks I have needed to perform within the past few months. I didn't seek out these keyholes. They found me.
- To convince you that keyholes are a *serious* problem, that keyholes can have disastrous consequences. Such consequences can range from security violations that disrupt networks and cost billions of dollars to—quite literally—death and dismemberment.
- To explain how keyholes in one part of a system often lead to undesirable (typically confusing or inconsistent) behavior in other parts of the system. Eliminating keyholes tends to improve the overall quality of a software system, because it eliminates the root cause of secondary problems.
- To describe practical steps that software developers can take to largely eliminate the keyhole problem. Many of these steps require little more than a determination to modify bad design habits that typically go unquestioned.

The examples and analysis in this article are primarily drawn from my experience with desktop applications under Windows 2000, but I believe that my observations and conclusions apply equally well to other desktop operating systems and environments, including MacOS, Unix in its various forms, and other desktop versions of Windows (e.g., Windows XP). I further believe that my reasoning is applicable to software in non-desktop environments, such as Palm and Pocket PC, though I recognize that the different physical characteristics of such devices (e.g., reduced screen size and resolution, lack of hard drives and virtual memory, etc.) may necessitate adjustments to the keyhole elimination strategies I propose. Finally, I believe that the ideas I present here are applicable to software running in embedded environments, though again the solutions I propose may need to be adjusted to be practical for such environments.

Fundamentally, the thesis I put forward in this article is that keyholes in software are extremely common, are irritating and frustrating at best, and are dangerous or harmful at worst. Regardless of the context in which keyholes occur, eliminating or reducing their impact virtually always leads to better software systems.

2 Visible Keyholes

The keyholes I have identified are either *visible* or *invisible*. Visible keyholes are present in user interface components and thus have some kind of screen presence. Invisible keyholes lack this presence. In this section, I introduce the visible keyholes. I discuss invisible keyholes in Section 3.

2.1 The Fixed Width Web Page Keyhole

For anyone who interacts with the world wide web, one of the most commonly encountered keyholes is the fixed width web page keyhole. Such a keyhole arises when a web page fails to take advantage of horizontal space in a browser window beyond that anticipated by the page's designer. Figure 1 shows two examples of this kind of keyhole. In the examples, notice how nearly half the horizontal window space is

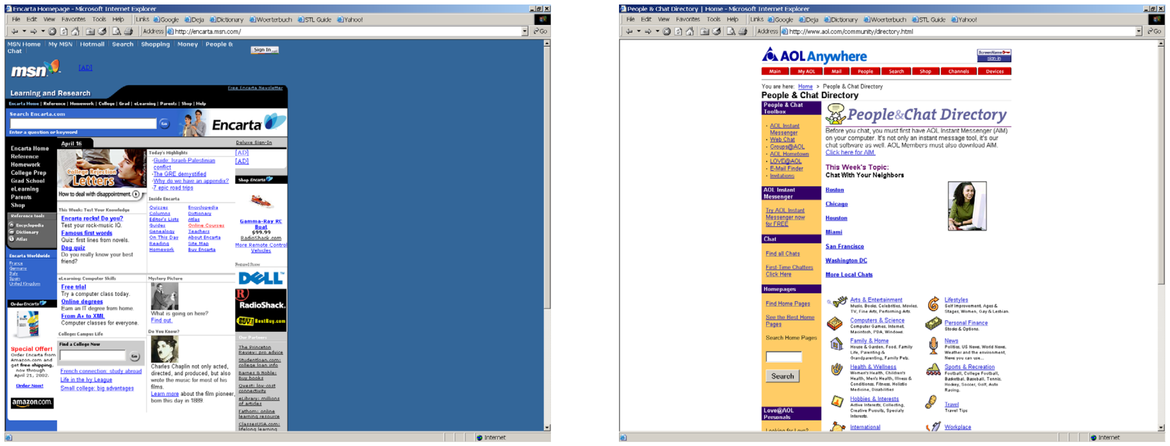

Encarta Homepage - Microsoft Internet Explorer
http://encarta.msn.com/
People & Chat Directory | Home - Microsoft Internet Explorer
http://www.aol.com/community/directory.html

Figure 1: Fixed width web page keyholes. The page on the left is from MSN, the page on the right from AOL.

unused[†], even though the vertical scroll bars indicate that there is additional content to be displayed. That additional content *could* be displayed; the client *has* made sufficient screen real estate available. Instead of taking advantage of the space, however, the site designer has limited the user's vision and has forced her to view the site through a keyhole.

Some web sites play games with the keyhole in an attempt to make it less obvious, generally by centering the fixed width page in the browser window. Figure 1 shows how the AOL home page does this. From a keyhole point of view, this is the moral equivalent of a child spreading his vegetables around on his plate

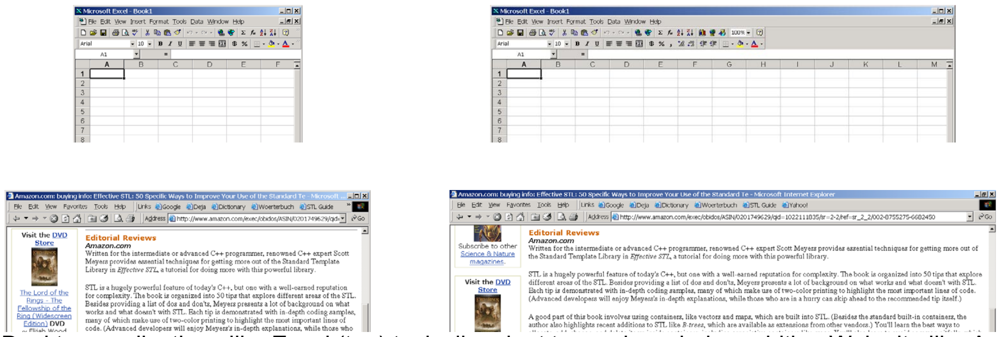
† The resolution of the screen shots in this article is 1280x1024, because that is the resolution I generally use. This resolution is presumably higher than that of most users, but that is not relevant to the issue at hand. The issue is that software (in this case, the HTML behind web sites) is gratuitously restricting the user's view of the world. At many web sites, the keyhole continues to exist even at lower resolutions. For example, the width of the content at the MSN web site is only 610 or 770 pixels, depending on the width of the browser window, and the AOL web site offers a fixed width of only 585 pixels.

in an attempt to convince his parents that he has eaten most of them. The amount of available (but unused) space remains the same.

Fixed width web page keyholes are particularly gratuitous, because both HTML and web browsers are designed for line breaks to be determined dynamically. Web site designers must take specific steps to *prevent* this from happening, e.g., by putting page content into tables and fixing the width of the table columns. Even so, fixed width web pages are distressingly common. Of the first 50 sites listed in *PC Magazine*'s Top 100 Classic Web Sites[†], two thirds use fixed width pages.

I speculate that the underlying reason for this is that many web sites were initially under the purview of corporate marketing departments, and the people who designed the sites were both more familiar and more comfortable with the static page layout typical of print media (e.g., brochures, magazine ads, etc.) than with the dynamic page layout proffered by HTML and web browsers. Rather than change their approach to page design, these designers found ways to eliminate browsers' inherent preference for dynamic layout.

This interpretation is consistent with the observation that desktop applications rarely exhibit this kind of behavior. Rather, they tend to show more content as the window in which they are running is expanded. That is, the *window itself* is the keyhole. Figure 2 shows how this is the case for Microsoft Excel 97. When Excel's window is made larger, the user sees more of the underlying spreadsheet. Figure 2 also shows that such behavior can be offered by web sites, as the screen shots from Amazon demonstrate.


Microsoft Excel - Book1
File Edit View Insert Format Tools Data Window Help
Amazon.com: buying info: Effective STL: 50 Specific Ways to Improve Your Use of the Standard Te - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links Google Deja Dictionary Woerterbuch STL Guide Yahoo!
Address http://www.amazon.com/exec/obidos/ASIN/0201749629/qid=1022111835/sr=2-2/ref=sr_2_2/002-8755275-6682450
Visit the DVD Store
The Lord of the Rings - The Fellowship of the Ring (Widescreen Edition) DVD
Subscribe to other Science & Nature magazines.

Editorial Reviews
Amazon.com
Written for the intermediate or advanced C++ programmer, renowned C++ expert Scott Meyers provides essential techniques for getting more out of the Standard Template Library in *Effective STL*, a tutorial for doing more with this powerful library.

STL is a hugely powerful feature of today's C++, but one with a well-earned reputation for complexity. The book is organized into 50 tips that explore different areas of the STL. Besides providing a list of dos and don'ts, Meyers presents a lot of background on what works and what doesn't with STL. Each tip is demonstrated with in-depth coding samples, many of which make use of two-color printing to highlight the most important lines of code. (Advanced developers will enjoy Meyers's in-depth explanations, while those who are in a hurry can skip ahead to the recommended tip itself.)

A good part of this book involves using containers, like vectors and maps, which are built into STL. (Besides the standard built-in containers, the author also highlights recent additions to STL like *B-trees*, which are available as extensions from other vendors.) You'll learn the best ways to

Figure 2: Desktop applications like Excel (top) typically adapt to varying window widths. Web site like Amazon (bottom) demonstrate that this is possible for web sites, too.

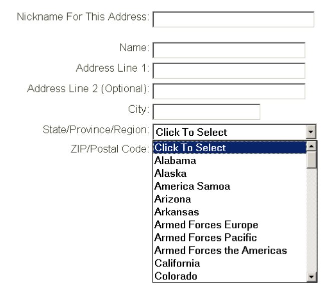
The predetermined nature of fixed width web page layouts dictates that such pages not only impose keyholes when viewed through wider-than-expected windows, they also impose a related problem: the pages are clipped (and thus give rise to horizontal scroll bars) when viewed through narrower-than-expected windows. Addressing this problem is more difficult than is addressing the keyhole problem, because adapting software to limited resources is more difficult than is removing restrictions on software in the presence of excessive resources. In this article, I shall not further consider what one might call the "reverse keyhole" problem.

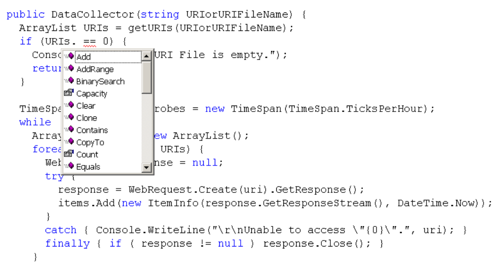
2.2 The ListBox/ComboBox Keyhole

ListBox and ComboBox controls (which I will refer to simply as "listboxes") give users a way of choosing from a list of alternatives, but the controls almost always limit the number of alternatives that are simultaneously visible, even when ample screen space is available to display them all. Figure 3 shows several examples of this phenomenon.

There are at least two drawbacks to this kind of keyhole, which is common in both desktop applications and web sites. First, it prevents users from seeing all of their options simultaneously, thus making it more difficult for them to decide what choice is most appropriate. Second, it requires an unnecessarily large amount of work for users to select an option that is not displayed in the keyhole. If all options were displayed, users would be able to move the mouse to the desired selection and click on it. Because of the key-

† http://www.pcmag.com/article/0,2997,s=25087&a=21914,00.asp





```
public DataCollector(string URIorURIFileName) {
  ArrayList URIs = getURIs(URIorURIFileName);
  if (URIs. == 0) {
    Consc                URI File is empty.");
    retur
  }

  TimeSpa                robes = new TimeSpan(TimeSpan.TicksPerHour);
  while
    Array                ew ArrayList();
    forea                 URIs) {
      Wel                nse = null;
      try {
        response = WebRequest.Create(uri).GetResponse();
        items.Add(new ItemInfo(response.GetResponseStream(), DateTime.Now));
      }
      catch { Console.WriteLine("\r\nUnable to access \"{0}\".", uri); }
      finally { if ( response != null ) response.Close(); }
    }
```

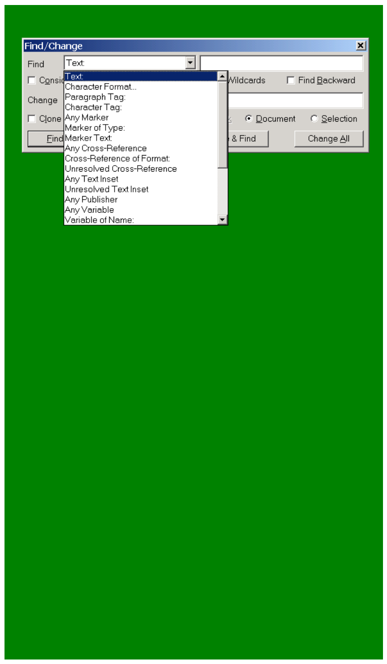


Figure 3: Listboxes from BooksAMillion's web site (upper left), Microsoft's Visual Studio .NET (lower left), and Adobe's FrameMaker 6 (right). Scroll bars show that only a subset of choices are displayed. The FrameMaker screen shot makes clear that lack of screen space is not the problem.

hole, however, users must first move the mouse to the scroll slider, down-click on it, move the slider down while holding the mouse button down, and release the mouse button. Only then may they select the option they desire.†

At web sites, listboxes are commonly employed to allow for client-side validation of user input, thus reducing network traffic and load on the server. These are legitimate goals, but listboxes come with drawbacks that are often underappreciated. Consider, for example, the BooksAMillion example in Figure 3 that uses a listbox to allow users to specify their State/Province/Region. Such listboxes are ubiquitous on the Web, but they can be intensely frustrating to users. One colleague of mine thunders, "Whoever designed those must live in Alabama!," because Alabama is the first state in an alphabetical listing. (The colleague is from Wyoming.) Anecdotal evidence suggests that my colleague is not alone in her strong negative feeling about such listboxes. Furthermore, it is not clear that such listboxes improve the accuracy of the data compared to simply letting users type it in. I live in Oregon, which is alphabetically the third US state beginning with O (after Ohio and Oklahoma), and I encounter state-selection listboxes so frequently, I now type O (to jump to the entry for Ohio), hit the down-arrow key twice (to advance over Ohio and Oklahoma to Oregon), and hit Enter without so much as thinking about it. Usually, this yields the correct result, but in listboxes that include Canadian provinces along with US states (such as the one in Figure 3), my ingrained procedure results in my erroneously selecting Ontario.

