Function Pointers in VBScript

Written by Meir Bar-Tal,
Senior Automation Expert
October 2007, Beer-Sheva, Israel
mailto:bartalme@freeshell.org
Version 1.00

Copyright © 2007 by Meir Bar-Tal. Permission is granted to copy distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Text being the text from the title up to the table of contents, and with no Back-Cover Texts.
A copy of the license can be obtained from http://www.gnu.org.
Be aware that there may be a newer version of this document!
Check http://www.advanced qtp.com for the latest release. If you want to distribute this document, I suggest you to link to the URL above to prevent the distribution of outdated versions.
You may also download the source code of the examples at http://www.advanced qtp.com. The example code is free software under the terms of the GNU General Public License.

Abstract: Function pointers are used in C and C++ to enable callbacks and as a result, more generic and parsimonious coding. This article shows how to implement function pointers in VBScript using the Scripting.Dictionary object and the Command Wrapper design pattern. It is also shown how to implement “constructor” functions, callbacks and event handlers.

Keywords: function pointer, callback, delegate, scripting dictionary, hash table, design pattern, command wrapper, event handler.

Contents

INTRODUCTION..................................................................................................................................................2
FUNCTION POINTERS...........................................................................................................................................2
DESIGN PATTERNS...........................................................................................................................................2
COMMAND WRAPPER....................................................................................................................................2
SCRIPTING DICTIONARY....................................................................................................................................4
METHOD...........................................................................................................................................................5
CONCLUSION.....................................................................................................................................................7
REFERENCES.....................................................................................................................................................7
APPENDIX 1: SAMPLE CODE...........................................................................................................................8
APPENDIX 3: CALLBACK EXAMPLE................................................................................................................14
Introduction

In the following sections I shall explain several basic concepts that are essential to understand the rationale behind the suggested method. The list might be non-exhaustive and the reader is hereby encouraged to recur to complementary sources to fill-in any eventual knowledge gaps.

Function Pointers

A function pointer is a variable that stores the memory address of a block of code that is programmed to fulfill a specific function. Function pointers are useful to avoid complex switch case structures; instead, they support direct access in run-time to previously loaded functions or class methods. This enables the construction of callback functions. A callback is, in essence, executable code that is passed as argument to a function. This enables more generic coding, by having lower-level modules calling higher-level functions or subroutines.

Function pointers are supported by programming languages like C and C++. A recent good introductory text on the subject with implementation examples is Lars Haendel (2005).

This article shows how to implement function pointers in VBScript.

Design Patterns

Design patterns are well established generic design techniques destined to make software design and the derived code more parsimonious, scalable, and reusable, as well as more easily maintainable. This article will delve only into the command wrapper design pattern, which is important for its purpose. For a more extensive explanation of design patterns, the reader is encouraged to read the literature in the field (e.g., Gamma, Helm, Johnson, & Vlissides, 1997).

The command-wrapper design pattern enables to load blocks of code dynamically to memory as well as other uses as will be reviewed below. The singleton design pattern is used to restrict instantiation of a class to one object. This is useful when there is no need for more than one object within a system (e.g., a reporter/logger object). For example, the factory design pattern usually implements the singleton design pattern to have a single interface from which to retrieve references to a specific class object or to a variety of class objects.

Command Wrapper

The command wrapper design pattern is a technique that implements a function as a class, usually with a single public method. During run-time, it is possible to instantiate the class and thus to get a reference (or pointer) to the object. This way, the code embodied in such methods can be dynamically loaded, a feature that can be of great value in systems that are poor in resources. This is especially true when the to-be called function is rarely used. After creating the object, it is possible to execute its method according to need. For example, let us assume a function that adds numbers (note: this example is, of course, trivial. It is done just to illustrate the point.)

```vbnet
Public Function Sum(ByVal arrNumbers)
    Dim ix
    If (Not IsArray(arrNumbers)) Then 'Not an array, so nothing to do – exit function
        'Add your error handling code here
        Exit Function
    End If

    Sum = 0
    For ix = LBound(arrNumbers) To UBound(arrNumbers)
        If IsNumeric(arrNumbers(ix)) Then
            Sum = Sum + arrNumbers(ix)
        Else
            'Add your error handling code here
        End If
    Next
End Function
```
Function Pointers in VBScript

'Test the function
MsgBox Sum(Array(23, 56, 78, 95, 114)), vbOKOnly, "Result"    'Display result returned by the Sum function

