
学校的理想装备
电子图书·学校专集
校园网上的最佳资源
ActiveX Scripting in MFC
BOOK
内部资料 非卖品

ActiveX Scripting in MFC

By Steve Wampler

Like Microsoft Internet Explorer, your applications can support VBScript (Visual Basic Scripting Edition) and JScript (JavaScript) with very little effort on your part. Microsoft has done most of the work by providing VBScript and JScript engines and by defining the ActiveX scripting COM interfaces that enable you to use the engines within your applications.

We'll take Microsoft's work one step further with an MFC-like C++ class called CScriptEngine that implements the ActiveX scripting interfaces in the context of a sample MFC application.

You should be familiar with MFC and automation. If not, read up on both in the Visual C++ online documentation.

Note that for ActiveX scripting to work in your MFC applications, your Visual C++ environment must be version 4.2b. The 4.2b upgrade patch is available from Microsoft at http://www.microsoft.com/visualc under "Downloads and Patches." You must also have the VBScript and JScript DLLs that come with Internet Explorer. You can download the DLLs from Microsoft at http://www.microsoft.com/msdownload/scripting.htm.

Why ActiveX scripting?

"Automation enables automation clients such as Visual Basic to use your applications. But why would you want your application to support ActiveX scripting as well? The answer depends on your application."

Automation enables automation clients such as Visual Basic to use your applications. But why would you want your application to support ActiveX scripting as well? The answer depends on your application.

For example, if your application is an ActiveX control container your users will likely want to write scripts that access the controls' automation events, properties, and methods. Your users may also want their scripts stored within the application's documents. And finally, they likely want a choice in scripting languages. By adding support for ActiveX scripting, your applications can do all of these things. Let's

examine a sample application called ScriptTest to see how.

The ScriptTest application from the user's perspective

ScriptTest is a Hello, World!–type application. It's a simple Single Document Interface (SDI) application generated using the MFC AppWizard. Here's what ScriptTest looks like in action:

To edit and run a ScriptTest script, you invoke the Enter Script dialog by selecting Run from the Script menu. Here's what the Enter Script dialog looks like:

In the Script Type box, you choose VBScript or JScript. You enter the script's text in the Script edit control.

To give it a try, download the ScriptTest application from the VCDJ web-site and run it. From the menu bar, select Script and Run, and enter the following VBScript:

```
Sub SayGoodbye
  Document.Text = "Goodbye, World!"
End Sub
SayGoodbye
```

Press OK to run the script.

This VBScript declares a SayGoodbye subroutine that simply sets the document's Text property (described shortly) to "Goodbye, World!" The script then calls the SayGoodbye subroutine. After running the script, ScriptTest displays "Goodbye, World!"

ScriptTest from a programmer's perspective

We enabled ActiveX scripting in ScriptTest in two steps: first, we add an automation method, and a property in ScriptTest's document class, CScriptTestDoc, gave a script something to do; second, two C++ classes enabled ScriptTest to create and use an ActiveX scripting engine.

Step 1: automating the ScriptTest application. Before adding support for ActiveX scripting, we use the Visual C++ ClassWizard to add an automation property called Text and a method called MsgBox() to ScriptTest's CScriptTestDoc class. The Text property is held in a CString variable within ScriptTest's document class. Its value is what you see written in ScriptTest's window.

In VBScript, you can get and set the document's Text property like so:

```
' get the Text property
strText = Document.Text
' set the Text property
Document.Text = "Goodbye, World!"
```

The MsgBox() method takes one string parameter containing the message text. For example, the VBScript statement Document.MsgBox("Goodbye, World!") causes the ScriptTest to display the following message:

Step 2: creating and using a scripting engine. To create and use an ActiveX scripting engine, ScriptTest includes the classes CScriptEngine and CScriptTestScriptEngine.

-

"The CScriptEngine class is an MFC-like wrapper around an ActiveX scripting engine. It's general enough that you should be able to reuse it in other MFC applications."

The CScriptEngine class is an MFC-like wrapper around an ActiveX scripting engine. It's general enough that you should be able to reuse it in other MFC applications.

The CScriptTestScriptEngine class is derived from CScriptEngine. It overrides one virtual function called OnGetItemInfo() that the scripting engine calls when it needs information about ScriptTest's automation

The CScriptEngine class. CScriptEngine plays two roles: From your application's perspective, it provides a set of methods for creating, initializing, and running a scripting engine. From the engine's perspective, it defines (but does not implement) the virtual OnGetItemInfo() method called by the script engine to obtain information about your application's automation objects. We'll look at the OnGetItemInfo() method shortly when we cover the CScriptTestScriptEngine class.

