Reach full functional equivalence with Trace-Match methodology

In a nutshell, the **Trace-Match** technology offers the ability to generate a trace file both in the original VB6 code and in the migrated .NET code, and later compare the two results to verify that the original project and the migrated project behave in the same way when the same sequence of actions is performed on them.

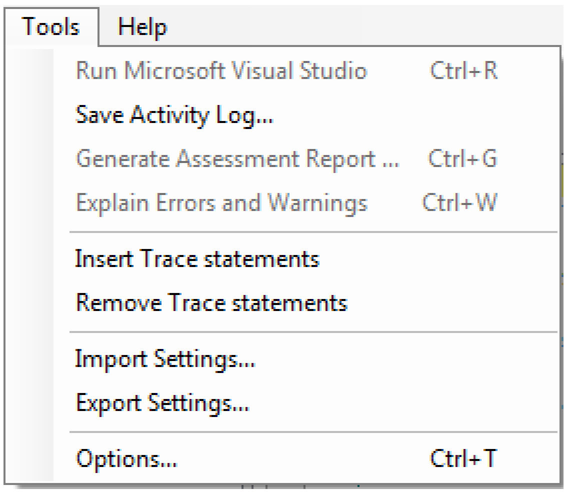
While the Trace-Match methodology alone can't warrant full functional equivalence, it is a powerful weapon in the hands of developers wishing to complete the migration process as quickly and correctly as possible.

We at Code Architects have successfully used the Trace-Match methodology for our migration services and have refined it over time. Starting with VB Migration Partner 1.32, we are releasing this methodology to the public, so that all our customers can benefit from it.

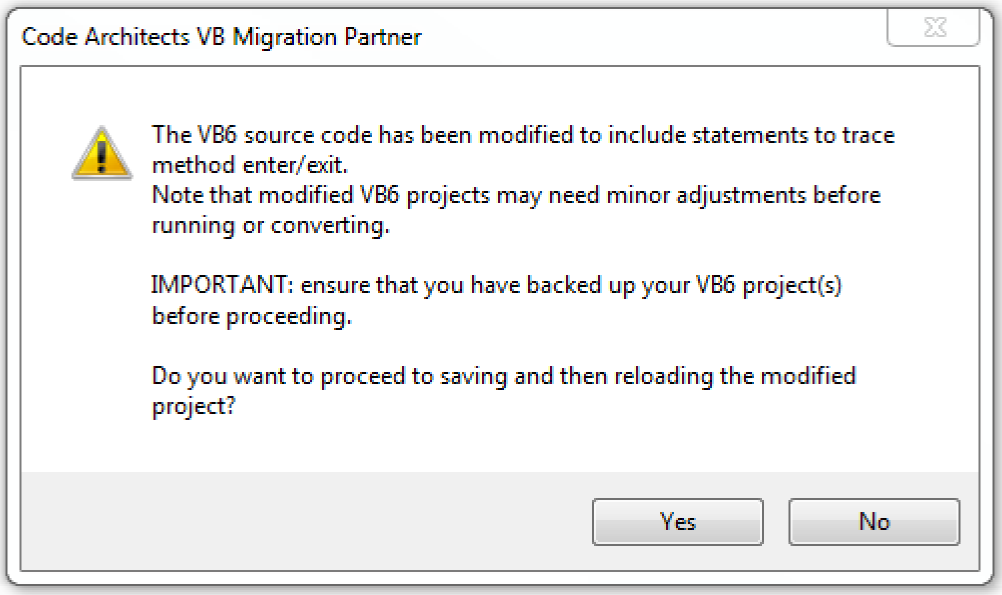
The steps to correctly implement the Trace-Match methodology are few and simple:

1. Instrument the original VB6 project with trace statements

Adding trace statements to a VB6 project is quite easy: just load a VB6 project inside VB Migration Partner (version 1.32 or later) and invoke the **Insert Trace statement** command from the Tools menu.



Once the operation is completed, the following message box appears:



Code Architects VB Migration Partner

The VB6 source code has been modified to include statements to trace method enter/exit.
Note that modified VB6 projects may need minor adjustments before running or converting.

IMPORTANT: ensure that you have backed up your VB6 project(s) before proceeding.

Do you want to proceed to saving and then reloading the modified project?