Of course, when the server receives my complete address, it detects the mismatch between my Oregon zip code and my claim to live in Ontario, but the fact that the server must check for this kind of error significantly weakens the justification for using a listbox in the first place. After all, one of the arguments in its defense was client-side data validation.

† There are keyboard-based mechanisms for navigating listboxes, but, in the case where users do not know the list of available choices in advance, they are more cumbersome and, presumably, less commonly employed than are mouse-based approaches.

In the particular case of listboxes for identifying states, it would be interesting to know whether listbox-related user errors (i.e., selecting the wrong value) actually occur at a lower rate than typing-relating errors when users are allowed to specify their state textually. I'm fairly certain that I've selected Ontario from a listbox far more times than I've mistyped "OR" or "Oregon."

The additional work users must do to overcome listboxes-related keyholes and the new opportunities for errors that listboxes may give rise to are more closely related that may at first appear. It is certainly possible to design listboxes that do not give rise to keyholes (see below), but the frustration and error factors inherent in listboxes remain even if the listbox-related keyholes are eliminated. In some cases (e.g., selecting states), listboxes are simply an inferior user interface component, and they should be eliminated anyway. The connection between this observation and keyholes is that one way to eliminate a keyhole is to eliminate the component giving rise to it, so if a listbox keyhole is eliminated by eliminating the listbox, the overall user interface may well improve beyond the improvement specifically brought about by elimination of the keyhole. As we shall see elsewhere in this paper, *ridding software of keyholes often yields benefits beyond those specifically traceable to the keyhole itself*.

Even when listboxes or listbox-like components are appropriate user interface elements, it is important to recognize that they need not give rise to keyholes. Figure 4 shows two listbox or listbox-like interface elements that do not introduce traditional listbox keyholes.

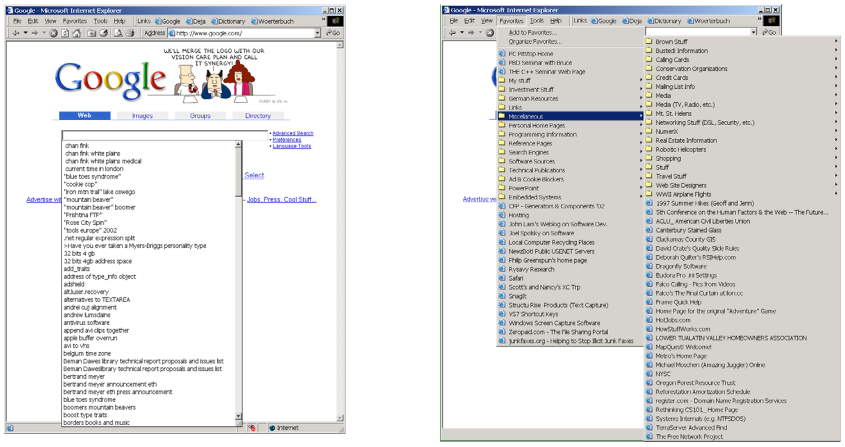


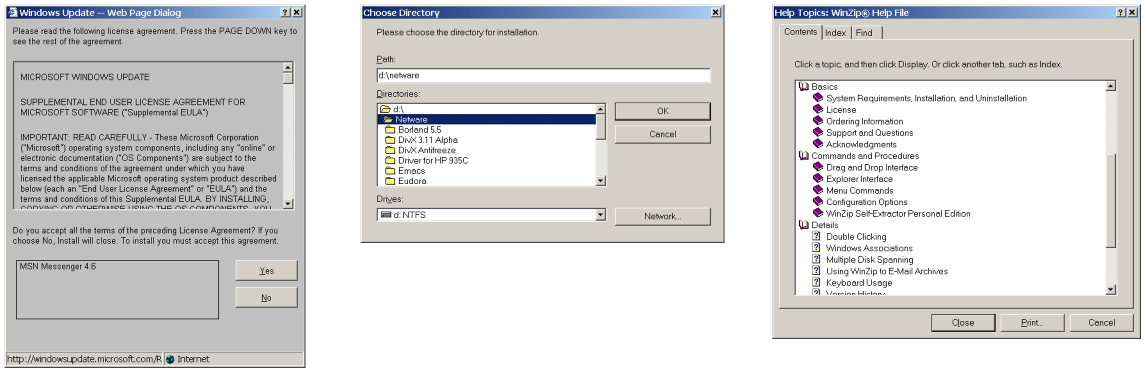
Figure 4: Left: Microsoft Internet Explorer 6 listbox at Google (left) that can be extended to the limits of the screen. Right: Internet Explorer 6 cascading favorites menus that dynamically size themselves up to the screen size.

2.3 The Fixed Size Window Keyhole

There are two problems with fixed size windows. First, they are almost always too small to display all the information they contain; users must typically do some scrolling. Second, fixed size windows are almost always smaller than the screen on which they are displayed[†]. Combined, these observations imply that there is typically more information to display and there is room to display it, but fixed size windows refuse to allow users to see the additional information in the additional space.

Figure 5 shows some of the many ways in which desktop applications typically employ fixed size windows. In each case, notice the presence of the vertical scroll bars. They make clear that the windows are too small to display all the information they contain. That being the case, why aren't the windows bigger? More importantly, why has the software chosen the size instead of letting the user determine it herself? Each of the applications depicted in Figure 5 features variable-sized windows for its primary functionality, so why do they take a different route here? Cynics have suggested that license agreements are deliberately

† Of course, a fixed size window may also be larger than the amount of screen real estate the user wishes to devote to it, but pursuit of that problem leads to the idea of allowing users to make such windows smaller, and *that* leads back to the reverse keyhole problem.



Please read the following license agreement. Press the PAGE DOWN key to see the rest of the agreement.

MICROSOFT WINDOWS UPDATE

SUPPLEMENTAL END USER LICENSE AGREEMENT FOR MICROSOFT SOFTWARE ("Supplemental EULA")

IMPORTANT: READ CAREFULLY - These Microsoft Corporation ("Microsoft") operating system components, including any "online" or electronic documentation ("OS Components") are subject to the terms and conditions of the agreement under which you have licensed the applicable Microsoft operating system product described below (each an "End User License Agreement" or "EULA") and the terms and conditions of this Supplemental EULA. BY INSTALLING,

Do you accept all the terms of the preceding License Agreement? If you choose No, Install will close. To install you must accept this agreement.

MSN Messenger 4.6

http://windowsupdate.microsoft.com/R Internet

Please choose the directory for installation.

Path: d:\netware

Directories:
d:\
Netware
Borland 5.5
DivX 3.11 Alpha
DivX Antifreeze
Driver for HP 935C
Emacs
Eudora

Drives: d: NTFS

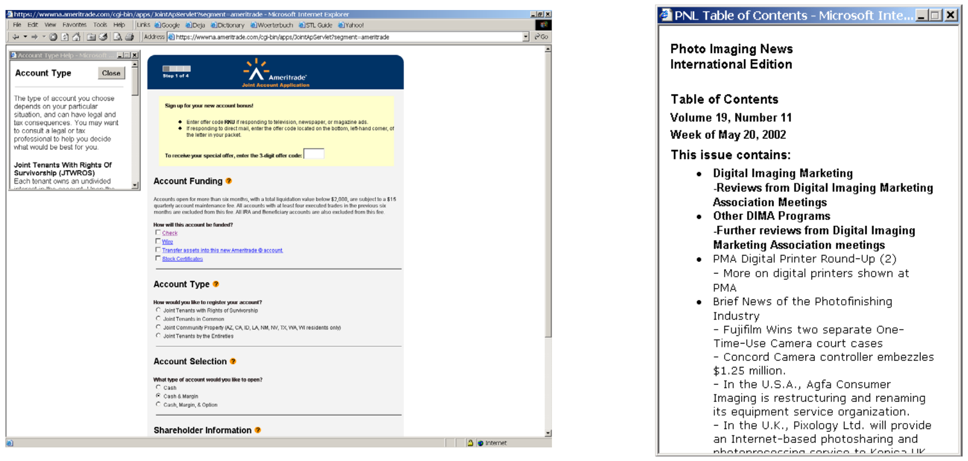
Click a topic, and then click Display. Or click another tab, such as Index.

Basics
System Requirements, Installation, and Uninstallation
License
Ordering Information
Support and Questions
Acknowledgments
Commands and Procedures
Drag and Drop Interface
Explorer Interface
Menu Commands
Configuration Options
WinZip Self-Extractor Personal Edition
Details
Double Clicking
Windows Associations
Multiple Disk Spanning
Using WinZip to E-Mail Archives
Keyboard Usage

Figure 5: Fixed size windows for (left) license agreements from Microsoft Windows Update, (middle) directory selection for Apple QuickTime 5 Installer, and (right) help for WinZip 8.1.

packaged in a manner to make them as difficult to read as follows[†], but why would an application want to make a user work harder than necessary to select a directory or to find a help topic?

Often, the size of fixed size windows fails to increase with increasing screen resolutions. This can lead to truly absurd situations, such as the one shown in Figure 6. Here, during the process of opening a broker-


File Edit View Favorites Tools Help
Links Google Deja Dictionary Woerterbuch STL Guide Yahoo!
Address
Account Type Help - Microsoft ...
Close
Step 1 of 4
Ameritrade
Joint Account Application
Internet
PNL Table of Contents - Microsoft Inte...

https://wwwna.ameritrade.com/cgi-bin/apps/JointApServlet?segment=ameritrade - Microsoft Internet Explorer

Account Type

The type of account you choose depends on your particular situation, and can have legal and tax consequences. You may want to consult a legal or tax professional to help you decide what would be best for you.

Joint Tenants With Rights Of Survivorship (JTWROS)
Each tenant owns an undivided

Sign up for your new account bonus!

- Enter offer code **RKU** if responding to television, newspaper, or magazine ads.
- If responding to direct mail, enter the offer code located on the bottom, left-hand corner, of the letter in your packet.

To receive your special offer, enter the 3-digit offer code:

Account Funding

Accounts open for more than six months, with a total liquidation value below $2,000, are subject to a $15 quarterly account maintenance fee. All accounts with at least four executed trades in the previous six months are excluded from this fee. All IRA and Beneficiary accounts are also excluded from this fee.

How will this account be funded?
☐ Check
☐ Wire
☐ Transfer assets into this new Ameritrade ® account
☐ Stock Certificates

Account Type

How would you like to register your account?
○ Joint Tenants with Rights of Survivorship
○ Joint Tenants in Common
○ Joint Community Property (AZ, CA, ID, LA, NM, NV, TX, WA, WI residents only)
○ Joint Tenants by the Entireties

Account Selection

What type of account would you like to open?
○ Cash
● Cash & Margin
○ Cash, Margin, & Option

Shareholder Information

Photo Imaging News
International Edition

Table of Contents
Volume 19, Number 11
Week of May 20, 2002
This issue contains:

- **Digital Imaging Marketing**
  -Reviews from Digital Imaging Marketing Association Meetings
- **Other DIMA Programs**
  -Further reviews from Digital Imaging Marketing Association meetings
- PMA Digital Printer Round-Up (2)
  - More on digital printers shown at PMA
- Brief News of the Photofinishing Industry
  - Fujifilm Wins two separate One-Time-Use Camera court cases
  - Concord Camera controller embezzles $1.25 million.
  - In the U.S.A., Agfa Consumer Imaging is restructuring and renaming its equipment service organization.
  - In the U.K., Pixology Ltd. will provide an Internet-based photosharing and

Figure 6: Left: Fixed size help window (in the upper left of the screen shot) from Ameritrade's web site. Right: Fixed size window without scroll bars from Photo Imaging News web site.

age account at Ameritrade, when the user requests help on selecting the appropriate account type, the site pops up a fixed size window that is 310x330 pixels. On a monitor of 1280x1024 pixels, this means that the web site allows the user to employ only 7.8% of the available pixels to consult the help.