Now, let us convert this function into an object using the Command Wrapper design pattern:

Class Sum
    Private m_arrVarNumbers
    Private m_varResult
    Private Sub Class_Initialize()
        'Initialize the Numbers member as an empty array
        ReDim m_arrVarNumbers(-1)
    End Sub
    Public Function Init(ByVal arrVarNumbers)
        Numbers = arrVarNumbers
    End Function
    Public Default Function Exec()
        Dim ix, arrNumbers
        If (Not IsArray(Numbers)) Then 'Not an array, so nothing to do – exit function
            'Add your error handling code here
            Exec = "Invalid data type was supplied to perform the operation."
            Exit Function
        End If
        Else
            arrNumbers = Numbers
        End If
        Result = 0
        For ix = LBound(arrNumbers) To UBound(arrNumbers)
            If (IsNumeric(arrNumbers(ix))) Then
                Result = Result + arrNumbers(ix)
            Else
                'Add your error handling code here
            End If
        Next
        Exec = Result
    End Function
    Public Property Get Numbers()
        Numbers = m_arrVarNumbers
    End Property
    Private Property Let Numbers(ByVal arrVarNumbers)
        m_arrVarNumbers = arrVarNumbers
    End Property
    Public Property Get Result()
        Result = m_varResult
    End Property
    Private Property Let Result(ByVal varResult)
        m_varResult = varResult
    End Property
End Class

'Public Function GetSum(ByVal arrNumbers)
    Set GetSum = New Sum
    GetSum.Init(arrNumbers)
End Function

1 Recall that QuickTest Pro does not support custom classes instantiation from within a test.

Meir Bar-Tal
'Test the class
Dim objSum, arrNumbers

arrNumbers = Array(23, 56, 78, 95, 114)  'Assign an array of numbers
Set objSum = GetSum(arrNumbers)  'Get an instance of class Sum and initialize it with numbers array
objSum.Exec()  'Execute Sum (Exec method)
MsgBox objSum.Result, vbOKOnly, "Result"  'Display result stored in Result property

Or:
MsgBox GetSum(Array(23, 56, 78, 95, 114)).Exec, vbOKOnly, "Result"  'Display result returned by the Exec method

Scripting Dictionary

In VBScript, the scripting dictionary is an object that stores key-item pairs using a hashing algorithm. The items can be accessed using their corresponding keys. Basically, the dictionary is useful to store variant type, non-structured data, such as the catalog number of an item in an inventory list, or the authors of a list of books, and to retrieve the data using the unique keys supplied, as in the following example:

Dim dicBooksAuthors

'Create an instance of the scripting dictionary class
Set dicBookAuthors = CreateObject("Scripting.Dictionary")

With dicBookAuthors
  'Add some books (keys) and authors (items)
  .Add "The Selfish Gene", "Richard Dawkins"
  .Add "The Mismeasure of Man", "Stephen J. Gould"
  .Add "The Da Vinci Code", "Dan Brown"

  'Answer the question: who wrote "The Selfish Gene"?
  strBook = "The Selfish Gene"
  MsgBox .item(strBook), vbOKOnly+vbInformation, "Author query result for book: " & strBook
End With

The example above demonstrates the power of a dictionary as a data storage and data retrieval device. For instance, it eliminates the need for item search functions (direct access) and avoids duplicate key creation.

However, the dictionary can be much more than just a data storage and retrieval device. Thanks to its capacity to store variant data types, it is actually possible to store references to objects of different types. For instance, it is possible to store complex data structures by using nested dictionaries. In the next chapter, we shall delve into a method to exploit this feature of the scripting dictionary object to implement function pointers in VBScript.

---

2 This issue, however, is out of the scope of the current paper, and will be covered separately in M. Bar-Tal, 2007b (forthcoming).
Method

Earlier in this article, we have shown that it is possible to encapsulate any function as a class by using the command wrapper design pattern. In addition, it has been indicated that the VBScript Scripting Dictionary object is equipped with the capacity of storing variant data types, including references to objects. In what follows a method will be described to combine these two features into a powerful design and programming technique.