IT IS ESSENTIAL THAT YOU HAVE PREPARED A BACKUP COPY OF YOUR VB6 PROJECT, because even if the Tools menu contains the **Remove Trace statements** command, we cannot guaranteed that it works well under all possible circumstances. Besides, this command doesn't automatically undo all the manual refinements you have done (see next section).

If you are sure that you have a backup copy of your project, click "Yes" and wait until the VB6 project is saved and reloaded.

The **Insert Trace statements** command performs the following actions:

a) it modifies the VBP file to add a reference to VB6TraceLib.dll type library, which includes two tracing classes (VB6AppTrace and VB6MethodTrace)

b) if the project includes at least one BAS module, it includes the following declaration inside that module:

```
Public AppTrace As New VB6AppTrace
```

I if the project doesn't contain any BAS module, VB Migration Partner automatically adds a module named VBMigrationPartner_AppTrace.bas.

c) it adds trace statements to all methods in the application.

For example, let's say that the application contains a Module1.Bas with this code:

```
Sub Test(ByRef x As String)
    If x = "" Then
        Exit Sub
    End If
    DoSomething x
End Sub
```

This is what the module looks like after trace statements have been inserted (added statements are in boldface)

```
Public AppTrace As New VB6AppTrace

Sub Test(ByRef x As String)
    Dim trace_ As VB6MethodTrace
    Set trace_ = AppTrace.EnterMethod("Module1.Test", x)
    If x = "" Then
        AppTrace.ExitMethod trace_, x: Exit Sub
    End If
    DoSomething x
    AppTrace.ExitMethod trace_, x
End Sub
```

NOTE 1: Trace statements include simple parameters (strings, numbers, dates, etc.) Object, array, and UDT parameters aren't included in the list of traced values.

NOTE 2: in rare circumstances, VB Migration Partner may incorrectly modify the VBP project file (see point A above) and the reference to the VB6TraceLib.dll isn't valid. When this happens, you need to manually add a reference to the VB6TraceLib.dll file, which you can find in VB Migration Partner's setup folder.

You can now exit VB Migration Partner and load the modified project inside the Visual Basic 6 IDE.

2. Refine the trace mechanism and add additional trace commands (optional)

The **Insert Trace statements** command saves you the drudgery of manually inserting trace commands in each and every method of your project, yet the code it produces uses the default trace settings. In many cases you might want to refine the generated code to precisely specify how the trace should look like.

The first kind of refinement consists of correctly initializing the VB6AppTrace object. This operation is typically done in the Sub Main method, or in the Form_Load event handler of the startup form:

```
Sub Main()
    AppTrace.Init IndentTrace Or IncludeImplicit Or ShowTime, ""
    ...
End Sub
```

The Init method accepts three arguments, all of which are optional.

Trace options

The first argument for the AppTrace.Init method is an enumerated value that specifies which tracing options should be enabled. Valid values are:

None: all options are disabled (this is the default behavior)

IndentTrace: indent trace output to reflect method call nesting

ShowTime: include time in trace output (number of seconds from when trace was initialized)

IncludeTerminate: include output from Class/Form Terminate methods

IncludeImplicit: include trace statements that mark when a method is exited implicitly (e.g. because of an unanticipated error)

OmitFile: don't send to trace file

Two additional available values are specific for .NET and have no effect under VB6:

AutoGC: invoke the GC.Collect method when EnterMethod is invoked

FastWrite: non-cached file writes

The default behavior is such that trace files produced under VB6 are easily comparable with trace files generated under .NET by means of most file comparison (Diff) utilities, such as WinMerge. (See this page for a complete list of free Diff tools.)

The **IndentTrace** option provides a better understanding of execution flow, but may cause several "false unmatches" when comparing files using a Diff tool. The **ShowTime** option provides a very basic profiling feature and is guaranteed to generate a different output under VB6 and .NET.

The **IncludeTerminate** and **IncludeImplicit** options call for an additional explanation. The VB6 tracing mechanism leverages the fact that a VB6/COM object fires its Class_Terminate event as soon as the object goes out of scope. This feature is dubbed *deterministic finalization* and the VB6TraceLib library uses it to achieve to important goals:

1. trace when exactly an object is destroyed, thanks to the trace statements in its Class_Terminate event handler
2. trace when any method exits because of an unhandled error. (In this case the auxiliary VB6MethodTrace object goes out of scope and fires its Class_Terminate event, thus the VB6TraceLib has an opportunity to detect that the method exited.)