As Figure 6 shows, it could be worse. Steve Brennen found that a pop-up window at the Photo Imaging News web site is not only fixed in size and too small to display all the text it contains, it also lacks scroll bars! It is left as an exercise for the reader how a visitor to this web site can view all the text in the window.

Some fixed size windows appear to be motivated by artistic considerations. An example appears in Figure 7, where the page design limits most content to a window of about 405x275 pixels, i.e., about

[†] Adherents to this conspiracy cannot help but notice that the license agreement for Windows Update not only comes in a fixed size window, the text is also unselectable, thus disabling the obvious workaround of copying the text to a less user hostile application.

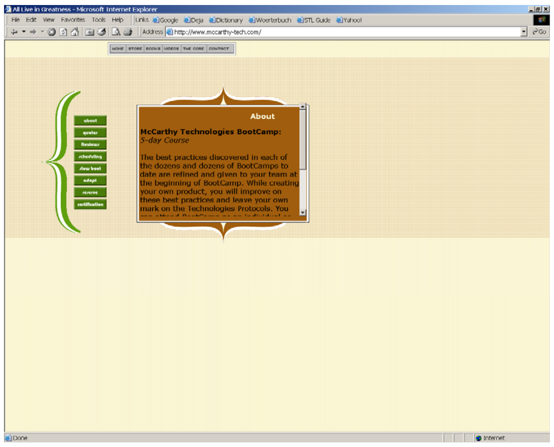

All Live in Greatness - Microsoft Internet Explorer
http://www.mccarthy-tech.com/
About
McCarthy Technologies BootCamp:
5-day Course
The best practices discovered in each of the dozens and dozens of BootCamps to date are refined and given to your team at the beginning of BootCamp. While creating your own product, you will improve on these best practices and leave your own mark on the Technologies Protocols.

Figure 7: Fixed size window keyhole at McCarthy Technologies' web site.

8.5% of the space available on a 1280x1024 display. This may be art, but I find it hard to believe that it leads to a truly satisfying user experience.

2.4 The Maximum Size Window Keyhole

Windows that may vary up to some predetermined maximum size are more flexible than windows of fixed size, but from a keyhole point of view, they lead to the same kinds of problems. Whether a window is fixed in size or limited in size makes little difference if it is smaller than the user desires. Both thwart the user's will. It is possible that windows with a maximum size lead to greater frustration, because users know that the software *can* resize the window, it simply refuses to do it beyond some arbitrary point.

Adobe Acrobat 4 offers a particular interesting maximum size window keyhole. Acrobat allows users to add notes to PDF documents, but no notes window may exceed 432x288 pixels. Figure 8 shows how

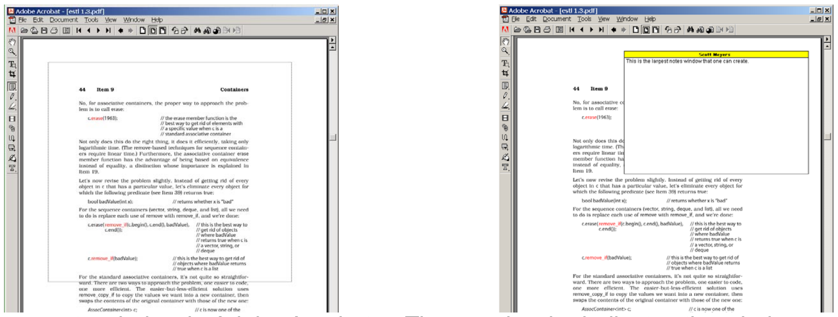

Adobe Acrobat - [estl 1.3.pdf]
Scott Meyers
This is the largest notes window that one can create.

Figure 8: Specifying a note window in Adobe Acrobat 4. The grey border indicates the window area swept out by the user (left), but the resulting window is smaller (right).

users employ the mouse to sweep out the area of the note window, but this sweeping fails to enforce the maximum window size. This allows users to specify larger note windows than the program supports. When this happens, the note window comes up at the maximum allowed size, a result sure to surprise some users. (I write this with certainty, because it happened to me, and when it did, I was surprised.)

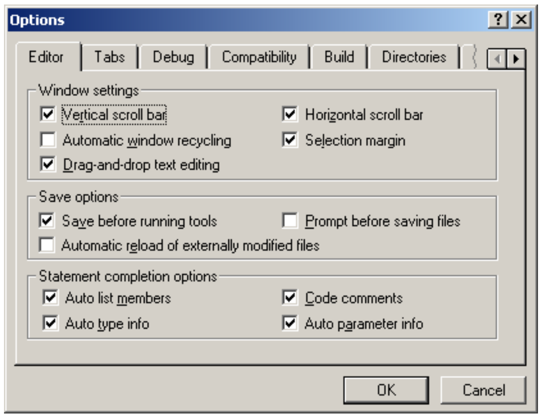
This is another case where eliminating a keyhole (the fixed size note window) would improve the overall user interface. If the note window were not arbitrarily limited to no more than 288x432 pixels, there would be no possibility of inconsistency between the part of the program responsible for providing feedback while a user sweeps out the desired window (clearly, that part of the program fails to enforce the 288x432 pixel limitation in Acrobat 4) and the part of the program responsible for implementing the note window. Keyholes can lead to dependencies between parts of programs that would better go uncoupled.

Incidentally, the restriction of 288x432 pixels is not quite as arbitrary as it may seem. An Adobe employee reminded me that there are 72 points per inch, so a window of size 288x432 points is 4x6 inches. The true underlying keyhole is thus 4x6 inches. This explains the numbers themselves, but it fails to explain the reason for the keyhole in the first place. What is the rationale behind limiting a note window to 4x6 inches? Why should a user be prohibited from creating a larger note window?

2.5 The Fixed Size Dialog Keyhole

Dialog boxes are generally fixed in size by definition. When they need to display more information than will fit in the allotted space, a common way to approach the problem is to divide the information into different tab sheets. There is nothing inherently wrong with this approach, but trouble can arise when the fixed size of the dialog is too small to allow all the tabs to be visible simultaneously. That can lead to dialog boxes with scroll controls for the tabs. An example from Microsoft's Visual C++ 6 is shown in Figure 9.

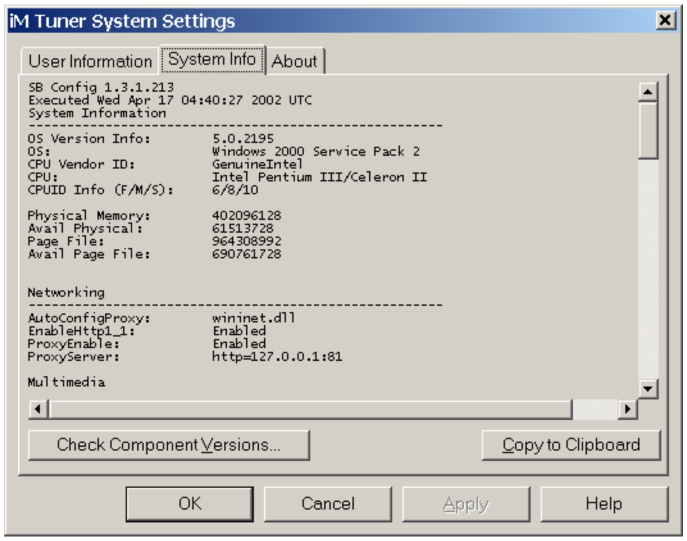
The wisdom of choosing a fixed size dialog to hold more tab sheets than easily fit is debatable, but it is probably fair to say that the need to add scroll bars *within a tab sheet* is always a sign of bad design. What *can* be done *is* done, however. Figure 9 shows a particularly bad example from Sonicbox's iMTuner: a tab sheet with both vertical *and* horizontal scroll bars.



Window settings
- [✓] Vertical scroll bar
- [✓] Horizontal scroll bar
- [ ] Automatic window recycling
- [✓] Selection margin
- [✓] Drag-and-drop text editing

Save options
- [✓] Save before running tools
- [ ] Prompt before saving files
- [ ] Automatic reload of externally modified files

Statement completion options
- [✓] Auto list members
- [✓] Code comments
- [✓] Auto type info
- [✓] Auto parameter info



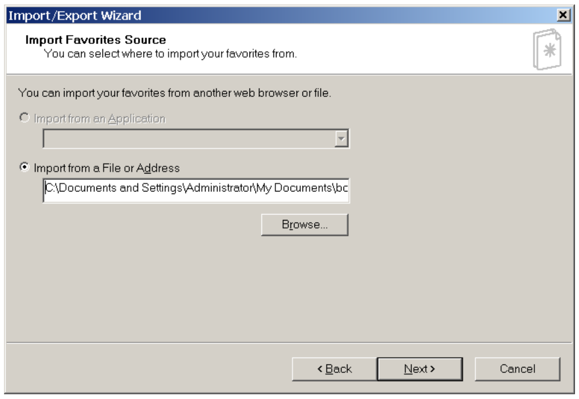
```
SB Config 1.3.1.213
Executed Wed Apr 17 04:40:27 2002 UTC
System Information
-------------------------------------------------------
OS Version Info:        5.0.2195
OS:                     Windows 2000 Service Pack 2
CPU Vendor ID:          GenuineIntel
CPU:                    Intel Pentium III/Celeron II
CPUID Info (F/M/S):     6/8/10

Physical Memory:        402096128
Avail Physical:         61513728
Page File:              964308992
Avail Page File:        690761728


Networking
-------------------------------------------------------
AutoConfigProxy:        wininet.dll
EnableHttp1_1:          Enabled
ProxyEnable:            Enabled
ProxyServer:            http=127.0.0.1:81

Multimedia
```

Figure 9: Fixed size dialog from Microsoft's Visual C++ 6 with more tab sheets than can be displayed (left) and a tab sheet with scroll bars from Sonicbox's iMTuner (right).

In both of these cases, the fundamental underlying problem is that the software designer is trying to put information into a space too small to contain it. Keyholes are the invariable result. This suggests that the need to introduce keyholes into a user interface is a general sign of bad design, that when designers are confronted with a keyhole, they should rethink their approach to the interface they are developing. For example, instead of trying to shoehorn more tab sheets than will fit onto a fixed size dialog, consider using a variable-size window to hold the tab sheets. Instead of adding scroll bars to a tab sheet, use a variable size window to hold the information, possibly in conjunction with a button on the tab sheet to bring up the window.

2.6 The Fixed Width Edit Control Keyhole

One of the most annoying of all keyholes is the fixed width edit control keyhole, whereby a single-line edit field is too narrow to show all the textual information in the field it displays. For example, Figure 10 shows how the edit control in the Import/Export Favorites dialog for Microsoft's Internet Explorer 6 is too narrow to hold even the default path with which it is automatically initialized. This particular example is irritating for at least two reasons. First, a user must use the arrow keys to scroll the default path to see if it is the one


Import/Export Wizard

Import Favorites Source
You can select where to import your favorites from.

You can import your favorites from another web browser or file.

Import from an Application

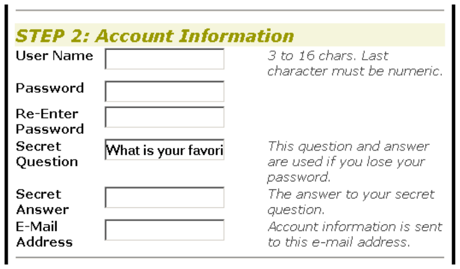
[✓ Import from a File or Address]

C:\Documents and Settings\Administrator\My Documents\bc

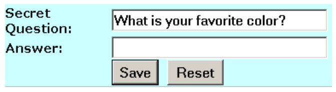
Figure 10: A fixed width edit control keyhole.

s/he wants, an irritation right off the bat. Second, the edit control itself is much narrower than the (fixed width) dialog that contains it. This is especially galling, because, on my system, the edit control displays 55 characters, while the full default path in the edit control is only 10 characters longer. That means that the full default path would fit if the width of the edit control had not been gratuitously limited. Of course, making the edit control as wide as the dialog box wouldn't help if the user wished to specify a path longer than the width of the dialog, a situation that is not uncommon when one wants to import or export favorites to or from a directory on network drive. In essence, this example demonstrates a theme we will return to later: nested frustration due to keyholes within keyholes. (In this case, a fixed width edit control within a dialog box, itself a fixed size window.)

Fixed width edit controls are as common on web pages as they are in desktop applications. As usual, they are annoying, and they also tend to give rise to ancillary user interface difficulties. Figure 11 shows a keyhole-related user interface inconsistency at the OneSuite.com web site. When a user initially signs up for


STEP 2: Account Information

User Name		3 to 16 chars. Last character must be numeric.
Password		
Re-Enter Password		
Secret Question	What is your favori	This question and answer are used if you lose your password.
Secret Answer		The answer to your secret question.
E-Mail Address		Account information is sent to this e-mail address.



