Gaurish Technologies Private Limited

Coding Style Guidelines

Module-2,3

This document focuses on the style with which your code should be written to make it easier for others to understand and maintain

Commenting

One of the keys to helping others understand, maintain, and debug your code is adequate commenting. Comments should be used liberally throughout your code such that the entire logic of your application can be followed and understood without having to actually read any of the code. This may seem excessive, but the long-term benefits are significant, making it well worth the extra time involved.

To accomplish this, virtually every line of code in your application should be preceded by a single line comment explaining what action is being performed. In cases where a few lines of code perform one logical operation, a single comment will suffice as long as it accurately and completely describes the actions being performed. To increase the readability of your comments, it is helpful to leave the line above the comment blank, thus visually breaking up the code into its logical steps

Formatting and Indenting

Another important aspect of writing maintainable code is consistent formatting and use of indentations. Visual Basic's default tab width is 4 spaces, however this tends to make it a little difficult to follow source code in situations where there are more than a few levels of indentation. This value should be change to 3 because it provides adequate horizontal spacing for clean formatting of code, yet doesn't become excessively wide during a larger number of levels of indentation. This value can be changed on the Editor tab of the Options dialog (use the Tools|Options menu item).

The following table provides a list of guidelines for proper code indentation style.

Consideration	**Description**
Declaration section of modules	• Variable declarations in the declaration section should not be indented, appearing at the left margin. • Each module should have the **Option Explicit** statement at the top to prevent bugs being introduced into code by misspelled variable names.
Procedure declaration/body	• Procedure declarations and their corresponding **End** *proc_type* statements should never be indented. • Procedure bodies should initially be indented one level to provide a nice visual offset from the procedure declaration. • There should always be one blank line between the end of one procedure and the declaration of the next.
Variable declarations	• All variables used by the procedure should be explicitly declared at the top of each procedure and indented exactly one level from the left margin. • Each declaration should include an explicit data type, even if the variable is a **Variant**. • All the "**As"** keywords in the declarations should be aligned horizontally so that the data types can be found more easily. • Commenting variables on the declaration line is optional, but

Consideration	Description
	can be helpful in some situations. Generally, variables should be given names that are self documenting.
Conditional/Looping Constructs	• Code should be indented one level whenever the flow of execution is affected by conditional statements or looping constructs. • Examples of conditional statements include: **If...Then...Else...End If** **Select Case...End Select** • Examples of looping constructs include: **For...Next** **Do...Loop**
Grouping constructs	• Code should be indented one level when grouping constructs of VBA are used. Examples include: **With...End With** **Enum** *enum_name*...**End Enum** **Type...End Type**.

Error Handling

In Visual Basic, errors become fatal errors if some portion of the application does not explicitly handle them. Obviously, this is unacceptable and for this reason every procedure must contain error handling (a VB 5 add-in to assist with this is under development). It is especially critical if a procedure could be the last line of defense for the application, meaning that there are no calling procedures below it in the call stack to handle errors it raises or cascades. The Sub Main procedure and event procedures of Forms fit this description. For example, if you had code in the Form_Click event procedure that caused an unhandled error, your application would die unceremoniously.

Class modules typically are not ultimately responsible for preventing fatal errors since their role is to provide services. However, they are responsible for returning intelligible error descriptions to clients. For example, it would be unacceptable to raise or cascade the “Invalid procedure call” error because it would make it difficult for a client to determine whether the error was intentionally raised, caused by a bug in the class module, or a result of invalid input parameters or settings.

All error handling code should use an approach consistent with the one used in the sample code below. An On Error Goto ErrorHandler statement should appear at the top of the procedure, following any variable declarations. The ErrorHandler label should appear at the bottom of the procedure, following an Exit Sub (or similar statement) to prevent the flow of execution entering the error handling block incorrectly. The error handling section should explicitly handle anticipated errors, translating them or taking an appropriate action. Other errors may simply be cascaded to the calling function using the Err.Raise method. Non-GUI based modules (i.e. standard and class modules) should never display error messages in message boxes since there may not be a user present to dismiss the dialog, causing the application to freeze until manual intervention arrives.

Code Samples

```
On Error GoTo ErrorHandler

'Make sure file was opened in read mode
If mintOpenMode <> OPEN_READ Then _
   Err.Raise 1000, , "Operation not supported for file in current mode"

Do Until IDataSource_EOF
   'Get a record using the physical file
   mobjPhysicalFile.GetData gstrLineData

   'Is the discriminator defined?
   If mintDiscriminatorWidth > 0 Then
      'Determine whose record it is, by it's disriminator
      lstrDisriminator = Mid$(gstrLineData, mintDiscriminatorStart, mintDiscriminatorWidth)
      Set lobjRecord = mcolFileRecordsByDisc(lstrDisriminator)

      With lobjRecord
         'Now tell the appropriate FileRecord object to retrieve the line
         .GetRecord

         'Let client know if a Header/Trailer record has arrived
         If Not .IsDetail_ Then _
            RaiseEvent RecordRead(.IsHeader_, lobjRecord)

         'Reinitialize the formulas
         lobjRecord.ReinitializeFormulas_
      End With

      'Keep looping until we have hit the end of the file or retrieved another detail record
      If lstrDisriminator = mobjDetailRecord.DiscriminatorValue_ Then _
         Exit Do
   Else
      'No disriminator defined, so just get the next record, reinitialize the formulas
      mobjDetailRecord.GetRecord
      mobjDetailRecord.ReinitializeFormulas_

      'Exit the loop
      Exit Do
   End If
Loop
```