Embedding Scripting in Your Applications

by Rod da Silva

CA-World 2000
eBusiness Solutions in Internet Time
JP175SN

Introduction

This paper represents an introduction into the fascinating world of scripting. It will show you how, with very little effort, you can produce truly open, robust software by adding scripting capabilities to your business applications. The focus of this script integration is the Microsoft Script Control—an OCX available for free download from http://www.microsoft.com\scripting. By integrating this ActiveX control into applications built with your development language of choice, you can quickly and easily support all of today's popular scripting languages such as VBScript and JScript (Microsoft's version of JavaScript), providing a degree of openness that was previously unavailable.

This paper endeavors to describe how to design your business applications in order to take full advantage of embedded scripts. It does not spend much time covering background information as to what scripting is and how Active Scripting technology (upon which the Script Control is based) works. For more information on this, please refer to the web site above. Further, while the techniques discussed here-in will work with any modern Windows development language that supports ActiveX, all example will be shown in CA-Visual Objects.

So Why Would I Want to Support Scripting?

Most of you are probably familiar with CA-Clipper, the father of CA-Visual Objects, and have programmed in that language at least once. If you have ever been a professional CA-Clipper programmer then you know full well the power of its code block data type. As useful as this data type was for the CA-Clipper programmer, I confess to never using the data type in any of my CA-Visual Objects applications. It's simply not required if you develop in an object-oriented style, since everything the code block offers is in my opinion better provided using methods of objects.

However, a very powerful code block technique used in 1000's of CA-Clipper applications was that of code block macro compilation—the ability to take a string representation of a code block at runtime and using the macro compiler operator '&' turn it into a real code block containing compiled, ready to execute code. Together, the code block and the macro compiler, as implemented in CA-Clipper, redefined what the term "data-driven" meant for tens of thousands of CA-Clipper programmers around the world. Macro compiled code blocks allowed developers to externalize real business logic from the main executable as simple text in a database or control file, read it in when required, compile it into actual executable code *once*, and then execute it as often as required at the same speed as the rest of the compiled application. This meant that complex business rules could be stored separately from the main executable "engine" and, importantly, edited and/or otherwise maintained (often by end users) without requiring the recompilation of the executable engine itself. With this capability sophisticated rule-based applications could easily be created that were very easy to customize, simply by editing a few business rules stored external to the main program. The next time the application was run, it would pick up the latest rule set and its execution behavior would be changed appropriately.

Scripting represents the modern way to do code block macro compilation. The goal is the same: through scripting we seek to store business logic external to the main business application in an easily editable text file (or memo field of a database) form, and have that business logic read in by the main program when necessary, and executed quickly and efficiently. It's true the end results are the same as the macro compiled code blocks of old, which begs the question, why change techniques? The answer is that using modern scripting techniques offers much more power and flexibility than the old tried and true macro compiled code blocks.

For starters, with scripting you are not limited to CA-Clipper expressions (less than 240 characters in length) the way you are with code blocks. A script can be any amount of valid code expressed in an appropriate Active Scripting language. This not only allows the script developer greater freedom of expression, but it also opens the door to the possibility of reusing existing, pre-tested and stable scripts written by 3rd parties. Thus, if your application is targeting an industry that is more likely to have a power Visual Basic end-user than a power CA-Clipper/CA-Visual Objects end-user, then you can still accommodate them with a sophisticated data-driven application, customizable with a scripting language that they are familiar with such as VBScript.

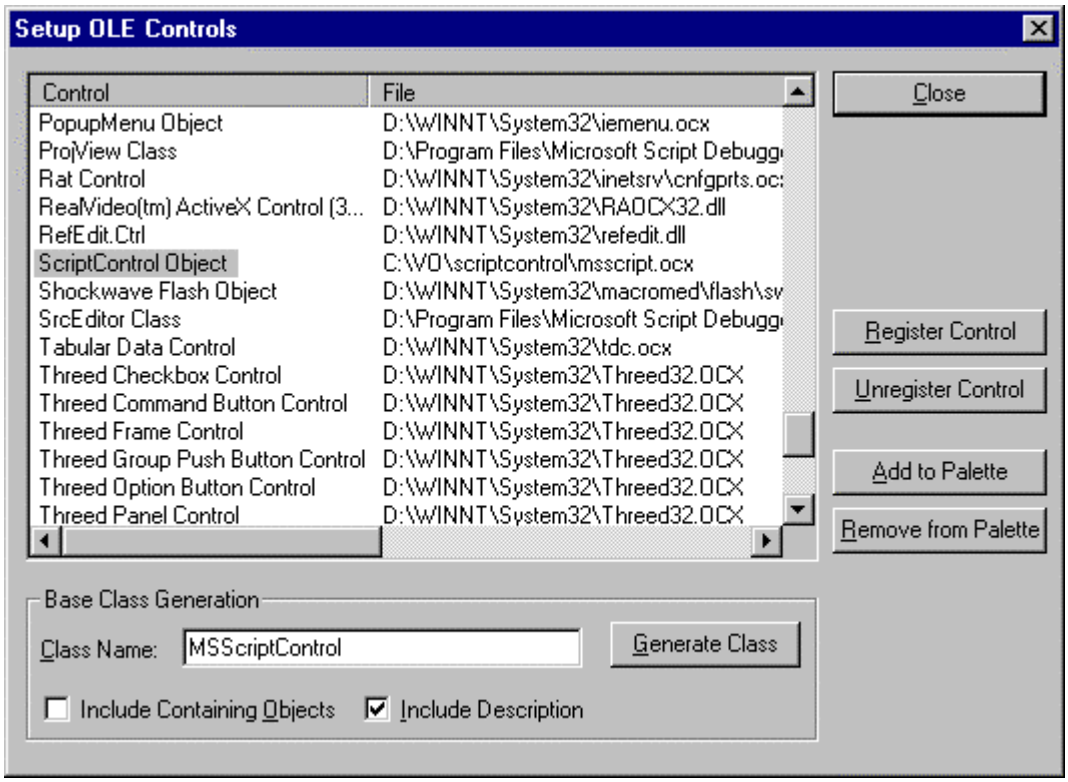
The second big (and arguably most important) advantage that modern scripting techniques have over macro compiled code blocks is that they inherently support the ActiveX (OLE Automation) interoperability standards of Microsoft. This means that not only can a business application load and execute a script to perform some calculation, for example, but the script itself can load an instance of any ActiveX (OLE Automation) server (including, as we shall see shortly, the hosting application itself) and interact with it as if it were a native object of the scripting language. This ability to tightly integrate disparate software components is what makes scripting an order of magnitude more powerful than the old macro compiled code block techniques with respect to data-driven development strategies.