Secret Question:	What is your favorite color?
Answer:	

Figure 11: Inconsistent fixed width edit controls for (left) initially specifying a secret question at the OneSuite.com web site and (right) editing the question later.

an account, s/he must specify a secret question. The edit control for this question displays only about 17 characters on my system, but the question itself is limited to 32 characters. Once an account is set up, however, the owner can edit the secret question, and the interface for this operation employs an edit control that displays about 31 characters. When revising the question, there appears to be no significant limit on its length; I was able to specify a question of 1000 characters.

The OneSuite example again demonstrates that where there are keyholes, there are opportunities for inconsistencies and confusion. Should the site display 31 or 17 characters of the secret question? Should the question be limited in length or should it not? Eliminate the keyholes, and these questions are easy to answer:

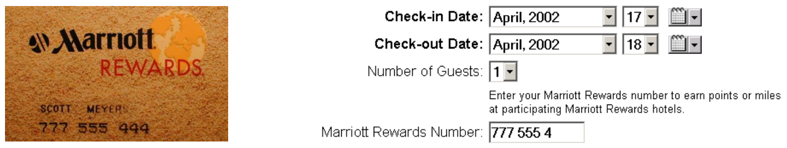
- *How much of the question should be displayed?* Ideally, as much as will fit on the screen, but certainly at least as much as will fit on one line in a window that can be resized to at least the width of the screen.

- *How long should the question be allowed to be?* As long as can be stored without undue difficulty, e.g., as long as the database will allow a string to be. (In Section 6, I discuss the interaction of databases and field size limitations.)

If you consistently eliminate keyholes or make them as large as possible (bigger keyholes let you see *more*), you reduce the friction your software imposes on its users and you reduce the opportunities for different parts of your software to behave differently.

2.7 The Lying Fixed Width Edit Control Keyhole

A particularly pernicious form of fixed width edit control is the lying variant, i.e., the fixed width edit control that appears to be one size but is actually smaller. I had a memorable encounter with this kind of keyhole at the web site for Marriott when I tried to enter my Marriott Rewards number in the web site's edit control for that information. Figure 12 shows my Marriott Rewards card with its nine-digit number for-



Check-in Date: April, 2002 17
Check-out Date: April, 2002 18
Number of Guests: 1
Enter your Marriott Rewards number to earn points or miles at participating Marriott Rewards hotels.
Marriott Rewards Number: 777 555 4

Figure 12: My Marriott Rewards membership card showing my (edited) membership number (left) and the result of trying to enter it into the lying fixed width edit control at the Marriott web site (right).

matted as three space-separated groups of three digits each. At the web site, I tried to type the exact same thing into the edit control, but I was unable to enter more than seven digits, even though there was ample room in the control for all nine. Eventually I figured out that (1) the browser was limiting me to nine characters, (2) the spaces were included in the nine characters allowance, and (3) the width of the edit control did not correspond to the maximum number of digits allowed.

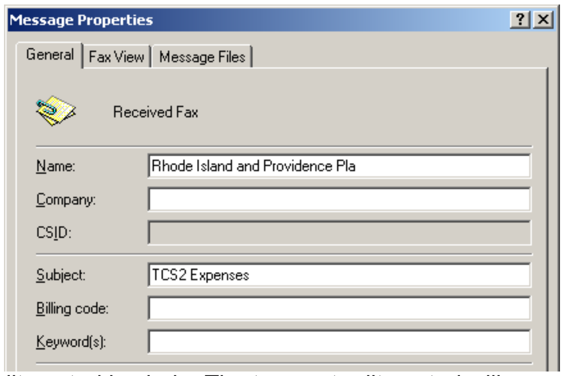
The reason for the nine-character limitation is almost certainly to allow client-side validation of the input, but, just as we saw earlier, this is misguided, because the server must validate the data anyway. Even if the client is to perform some preliminary validation, it shouldn't count spaces as part of the number, because they clearly aren't. In this case, the web site's failure to accept spaces leads to an inconsistency with the formatting of the number on the membership card, something sure to irritate some users. Again, I speak from personal experience.

(In general, web sites that demand long strings of digits — such as credit card numbers or telephone numbers — should always allow common separators such as spaces and dashes. They make it much easier for the humans entering the data, and they are trivial for the software to remove during validation.)

If the edit control eliminated the nine-character keyhole, not only would it allow a user to format the entry in a way s/he finds natural (e.g., copying it off the membership card), it would also eliminate the lying aspect of the fixed width edit control. Once again, eliminating a keyhole would improve other aspects of the user interface.

Desktop applications also employ lying fixed width edit controls. Figure 13 shows a dialog from Symantec's Winfax Pro 10, where I am attempting to specify that a fax I've received came from Rhode Island. In an interesting quirk of fate, the smallest state in the country boasts the longest official name, "Rhode Island and Providence Plantations," and it is this official name I would like to enter. Unfortunately, though the edit control appears to be willing to accept at least 49 characters, it stops accepting input once 31 characters have been entered. In this case, the reason for the lying fixed with edit control appears to be entirely aesthetic. Fax subject lines are allowed to fill the entire edit control, so the designer of this tab sheet apparently felt that it was more important for all the edit controls to look the same than for the width of each edit control to correspond to the maximum amount of data that could be entered.

Of course, if the underlying field values were unlimited in length (thus eliminating a keyhole we shall discuss shortly), there would be no such thing as a lying fixed width edit control keyhole, because edit con-



Name: Rhode Island and Providence Pla
Company:
CSID:
Subject: TCS2 Expenses
Billing code:
Keyword(s):

Figure 13: A lying fixed width edit control keyhole. The topmost edit control will accept no more input than is shown.

trols would never accept fewer characters than they appear to be willing to hold. Once again, elimination of a keyhole in one part of a program would eliminate interface difficulties in other parts.

2.8 The Restricted Range Keyhole

When a user interface component refuses to display part of its natural range, the result is a restricted range keyhole. I will present an example of such a keyhole in Section 5, so I defer discussion of this keyhole type until then.

3 Invisible Keyholes

Visible keyholes are common and are easy to, well, see, but equally common and equally troublesome keyholes are *invisible* — not parts of graphical user interfaces. Being invisible doesn't make them undetectable, however. Far from it. Invisible keyholes assert their existence in myriad ways, some merely frustrating, others substantially more serious. In this section, I introduce four types of invisible keyhole.

3.1 The Too Few Bits Keyhole

A too few bits keyhole arises when a developer has allotted too few bits to hold a value. My favorite manifestation of this problem arose when I ran into trouble installing Adaptec's Easy CD Creator 4†. In consulting the web site for this product, I came upon the following truly astonishing advice:

> If you have Easy CD Creator 4.00 and you have more than 8Gb of free space on the drive or partition where Windows is installed, please create TEMP directories and fill up space in these until less than 8Gb of free space remains on that disk or partition. (A quick way to do this is to copy all the files on the Easy CD Creator installation disc into your temp directories!)

Let us be utterly clear about this: the problem is *too much* free disk space. What makes the problem especially interesting is the maximum free space value: 8GB. If the value were 4GB, we'd realize that this is the maximum value representable in a 32 bit number, and we might shrug it off as an understandable artifact of a developer who takes a 32 bit world for granted. But the value is 8GB. 8GB corresponds to 33 bits! I am at a loss to explain how a well-meaning programmer can accept a value requiring 33 bits but cannot accept a value requiring 34 bits. (Frankly, I'd expect that if the software offers 33 bits, it would offer 64, or at least 48.)

The strange value of this keyhole (33 bits) is interesting, but more germane to me as I was installing the software was the fact that I was installing it on my new almost-completely-empty *80GB* hard drive. Even though the Adaptec web site helpfully offered advice on how to fill 90% of my hard drive, that was little consolation for the realization that had their keyhole been a more reasonable size, I would never have run

† This product is now owned by Roxio.

into the problem in the first place. In Section 6, I discuss how to determine what a “reasonable size” is likely to be when it comes to allotting bits to values.

3.2 The Unnecessarily Signed Int Keyhole

Sometimes a contributing factor towards there being too few bits available is that one of them — the sign bit — has been squandered. For example, I run a program called Naviscope that, among other things, filters ads out of web pages. Naviscope keeps track of how many ads it has eliminated, and I watched with great interest as it filtered out ten thousand ads, then twenty thousand ads, then thirty thousand ads, then thirty two thousand five hundred and sixty five ads ... and then *negative* thirty two thousand five hundred and sixty six ads. Figure 14 shows a Naviscope status message some time thereafter.

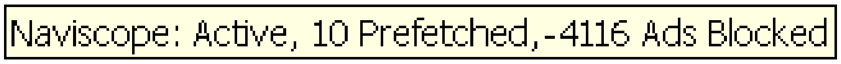

Naviscope: Active, 10 Prefetched,-4116 Ads Blocked

Figure 14: Evidence of an unnecessarily signed int keyhole.

As a general rule, when counting things, the count cannot go below zero. When that is the case, it is better design to use an unsigned integer to store the count. For one thing, it directly expresses the fact that negative values are not valid. For another, it avoids the creation of unnecessarily signed int keyholes.

In the particular case of Naviscope, the unnecessarily signed int keyhole simply delayed the inevitable manifestation of a too few bits keyhole, because a 16 bit value, even when unsigned, is insufficient to represent the number of blocked ads for an active user of the world wide web over a period of several months. Once again I speak from personal experience. Given that Naviscope is a 32 bit application running on a a 32 bit operating system that supports virtual memory, the choice of a 16 bit counter to keep track of how many ads have been blocked is penny wise and pound foolish. We are thus confronted with another case of nested keyholes: an unnecessarily signed int keyhole inside a too few bits keyhole.

3.3 The Maximum Field Size Keyhole

A maximum field size keyhole arises when a limitation is imposed on the amount of data (often the number of characters) that may be stored for a particular field value. Designers typically rationalize the introduction of such keyholes into their software by arguing the imposed limit is “obviously big enough.”

My experience has been that limits that are “obviously big enough” almost always turn out to be too small. As an example, consider the following phrase from one of my books for C++ programmers:

Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator.

This phrase gives rise to several distinct index entries:

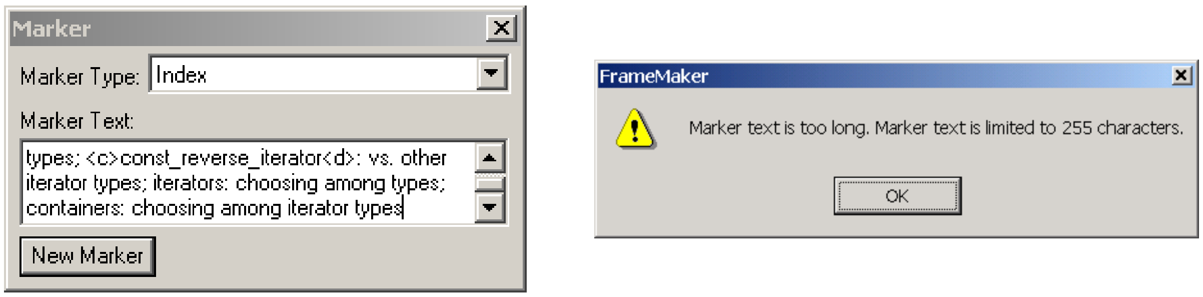
iterator: vs. other iterator types
const_iterator: vs. other iterator types
reverse_iterator: vs. other iterator types
const_reverse_iterator: vs. other iterator types
iterators: choosing among types
containers: choosing among iterator types

The book preparation program I use, Adobe’s FrameMaker 6, requires that markup be added in order for these entries to be typeset correctly. With markup added, the text for the index entries looks like this:

```
<$startrange><c>iterator<d>: vs. other iterator types; <c>const_iterator<d>: vs. other iterator
types; <c>reverse_iterator<d>: vs. other iterator types; <c>const_reverse_iterator<d>: vs.
other iterator types; iterators: choosing among types; containers: choosing among iterator types
```

Figure 15 shows the fixed size window keyhole one must use to add this “index marker text” to the book, and it also shows the error message that resulted when I attempted to add the text. Alas, the marker text for my index entries exceeds the 256 character maximum field size keyhole imposed by FrameMaker.

The point of this example is to demonstrate that users who run afoul of field size keyholes are often doing something reasonable, though it may not have been anticipated by the developers of the software. That’s



Index

types; <c>const_reverse_iterator<d>: vs. other iterator types; iterators: choosing among types; containers: choosing among iterator types

Marker text is too long. Marker text is limited to 255 characters.

Figure 15: Fixed size window keyhole in Adobe's FrameMaker 6 through which one must add index marker text (left) and the result of submitting that text (right).

the first problem with maximum field size keyholes: they frustrate users who are trying to do reasonable things. Often such users are among the software's most sophisticated, hence among its most important.

The second problem is that field sizes that are "obviously big enough" today tend to become obviously insufficient tomorrow. Consider, for example, the increase in zip code length from five to nine digits, the need to use ten-digit dialing for telephone numbers in places where historically seven digits sufficed, and the Y2K shift from two to four digits to hold the year.