As you may know, in the .NET Framework object destruction isn't deterministic, and objects are finalized *some time* after they go out of scope or are explicitly set to Nothing. Because of this difference, the .NET version of the VB6AppTrace class *can't rely on deterministic finalization to detect when an object is destroyed or a method exits because of an error*. For this reason, the IncludeTerminate or IncludeImplicit options often generate .NET trace files that somehow differ from the VB6 trace file.

We have adopted many advanced programming techniques to reduce the number of differences between VB6 and .NET trace files. (More details later in this document.)

Finally, the **OmitTrace** flag omits writing to trace file. In this case, trace output from the VB6 or .NET application can be intercepted by means of DebugView or similar utilities. This feature can be very helpful for debug and test purposes, not just when implementing the Trace-Match methodology.

Exclude pattern

The second argument to the **AppTrace.Init** method is an optional regular expression pattern that defines which methods should be excluded from tracing. For example, you may want not to include tracing from inside MouseDown, MouseMove, and MouseUp event handlers, which you can easily achieve with this code:

```
AppTrace.Init , "_Mouse(Down|Move|Up)$"
```

The name of the method being traced is in the format *"classname.methodname"*, thus for example the MouseMove hander for the Text1 control on the Form1 form appears as "Form1.Text1_MouseMove" in the trace file. The above regular expression discards trace for any method whose name ends with "MouseDown", "MouseMove", or "MouseUp".

Excluding all the methods from a specific form, class, or module is equally easy. For example, the following code disables trace inside all the methods inside the Widget class:

```
        AppTrace.Init , "^Widget\."
```

Notice that the exclude pattern only works at runtime. In other words, VB Migration Partner always adds trace statements to all the methods of all files in the current project or project group, but you can decide which methods should *not* emit trace output when you run the VB6 (or .NET) project.

NOTE: the VB6TraceLib.dll component uses the **Microsoft VBScript Regular Expression 5.5** type library to implement the exclude feature. If such a library isn't installed on the current machine, specifying a nonempty string for the ExcludePattern argument causes a runtime error.

Trace file name

The third optional argument to the **AppTrace.Init** method is the name of the output file. By default, this file is C:\TRACE_VB6.TXT, but you can specify another file if you wish:

```
        AppTrace.Init , "^Widget\.", "c:\tracefiles\test_VB6.txt"
```

When you later convert this code to .NET and run the migrated project, the corresponding VB6AppTrace.Init method in the VB Migration Partner's support library automatically changes "VB6" (uppercase) into "NET". This means that this code will create the **test_VB6.txt** file under VB6 and the **test_NET.txt** file in the migrated project. You never need to worry that the converted .NET project might accidentally overwrite the VB6 trace file.

Additional trace output

In addition to the statements automatically added by the **Insert Trace statements** command, you are free to decorate the existing VB6 code with additional trace output calls. Both the VB6AppTrace and VB6MethodTrace objects expose the Trace method, therefore you can use either of the following syntaxes:

```
Sub Test(ByRef x As String)
    Dim trace_ As VB6MethodTrace
    Set trace_ = AppTrace.EnterMethod("Module1.Test", x)
    ...
    ' syntax #1
    AppTrace.Trace "just before calling DoSomething"
    DoSomething
    ' syntax #2
    trace_.Trace "just after calling DoSomething"
    AppTrace.ExitMethod trace_, x
End Sub
```

The output from these two methods is only slightly different: the former syntax emits the string as-is, whereas the latter one prefixes the text with the method name:

```
just before calling DoSomething
<trace from DoSomething method here>
Module1.Test: just after calling DoSomething
```

You can use either form of Trace method to trace the values of your variables, the state of the running program, details on the execution flow, and any other piece of information that is useful to understand what happens inside the application about to be migrated to .NET.

Selectively enable/disable tracing

Another way to improve the quality of your trace files is to selectively disable (and then re-enable) tracing in portions of code that are of no interest for you. For example, let's assume that you have thoroughly tested the DoSomething method and that you are sure that it works correctly. If this is the case, you might want to disable tracing before calling that method, and re-enable it immediately afterwards:

```
Sub Test(ByRef x As String)
    Dim trace_ As VB6MethodTrace
    Set trace_ = AppTrace.EnterMethod("Module1.Test", x)
    ...
    AppTrace.Enabled = False
    DoSomething
    AppTrace.Enabled = True

    AppTrace.ExitMethod trace_, x
End Sub
```

3. Run the original VB6 project and produce a set of trace files.

Conceptually this step is very simple: you just run a well-defined set of actions on the original VB6 application and produce one or more trace files.

