Intelligent Combo Boxes

Chris Barker

2.0 95 97 2000 DOWNLOAD

What can you do to improve the accuracy of data entry? With only a few lines of code, you can catch one of the most common data-entry errors in combo boxes. In this article, Chris Barker shows how this can be done.

THE phrase, "Garbage in—garbage out" has been heard by most everyone, but what can be done about it? A common contributor to the garbage in syndrome is data-entry mistakes, and one contributor to bad data is the searching combo box. Searching combo boxes can be an excellent tool for reducing data-entry errors while also making it easier for users to enter data. Your typical combo box performs the following roles:

1. Provides the user with a limited list of alternatives with which to fill a field (provided that you've set the box's Limit-to-list property to True).
2. Enhances data-entry proficiency by automatically finding the closest match in the list as the user types (this features uses the box's Autofill property).
3. Reduces the need for users to know system codes by providing them with information they recognize (here, you use bound columns to display meaningful data while updating with the related code).
4. Allows users to see their choices directly in front of them without having to open another form.

However, there's one downside to all of these benefits—users get lazy. For example, imagine that you're a user entering a customer order for "MackDonald's Restaurants." You begin typing in "m," "a," "c," and "k." At this point, Access has probably already selected "MackDonald's Restaurants" in the combo box. This works great, and your users love this feature because they don't have to know any archaic code like "Customer #39" or "MC017."

However, two months later there's a new customer added—"MackDonald's Farms." Assuming that you have the combo box sorted alphabetically by customer name, this new customer will appear before "MackDonald's Restaurants." So what will happen over the next few weeks? Data-entry errors will increase substantially because the users who have adapted to typing "mack" (or even fewer keystrokes) will now be selecting "MackDonald's Farms" because it appears in the list before "MackDonald's Restaurants."

I doubt there are many users out there who haven't encountered this problem at some point in time. The severity of the problem will, of course, vary depending on the volume of entries. Salespeople that enter a few customer orders throughout the day are more likely to notice this problem. People whose primary job function is entering hundreds or thousands of items per day will likely be oblivious to the problem until it's far too late.

Solutions

A very simple solution is to use the combo box's Dropdown method (MyComboBox.Dropdown) in the OnEnter or OnGotFocus event of the box. This will cause the full list to display. The goal is to have the user see the two similar entries and note which one is highlighted. As a solution to the problem, however, it has a number of drawbacks:

- In most cases, this is unnecessary.
- It's distracting to the user to have the screen suddenly flash a list at them.
- It doesn't really provide any check on the data the user has entered. Users who are susceptible to this problem are the ones who perform high volumes of entries—and they probably aren't even looking at the screen as they enter. Instead, they're looking at the paper they're entering data from.

Dropping the list is probably a worthwhile solution to consider for those forms that are used infrequently for data entry (my salesman example, for instance). For a high-entry scenario, this solution isn't good enough.

The problem is that a user can enter enough characters to select two values. In my example, "mack" selects both "MackDonald's Restaurants" and "MackDonald's Farms." However, only the first item in the list will be selected, which might or might not be the right one. What's needed is some way to check for similar values in the combo box list to determine whether a user hasn't uniquely identified an entry. With this feature available, I can alert the user only when there's a possibility of error. To that end, I created the

cboValCheck function.

Checking the combo

The cboValCheck function is passed two parameters: a combo box (cbo) and the column to match (intMatchColumn). The function returns a Boolean value indicating whether or not a match was found. The function assumes that the column to match against is sorted so that similar values will appear one after the other in the list.

The function begins by determining the user's current selection using the Column property of the combo box. I add a null string to the value returned by the combo box for those situations when there's no value in the combo box yet:

```
strcboText = cbo.Column(intMatchColumn) _
                    & vbNullString
```

This strcboText value now needs to be compared to the other strings in the combo box to determine whether the user has uniquely selected one entry in the box. Since I assume that the list is sorted, I can optimize the search by checking only the values above and below the user's currently selected value. After all, in a sorted list if the values directly above and below are not a match, then no other value will be either.

To get the values above and below the current value, I retrieve the position of the currently selected value using the ListIndex property. As I'll explain, I also need to know how many items are in the list, so I also retrieve the ListCount property:

```
intcboIndex = cbo.ListIndex
intcboCount = cbo.ListCount
```

Now that I have the selected entry's position, I can get the value above and below the current value using the Column property of the combo box. In the following code, the first line gets the item after the current entry, and the second line gets the item above the current entry:

```
cbo.Column(intMatchColumn, intCboIndex + 1)
cbo.Column(intMatchColumn, intCboIndex - 1)
```

One catch: What if the selected item is the first or last item in the list? The answer is that this test will generate an error. This is where the list count that I mentioned earlier comes into play. I use that information to determine whether the item that I'm checking is at the end of the list:

```
If (intCboIndex + 1 <= intCboCount - 1) Then
  cbo.Column(intMatchColumn, intCboIndex + 1)
End If
  To check for the first item in the list, I just _
compare against 0:
If (intCboIndex - 1 >= 0) Then
  cbo.Column(intMatchColumn, intCboIndex - 1)
End If
```