Microsoft's Visual C++ 6 debugger limits the size of program identifiers to 256 characters under the assumption that no programmer would choose an identifier longer than that. During the lifetime of Visual C++ 6, however, an aspect of C++ known as templates has become increasingly popular. Templates force the C++ compiler to generate unique identifiers, and the algorithm used to generate identifiers often yields names that exceed 256 characters. As a result, tens or hundreds of thousands of Visual C++ users now routinely encounter warnings such as this:[†]

```
warning C4786:
'std::rb_tree<CAiSpanningTree<State,std::less<State>>::TransClosureNode, CAiSpan-
ningTree<State,std::less<State>>::TransClosureNode,std::ident<Cai Span-
ningTree<State,std::less<State>>::TransClosureNode,CAiSpanningTree<S
tate,std::less<State>>::TransClosureNode>,std::less<CAiSpanningTree<Stat
e,std::less<State>>::TransClosureNode>>' : identifier was truncated to '255' characters in the
debug information
```

The warnings themselves are irritating and, for development shops where programs are required to compile without warnings, troublesome, but a more serious ramification of the underlying 256 character keyhole is that it prevents users from using the debugger to examine variables with the long names. That is, the compiler itself generates identifiers that the debugger cannot handle!

Once again, we see how a keyhole in one part of a system can interact with other parts of the system to yield behavior that is irritating, frustrating, and limiting. Once again, had the system avoided introduction of the keyhole in the first place, the derivative problems would not have arisen.

Some field size keyholes are imposed to enforce semantic constraints, but even this can be misguided, because it can preclude the possibility of entering meaningful, but unconventional, data values that are longer than expected. Alan Cooper has proposed, for example, that it's not unreasonable to allow the value "213/310?" as an area code in places where the exact area code is not known but is not yet critical. Anticipating objections to this idea, he writes:

> I can hear the squeals of protest: "But, but, but the area code field is only big enough for three digits! I can't fit "212/310?" into it!" Gee, that's too bad. You mean that rendering the user interface of your program in terms of the underlying implementation model—a rigidly fixed field

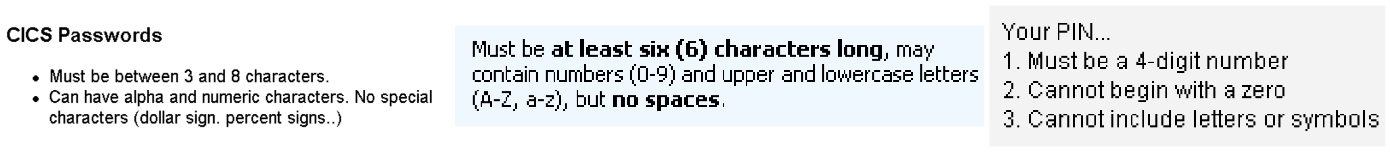
† This example warning is taken from a Microsoft support article on the topic, http://support.microsoft.com/default.aspx?scid=kb;EN-US;q167355.

width— forces you to reject natural human behavior in favor of obnoxious, computer-like, inflexibility supplemented with demeaning error messages?[†]

Maximum field size keyholes frustrate users, decrease the robustness of software over time, give rise to inconsistencies in different parts of a system, and preclude unconventional "human friendly" data values. Eliminating them is always a good idea.

3.4 The Restricted Domain Keyhole

When some elements of the natural input domain are excluded, the result is a restricted domain keyhole. The most common scenario is when a textual input is required, but some characters are forbidden. This situation arises with great regularity at web sites that require users to identify themselves with usernames and passwords, because most web sites limit not only the length of the username and password (maximum field size keyholes), they also restrict the characters that may be used in the password. Figure 16 shows three examples.



CICS Passwords

- Must be between 3 and 8 characters.
- Can have alpha and numeric characters. No special characters (dollar sign. percent signs..)

Must be **at least six (6) characters long**, may contain numbers (0-9) and upper and lowercase letters (A-Z, a-z), but **no spaces**.

Your PIN...
1. Must be a 4-digit number
2. Cannot begin with a zero
3. Cannot include letters or symbols

Figure 16: Restricted domain keyholes for passwords at web sites from the University of Arkansas (left), Hotmail (center), and Ameritrade (right).

As the figure indicates, the University of Arkansas prohibits "special characters" such as dollar and percent signs, Hotmail disallows spaces, and Ameritrade restricts PINs to four-digit numbers, the first of which must not be a zero. None of these restrictions makes much sense.

We begin with the observation that there is nothing "special" about dollar or percent signs. If we make the reasonable assumption that we are interested only in printable characters[†] represented as values in ASCII or EBCDIC, no character that can be represented in both encodings is any more special than any other. For passwords, the characters have no meaning; they aren't interpreted in any way. That being the case, there is no way that the sequence "A%$" can be considered special compared to, say, "ABC".

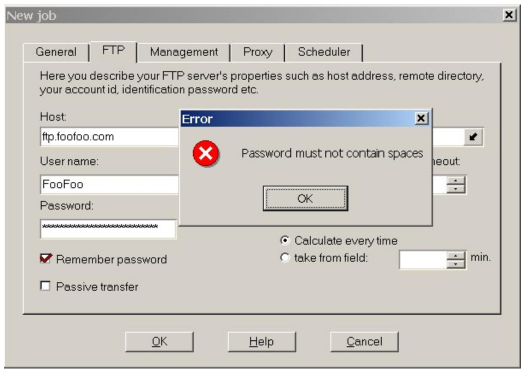
By the same reasoning, spaces aren't special either. Unfortunately, the notion that spaces are somehow "different" is widespread. It is not universal, however, and this can lead to problems that border on comical. Figure 17 shows what happened when I tried to use Rifco's DC SmartFTP to communicate with my web server.

DC SmartFTP rejected my password, because it "may not contain spaces". However, the password at my web server *does* contain spaces. This keyhole is particularly galling, because SmartFTP's only job is to convey the password to the FTP site. There's no reason for SmartFTP to take even the most cursory glance at it. Getting an error message from SmartFTP about my password would be like getting a message from the post office that they were rejecting a letter I'd mailed, because they didn't like what I wrote.

I suspect that the restrictions on Ameritrade PINs are inherited from the PINs Ameritrade uses in its telephone interface. That would explain why PIN values must consist of digits only, and it would likely explain why the first digit must not be zero. (Presumably entering a leading zero would summon an operator or would otherwise throw the system out of PIN-entry mode.) I can understand the desire for uniformity between two interfaces (telephone and web) to the same underlying system, but it makes no sense to me to restrict the comparatively rich input capabilities available to web users (typically a keyboard) to those

[†] Alan Cooper, *About Face*, IDG Books, 1995, p. 430.

[†] The restriction to printable characters isn't really necessary, but it allows us to ignore the fact that some unprintable characters are interpreted by operating systems or web browsers and are hence difficult for web users to enter into password fields. Problematic characters include ESC, ^C, ^Z, etc.



Here you describe your FTP server's properties such as host address, remote directory, your account id, identification password etc.

Host: ftp.foofoo.com

User name: FooFoo

Password: ***********************

[✓] Remember password

[ ] Passive transfer

(●) Calculate every time

( ) take from field: min.

Error

Password must not contain spaces

Figure 17: Restricted domain keyhole in Rifco's DC SmartFTP.

available to users having only a telephone keypad. An alternative would have been to *allow* use of a telephone PIN as a web site password, but not require it.

Many software systems impose length limitations (maximum field size keyholes) on passwords. This is misguided. Passwords should be hard to guess and easy to remember, and one of the easiest ways to facilitate good passwords is to stop calling them pass*words* and start encouraging people to use phrases instead. Consider, for example, this password:

Mary had a little lamb.

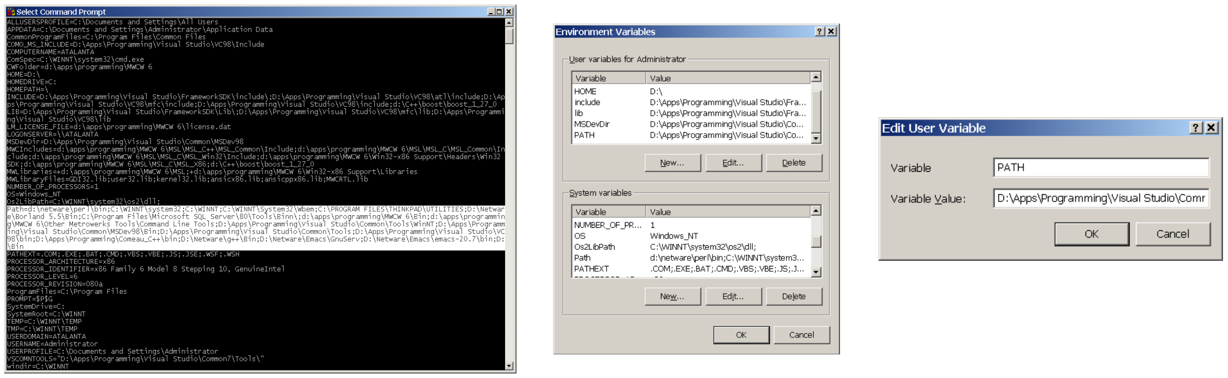
It's easy to remember, but it won't be found by a brute force dictionary search. It's even got both upper and lower case letters. Imposing a restricted domain keyhole that prohibits spaces precludes passwords like this. Furthermore, imposing a maximum field size keyhole of fewer than 23 characters precludes it, too. On this latter point, Unix systems adopt an interesting approach. Unix passwords are hashed before validation, and the length of the input into the hash algorithm is restricted, e.g., to 16 or 32 characters. However, the passwords users type in are not limited to this length. Instead, if they are too long, they are truncated before being hashed. On a system with a 16 character limit on password length, then, a user would be allowed to enter the entire phrase above, but the input to the hashing algorithm would be "Mary had a littl". Users are thus allowed to use easy-to-remember phrases, but the length-restricted hashing algorithm (for which I assume there is a legitimate technical justification) continues to be viable. As long as the truncating behavior of the password process is made clear to users, this strikes me as a reasonable design that largely shields users from keyholes.

4 Combinations of Keyholes

Keyholes irritate and frustrate users in a variety of ways, but perhaps the single biggest problem with keyholes is their ubiquity. Encountering the occasional edit control keyhole might not be so bad were it not that several often occur together in the same fixed size window keyhole and interact with underlying restricted field width keyholes (sometimes in conjunction with restricted domain keyholes). Working with software is a vexing experience when you are gratuitously prevented from seeing everything you want to see and from saying everything you want to say.

We have seen several examples of how software often confronts users with multiple keyholes simultaneously. For example, Figure 1 shows how fixed width edit control keyholes are embedded within fixed width web page keyholes, and Figure 11 depicts multiple fixed width edit control keyholes atop restricted field width keyholes. It is worth looking at three additional examples, however, if for no other reason than to make clear just how bad things are.

Figure 18 highlights the value of my PATH environment variable and the tools Microsoft's Windows 2000 gives me to edit it. My PATH is 778 characters long. It is formed by concatenating my user PATH to my system PATH. To edit my PATH, I must open the Environment Variables dialog, a fixed size window keyhole inside which are nested two scrolling fixed size window keyholes (one for user environment variables, one for system environment variables). Once I've navigated to and selected the PATH component I



```
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
CommonProgramFiles=C:\Program Files\Common Files
COMO_MS_INCLUDE=D:\Apps\Programming\Visual Studio\VC98\Include
COMPUTERNAME=ATALANTA
ComSpec=C:\WINNT\system32\cmd.exe
CWFolder=d:\apps\programming\MWCW 6
HOME=D:\
HOMEDRIVE=C:
HOMEPATH=\
INCLUDE=D:\Apps\Programming\Visual Studio\FrameworkSDK\include\;D:\Apps\Programming\Visual Studio\VC98\atl\include;D:\Ap
ps\Programming\Visual Studio\VC98\mfc\include;D:\Apps\Programming\Visual Studio\VC98\include;d:\C++\boost\boost_1_27_0
LIB=D:\Apps\Programming\Visual Studio\FrameworkSDK\Lib\;D:\Apps\Programming\Visual Studio\VC98\mfc\lib;D:\Apps\Programmi
ng\Visual Studio\VC98\lib
LM_LICENSE_FILE=d:\apps\programming\MWCW 6\license.dat
LOGONSERVER=\\ATALANTA
MSDevDir=D:\Apps\Programming\Visual Studio\Common\MSDev98
MWCIncludes=d:\apps\programming\MWCW 6\MSL\MSL_C++\MSL_Common\Include;d:\apps\programming\MWCW 6\MSL\MSL_C\MSL_Common\In
clude;d:\apps\programming\MWCW 6\MSL\MSL_C\MSL_Win32\Include;d:\apps\programming\MWCW 6\Win32-x86 Support\Headers\Win32
SDK;d:\apps\programming\MWCW 6\MSL\MSL_C\MSL_X86;d:\C++\boost\boost_1_27_0
MWLibraries=+d:\apps\programming\MWCW 6\MSL;+d:\apps\programming\MWCW 6\Win32-x86 Support\Libraries
MWLibraryFiles=GDI32.lib;user32.lib;kernel32.lib;ansicx86.lib;ansicppx86.lib;MWCRTL.lib
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Os2LibPath=C:\WINNT\system32\os2\dll;
Path=d:\netware\perl\bin;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem;C:\PROGRAM FILES\THINKPAD\UTILITIES;D:\Netwar
e\Borland 5.5\Bin;C:\Program Files\Microsoft SQL Server\80\Tools\Binn\;d:\apps\programming\MWCW 6\Bin;d:\apps\programmin
g\MWCW 6\Other Metrowerks Tools\Command Line Tools;D:\Apps\Programming\Visual Studio\Common\Tools\WinNT;D:\Apps\Programm
ing\Visual Studio\Common\MSDev98\Bin;D:\Apps\Programming\Visual Studio\Common\Tools;D:\Apps\Programming\Visual Studio\VC
98\bin;D:\Apps\Programming\Comeau_C++\bin;D:\Netware\g++\Bin;D:\Netware\Emacs\GnuServ;D:\Netware\Emacs\emacs-20.7\bin;D:
\Bin
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 8 Stepping 10, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=080a
ProgramFiles=C:\Program Files
PROMPT=$P$G
SystemDrive=C:
SystemRoot=C:\WINNT
TEMP=C:\WINNT\TEMP
TMP=C:\WINNT\TEMP
USERDOMAIN=ATALANTA
USERNAME=Administrator
USERPROFILE=C:\Documents and Settings\Administrator
VSCOMNTOOLS="D:\Apps\Programming\Visual Studio\Common7\Tools\"
windir=C:\WINNT
```

Variable	Value
HOME	D:\
include	D:\Apps\Programming\Visual Studio\Fra...
lib	D:\Apps\Programming\Visual Studio\Fra...
MSDevDir	D:\Apps\Programming\Visual Studio\Co...
PATH	D:\Apps\Programming\Visual Studio\Co...