Recall that a function pointer is a variable that stores the memory address of a function or instantiated class object. Following the example given above in the Command Wrapper section, we shall now show how it is possible to build function pointers using a Scripting Dictionary.

```
Dim dicFunctionHandler

' Create an instance of the scripting dictionary class
Set dicFunctionHandler = CreateObject("Scripting.Dictionary")

' Load some functions
With dicFunctionHandler
    .Add "+", New Sum
    .Add ",", New Subtract
    .Add "+", New Multiply
    .Add ",", New Divide
End With

' Execute the functions using the Function Handler
With dicFunctionHandler
    MsgBox .item("+").Exec(), vbOKOnly+vbInformation, "Result (+)" ' Display result returned by the Exec method
End With
With .item("-")
    MsgBox .Exec(), vbOKOnly+vbInformation, "Result (-)" ' Display result returned by the Exec method
End With
With .item("*")
    MsgBox .Exec(), vbOKOnly+vbInformation, "Result (*)" ' Display result returned by the Exec method
End With
With .item("/")
    MsgBox .Exec(), vbOKOnly+vbInformation, "Result (/)" ' Display result returned by the Exec method
End With

Or, using the “constructors”:

' Load some functions
With dicFunctionHandler
    .Add "+", GetSum(Array(23, 56, 78, 95, 114))
    .Add ",", GetSubtract(Array(117, 23))
    .Add "+", GetMultiply(Array(7, 5))
    .Add ",", GetDivide(Array(84, 12))
End With

' Execute the functions using the Function Handler
With dicFunctionHandler
    MsgBox .item("+").Exec(), vbOKOnly+vbInformation, "Result (+)" ' Display result returned by the Exec method
    MsgBox .item("-").Exec(), vbOKOnly+vbInformation, "Result (-)" ' Display result returned by the Exec method
    MsgBox .item("*").Exec(), vbOKOnly+vbInformation, "Result (*)" ' Display result returned by the Exec method
    MsgBox .item("/").Exec(), vbOKOnly+vbInformation, "Result (/)" ' Display result returned by the Exec method
End With
```

3 The source code of the classes used in the example can be found in Appendix 1.
In the above example we have shown how to:

1. Implement the Command Wrapper design pattern in VBScript;
2. Implement a “constructor” for a class in VBScript;
3. Implement a function handler using a scripting dictionary;
4. Instantiate such custom classes and load them to a scripting dictionary;
5. Call the loaded function via the dictionary key and retrieve the result.

This is the method suggested in this paper to implement a function pointer in VBScript.

There is also another possible way to call a function implemented with this method, as follows:

```vbscript
'Execute the functions using the Function Handler
MsgBox dicFunctionHandler(“+”).Exec(), vbOKOnly+vbInformation, "Result (+)"
MsgBox dicFunctionHandler(“-”).Exec(), vbOKOnly+vbInformation, "Result (-)"
MsgBox dicFunctionHandler(“*”).Exec(), vbOKOnly+vbInformation, "Result (*)"
MsgBox dicFunctionHandler(“/”).Exec(), vbOKOnly+vbInformation, "Result (/)"
```

and this is because the item property is the scripting dictionary’s default property.

In a similar fashion, it is possible to define the Exec methods of the above mentioned classes as Default (by declaring it: Public Default Function) and then the code above can be further reduced to:

```vbscript
'Execute the functions using the Function Handler
MsgBox dicFunctionHandler(“+”), vbOKOnly+vbInformation, "Result (+)"
MsgBox dicFunctionHandler(“-”), vbOKOnly+vbInformation, "Result (-)
MsgBox dicFunctionHandler(“*”), vbOKOnly+vbInformation, "Result (*)"
MsgBox dicFunctionHandler(“/”), vbOKOnly+vbInformation, "Result (/)"
```

The readers are encouraged to try to execute the sample code shown in this paper, as well as to try the method to implement their own functionality.

**Discussion**

We have seen so far how to implement the Command Wrapper design pattern in VBScript and a “constructor” for a class in VBScript, as well as a function handler using a scripting dictionary. We have also shown how to instantiate such custom classes and load them to a scripting dictionary, together with different ways to call the loaded function via the dictionary key and to retrieve the result. We have also indicated that this method is, in fact, equivalent to the implementation of a function pointer in C or C++.

The general uses and benefits of function pointers are explained elsewhere (e.g., Lars Haendel, 2005), and hence they will not be covered here. In what remains I shall attempt to convey in which cases the implementation of this design pattern in VBScript in general, and with Quicktest Pro (QTP) in particular, might be of benefit.

First, the method presented in this paper should be of great value when the basic hardware configuration is poor, i.e., when the system is low in RAM, by means of dynamic loading of code blocks. Recall that common QTP usage requires the automation developer to add every single function library to the test resources.