The rest of this paper will show by way of examples just how easy it is to achieve this new level of flexibility and expressive power in your own CA-Visual Objects applications.

A Simple Script Embedded Application

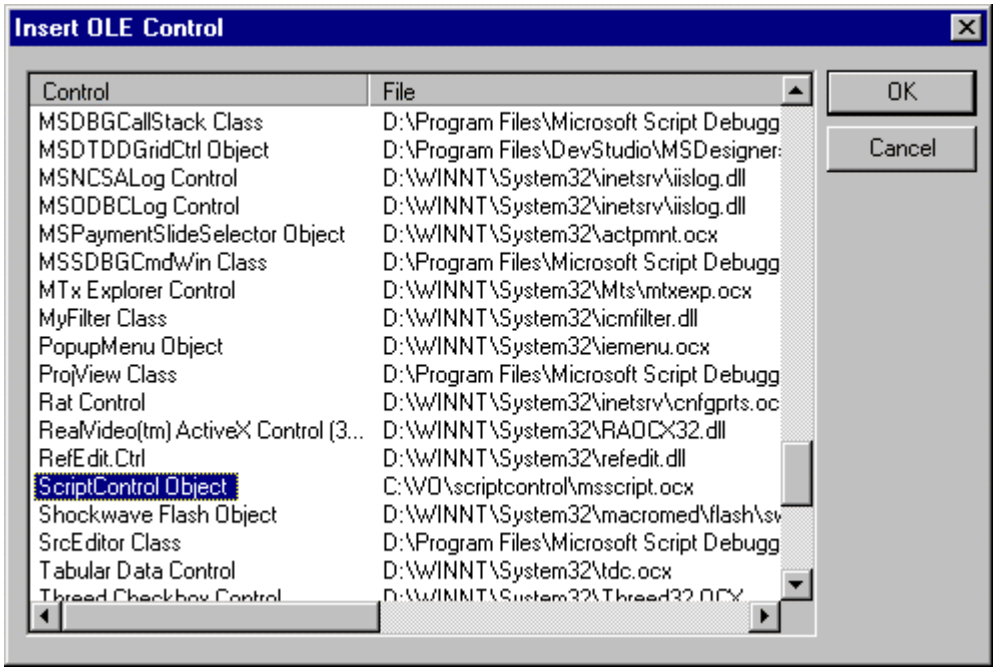
Assuming that you have downloaded and installed the Microsoft Scripting Control from the above referenced web site, you are ready to begin adding scripting to your applications. To get a feel for how to use the control we start with a very simple example. Taking the CA-Visual Objects standard MDI application as a starting point, we will modify it in order to test the OCX scripting control.

The first order of business is to create a class wrapper for the scripting OCX using standard CA-Visual Objects OCX handling procedures. Before proceeding, make sure you have included CA-Visual Objects' OLE library in your application in order to have support for OCXs. Next create a new module named MSScriptControl or something similar and then select it and choose the Tools | Setup OLE Control… menu option to display a list of available OCXs on your system that you can work with. Find and select the one called ScriptControl.Object, enter an appropriate class name for the class wrapper that will be generated for this control, and then press the "Generate Class" button.