Variable	Value
NUMBER_OF_PR...	1
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\netware\perl\bin;C:\WINNT\system3...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.J...

Variable: PATH

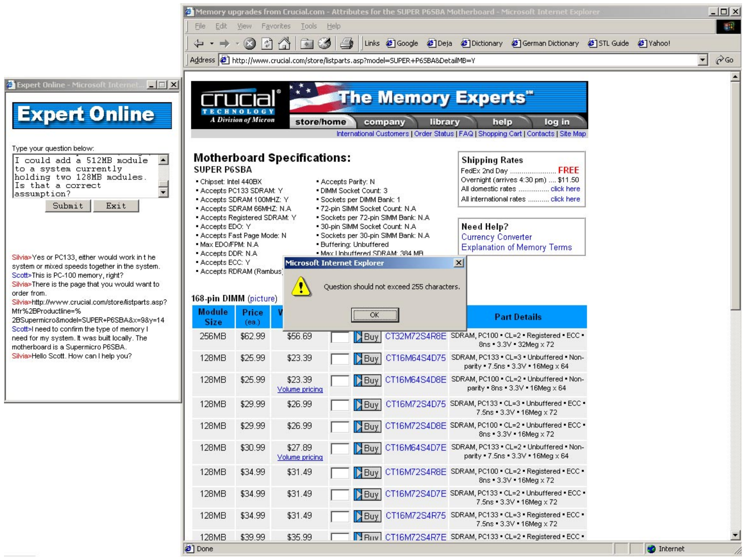
Variable Value: D:\Apps\Programming\Visual Studio\Comr

Figure 18: Left: DOS box showing my environment variables with my PATH highlighted. Middle: Windows 2000 Environment Variables dialog. Right: dialog for editing a specific variable.

wish to edit, I must work with a fixed size window keyhole containing a fixed size edit control keyhole. This latter keyhole displays only 39 characters at a time—a mere 5% of the total PATH length. It is literally useless. When I need to edit my PATH (not a terribly rare occurrence), I copy the text from the edit control, modify it in a text editor, then copy it back into the edit control. Thanks to the many keyholes my operating system puts between me and my environment variables, editing my PATH is significantly harder in Windows 2000 than it was in DOS.

Figure 19 is a screen shot from a recent session I had at the Crucial Technology web site. I was shopping for memory, but I had some questions, so I availed myself of the "Expert Online" feature, which allowed me to textually chat with a company representative. The web site directed me to "type your question below," so I did. When I tried to submit it, however, it was rejected. The cause? I had exceeded the 255 character limit for questions. Clearly, nobody would ever want to ask a question any longer than that.

Figure 19 also shows that Crucial's web site is a fixed width web page keyhole, and the edit control into which I typed my overly lengthy question is an edit control keyhole. Of the 255 character maximum imposed by the maximum field size keyhole, this window displays no more than about 140.



Type your question below:

I could add a 512MB module to a system currently holding two 128MB modules. Is that a correct assumption?

Silvia>Yes or PC133, either would work in t he system or mixed speeds together in the system.
Scott>This is PC-100 memory, right?
Silvia>There is the page that you would want to order from.
Silvia>http://www.crucial.com/store/listparts.asp?Mfr%2BProductline=%2BSupermicro&model=SUPER+P6SBA&x=9&y=14
Scott>I need to confirm the type of memory I need for my system. It was built locally. The motherboard is a Supermicro P6SBA.
Silvia>Hello Scott. How can I help you?

Memory upgrades from Crucial.com - Attributes for the SUPER P6SBA Motherboard - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Links
Google
Deja
Dictionary
German Dictionary
STL Guide
Yahoo!

Address http://www.crucial.com/store/listparts.asp?model=SUPER+P6SBA&DetailMB=Y

Motherboard Specifications:
SUPER P6SBA

- Chipset: Intel 440BX
- Accepts PC133 SDRAM: Y
- Accepts SDRAM 100MHZ: Y
- Accepts SDRAM 66MHZ: N.A
- Accepts Registered SDRAM: Y
- Accepts EDO: Y
- Accepts Fast Page Mode: N
- Max EDO/FPM: N.A
- Accepts DDR: N.A
- Accepts ECC: Y
- Accepts RDRAM (Rambus
- Accepts Parity: N
- DIMM Socket Count: 3
- Sockets per DIMM Bank: 1
- 72-pin SIMM Socket Count: N.A
- Sockets per 72-pin SIMM Bank: N.A
- 30-pin SIMM Socket Count: N.A
- Sockets per 30-pin SIMM Bank: N.A
- Buffering: Unbuffered
- Max Unbuffered SDRAM: 384 MB

Shipping Rates
FedEx 2nd Day .... FREE
Overnight (arrives 4:30 pm) $11.50
All domestic rates .... click here
All international rates .... click here

Need Help?
Currency Converter
Explanation of Memory Terms

Microsoft Internet Explorer

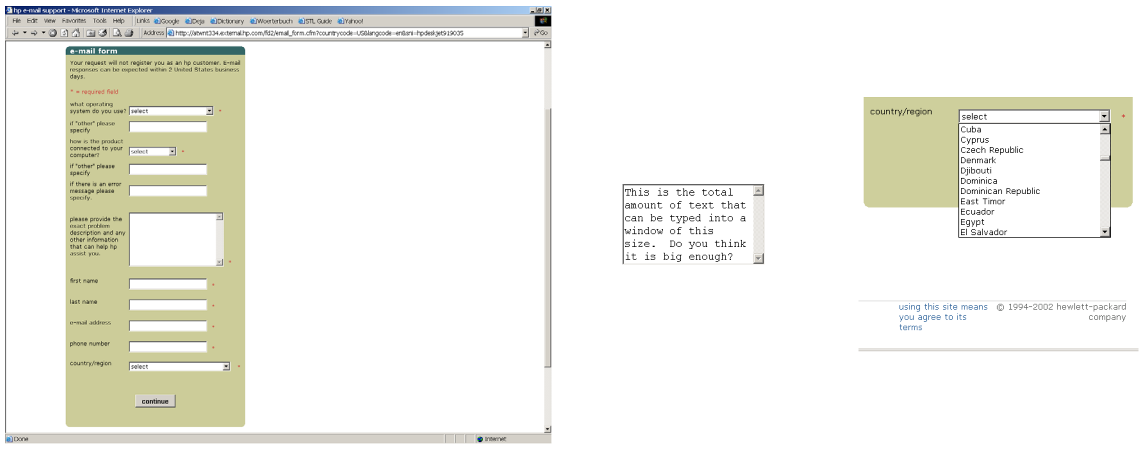
Question should not exceed 255 characters.

168-pin DIMM (picture)

Module Size	Price (ea.)	W				Part Details
256MB	$62.99	$56.69		Buy	CT32M72S4R8E	SDRAM, PC100 • CL=2 • Registered • ECC • 8ns • 3.3V • 32Meg x 72
128MB	$25.99	$23.39		Buy	CT16M64S4D75	SDRAM, PC133 • CL=3 • Unbuffered • Non-parity • 7.5ns • 3.3V • 16Meg x 64
128MB	$25.99	$23.39 Volume pricing		Buy	CT16M64S4D8E	SDRAM, PC100 • CL=2 • Unbuffered • Non-parity • 8ns • 3.3V • 16Meg x 64
128MB	$29.99	$26.99		Buy	CT16M72S4D75	SDRAM, PC133 • CL=3 • Unbuffered • ECC • 7.5ns • 3.3V • 16Meg x 72
128MB	$29.99	$26.99		Buy	CT16M72S4D8E	SDRAM, PC100 • CL=2 • Unbuffered • ECC • 8ns • 3.3V • 16Meg x 72
128MB	$30.99	$27.89 Volume pricing		Buy	CT16M64S4D7E	SDRAM, PC133 • CL=2 • Unbuffered • Non-parity • 7.5ns • 3.3V • 16Meg x 64
128MB	$34.99	$31.49		Buy	CT16M72S4R8E	SDRAM, PC100 • CL=2 • Registered • ECC • 8ns • 3.3V • 16Meg x 72
128MB	$34.99	$31.49		Buy	CT16M72S4D7E	SDRAM, PC133 • CL=2 • Unbuffered • ECC • 7.5ns • 3.3V • 16Meg x 72
128MB	$34.99	$31.49		Buy	CT16M72S4R75	SDRAM, PC133 • CL=3 • Registered • ECC • 7.5ns • 3.3V • 16Meg x 72
128MB	$39.99	$35.99		Buy	CT16M72S4R7E	SDRAM, PC133 • CL=2 • Registered • ECC •

Figure 19: Crucial Technology web site with fixed width web page keyhole, fixed size window keyhole, and error message arising from maximum field size keyhole.

Figure 20 is a wonder to behold. It shows an email contact form from the Hewlett-Packard product support web site, and, to the best of my recollection, it features the narrowest fixed width web page keyhole I have


hp e-mail support - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links Google Deja Dictionary Woerterbuch STL Guide Yahoo!
Address http://atwnt334.external.hp.com/fd2/email_form.cfm?countrycode=US&langcode=en&sni=hpdeskjet919035
Go
Done
Internet

e-mail form

Your request will not register you as an hp customer. E-mail responses can be expected within 2 United States business days.

* = required field

what operating system do you use?	select *
if "other" please specify	
how is the product connected to your computer?	select *
if "other" please specify	
if there is an error message please specify.	
please provide the exact problem description and any other information that can help hp assist you.	*
first name	*
last name	*
e-mail address	*
phone number	*
country/region	select *

continue

```
This is the total
amount of text that
can be typed into a
window of this
size.  Do you think
it is big enough?
```

country/region

select *

Cuba
Cyprus
Czech Republic
Denmark
Djibouti
Dominica
Dominican Republic
East Timor
Ecuador
Egypt
El Salvador

using this site means you agree to its terms © 1994-2002 hewlett-packard company

Figure 20: HP web site with extreme fixed width web page keyhole and several embedded fixed width edit control keyholes (left), tiny fixed size edit control keyhole for problem description (middle), and listbox keyhole (right).

ever seen: content takes up only 420 pixels. (AOL's fixed width web page keyhole in Figure 1 is some 40% wider.) This sliver-thin page manages to contain seven fixed width edit control keyholes, one lying fixed width edit control keyhole, and one listbox keyhole. The edit control for entering the "exact problem description and any other information that can help hp assist you" displays no more than 120 characters (e.g., 20 five-character words, including spaces to separate them), and the listbox keyhole displays precisely 11 of over 200 choices. It is difficult to come away from this form with the feeling that HP truly wants to hear from its customers.

5 Beyond Inconvenience

The examples in this article have so far focused on how keyholes lead to software systems that needlessly irritate and frustrate their users. That should be reason enough to eliminate keyholes whenever practical, but the stakes are much higher than irritation and frustration. Some keyholes can truly lead to disaster.

Too few bits keyholes, unnecessarily signed int keyholes, and maximum field size keyholes all imply that somewhere in a software system, only a certain amount of storage has been set aside to hold a value. For such systems to function correctly and reliably, great care must be exercised to ensure that the program never tries to store information that requires more space than has been set aside to hold it. When this care is missing, the consequences can be dire.