We recommend that you prepare a script of these actions: what menu commands you select; which push buttons you click (and whether you activate them using the mouse or the keyboard); which characters you enter in fields, and so forth. It is important that you precisely describe these actions, because you're going to replicate them in the converted .NET application.

We also recommend that you run the VB6 project as a compiled stand-alone file, because the behavior of interpreted and compiled VB6 projects can sometime differ.

Finally, we suggest that you produce a separate trace file for each case test. The easiest way to do so is renaming the resulting trace file at the end of each test case from Windows Explorer, so that the file isn't overwritten when you run the next test case.

Here is a very short example of trace file that might be produced by a simple VB6 application that has one form (Form1) and one class (Widget). The trace was obtained by loading Form1 and then clicking on its two button (Command1 and Command2), where the Command2_Click event handler creates an instance of the Widget class and invokes its One method, which in turn invokes its Two method. Notice that the IndentLevel, ShowTime, and IncludeTerminate options are all enabled.

```
0000.00: --- START -------------------------
0000.00: Enter Form1.Form_Load
0000.00: Exit   Form1.Form_Load
0003.77: Enter Form1.Command1_Click
0003.77:   Enter Form1.DoSomething
0003.77:     Enter Form1.DoSomethingElse
0003.77:     Exit  Form1.DoSomethingElse
0003.77:   Exit  Form1.DoSomething
0003.77: Exit  Form1.Command1_Click
0005.02: Enter Form1.Command2_Click
0005.02:   Enter Widget.One
0005.02:     Enter Widget.Two
0005.02:     Exit  Widget.Two
0005.02:   Exit  Widget.One
0005.02: Exit  Form1.Command2_Click
0005.02: **Enter Widget.Class_Terminate
0005.02: **  Enter Widget.Two
0005.02: **  Exit  Widget.Two
0005.02: **Exit  Widget.Class_Terminate
0007.05: Enter Form1.Form_Unload
0007.05: Exit  Form1.Form_Unload
0007.06: --- END   -------------------------
```

The lines that contain two asterisks are produced by calls that originate from inside a Terminate event (in this case the Class_Terminate event of the Widget class).

By comparison, this is the trace file produced by the same series of actions when no trace option is enabled:

```
--- START -------------------------
Enter Form1.Form_Load
Exit  Form1.Form_Load
Enter Form1.Command1_Click
Enter Form1.DoSomething
Enter Form1.DoSomethingElse
Exit  Form1.DoSomethingElse
Exit  Form1.DoSomething
Exit  Form1.Command1_Click
Enter Form1.Command2_Click
Enter Widget.One
Enter Widget.Two
Exit  Widget.Two
Exit  Widget.One
Exit  Form1.Command2_Click
Enter Form1.Form_Unload
Exit  Form1.Form_Unload
--- END   -------------------------
```

As you see, calls that originate from the Class_Terminate event of the Widget class aren't displayed any longer and that the lack of indentation makes it more difficult to follow the execution path.

4. Convert the VB6 project to .NET

Nothing special here: this is standard stuff for all VB Migration Partner users.

It is essential, however that you strictly adhere to the convert-test-fix methodology we recommend in our documentation. This methodology dictates that you never modify the converted .NET code to solve compilation and runtime errors; instead, you add one or more pragmas to the original VB6 source code and then re-convert the project to .NET.

NOTE: Because pragmas are just special VB6 comments, the convert-test-fix approach ensures that the trace files you have created at previous steps continue to be valid and correctly describe how the VB6 code behaves.

This step is complete when you have a .NET project that compiles correctly and run without any runtime exception. Unfortunately, running without errors doesn't necessarily mean that the .NET code is functionally equivalent to the original VB6 project.

Ensuring functional equivalence is the purpose of the Trace-Match methodology.

5. Run the same tests on the migrated .NET project and produce a set of trace files

Now that the .NET code runs with no visible exceptions, you can perform the same set of scripts and test cases seen in step 3, and produce a similar set of trace files.

6. Compare the VB6 and .NET trace files

You now have all the trace information you need, both from the original VB6 project and the converted .NET project. Using a Diff tool like WinMerge it's easy to spot any major and minor difference between the VB6 and .NET trace files.