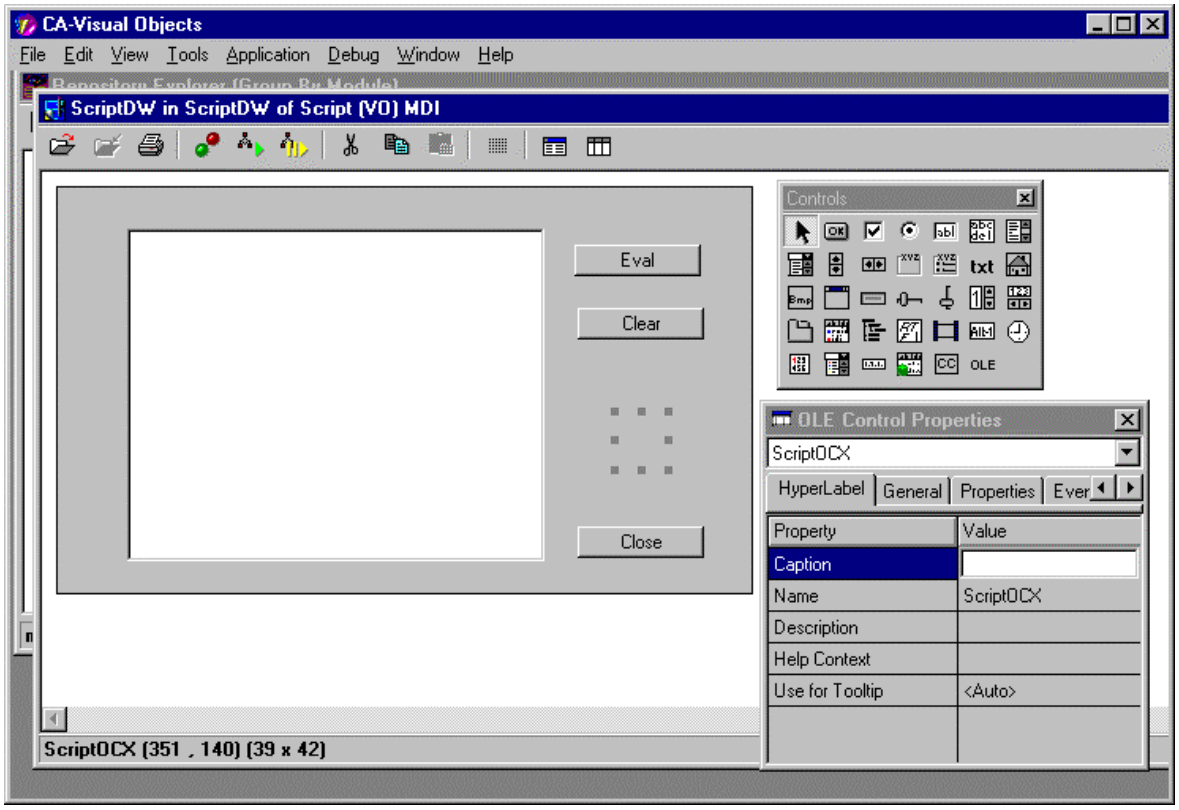
The class should be generated immediately, after which simply press the Close button to dismiss the dialog. Then, immediately compile the application. You should receive no warnings or errors.

Next, create another new module called ScriptDW or something similar which contains a standard CA-Visual Objects DataWindow. From within the Window Editor select the Edit | Insert OLE Control… menu option to display the following dialog.



As before, find and select the ScriptControl.Object OCX and press the OK button to add an instance of this control to your form. The OCX will be represented by a grey resizeable box only, since this ActiveX control has no runtime user interface associated with it. That is, at runtime this OCX will in fact be invisible.

Next simply add, in any layout that is appealing to you, three buttons (Eval, Clear and Close) and a multi-line edit control. These controls will be used to exercise the script control. My screen looks like:



Next select the Script control you placed on the form and choose the "General" tab of the floating properties window and select the "MSScriptControl" entry in the "Inherit from Class" property combo box. This ensures that the code generated by the Window EDitor (WED) to manipulate the OCX inherits from our generated script control class wrapper from above.

Next add the following code in the PostInit() method of the DataWindow:

```
METHOD PostInit(oWindow,iCtlID,oServer,uExtra) class ScriptDW
       //Put your PostInit additions here
       oDCScriptOCX:Language := "VBScript"
       return NIL
```