A colleague of mine once wrote software to control the machinery in a lumber mill. The software used 16 bit unsigned integers to represent the width of a tree in thousands of an inch. The colleague described what happened when 16 bits proved insufficient and the software failed to detect it:

> The system had been running without trouble for two or three months, but they had only been cutting smaller logs. The problem occurred the first time they tried to saw really large old growth logs that were near the capacity of the equipment, close to 65 inches in diameter. The operator was loading a big log on the carriage and positioning it for the first cut. As he moved the log away from the sawline, the system determined the knee was something like 64.000 inches from the sawline and wanted to move another two inches. The target position was calculated as 64,000 + 2,000 = 66,000, which becomes 464, because the overflow causes a modulo operation of 65,535. So the new target was about a half inch from the sawline.
>
> The log had been moving away from the sawline, but now it started moving the other way. The operator cab was on the other side of the sawline from the carriage, right next to the band saw. The log started moving right at the operator, he hit the emergency shutoff and bailed out the back of the cab. He got away unhurt, but he was too slow with the shutoff to stop the log before it pushed through the Plexiglas window into the operator cab.

It was a very serious bug. It really could have killed someone. If the log had gone into the spinning saw it could easily have broken it, sending shrapnel flying through the whole mill.

Software developers more accustomed to web sites than sawmills have no less reason to worry. One of the most common causes of security problems in contemporary software are buffer overruns: writing data into memory beyond the area set aside for the data. The prevalence of this type of bug is a virtual indictment of the software industry, because it indicates that a basic rule of data storage is frequently ignored: *before you put something somewhere, make sure it will fit*. Last year's Code Red and Code Red II Internet worms both exploited the ability to perform buffer overruns in web server software, but the problem is much more widespread than that. Table 1 summarizes the results of a recent Google search for software where buffer overrun vulnerabilities have been identified.

Program(s)	Producer	Operating System
gzip		NetBSD
xdat	IBM	AIX
count.cgi		Many
Kerberos	MIT	
df, pset, eject	SGI	IRIX
Openview Network Node Manager Alarm Service	HP	Many
Clip Art Gallery	Microsoft	Various Windows
Index Server 2.0	Microsoft	Windows NT and 2000
rpc.espd daemon	SGI	IRIX
xterm		OpenBSD
statd		Various Unix
Windows Shell	Microsoft	Windows
talkd	Sun	Solaris and SunOS
AOL Instant Messenger (AIM)	AOL	Windows
Windows Media Player	Microsoft	Windows
HyperTerminal	Microsoft	Windows
SQL Server	Microsoft	Windows
Internet Explorer	Microsoft	MacOS
Internet Information Server	Microsoft	Windows
Web+	TalentSoft	Many
dtspcd daemon		Various Unix
login		System-V-derived Unix
C Runtime Library	Microsoft	Windows
Yahoo Instant Messenger	Yahoo	Many

Table 1: A sampling of programs in which buffer overrun vulnerabilities have been identified. (Patches have typically since been released.)

Buffer overruns are possible only if buffers are fixed in size, and fixed size buffers are, by definition, keyholes. Such keyholes (and the attendant programmer sloppiness about guarding them appropriately) has allowed malicious hackers to interrupt the functioning of the Internet, cause chaos within corporate Intranets, and inflict billions of dollars in damage. Code Red alone is reported to have caused over $1 billion in damage[†].

Still, it could be worse, and in 1979, it nearly was. In March of that year, a nuclear power plant at Three Mile Island in Pennsylvania suffered a catastrophic loss of coolant. It is estimated that the plant came within 30 minutes of total meltdown. Had that occurred, it is likely that there would have been a signifi-

† http://www.usatoday.com/life/cyber/tech/2001-08-01-code-red-costs.htm.

cant release of radioactive material, possibly in the form of a giant cloud of radioactive steam. The underlying cause of the disaster was a release valve that stuck open, but, as far as I know, that has nothing to do with software keyholes. What does have to do with keyholes is the software that reported the temperature of the water in the release valve drain pipe.

When the release valve at Three Mile Island stuck open, the behavior of other systems in the plant made clear to the operators that something was wrong. They didn't know *what* was wrong, however, and it didn't help that an indicator in the control room appeared to show that the release valve was closed when it wasn't. As the operators initiated what was in all likelihood the most important real-time debugging session of their lives, one of the things they checked was the temperature of the water in the release valve drain pipe. If the temperature was unusually high, that would have suggested that, regardless of the reading on the control panel, the release valve was actually open. Such a realization might have led them to force the valve closed, an action they did take later and which ultimately prevented the meltdown.

Checking the temperature of the water in the release valve drain pipe wasn't a simple matter of looking at a gauge. A computer monitored that value, so the operators queried the computer for the temperature. The number they got back was 280 degrees Fahrenheit, not an unusual value for water vapor under the pressurized conditions of the plant, and far below the roughly 600 degree temperature of the water the valve was designed to vent. Unfortunately, the reported value was inaccurate. It was low. Very low. The actual drain pipe water temperature was considerably higher, but *the software had been designed never to display a value for this temperature that was greater than 280F.*"[†] This is an example of the restricted range keyhole I introduced in Section 2.8, a keyhole that refuses to display some values in its natural range. Like most keyholes, it doesn't sound inherently dangerous, but also like most keyholes, it can lead to significant problems under the right conditions.

As I said, keyholes can lead to consequences much more serious than irritation and frustration.

6 Eliminating Keyholes

One of the most annoying things about keyholes is that few can be justified on technical grounds. For those rare keyholes where a technical justification does exist, it's still not uncommon to find that the size of the keyhole is smaller than necessary, i.e., is overly restrictive. In this section, I review each keyhole type and propose practical approaches to eliminating (or at least enlarging) such keyholes.

My emphasis is on *practical* approaches to keyhole elimination. In an ideal world, our software would impose no keyholes at all, but we are unlikely to experience that kind of bliss in the foreseeable future. Implementing such systems would simply be too difficult. My focus is therefore on the elimination or mitigation of keyholes that are, as I noted in Section 1, *gratuitous* or *artificial*. Such keyholes are the low-lying fruit of the keyhole universe. It makes sense to pick them first.

Some developers are likely to object to my proposed remedies on the basis that they are too much work or are inconsistent with established conventions. I address these concerns in Section 7.

6.1 Eliminating Fixed Width Web Page Keyholes

I noted in Section 2.1 that the default behavior for web pages is for their layout to be determined dynamically based on the width of the browser window, and I explained that page designers must take explicit steps to disable this default behavior. That being the case, the way to eliminate fixed width web page keyholes is clear: stop disabling the default behavior.

Some page designers are likely to resist this perceived loss of control, so it may be instructive to review the many aspects of page appearance that site designers cannot alter.

[†] James R. Chiles, *Inviting Disaster*, HarperBusiness, 2001, pp. 59-60. My account of the Three Mile Island disaster is taken from this book.

- The user's screen size and resolution.
- Whether the size of the virtual screen is the same as the size of the physical screen. Many computers, for example, offer virtual screens of 1280x1024 pixels that run on physical screens of 1024x768 pixels.
- The size of window attributes such as fonts and border widths.
- Whether the monitor is color and, if so, how many bits/pixel are available.
- Under Windows, whether the task bar is hidden, its size in pixels, and whether it appears at the bottom, top, left, or right edge of the screen.

Page designers already lack control over many aspects of a user's viewing experience. With that in mind, designers for the web need to abandon the fiction that layout for web pages is akin to layout for printed pages. A good (and easy) way to start is to abandon fixed width web pages.

6.2 Eliminating ListBox/ComboBox Keyholes

We have already discussed three ways of mitigating or eliminating keyholes arising from listbox and combobox keyholes. The first is to have the size of the drop-down list extend the full length of the screen, if need be. Ideally, this would happen automatically as it does with the Internet Explorer Favorites menu shown in Figure 4. A less desirable alternative is to have the initial listbox/combobox dropdown appear as a relatively small keyhole, but to allow users to expand it. This is also shown in Figure 4. In general, the first approach is more desirable than the second, because it eliminates work on the user's part, but implementation considerations may tilt the balance towards the second one for web applications. After all, the user-resizable combobox has clearly already been written, and reusing that technology may be preferable to creating something better from scratch. (I'll examine the write-vs.-reuse question in Section 7.)

The third approach to eliminating listboxes and comboboxes is to rethink the need to present users with a fixed set of choices in the first place. In many cases, the set of choices is well known, and allowing users to type in their selection manually will be faster, more natural, and less error-prone than having them select from a set of choices. We discussed this issue in Section 2.2 with respect to using a listbox to have users specify a U.S. state.

The more listboxes or comboboxes that are needed to fully specify a field, the better the case for scrapping all of them. For example, having users specify a date by using one listbox for the month, another for the day, and a third for the year (as some web sites do) is a sure way to drive people crazy. Such designs are typically motivated by an understandable desire to maximize client-side data validation and minimize round trips between client and server, but Section 2.2 notes that server-side validation is typically required in any case. Furthermore, asking users to specify information such as dates, addresses, etc., in this awkward, unnatural manner almost certainly leads to increased error rates and elevated levels of irritation.

6.3 Eliminating Fixed Size Window Keyholes

Fixed size windows are easy to eliminate: just stop using them. The next time you reach for a fixed size window control, choose a variable size window control instead.

6.4 Eliminating Maximum Size Window Keyholes

Maximum size windows are even easier to eliminate than fixed size windows, because the windows already allow users to change their size. All you need to do, then, is to eliminate the constraint that prevents their size from growing as large as users want it to be.

6.5 Eliminating Fixed Size Dialog Keyholes

Dialog boxes are really just a specialized type of fixed size window, so one of two situations applies: either (1) all the information to be placed into the dialog will always fit, or (2) it won't. When it won't, the proper solution is to replace the fixed-size dialog with a different control, one that causes no keyhole (or possibly a larger keyhole). The natural choice is a variable-sized window.

6.6 Eliminating Fixed Width Edit Control Keyholes

Eliminating a fixed width edit control typically calls for two actions. First, follow the advice in Section 6.11 to eliminate any maximum field size keyhole that may apply to the data displayed in the edit control. Doing so eliminates any data-inspired constraints on the edit control width. Second, have the width of the edit control vary with the width of the window. Assuming you are following the advice in Sections 6.1 and 6.3-6.5, almost all the windows in your software will allow users to resize them.

On web pages, having the width of an edit control vary with the window width is easy, but, based on the behavior of most web pages I see, few web page designers know it. All that's needed is to set the style attribute to "width:100%". Here are examples for a single-line edit box and a multiple-line edit box.

```
<input name="textbox" style="width:100%"></input>

<textarea name="textarea" rows="7" cols="80" style="width:100%"></textarea>
```

My simple tests show that this works as expected in the latest browsers from Microsoft, Netscape, and Opera. Certainly some users work with browsers that fail to support this attribute (e.g. older browsers), but that is hardly an excuse for avoiding it. Some users work with monochrome monitors, too, but that's no reason not to provide color to users whose monitors offer it.

6.7 Eliminating Lying Fixed Width Edit Control Keyholes

A fixed width edit control can lie only if it limits the number of characters that may be entered, so the best way to eliminate lying fixed width edit controls is to eliminate restrictions on the size of the field being edited by the control. Section 6.11 explains how to do that.

If a fixed field width is essential, an alternative is to make sure that the fixed width edit control doesn't lie. That means ensuring that the width of the control corresponds to the maximum number of characters that may be entered. Because this is possible only with fixed width fonts, it implies that the font used for characters in the edit control must be fixed in width (e.g., Courier), and the width of the control must take into account the font size being used. Determining this size may not be straightforward. On my notebook computer, I use Windows' "Large Fonts" setting (125% of normal size) when hooked up to an external monitor, but I apply the "Small Fonts" setting when using the notebook's native screen. Many applications display text improperly when I'm using the larger fonts. Subtle pitfalls such as this lend additional weight to the preferred solution of eliminating the problem by removing any underlying maximum field size keyholes.

6.8 Eliminating Restricted Range Keyholes

A restricted range keyhole such as the temperature display at Three Mile Island (Section 5) arises when an output device fails to show some values in the natural range of whatever is being displayed, so the way to eliminate the keyhole is clear: show all values in the natural range of whatever is being displayed. However, the form such display takes may vary over the range.

Taking the temperature display at Three Mile Island as an example, it is possible to speculate on reasons why no temperature above 280 degrees was displayed.† One possibility is that temperatures above 280 degrees could not be accurately measured by the underlying instrumentation. If that was the case, the proper display for temperatures above 280 might have been ">280" or "Out of range" or some other indicator that the temperature was beyond 280. A second possible reason for the restricted range keyhole is that temperatures above 280 degrees could not be reliably measured by the underlying instrumentation. If that was the case, similar outputs could have been used to convey it.

The crux of the matter is that the natural range of temperatures included at least one additional value ("above 280"), and the computer system failed to display that value. That was the fundamental cause of the keyhole. Displaying all possible values in the natural range would have eliminated it.

† I was unsuccessful in my attempts to discover the reason for this behavior.

6.9 Eliminating Too Few Bits Keyholes

Some too few bits keyholes are easy to eliminate. Instead of using an 8 or 16 bit data type on a 32 bit architecture, use a 32 or 64 bit data type. Adherence to this simple rule would have eliminated both of the too few bits keyholes described in Section 3.1, and any change in program performance (in either space or time) would almost certainly go unnoticed.† Another easy approach is to note Section 6.10's observation that it is typically easy to add a bit to an inherently nonnegative value by using an unsigned variable instead of a signed one. In general, it's easy to increase keyhole sizes up to the point where they correspond to the number of values representable in a word (or sometimes two) of the underlying hardware.