Notice that a difference in trace files doesn't really give you the absolute certainty that the converted code isn't functionally equivalent to the original project. For example, the .NET trace file might look like this:

```
0000.00: --- START -------------------------
0000.00: Enter Form1.Form_Load
0000.00: Exit  Form1.Form_Load
0003.22: Enter Form1.Command1_Click
0003.22:   Enter Form1.DoSomething
0003.22:     Enter Form1.DoSomethingElse
0003.22:     Exit  Form1.DoSomethingElse
0003.22:   Exit  Form1.DoSomething
0003.22: Exit  Form1.Command1_Click
0004.81: Enter Form1.Command2_Click
0004.81:   Enter Widget.One
0004.81:     Enter Widget.Two
0004.81:     Exit Widget.Two
0004.81:   Exit Widget.One
0004.81: Exit Form1.Command2_Click
0004.81: Enter Form1.Form_Unload
0005.15: Exit  Form1.Form_Unload
0005.22: **Enter Widget.Class_Terminate
0005.22: **  Enter Widget.Two
0005.22: **  Exit  Widget.Two
0005.22: **Exit  Widget.Class_Terminate
```

Even not considering the difference in timing, the most obvious difference with the original VB6 trace file (see step 3 above) is that calls originated from the Class_Terminate event in the Widget class come after the Form_Unload method in the .NET code, whereas they followed the exit from Command1_Click method in the VB6 project.

The reason for this difference is that .NET objects are usually destroyed later than the corresponding VB6 object. Worse, if the object is being destroyed when the entire application is shutting down you might not see any trace from their Class_Terminate event handler.

For the same reason, you might not see the "---- END --- " line in trace files produced under .NET. In fact, the VB6AppTrace object might be destroyed before the last operation.

To avoid false mismatches caused by *undeterministic finalization*, by default VB Migration Partner's trace mechanism omits trace output originated from inside the Terminate event handler of forms, classes, and user controls. However, if you specified the **IncludeTerminate** option (see step 2 above), this output is included in the trace file but is marked with a double asterisks.

Alternatively, you can decide to enable the IncludeTerminate option and later use a Grep utility to discard those lines from the output. For example, you might use the FIND utility from Windows command line, as follows:

```
FIND c:\trace_net.txt "**" /V
```

Forced garbage collections

As explained previously, the different finalization mechanism explains why the VB6 and .NET versions of the trace file often differ. A simple trick that greatly reduces these differences is the **AutoGC** trace option, which you enable with this code (in the original VB6 project):

```
AppTrace.Init IndentTrace Or IncludeImplicit Or AutoGC
```

As its name suggests, the **AutoGC** option forces a full garbage collection each time a method is exited (that is, when the VB6AppTrace.ExitMethod is called). Such forced garbage collections reduces the gap between deterministic (VB6) and undeterministic (.NET) finalization, which in turn reduces the differences between VB6 and .NET trace files.

Keep in mind, however, that even when you enable this option, the deep differences between the two finalization mechanisms aren't solved completely. Also, each garbage collection adds overhead to your code, so be prepared for higher execution timings when this option is enabled.