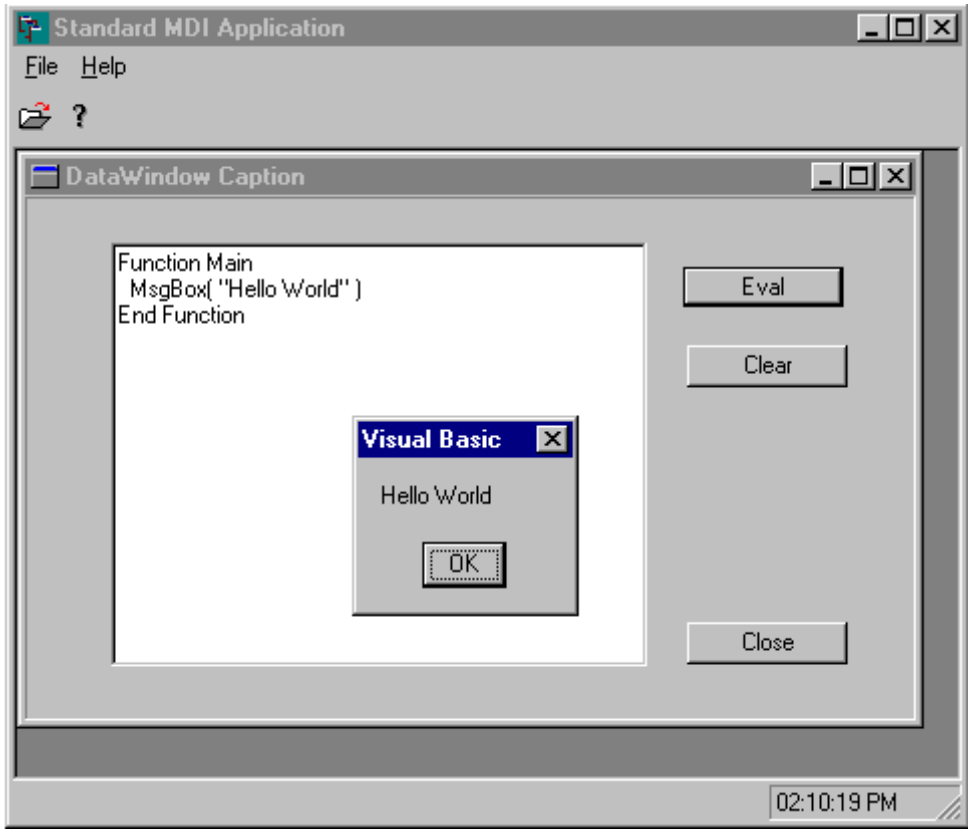
Next add the following code respectively as the “Click Event” for each of the three buttons you placed on your form (Eval, Clear and Close):

```
METHOD EvalPB( ) CLASS ScriptDW
	LOCAL cb                    AS CODEBLOCK
	LOCAL uxError AS USUAL
	LOCAL uValue  AS USUAL

	IF ! Empty( oDCScriptMLE:TextValue )
		cb := ErrorBlock( {|o| _Break(o) } )
		BEGIN SEQUENCE
			oDCScriptOCX:Reset()
			oDCScriptOCX:AddCode( oDCScriptMLE:TextValue )
			uValue := oDCScriptOCX:Run( "Main" )
			TextBox{ , "Result of Script:", AsString( uValue ) }:Show()
		RECOVER USING uxError
			TextBox{,"Error Executing 'Main' Script", ;
					IIF( IsInstanceOf( uxError, #Error ) .AND.;
						!EMPTY( uxError:Description ),;
					uxError:Description, "Unknown Error" ) ;
				}:Show()
		END
		ErrorBLock( cb )
	ELSE
		MessageBeep( MB_OK )
	ENDIF

METHOD ClearPB( ) CLASS ScriptDW
	oDCScriptMLE:TextValue := ""

METHOD ClosePB( ) CLASS ScriptDW
	oDCScriptOCX:Reset()
	self:EndWindow()
```

Finally, hook the ScriptDW DataWindow up to your empty shell menu, and compile the applicaton. When you run it, enter the following VBScript in the multi-line edit control and press the Eval button to test the script control:

```
Function Main
  MsgBox( “Hello World” )
End Function
```

If all is well you should see a “Hello World” dialog box, indicating that the script control has successfully interpreted and executed your script:



The most important code to observe is in the PostInit() and Eval() methods of the ScriptDW DataWindow class as shown above. The important (and only) line in the PostInit() is that which sets the language the script control will use— oDCScriptOCX:Language := "VBScript". Since the script control is capable of dealing with any Active Scripting compatible scripting language, you have a choice here. This line clearly says that VBScript is the language of choice for this example. However, if you like, you can change this to “JScript” for example if you prefer to work with Microsoft’s JavaScript implementation instead. Likewise, any other Active Script compatible scripting language that emerges in the future will also automatically be supported by the control simply by changing this one line.

Clearly, therefore, when saving your scripts to a database memo field, for example, you should also consider saving this language property string so that it can be set appropriately for each script separately at script load time prior to executing it. This would make for a much more flexible solution than the hard coding to VBScript that this example demonstrates.

Once the language property of the script control has been established, all of the real interesting code appears in the Eval() method of the DataWindow. This method attempts to evaluate any script code found in the multi-line edit, sounding a beep if no script is found.

If a script is found, the method sets up a local error handler to catch any errors reported back from the script control. The real meat of the code boils down to the three lines:

```
…
oDCScriptOCX:Reset()
oDCScriptOCX:AddCode( oDCScriptMLE:TextValue )
uValue := oDCScriptOCX:Run( "Main" )
…
```

These lines are fairly self explanatory. They respectively reset the script control to receive a new script, add the script code to execute to the script control, and then run the script code. You can see that this example expects that the script code being added to the control has a subroutine or function named "Main", as this is the routine name it attempts to run. However, you are free to use any function name you would like. In practice a more flexible approach would be to have this routine name saved along with the script itself so that it could be read in as well at runtime, rather than forcing it to be hard coded as above. Also, any return value the script may produce is returned to your program from the call to Run().