Increasing too few bits keyholes beyond that is more difficult. For example, writing software to represent more than 4,294,967,296 objects on a 32 bit architecture is nontrivial, because that's as many values as a word-sized pointer can take. It's not impossible, of course, and some applications do it as a matter of course (e.g., database systems often implement at least a 96 bit address space), but I don't expect software developers to work that hard for mainstream applications. Before worrying about keyholes grounded in architectural decisions in the underlying hardware, it makes sense to focus on eliminating keyholes grounded in dubious software design or implementation decisions.

6.10 Eliminating Unnecessarily Signed Int Keyholes

The obvious way to eliminate unnecessarily signed int keyholes is to use unsigned types whenever that makes sense, i.e., whenever negative values are semantically excluded. Unfortunately, not all programming languages make this as easy as it should be. Unsigned types are directly available in C, C#, and C++, but they are absent from Java and Visual Basic. Languages lacking native support for unsigned types may be able to access them through libraries (e.g., Visual Basic .NET can take advantage of them through its support for the Microsoft .NET Common Type System), and where this can be done without undue difficulty, it should certainly be considered. In cases where heroic effort would be required to employ unsigned types, it's more reasonable to simply bump the size of the signed data type up to the next natural size boundary, e.g, use a signed 32 bit int where a 16 bit unsigned int might initially have been considered. Such a strategy has the additional beneficial side effect of reducing the likelihood of imposing too few bits keyholes on users.

6.11 Eliminating Maximum Field Size Keyholes

In my experience, the vast majority of maximum field size keyholes arise not from semantic constraints inherent in the domain, but from arbitrary implementation decisions made regarding how the field value is to be stored. For example, many programmers use fixed-size arrays to store field values, and the size they choose for the array almost invariably becomes the size of the field size keyhole. How are such array sizes chosen? More often than not, they're based on a programmer's intuition of "the smallest size that's obviously big enough." This is just plain bad programming. Not because it's based on intuition (though that itself is awful enough), but because it's based on fixed size arrays. For the kinds of systems with which this article is primarily concerned (e.g., those with extensive real memory and more extensive virtual memory), there is almost no case to be made for using fixed-size arrays. Rather, input of unknown length† should be read into data structures that dynamically and automatically extend themselves to accommodate the amount of data available. Some languages have such types built in [GIVE EXAMPLES], while others have ready access to them via class libraries (e.g., C++'s basic_string and vector templates, GIVE OTHER EXAMPLES HERE (e.g., Visual Basic's String data type)). For programmers who find themselves in the extraordinary position of working with a language and library that fails to offer an array-like data structure

† There are exceptions, of course. Embedded applications are more sensitive to increases in data space than are hosted applications, and some high performance applications suffer noticeably reduced locality of reference when data are too big. Such programs are outside the mainstream I consider in this paper, however, and neither of the examples I give in Section 3.1 qualify as exceptional.

† In robust systems, this is virtually all input, because even input that is supposed to be in a known format (e.g., all fields of predefined lengths) may have become corrupted (possibly maliciously) since it was last written. Reliable systems verify the validity of all data before working with it.

that automatically expands to hold the data that is inserted into it, the solution is to write such a data structure themselves. The time needed to write, test, verify, and document the code can be amortized over the many uses the code will doubtless enjoy.

Let me be utterly explicit about this: *there is no excuse for using a fixed-size array where a dynamically extensible array is more appropriate, and every place that accepts input is a place where a dynamically extensible array is more appropriate*.

Switching from fixed size to dynamically extensible arrays eliminates any in-memory justification for maximum field size keyholes, but it fails to address constraints that may arise from the need for persistent storage of such fields. There are two common situations. Either the persistent data are stored in a flexible format where different values for the same field may vary widely (e.g., an XML file where the value for the <name> field may vary from instance to instance[†]) or the data are stored in a database with a schema where all values for a particular field must fit in the same fixed-size field. The former case — that of the flexible storage format — is easy to handle, because the format imposes no keyhole that the software must honor. The latter case — that of data to be stored in a database with a schema using fixed-size fields— is only slightly more challenging.

Database field types can allow for flexibility in the amount of data to be stored, especially when it comes to storing strings. Virtually all database systems support VARCHAR data types, which are variable length strings of up to some system-specified maximum length, e.g., 8000 characters for SQL Server or 255 characters for Oracle, though the latter also offers the VARCHAR2 data type, which allows for strings of up to 2000 characters. Strings of essentially unlimited length may be stored in fields of types such as TEXT (for MySQL) and BLOB (for virtually all database systems). Compared to fixed-length CHAR fields, there may be a (typically small) performance penalty for using VARCHAR, VARCHAR2, TEXT, or BLOB fields, but this is akin to the (typically inconsequential) hit incurred by the use of dynamically extensible arrays for in-memory string storage and is unlikely to be prohibitive in most applications.

Setting aside performance for a moment, my point is that implementation difficulty is no excuse for database-inspired maximum field size keyholes, because using the data types VARCHAR or TEXT or BLOB instead of CHAR in a schema definition is not very difficult. Where programmers have control over the schema, such field types should be the default in the same way that dynamically extensible arrays should be preferred over fixed size arrays for in-memory data. Even the simple act of using 255-character VARCHARs instead of smaller fixed size CHAR fields would eliminate a large number of maximum field size keyholes that users of both native-platform and web applications routinely encounter.

As regards performance considerations, it is important to bear in mind that they will arise only some of the time, and even then only for some of the fields being stored. Occasionally, there may be a need to replace variable size data types with those of fixed size, but this doesn't change the fact that the *default* design decision should be to use variable size strings, because such strings are more robust in the face of changing technology (e.g., Section 3.3's phone number and zip code examples) and, by eliminating the irritation and frustration that go with maximum field size keyholes, lead to a more satisfying user experience.

Some designers worry that allowing users to specify long strings as field values will lead to unacceptable data bloat, either on disk or over the network between client and server. Such worries are largely unfounded. Systems built on fixed size strings typically choose the size of the strings to be "obviously big enough." Call this limit n. When no size limit is imposed, it is reasonable to assume that almost all users will continue to specify field values no larger than n. After all, n is "obviously big enough" for almost all users. Hence, the total data stored will remain essentially unchanged.

The only time when fields longer than n will be encountered will be when (a) n was too small for a user's legitimate need, (b) a user makes an error entering a field value, or (c) a malicious user deliberately enters a long field value, possibly in an attempt to cause a buffer overrun. In cases (a) and (c), allowing long strings results in better software behavior, either because legitimate user data is not rejected or because the software remains intact in the face of a potential attack. In interactive scenarios, case (b) can be dealt with

† This is the case for the Adobe FrameMaker indexing example of Section 3.3, because FrameMaker documents are stored in a binary file whose layout is determined by the text in the document being stored.

by querying the user to verify that suspicious entries are intentional (e.g., "Are you sure that the first name is really 'The rain in Spain falls mainly on the plain"?). In non-interactive scenarios, a warning can be written to a log file or similar action taken.

6.12 Eliminating Restricted Domain Keyholes

At one level, restricted domain keyholes are one of the easiest of all keyholes to eliminate, because all that is required is to *not* check to see if a "forbidden" input element has been entered. The result is less code for programmers to produce, test, document, and maintain and fewer restrictions on users, truly a win-win situation. This "just don't check" approach is viable in situations where the domain restrictions are gratuitous, i.e., have no underlying basis in other aspects of the software design. This should virtually always be the case for passwords, for example, and is almost certainly the case for all the examples in Section 3.4.

It is possible to imagine situations where restricted domain keyholes are not gratuitous. For example, one might imagine a system where users employ ASCII-based input devices (e.g., programs running under Windows), but the resulting input is stored on an EBCDIC-based system. Such systems might reasonably choose to limit user inputs to characters present in both ASCII and EBCDIC. Even this need not always be the case, however. If the system uses passwords and follows a Unix-like policy of not storing the password text, there is no reason why users should be prevented from using the full ASCII character set, because the characters in the password can be interpreted as a sequence of integers rather than a sequence of characters, with password validation performed using the integers. With such a design, the fact that a sequence of integers may correspond to different character sequences under ASCII and EBCDIC is immaterial, because it will never be interpreted as a sequence of EBCDIC characters.

7 Discussion

In Section 6, I explained how each of the keyhole types discussed in this paper can be enlarged or eliminated. Most of my prescriptions involve little more than a change in habits (e.g., employing user-resizeable windows instead of fixed- or maximum-sized windows, using unsigned instead of signed data types for values that are meaningful only when nonnegative, choosing variable sized arrays and database field types instead of their fixed-size counterparts). Some actually reduce your workload as a software developer (e.g., *don't* specify web page line breaks manually, *don't* gratuitously filter out natural domain elements). A few call for a trivial amount of additional effort (e.g. when you use an edit control on a web page, specify that the width of the control should change as the page width does). None entail any significant additional design or implementation work. Most keyholes are easy to eliminate. All that is required is the desire to do it.

A small minority of keyholes arise because of more fundamental underlying restrictions. As I noted in Section 6.9, for example, papering over keyholes based on the maximum number of values representable in a machine word is difficult. Similarly, the opaque nature of database BLOBs (Section 6.11) may lead to querying restrictions that justify the use of more query-friendly (but maximum-sized) VARCHARs. When you confront troublesome keyholes such as these, my general advice is to leave them be; devote your energies elsewhere. In Section 1, I explained that my goal in this paper is to advocate the mitigation or elimination of keyholes that are gratuitous and artificial, and in Section 6 I noted that it makes sense to pick the lowest-lying keyhole fruit first. My philosophy about keyholes is designed to be practical: enlarge or eliminate them when it's easy, but don't torture yourself. Many things are easy, but a few are hard. Do the easy things as a matter of course. Do the hard things only when you have to.

In Section 6.11, I discussed the (largely theoretical) scenario whereby you wish to avoid fixed size arrays but are programming in a language and with libraries that fail to offer an array-like data type that automatically grows to accommodate however much data is inserted into it. My advice was to write such a type yourself, and my reasoning was that the cost of creating such a type would be quickly recovered by the numerous uses it would receive and the large number of maximum field size keyholes it would eliminate. This is but a specific example of a more general question: faced with a choice between using an off-the-shelf software component that leads to keyholes or creating a similar component that enlarges or eliminates the keyholes, what is the most appropriate course of action?

The proper solution to every write-or-reuse question boils down to the results of a cost-benefit analysis. Developing a new component (i.e., writing it, testing it, documenting it, etc.) takes time and energy that could be spent elsewhere. This is a one-time up-front cost. Maintaining the component once it has been developed, however, is an ongoing cost. On the other hand, keyhole enlargement or elimination can lead to decreased user error rates, greater user satisfaction, and more effective product differentiation (see below). These benefits accrue each time the component is used. As a general rule, then, it's easiest to justify the creation of new components that are simple to create and maintain (i.e, whose cost is low) and that address either keyholes with enormous drawbacks (i.e., feature a large per-use payback) or that arise in a large number of situations (i.e., have a smaller per-use payback, but a large multiplicative factor).

Keyholes are such a fixture of contemporary software systems, eliminating them can lead to charges of failing to adhere to established conventions. One might argue, for example, that users *expect* comboboxes and listboxes to exhibit the keyhole behavior described in Section 2.2 and that failure to adhere to these expectations will surprise and confuse users. Or one might complain that because approximately two thirds of the most popular web sites fix or limit the content area of their web pages (Section 2.1), that's what users are accustomed to, and that alone makes fixed-width pages a good design.

Such arguments are ill-conceived for a number of reasons:

- Most people dislike things that hurt, frustrate, or irritate them, even if those things follow convention. Such people are receptive to new, less unpleasant conventions, and, all other things being equal, they'll happily switch from systems that are established and annoying to new ones that are innovative and satisfying.
- When your software acts just like everybody else's, it is difficult to make your product stand out from the crowd. Differences in behavior — such as failing to gratuitously irritate your users in situations where your competition does — can favorably distinguish your software in the eyes of your prospects and customers. Microsoft, for example, is famous for providing development tools that make it easy to produce applications with a "Windows look and feel," then turning around and writing custom controls for their own applications (e.g., Office) that have a slightly different — and slightly better — look and feel.
- We have a word for what happens when most people are doing one thing but you come up with something different that works better: *progress*. The fact that keyholes are common and conventional does not mean that they are good or that users like them. Software that can be exploited via buffer overruns is common and conventional, too, but increasingly we understand just how dangerous that can be.

About the Author

Scott Meyers is a software development consultant with over three decades of programming and industry experience; he provides consulting and training services to clients worldwide. He is a member of the advisory boards for several start-up companies as well as *Software Development* magazine. The author of three best-selling books on C++ programming, he received his Ph.D. in Computer Science from Brown University. His current work focuses on identifying fundamental principles for improving software quality. His web site is http://www.aristeia.com/.