Second, by implementing a function handler as illustrated above, it would be possible to build a generic controller component that would enable execution of real keyword-driven scripts. With such a component it would be possible, for example, to define the flow of the different code blocks in an external file, such as an XML file.

Third, the method can be used to emulate callbacks, which is a central feature of function pointers. This can be easily done by passing the function handler entry (dictionary item) to another function.

Fourth, the method can be used to emulate event handling. This can be achieved by returning a string with the name of the function to be called. Please notice that this technique would yield a highly parsimonious coding style, for it makes the need for decision structures to analyze the return code of a function obsolete. An example for this can be found in Appendix 2.
Conclusion

This paper attempted to demonstrate how to implement function pointers in VBScript, and pinpointed the possible uses and advantages of the method in general and particularly for QTP. It is concluded that the technique can help developers to achieve more efficient and generic design and code, and better run-time resources management. Future forthcoming articles will further expand on several topics mentioned throughout this paper.

References


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1997). Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Publishing.
Appendix 1: Sample Code

Class Sum
    Private m_arrVarNumbers
    Private m_varResult

    Private Sub Class_Initialize()
        'Initialize the Numbers member as an empty array
        ReDim m_arrVarNumbers(-1)
    End Sub

    Public Function Init(ByVal arrVarNumbers)
        Numbers = arrVarNumbers
    End Function

    Public Default Function Exec()
        Dim ix, arrNumbers
        If (Not IsArray(Numbers)) Then 'Not an array, so nothing to do – exit function
            Exec = "Invalid data type was supplied to perform the operation."
            Exit Function
        End If
        If (UBound(arrNumbers) - LBound(arrNumbers) + 1 <= 1) Then
            'Array is empty or has single item - Add your error handling code here
            Exec = "Not enough data was supplied to perform the operation."
            Exit Function
        End If
        Else
            arrNumbers = Numbers
        End If
        Result = 0
        For ix = LBound(arrNumbers) To UBound(arrNumbers)
            If (IsNumeric(arrNumbers(ix))) Then
                Result = Result + arrNumbers(ix)
            Else
                'Add your error handling code here
            End If
        Next
        Exec = Result
    End Function

    Public Property Get Numbers()
        Numbers = m_arrVarNumbers
    End Property

    Private Property Let Numbers(ByVal arrVarNumbers)
        m_arrVarNumbers = arrVarNumbers
    End Property

    Public Property Get Result()
        Result = m_varResult
    End Property

    Private Property Let Result(ByVal varResult)
        m_varResult = varResult
    End Property
End Class

'This function behaves as a constructor and returns an initialized instance of the class'
Public Function GetSum(ByVal arrNumbers)
    Set GetSum = New Sum
    GetSum.Init(arrNumbers)
End Function

4 Recall that QuickTest Pro does not support custom classes instantiation from within a test.
Meir Bar-Tal
Class Subtract
    Private m_arrVarNumbers
    Private m_varResult

Private Sub Class_Initialize()
    'Initialize the Numbers member as an empty array
    ReDim m_arrVarNumbers(-1)
End Sub
Public Function Init(ByVal arrVarNumbers)
    Numbers = arrVarNumbers
End Function
Public Default Function Exec()
    Dim ix, arrNumbers
    If Not IsArray(Numbers) Then 'Not an array, so nothing to do – exit function
        'Add your error handling code here
        Exec = "Invalid data type was supplied to perform the operation."
        Exit Function
    Else
        arrNumbers = Numbers
        If UBound(arrNumbers) - LBound(arrNumbers) + 1 <= 1 Then
            'Array is empty or has single item - Add your error handling code here
            Exec = "Not enough data was supplied to perform the operation."
            Exit Function
        End If
    End If
    Result = arrNumbers(LBound(arrNumbers)) - arrNumbers(LBound(arrNumbers)+1)
    For ix = LBound(arrNumbers)+2 To UBound(arrNumbers)
        If IsNumeric(arrNumbers(ix)) Then
            Result = Result - arrNumbers(ix)
        Else
            'Add your error handling code here
        End If
    Next
    Exec = Result
End Function
Public Property Get Numbers()
    Numbers = m_arrVarNumbers
End Property
Private Property Let Numbers(ByVal arrVarNumbers)
    m_arrVarNumbers = arrVarNumbers
End Property
Public Property Get Result()
    Result = m_varResult
End Property
Private Property Let Result(ByVal varResult)
    m_varResult = varResult
End Property
End Class