One thing you may wish to do is pass one or more arguments to the script routine you are trying to execute. You do this by specifiying *one* optional argument to the script control's Run() method as in:

```
Arg := "Hello World"
uValue := oDCScriptOCX:Run( "Main", Arg )
```

On the script side you would access the argument as a declared passed parameter as in:

```
Function Main( sParam )
       MsgBox( sParam )
End Function
```

If you need more than one argument, simply pass an array for the value of Arg as in:

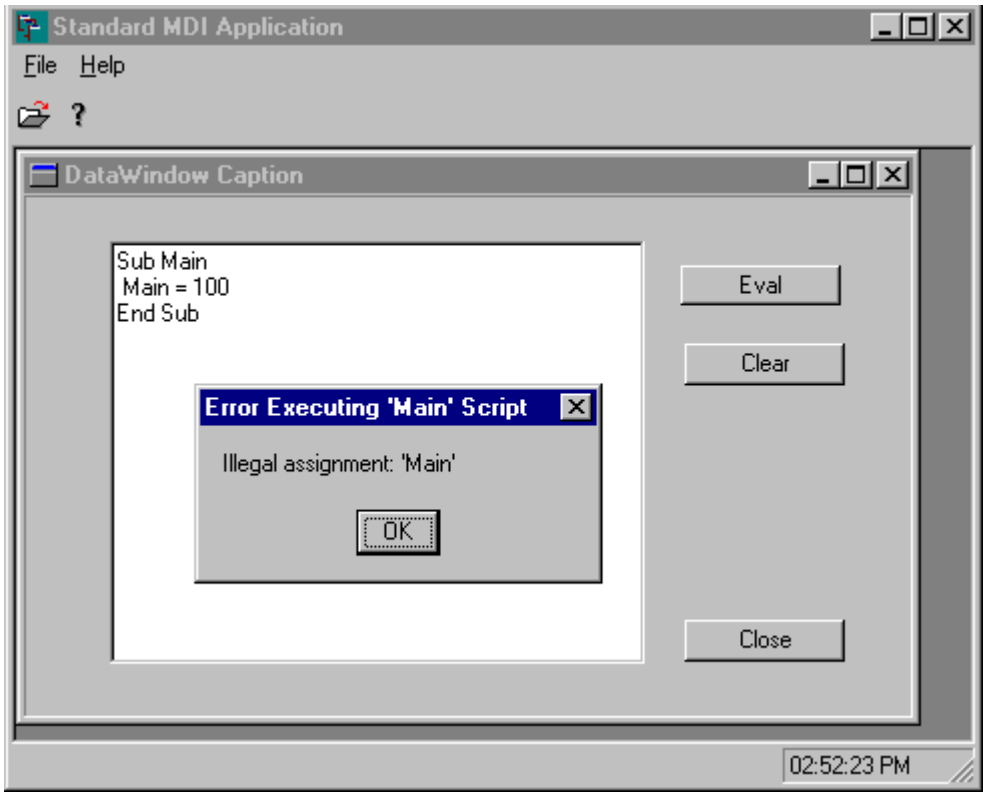
```
Arg := {"Hello", "World"}
uValue := oDCScriptOCX:Run( "Main", Arg )
```

With the script looking like:

```
Function Main( aParams )
       MsgBox( aParams(1) )
       MsgBox( aParams(2) )
End Function
```

The script code you add to the control can be any legal VBScript code. You can add global DIM variable statements, scripts containing multiple functions, and of course scripts that have non-trival code within their declared functions. Moreover, you can call the AddCode() method of the script control multiple times to "build up" large scripting contexts containing many pieces of script control. The Reset() method is used to reset the scripting context. In the above example, I do this for every press of the Eval() button, but this is not necessary.

Finally, any syntax or semantic errors encounterd while the script control is processing the script will be automatically reported back to the hosting CA-Visual Objects application in the form of standard CA-Visual Objects error object. The local error handler in the Eval() method is therefore used to trap and display the error. The following screen shot depicts the error message that results when you try and return a value from a VBScript subroutine (subroutines cannot return values in VBScript; you must use a function for this).



While the script control has other properties and methods you can use to configure its operation, the Language property and the Reset(), AddCode() and Run() methods discussed above are about all you have to know to embed basic scripting mechanisms into your business applications. The script control makes it almost trivial to add powerful scripting capabilities to your programs. I encourage you to play with the scripting control on your own. It comes with complete on-line documentation which discusses all of the features it provides.

Designing Your Application for Embedded Scripting

If all you want to do is be able to load a script written in your favorite scripting language and execute it at runtime from within your business application, then you can save yourself some time and stop reading now. I have already given you all the information you need to do this. However, if you want to exploit the inherent power of a fully integrated scripting design approach to reach new levels of software interoperability and reuse from within your business applications, then read on.

As powerful as the script control may seem to you at the moment, it is actually capable of a lot more. In fact, the examples shown so far are fairly limiting in that they demonstrate a model whereby the script being executed has no context in the outside world beyond that which has been passed in to it via the entry routine's argument list (if any). A more powerful model would allow for the script to access software components outside itself and even to be able to reach up into the application that is hosting the embedded script and access its reusable software components. If such a thing were possible, it would allow scripts to manipulate the data and services of the host application itself as well as those of remote software components. This is in fact possible with the script control's AddObject() method.

The AddObject() method is used to add context to the script control from the world around it. Arguably, the most import context the script control should care about is that of the host application. Ideally the script control should be able to directly use the object model of its hosting application.

