No More Write-only Code

Peter Vogel

Peter Vogel takes a quick look at two topics that will help you create code that can be maintained: error handling and program documentation.

BACK when I started writing code, the battle over structured code was just in the process of wrapping up (structured programming won). That didn't stop me from writing a lot of "write-only" code with two-letter variable names, no comments, and GoTos. My excuse is that I was writing in AppleBasic on an Apple IIe with 48KB of memory. Gerry Weinberg commented once that many earlier programs were really love letters from the programmer to the computer. And, like all love letters, they were so full of inside jokes and pet phrases that only the two lovers could understand them.

Of course, back then the emphasis was on building programs—which, sadly enough, it still is. Yet we know that we spend far more time modifying programs already in production than we do writing totally new programs. Typical estimates suggest that 65 to 75 percent of a developer's time is spent in working on existing applications. "Write-only" programs are the bane of efficient and effective system maintenance. Being able to understand what was meant by the previous programmer is critical to being able to fix or enhance an application. And, of course, six months after you write some code you will be "the previous programmer," even when you're reading your own code.

There aren't any rules for writing readable code. Writing readable code is more about the attitude that you take when you write than a particular set of rules. In this article I'll discuss two topics that will let me illustrate the kinds of things that you can do in order to create readable code: error handling and commenting.

Error handling

In his article in this month's issue, Keith Bombard stresses the importance of error handling. In VBA environments, error handling has been limited to the On Error construct. One way to use VBA error handling is like this:

```
Public Sub..

On Error GoTo ErrorRoutine
…potential problem…
Exit Sub
ErrorRoutine:
  …error handling code…
  Exit Sub

End Sub
```

The readability problem with this strategy doesn't become obvious just by looking at this skeleton. The problem that exists is in the distance that may separate the code between the "...potential problem..." section and the "...error handling code..." area. Separating two sections of code that are as tightly related as a line of code and its error handler makes it harder to understand how they relate to each other.

For the past five or six years I've been writing my error handling code this way:

```
Public Sub..

On Error Resume Next
…potential problem…
If err.Number > 0 Then
  …error handling code…
    err.Clear
End If

End Sub
```

Using this error handling strategy, my error handling code can immediately follow the code that may invoke it. The point that I'm trying to make is that I didn't adopt this style because it was more efficient. My only reason for using On Error Resume Next was that I believed that it would make it easier for others (and me) to read my code.

It's not a perfect solution (of course, when you start off assuming that you have an error, it's hard to be perfect). The Try...Catch processing that's introduced with Visual Basic .NET is a cleaner solution. In the meantime, this is what I'm doing to make life easier for my successors.

Commenting

My bosses always told me that I should comment my code. They never told me what that meant, but they were confident that I should do it.

At one time there was an ongoing argument about what the right ratio of lines of code per comment line was. In the end, people realized that the problem wasn't about the quantity of comments but the quality of the comments. To paraphrase Kernighan and Plauger in their classic work, *The Elements of Programming Style*, the value of a comment is zero or negative if it's wrong.

Commenting a program is fundamentally wrong. By commenting a program you actually create two versions of the program: the one described in the comments and the one that exists in the code. As any database developer knows, if you have two versions of a piece of data, over time, they will disagree.

Much of what was supposed to be handled by commenting—making clear what a program does—is now handled by structured coding and meaningful variable names. There is still a role for comments to play, though.

I should give credit where credit is due. Here, I'm just echoing a comment that Paul Litwin, the founding editor of *Smart Access*, made many years ago: Comments should describe the "why," not the "how," of the code. The first purpose of a comment is to describe what isn't in the code. Comments should describe why a piece of code exists because nothing in the code can show that.

There's one other reason you shouldn't comment: Comments get in the way of the code. I restrict comments to the top of each procedure where they don't get in the way of the code. This also defines the scope of what I'm commenting—it describes the routine that the comment starts. Refusing to put comments in my code also forces me to write clear code.

There aren't any hard and fast rules here, though. The goal is to make your code readable, not to enforce a rule. Keith's article has a good example of when you can be forced to put comments inside a routine. In his code, in order to make the code work for all versions of Access, Keith had to embed the numbers 0 through 3 in a Select Case statement. This is almost the definition of write-only code:

```
Select Case someVariable
     Case 1
               …
     Case 2
               ..
     etc.
```

Faced with this code, the developer is forced to guess what the numbers 1 and 2 mean. The right answer, of course, is to use the system's predefined constants. This wasn't possible if Keith was going to meet his design goal of supporting all versions of Access. Keith could, of course, have defined his own set of variables with meaningful names to use in the code (meaningful variables, by definition, convey meaning). However, since the variables would only be used in one place, Keith used the numeric constants in his Select Case statement and inserted comments to describe what each "magic number" meant:

```
Select Case someVariable
     Case 1   'System procedure
               …
     Case 2   'Property Set
               ..
     etc.
```

Since comments are typically longer and easier to read than meaningful variable names, I could make a case that Keith's use of comments is superior to using variable names.

There's one other purpose that only comments can provide: Comments can summarize a procedure to describe to programmers what the routine does. This must be done at a sufficiently high level so that the comments don't describe the routine's internals. When I've used comments for this purpose, my comments (still at the level of the subroutine and the function) described:

- The kind of information to be passed to the routine's parameters (but not the parameter's datatypes, because that information is embedded in the code).
- The kind of information that will be returned by the routine (only appropriate for functions).
- Any side effects of the routine (that is, things changed by the code that are visible to the calling code: a file left open, a record deleted, and so on).
- The relationship between the routine's inputs (parameters) and its outputs and/or side effects.

I've put these options in order of "increasing problems." Describing parameters, for instance, is easy to do without exposing anything about how the routine works. The comments can provide guidance as to what data should be passed to the routine, information that may not be expressed anywhere in the routine's code. (Some of that information can be conveyed by giving parameters names that describe the data that should be passed to them.)

However, by the time that you get to the last item in the list, the relationship between the inputs and the outputs, it becomes increasingly more difficult to not describe how the routine works. ▲

Peter Vogel (MBA, MCSD) is a principal in PH&V Information Services. PH&V specializes in design and development for systems that use Microsoft tools. Peter has designed, built, and installed intranet and component-based systems for Bayer AG, Exxon, Christie Digital, and the Canadian Imperial Bank of Commerce. He's also the editor of the *Smart Access* newsletter, and wrote *The Visual Basic Object and Component Handbook* (Prentice Hall, currently being revised for .NET). In addition to teaching for Learning Tree International, Peter wrote its Web application development, ASP.NET, and technical writing courses, along with being technical editor of its COM+ course. His articles have appeared in every major magazine devoted to VB-based development, can be found in the Microsoft Developer Network libraries, and are included in Visual Studio .NET. Peter also presents at conferences around the world. peter.vogel@phvis.com.