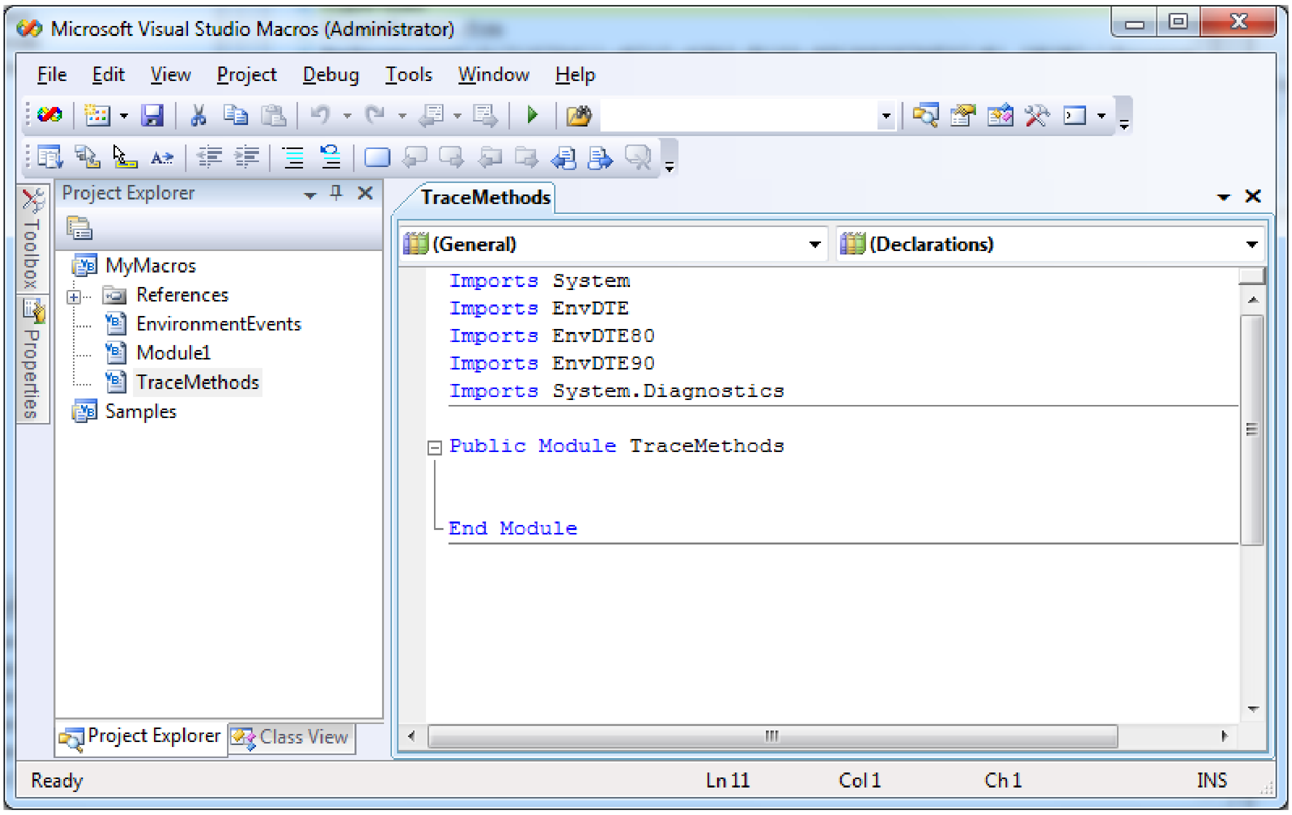
7. Removing trace statements

Once you are 100% sure that the converted code behaves exactly like the original VB6 project and that functional equivalence has been reached, you can remove all trace statements from the VB6 project and perform the final conversion, to produce a fully working .NET project that contains no trace statements.

This step is fully automatic, thanks to the **Remove Trace statements** command from VB Migration Partner's Tools menu.

If you have decided not to stick to the convert-test-fix methodology, however, you will have to manually remove all trace statements from the .NET project. You can easily do it with a global Find and Replace command, or you can define a Visual Studio macro that does the job for you. The following text describes how to create such a macro.

1. Launch Visual Studio and select the Macro IDE command from the Macros submenu of the Tools menu. This action brings you to the Microsoft Visual Studio Macros environment.
2. Select the MyMacros node in the Project Explorer window, then select the Add Module command from the Project menu.
3. Name the new module appropriately – for example, “TraceMethods” in the dialog box that appears, then click on the Add button to create the module. This is what you should see now inside the Macro IDE:



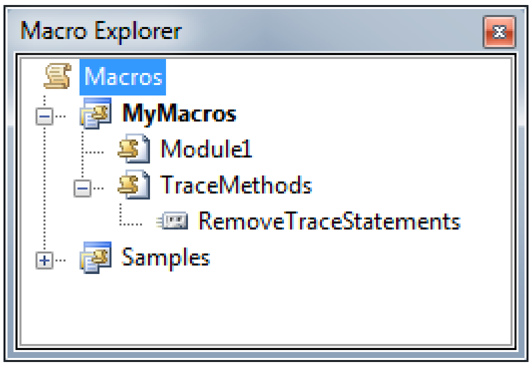
```
Imports System
Imports EnvDTE
Imports EnvDTE80
Imports EnvDTE90
Imports System.Diagnostics

Public Module TraceMethods


End Module
```

Enter the following code inside the module:

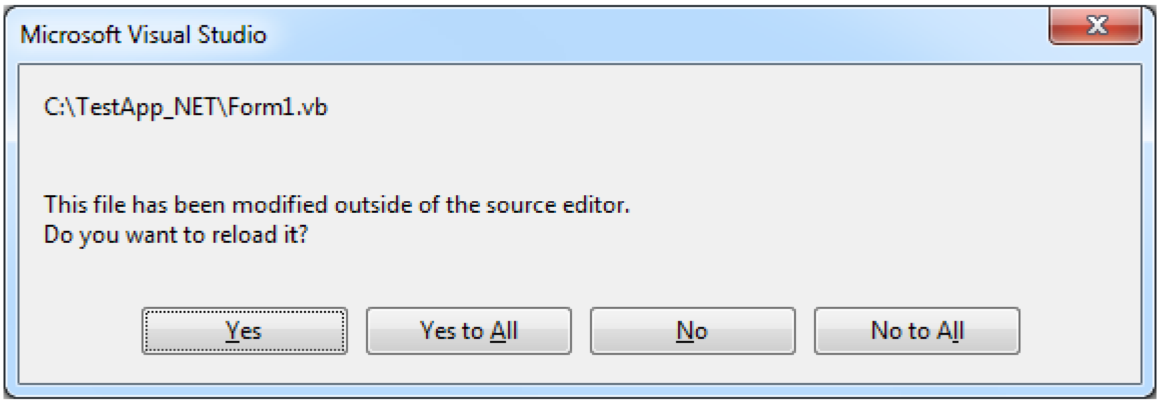
```
Public Module TraceMethods

   Sub RemoveTraceStatements()
      Dim pattern As String = "([ \t]*Dim trace_ As VB6MethodTrace.+?\r\n(\r\n)?)" _
        & "|([ \t]*AppTrace\.(ExitMethod|Init|Trace|Enabled)\(.+?(:|\r\n))" _
        & "|([ \t]*trace_\.Trace\(.+?(:|\r\n))" _
        & "|(Public AppTrace As New VB6AppTrace\r\n(\r\n)?)"
      Dim reTrace As New System.Text.RegularExpressions.Regex(pattern, _
        System.Text.RegularExpressions.RegexOptions.IgnoreCase)
      ' iterate over all projects in current solution
      For Each  prj As EnvDTE.Project In DTE.Solution.Projects
         ' iterate over all project items in this project
         For Each prjItem As EnvDTE.ProjectItem In prj.ProjectItems
            ' iterate over all files in this project item
           For i As Short = 0 To prjItem.FileCount - 1
               ' read this file
               Try
                  Dim filename As String = prjItem.FileNames(i)
                  ' skip if in the My Project folder
                  If filename.Contains("\My Project\") Then Continue For
                  Dim text As String = System.IO.File.ReadAllText(filename)
                  ' remove all trace statements and save
                  text = reTrace.Replace(text, "")
                  System.IO.File.WriteAllText(filename, text)
               Catch ex As System.Exception
                  ' ignore errors
               End Try
             Next
           Next
       Next
   End Sub

End Module
```

You can now save the macro you've just created and return the Visual Studio (use the Close and Return command from the File menu).

From inside Visual Studio, select the Macro Explorer command from the Macros submenu of the Tools menu, which brings up the Macro explorer window.



You can finally right-click on the RemoveTraceStatements element and select the "Run" command to remove all trace statements. The following dialog might appear: if so, just click "Yes" to reload all modified files.


Microsoft Visual Studio

C:\TestApp_NET\Form1.vb

This file has been modified outside of the source editor.
Do you want to reload it?

IMPORTANT: this action is destructive and overwrites all the files in the current VB.NET solution. It is therefore essential that you run the macro only after creating a backup of the solution, in case the remove operation mistakenly deletes vital portions of your code.

Conclusions

The Trace-Match methodology is a powerful tool that allows VB Migration Partner users to quickly reach functional equivalence and to obtain "objective" evidence that such equivalence has been reached.

By adding additional trace statements, developers can easily create a set of test cases that can run unattended. For example, such additional statements can trace the contents of user interface elements – e.g. the text inside a TextBox, or the selected item in a ListBox control – so that no human intervention is necessary to confirm that the converted project delivers correct results.

Code Architects is the only vendor that offers Trace-Match or a comparable tracing methodology. Not only do we allow you to *prove* that your code works and behaves like the original VB6 code, we even provide you with the tools that quickly insert and remove all the trace statements on your behalf.