Please note that the follow example was build using CA-Visual Objects version 2.0b-x and VOCOM 2.01, a 3rd party product I have written, generally available and in use by 100's of CA-Visual Objects developers since September 1997. At the time of this writing, CA-Visual Objects 2.5 was still in beta and all of my attempts to create a working sample using only the current beta build, without VOCOM, failed. In the interest of showing a complete working sample of the material to follow I have therefore substituted a VOCOM solution. However, by the time CA-Visual Objects version 2.5 has been released, the example should easily convert to an all native CA-Visual Objects 2.5 solution without requiring VOCOM. Regardless, however, the concepts illustrated by the example remain relevant even as the implementation details do not.

Since this sample is built using VOCOM, we have to do things slightly differently. It's best that we start with a fresh application. To begin, create a new standard application and name it whatever you want. As before, create a ScriptDW DataWindow and construct a similar form as we did previously, dragging and dropping the Multi-line edit control and the three buttons (Eval, Clear and Close). However, this time do NOT insert the script OLE control, since we will not be using an OCX for this example. This means that you do NOT need to include CA-Visual Objects' OLE library and you do NOT need to generate a class wrapper for the script control as we did previously.

With your form looking as it did previously (with the exception of the invisible script control OCX), we are now ready to add our custom code. The first thing we must do is add a custom instance variable to our ScriptDW class. Still in the Window EDitor and with the ScriptDW selected, choose
"Class Declaration" on the DataWindow property tab and add a line to declare a protected instance variable named oScriptControl as an object and save.

```
class ScriptDW inherit DATAWINDOW

      protect oDCScriptMLE as MULTILINEEDIT
      protect oCCEvalPB as PUSHBUTTON
      protect oCCClearPB as PUSHBUTTON
      protect oCCClosePB as PUSHBUTTON
      instance ScriptMLE

  //{{%UC%}} USER CODE STARTS HERE (do NOT remove this line)
  PROTECT oScriptControl   AS OBJECT
```

Next, further down on the same DataWindow property table select “PostInit Actions” and enter the following code and save:

```
METHOD PostInit(oWindow,iCtlID,oServer,uExtra) class ScriptDW
       //Put your PostInit additions here

       LOCAL oSC                  AS OBJECT
       LOCAL hResult       AS LONG

       // Create an instance of the Microsoft Script Control
       // NOTE:  We use a local oSC variable here since you should never
       //         pass an instance variable of a class by reference
       hResult := VOCOMCreateObject( "MSScriptControl.ScriptControl",;
                          @oSC )

       IF SUCCEEDED( hResult )
              oSC:Language := "VBScript"
              self:oScriptControl := oSC    // Assign object to ScriptDW ivar
       ELSE
              TextBox{, "Error Creating Script Control"}:Show()
              self:EndWindow()
       ENDIF
       return NIL
```

This code attempts to create an instance of the script control using VOCOM Although the script control is an OCX, since it has no user interface we can, for all intents and purposes, treat it like a standard ActiveX (OLE Automation) server and instantiate it directly from its PROG_ID (Program IDentifier), which is “MSScriptControl.ScriptControl”. Note that if a problem should arise attempting to instantiate the script control (e.g., it is not found on the machine), the ScriptDW window never shows but instead displays an appropriate error message dialog box.

Finally, we add the support code for the three buttons—Eval, Clear and Close—similar to before. *(NOTE: the only change to the previous version of these methods is that references to the* ***oDCScriptOCX*** *instance variable have been changed to* ***oScriptControl****.*)

```
METHOD EvalPB( ) CLASS ScriptDW
       LOCAL cb                   AS CODEBLOCK
       LOCAL uxError AS USUAL
       LOCAL uValue  AS USUAL

       IF ! Empty( oDCScriptMLE:TextValue )
              cb := ErrorBlock( {|o| _Break(o) } )
              BEGIN SEQUENCE
                     oScriptControl:Reset()
                     oScriptControl:AddCode( oDCScriptMLE:TextValue )
                     uValue := oScriptControl:Run( "Main" )
                     TextBox{ , "Result of Script:", AsString( uValue ) }:Show()
              RECOVER USING uxError
                     TextBox{,"Error Executing 'Main' Script", ;
                                   IIF( IsInstanceOf( uxError, #Error ) ;
                                          .AND. !EMPTY( uxError:Description ),;
                                   uxError:Description, "Unknown Error" );
                            }:Show()
              END
              ErrorBLock( cb )
       ELSE
              MessageBeep( MB_OK )
       ENDIF

METHOD ClearPB( ) CLASS ScriptDW
       oDCScriptMLE:TextValue := ""

METHOD ClosePB( ) CLASS ScriptDW
       oScriptControl:Reset()
       self:EndWindow()
```

At this point, you have an equivalent program to the one we had previously. As before, hook up the ScriptDW DataWindow to your empty shell menu and compile and run the application. You will be able to test your scripts in identical fashion as before with this VOCOM version of the application.