Now that you have the items above and below the selected item, you can perform the actual comparison. Which leads to the next question: How close a match is considered grounds for warning the user that there's a potential problem? Truth is, it depends on you and your application. I base the match factor on percentages. I grab the first 50 percent of the selected value and then compare that with the same length text from the items above and below it in the combo box. If you specify a low percentage match (25 percent), you have a better chance of catching a data-entry error, but the users might not like the number of interruptions. If you have a high percentage match (75 percent), your users won't likely complain, but they might also not be prompted for possible data-entry errors as often as they should be.

Another solution is to determine how many characters the user has entered and check that many characters. As users type in their entry, Access displays the current matching result with the letters that you haven't typed but Access has provided. You can calculate the number of letters that the user has entered by subtracting the number of selected letters (the part supplied by Access) from the total number of letters in the combo box and then adding 1:

```
Len(cbo.Text) - cbo.SelLength + 1
```

For the sake of simplicity, I'll use a 50 percent match factor in my sample code. In your function, you can specify your own percentage, or even pass the match percentage as another argument to the function. The variable intSearchChar is used to store the number of characters on which we're going to perform the match:

```
intSearchChar = _
  ((Len(strcboText) * .5) \ 1)
```

The final step is to perform the comparison using the Left$ function and the intSearchChar variable. For the test on the item following the selected value, the code looks like this:

```
If Left$( _
  cbo.Column(intMatchColumn, intCboIndex + 1), _
  intSearchChar) = _
  Left$(strcboText, intSearchChar) Then
       cboValCheck = True
End If
```

Using the result

Now what do you do once you've found a match? Once again, this is up to you and your application. At the very least, I recommend beeping to inform the user and dropping down the combo box. This provides both auditory and visual cues that there might be a problem. The code to do that looks like this:

```
If cboValCheck Then
  cbo.Dropdown
```

```
  DoCmd.Beep
End If
```

You might wish to provide additional information to the user by using the MsgBox function.

To use the cboValCheck function, just call it from the BeforeUpdate event of the combo box on which you want to perform the validation. The BeforeUpdate event is passed to a parameter called Cancel by Access. If you set the Cancel parameter to False, any pending updates to your tables are put on hold. Since the cboValCheck routine returns False when there's a potential problem, I just pass this value to the Cancel parameter to prevent updates until the user has a chance to confirm their choice.

There are two caveats (which I'm sure you were expecting, right?). The first caveat in using this function is that you also need to have a module-level flag variable to indicate whether a check has already been performed on the user's current selection—otherwise, the function could continually return a result that Cancels the BeforeUpdate. You should check that flag before calling the cboValCheck function. You'll also need to set the check flag variable back to False on the AfterUpdate event of the same combo box. The code reads like this:

```
Private mblnChecked As Boolean

Private Sub _
 MyCombo box_BeforeUpdate(Cancel As Integer)
If Not mblnChecked Then
  Cancel = cboValCheck(MyCombo box, 1)
  mblnChecked = True
End If
End Sub

Private Sub CustomerID_AfterUpdate()
  mblnChecked = False
End Sub
```

The second caveat is that this routine could interfere with other BeforeUpdate code. If there's other code in the BeforeUpdate event, you'll need to decide whether it's better to put the call to cboValCheck before or after the existing code. Additionally, you might want to consider exiting out of the BeforeUpdate event if cboValCheck returns True, so as not to conflict with other code in the event:

```
If cboValCheck(MyCombo box, 1) Then
  Cancel = True
  Exit Sub
End If
```

This function is really just the beginning. You can modify it to handle numerous scenarios. For example, I've modified this function to account for non-sorted columns, to match a specific number of characters (passing number of characters vs. a percentage to the function), to provide for auto-detection of the first visible column, and to match only on the characters the user physically typed in the box. The single most important modification was to put the functionality into a Class in order to provide ultimate flexibility. Once it's in a Class module, I can reuse the code with various combo boxes on the form and even trap the combo box BeforeUpdate events in my Class module (see Shamil Salakhetdinov's article "Using Dynamic External Event Procedures" in the January 1999 issue of *Smart Access*).

But really, the tip of the iceberg is in the changes to how we look at the business problems that confront us. The next time you hear a manager comment on the accuracy of report data, consider where the report data is coming from. What other functions can be written to improve the quality of data your systems are collecting? ▲

INTCOMBO.ZIP at www.smartaccessnewsletter.com

Chris Barker is an IT manager for a transportation company and the owner of Genesis Technology Alliance, a technology solution provider. He has a Bachelor of Commerce degree from Dalhousie University with Honors in MIS. Chris's development experience has focused primarily on Access, but he has also branched out into Office, Visual Basic, and SQL Server. Cbarker@canada.com.