'This function behaves as a constructor and returns an initialized instance of the class
Public Function GetSubtract(ByVal arrNumbers)
    Set GetSubtract = New Subtract
    GetSubtract.Init(arrNumbers)
End Function
Class Multiply
   Private m_arrVarNumbers
   Private m_varResult

Private Sub Class_Initialize()
   'Initialize the Numbers member as an empty array
   ReDim m_arrVarNumbers(-1)
End Sub
Public Function Init(ByVal arrVarNumbers)
   Numbers = arrVarNumbers
End Function
Public Default Function Exec()
   Dim ix, arrNumbers
   If (Not IsArray(Numbers)) Then 'Not an array, so nothing to do – exit function
      'Add your error handling code here
      Exec = "Invalid data type was supplied to perform the operation."
      Exit Function
   Else
      arrNumbers = Numbers
      If (UBound(arrNumbers) - LBound(arrNumbers) + 1 <= 1) Then
         'Array is empty or has single item - Add your error handling code here
         Exec = "Not enough data was supplied to perform the operation."
         Exit Function
      End If
      Result = arrNumbers(LBound(arrNumbers)) * arrNumbers(LBound(arrNumbers)+1)
      For ix = LBound(arrNumbers)+2 To UBound(arrNumbers)
         If IsNumeric(arrNumbers(ix)) Then
            Result = Result * arrNumbers(ix)
         Else
            'Add your error handling code here
         End If
      Next
      Exec = Result
   End If
End Function
Public Property Get Numbers()
   Numbers = m_arrVarNumbers
End Property
Private Property Let Numbers(ByVal arrVarNumbers)
   m_arrVarNumbers = arrVarNumbers
End Property
Public Property Get Result()
   Result = m_varResult
End Property
Private Property Let Result(ByVal varResult)
   m_varResult = varResult
End Property
End Class

'This function behaves as a constructor and returns an initialized instance of the class
Public Function GetMultiply (ByVal arrNumbers)
   Set GetMultiply = New Multiply
   GetMultiply.Init(arrNumbers)
End Function
Class Divide
    Private m_arrVarNumbers
    Private m_varResult

    Private Sub Class_Initialize()
        'Initialize the Numbers member as an empty array
        ReDim m_arrVarNumbers(-1)
    End Sub

Public Function Init(ByVal arrVarNumbers)
    Numbers = arrVarNumbers
End Function

Public Default Function Exec()
    Dim ix, arrNumbers
    If (Not IsArray(Numbers)) Then 'Not an array, so nothing to do – exit function
        'Add your error handling code here
        Exec = "Invalid data type was supplied to perform the operation."
        Exit Function
    Else
        arrNumbers = Numbers
        If (UBound(arrNumbers) - LBound(arrNumbers) + 1 <= 1) Then
            'Array is empty or has single item - Add your error handling code here
            Exec = "Not enough data was supplied to perform the operation."
            Exit Function
        End If
        If (IsNumeric(arrNumbers(LBound(arrNumbers))) And IsNumeric(arrNumbers(LBound(arrNumbers)+1))) And (arrNumbers(LBound(arrNumbers)+1) <> 0) Then
            Result = arrNumbers(LBound(arrNumbers)) / arrNumbers(LBound(arrNumbers)+1)
        Else
            'Add your error handling code here
            Exec = "Invalid data was supplied to perform the operation."
            Exit Function
        End If
        For ix = LBound(arrNumbers)+2 To UBound(arrNumbers)
            If (IsNumeric(arrNumbers(ix)) And (arrNumbers(ix) <> 0)) Then
                Result = Result / arrNumbers(ix)
            Else
                'Add your error handling code here
            End If
        Next
        Exec = Result
    End If
End Function

Public Property Get Numbers()
    Numbers = m_arrVarNumbers
End Property

Private Property Let Numbers(ByVal arrVarNumbers)
    m_arrVarNumbers = arrVarNumbers
End Property

Public Property Get Result()
    Result = m_varResult
End Property

Private Property Let Result(ByVal varResult)
    m_varResult = varResult
End Property

End Class

' This function behaves as a constructor and returns an initialized instance of the class
Public Function GetDivide(ByVal arrNumbers)
    Set GetDivide = New Divide
    GetDivide.Init(arrNumbers)
End Function
Appendix 2: Event Handling Example