Now let's modify the application a little. Change the ScriptDW:Eval() method to look like:

```
METHOD EvalPB( ) CLASS ScriptDW
       LOCAL cb            AS CODEBLOCK
       LOCAL uxError AS USUAL
       LOCAL uValue        AS USUAL

       LOCAL hResult       AS LONG
       LOCAL oApp          AS OBJECT


       IF ! Empty( oDCScriptMLE:TextValue )
              cb := ErrorBlock( {|o| _Break(o) } )
              BEGIN SEQUENCE
                     oScriptControl:Reset()
                     oScriptControl:AddCode( oDCScriptMLE:TextValue )

                     hResult := VOCOMCreateObject( ;
                                   "STDActiveXServerApp.Application", @oApp )
                     IF SUCCEEDED( hResult )
                            oScriptControl:AddObject( "MiddleTier",;
                             oApp, TRUE )
                     ELSE
                            BREAK VOCOMOLEBaseError{ hResult }
                     ENDIF

                     uValue := oScriptControl:Run( "Main" )

                     oApp:Release()

                     TextBox{ , "Result of Script:", AsString( uValue ) }:Show()
              RECOVER USING uxError
                     TextBox{,"Error Executing 'Main' Script", ;
                                   IIF( IsInstanceOf( uxError, #Error ) .AND. ;
                                                 !EMPTY( uxError:Description ),;
                                   uxError:Description, "Unknown Error" ) ;
                            }:Show()
              END
              ErrorBLock( cb )
       ELSE
              MessageBeep( MB_OK )
       ENDIF
```

The additions made to the method have been highlighted. This code uses VOCOM to create an instance of a second COM component—STDActiveXServerApp.Application—which is a sample VOCOM ActiveX server which comes with the product. If the object is successfully created, then we pass it to the script control in the line:

```
oScriptControl:AddObject( "MiddleTier", oApp, TRUE )
```

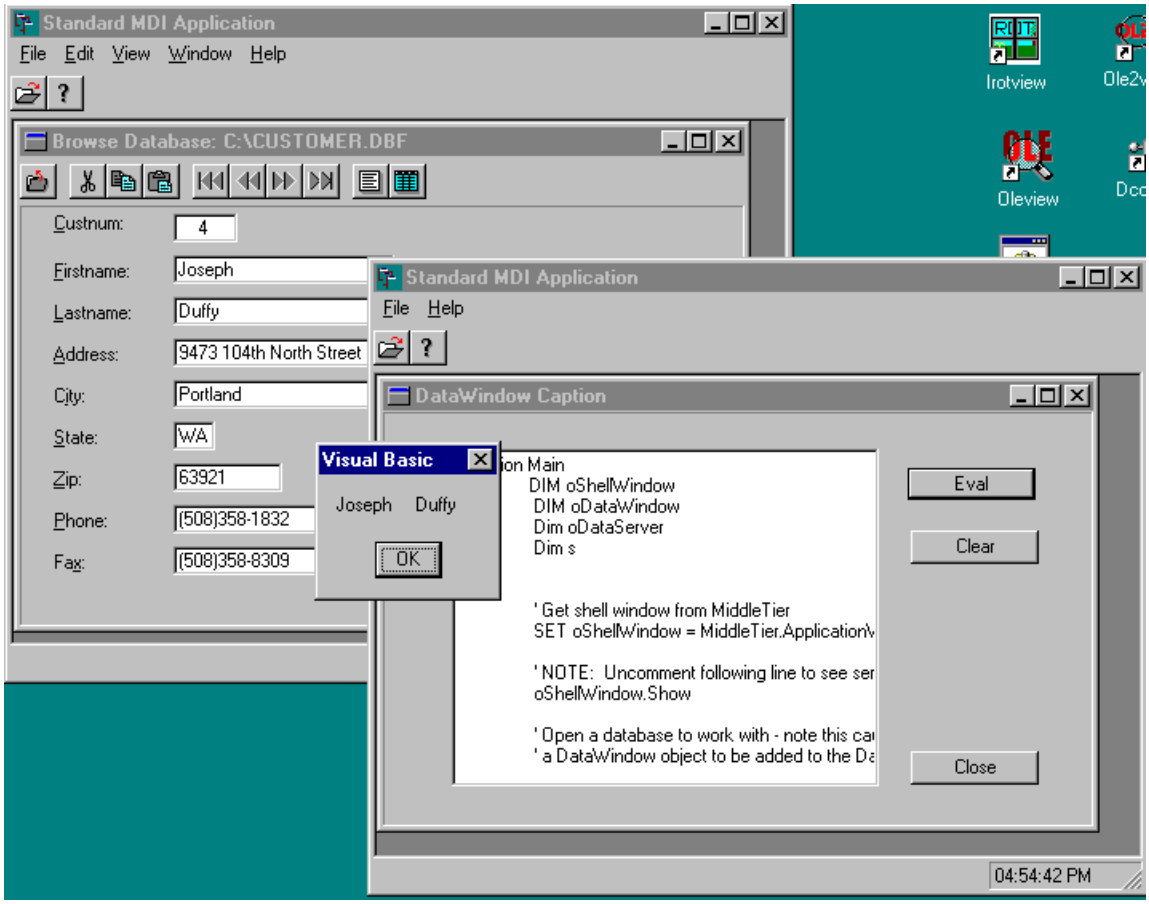
This line tells the script control to add an object named "MiddleTier" to the global context of the current script. As long as oApp is a valid ActiveX server, this line will succeed, and from that point forward you will be able to refer directly to the symbol "MiddleTier" from within all code in the current script context (i.e., up to the next call to Reset()).