We shall show here how to implement an event handler in VBScript for a typical login flow in QTP. The flow includes three functions: Login, FirstTest and Logout. In addition, two additional functions (AbortTest and Cleanup) are included for error handling and orderly exiting the test flow. Within the TestFlow Sub, the functions are loaded using a generic function (class) loader function (GetFunction), and then called according to their order in the arrRunActions, with their corresponding arguments from arrActionArgs. As can be seen in the code, in case of error the Login.Exec function returns the name of the action to be performed (AbortTest). This action is then performed by means of the statement: 

\[ \text{strNextAction} = \text{dicFunctionHandler(strNextAction)}(\text{arrArgs}) \]

and since the for loop continues to advance, the normal course of the test flow is altered such that the FirstTest action is never executed.

```
' VB Script Document
Option Explicit

Class Login
    Public Default Function Exec(ByVal arrArgs)
        'Add your code here
        If (Not isArray(arrArgs)) Then
            MsgBox "Error: username & password are mandatory parameters.", vbOKOnly+vbcritical, "Login"
            Exec = "AbortTest"
        Else
            MsgBox Join(arrArgs, ","), vbOKOnly+vbinformation, "Login"
            Exec = ""
        End If
    End Function
End Class

Class Logout
    Public Default Function Exec(ByVal arrArgs)
        MsgBox "Exec", vbOKOnly+vbinformation, "Logout"
        'Add your code here
    End Function
End Class

Class AbortTest
    Public Default Function Exec(ByVal arrArgs)
        MsgBox "Exec", vbOKOnly+vbcritical, "AbortTest"
        Exec = "Cleanup"
        'Add your code here
    End Function
End Class

Class FirstTest
    Public Default Function Exec(ByVal arrArgs)
        MsgBox "Exec", vbOKOnly+vbinformation, "FirstTest"
        'Add your code here
    End Function
End Class

Class Cleanup
    Public Default Function Exec(ByVal arrArgs)
        MsgBox "Exec", vbOKOnly+vbinformation, "Cleanup"
        'Add your code here
    End Function
End Class

' This generic function will load any class
Public Function GetFunction(ByVal strFunctionName)
    Execute "Set GetFunction = New " & strFunctionName
End Function
```

Meir Bar-Tal
Function Pointers in VBScript

This sub will run the flow

Public Sub TestFlow(ByVal arrLoadActions, ByVal arrRunActions, ByVal arrActionArgs)
    Dim dicFunctionHandler, ix, arrArgs, strNextAction

    Set dicFunctionHandler = CreateObject("Scripting.Dictionary")
    Set arrArgs = CreateObject("Scripting.Dictionary")

    'Load the required functions
    With dicFunctionHandler
        For ix = LBound(arrLoadActions) To UBound(arrLoadActions)
            .Add arrLoadActions (ix), GetFunction(arrLoadActions (ix))
        Next
    End With

    'Run the required flow
    strNextAction = ""
    For ix = LBound(arrRunActions) To UBound(arrRunActions)
        'Get the action arguments
        arrArgs = split(arrActionArgs(ix), ",")
        If (UBound(arrArgs) - LBound(arrArgs) + 1 = 0) Then
            'If no args are found, pass an empty string
            arrArgs = ""
        End If
        If (strNextAction = "") Then
            'Run the next planned action
            strNextAction = dicFunctionHandler(arrRunActions(ix))(arrArgs)
        Else
            'Run the action returned by the previously call action
            strNextAction = dicFunctionHandler(strNextAction)(arrArgs)
        End If
    Next
End Sub

Call TestFlow(Array("Login", "FirstTest", "Logout", "AbortTest", "Cleanup"), Array("Login", "FirstTest", "Logout"), Array("User;123456", ",", ""))
Appendix 3: Callback Example

We shall show here how to implement a callback with the classes defined in Appendix 1 and used in this paper. The example used here will be the calculation of the total amount plus the tax on the sum of several items prices.

'Define a constant with the tax value multiplier
Const CONST_DBL_VAT = 1.155

Now, by dynamically loading the functions without the function handler (as shown in the Method chapter):

'Execute the (*) function on the result of the (+) function to calculate the tax on an item
MsgBox GetMultiply(Array(GetSum(Array(23, 56, 78, 95, 114)), CONST_DBL_VAT)), vbOKOnly, "Result (Tax)"

True, the syntax looks complicated, but it is very useful.