With these changes made, rebuild the application and run it, cutting and pasting the following script:

```
Function Main
	DIM oShellWindow
	DIM oDataWindow
	Dim oDataServer
	Dim s


	' Get shell window from MiddleTier
	SET oShellWindow = MiddleTier.ApplicationWindow

	' NOTE:  Uncomment following line to see server
	'oShellWindow.Show

	' Open a database to work with - note this causes
	' a DataWindow object to be added to the DataWindows
	' collection
	s = "C:\CUSTOMER.DBF"
	oShellWindow.FileOpen( s  )

	' Retrieve newly created DataWindow indexed by database name
	Set oDataWindow = oShellWindow.DataWindows( s )

	' NOTE:  Uncomment following line to see server
	'oDataWindow.Show


	' Retrieve the DataServer reference from the DataWindow
	Set oDataServer = oDataWindow.Server


	' Process the file in some way...
	DO WHILE NOT oDataServer.EOF
		' Lets see the data
		MsgBox( oDataServer.FirstName & " " & oDataServer.LastName )
		oDataServer.Skip
	LOOP

	' Don't forget to clean up
	oDataWindow.FileClose
	oShellWindow.EndWindow
End Function
```

This code will use the object passed in (which is just an ActiveX server wrapper around CA-Visual Objects' own standard application—a sample that comes with VOCOM) to open and traverse a DBF file displaying the names of all of the customers. It is clearly not intended to be efficient, but it does demonstrate how the entire component, including its nested objects, has become available to the script through the "MiddleTier" global variable referenced near the top of the script. Notice how we are able to reach into the MiddleTier object and pull out the ShellWindow, DataWindow and DataServer objects respectively as we drill deeper into the object model of the VOCOM component. These objects are simply proxies onto their actual counterparts in the standard application of CA-Visual Objects. Whatever you can do normally to these CA-Visual Objects objects, you can now do through script, ensuring complete access to the business logic of this (or any other) component!

If you want to have some fun, try the sample again but this time uncomment the **`oShellWindow.Show`** and **`oDataWindow.Show`** lines.



I named the object “MiddleTier” to underscore the role the object plays in the script. Admittedly a standard application component is not a good example of a middle tier component. However it is not hard to think of real, user interface-less, business logic components that could be substituted to provide valuable services to your script.

Moreover, you are not restricted to adding only one object. You can call the script control’s AddObject() method as many times as you need to to populate the current script context with components you need to work with. In this manner you begin to exploit the script as being the “glue” that ties together a collection of disparate components!

As a final example to demonstrate this, add the following line immediately after the AddObject() call in the ScriptDW:Eval() method above:

```
oScriptControl:AddObject( "UI", self:Owner, TRUE )
```

This line adds the ShellWindow (the owner of the ScriptDW window) to the script control context, assigning it a name of "UI". This effectively gives the script access to the ShellWindow of the hosting application (including all nested objects reachable from it) along with the already accessible MiddleTier component. VOCOM takes care of the ugly details of automatically exposing an otherwise ordinary CA-Visual Objects object into a properly formed ActiveX server object which the script control can deal with.

With this change in place, rebuild and rerun the application and reexecute the script sample from above to make sure nothing broke. Now replace the MsgBox() call in the loop of the script with:

```
UI.Caption = oDataServer.FirstName & " " & oDataServer.LastName
```

and run it again. Watch the title bar of the application closely, as the script will rapidly change the caption as it loops through the records. The script functions as "glue" code in that it is actually taking data from the "MiddleTier" component and effecting a change on the "UI" component!

Conclusion

This paper has endeavored to show you how scripting can be used to open your applications up to new possibilities. Through the use of runtime scripts, a division between user interface components and business logic components can be better made. This will in turn result in software that is more componentized and therefore more reusable, scalable and easier to deploy—the promise of n-tier systems. If you are already designing your software components as ActiveX servers, then they will be immediately accessible from within your data-driven scripts. If you are not, then perhaps it's time you began exploring the world of ActiveX.

As ActiveX components proliferate, the importance of scripting as component glue will increase. CA-Visual Objects is a natural language to write ActiveX servers in, and the Microsoft Script Control makes it easy to incorporate flexible, script-based, data-driven design concepts that are open to the world of ActiveX. Download the script control and start playing today!

Rod da Silva is the principal of Software Perspectives, Toronto, Canada. He has been a featured speaker at Developer's Conferences all over the world for several years. Rod has a Bachelor of Mathematics degree with Honors in Computer Science from the University of Waterloo, and his areas of interest include compilers, client/server DBMS system software and Windows OLE technologies. He is author of the VOCOM library which among other things provides CA-Visual Objects 2.x developers the ability to create ActiveX Servesr, as well as CULE (Component Unification Language Everywhere), the world's first "component-oriented" language (http://www.SoftwarePerspectives.com/CULE). He is available for general purpose Windows consulting and/or training and can be reached by e-mail at RodDaSilva@SoftwarePerspectives.com.