The best way to describe CScriptEngine is in the context of ScriptTest's CScriptDialog::OnOK() method. CScriptDialog is the class that implements ScriptTest's Enter Script dialog, and CScriptDialog::OnOK() is the method MFC calls when you press the dialog's OK button. CScriptDialog::OnOK() is responsible for creating a CScriptEngine object and using that object to run a script.

The CScriptDialog::OnOK() method appears below with comments and with all of the error-handling code removed.

```
void CScriptDialog::OnOK()
{
```

The first few lines of OnOK() simply call a few helper functions defined in the CScriptDialog class. The helpers obtain the script's text and type from the dialog's controls.

```
// get the script text
  CString strScript = GetScript();
  // get the script engine's class identifier (CLSID)
  const CLSID* pCLSID = GetScriptEngineCLSID();
  ASSERT(pCLSID != NULL);
```

Next, OnOK() creates a new CScriptTestScriptEngine that for our purposes is treated as a CScriptEngine.

```
// create the script engine proxy on heap (already AddRef'd upon return)
  CScriptEngine* pScriptEngine = new CScriptTestScriptEngine(m_pDoc);
  ASSERT(pScriptEngine != NULL);
```

Like many MFC objects such as a CWnd, a CScriptEngine is really a proxy for another object, in this case an ActiveX scripting engine. With a CScriptEngine object in place, it's time to create the real engine using the CScriptEngine::Create() method. Create() is implemented in the CScriptEngine class and takes the class identifier (CLSID) of the type of engine you'd like to create.

```
// create the actual scripting engine
  pScriptEngine->Create(*pCLSID);
```

Now we're ready to feed the engine the script using the CScriptEngine::ParseScriptText() method. If you look in the ScriptEngine.h file, you'll see ParseScriptText() takes a lot of parameters. All the parameters but the first—the script's text—have default values, so the first parameter is the only one provided.

```
// give the engine the script
  USES_CONVERSION; // needed by the T2COLE macro
  pScriptEngine->ParseScriptText(T2COLE(strScript));
```

The only other thing to note here is the use of MFC's USES_CONVERSION and T2COLE macros. T2COLE converts a single-byte string into an OLE (double-byte) string. USES_CONVERSION defines a local variable used by T2COLE. For more information on using MFC conversion macros, see technical note #59 in the Visual C++ online documentation.

Next, the OnOK() method calls the CScriptEngine::AddNamedItem() method to tell the engine the name of ScriptTest's "Document" object.

```
// add the document as a named item
  pScriptEngine->AddNamedItem(L"Document");
```

You can call AddNamedItem() for other items as well, and you can name them anything you want. Typically, though, you only explicitly name your application's "top level" objects. Objects embedded within your application's top-level objects, such as ActiveX controls, are usually accessed through the top-level objects' automation properties and methods. For each item you add using AddNamedItem(), you should expect one or more calls to the virtual CScriptEngine::OnGetItemInfo() method.

Now we're ready to run the script by calling the Run() method.

```
// run the script
  pScriptEngine->Run();
```

The Run() method immediately executes the script and returns. If the engine encounters an error in the script, it calls the CScriptEngine::OnScriptError() method which, by default, displays a message box:

Note that if your script has any event-handling methods, they aren't called until an event occurs. Say your application has an object named MyButton that generates a Click event each time the user clicks a particular button. Using Visual Basic's event-handler naming convention, the VBScript code to handle MyButton's Click event would look something like this:

```
Sub MyButton_Click
  ...
End Sub
```

As long as the VBScript engine is running, it calls the MyButton_Click event whenever the user clicks on the button.

Because ScriptTest's automation objects don't generate any events, it's safe to terminate the script engine before exiting the CScriptDialog::OnOK() method. You may want to let the script engine continue to run, though. We'll look at how to do that at the end of this article.

You terminate the script engine using the CScriptEngine::Close() and CCmdTarget::ExternalRelease() methods.

```
// close the script engine
  pScriptEngine->Close();
  // release the script engine (it'll delete itself)
  pScriptEngine->ExternalRelease();
```

Finally, we let the CDialog base class do its thing.

```
  CDialog::OnOK();
}
```

That's all it takes to run a simple script.

The CScriptTestScriptEngine class.The CScriptDialog::OnOK() method demonstrates how to create, initialize, and use an ActiveX scripting engine through the CScriptEngine class. Part of that initialization involves naming the application's automation objects so you can refer to them by name within a script. But a scripting engine needs more than just the object's name.

That brings us back to the OnGetItemInfo() method, which you'll recall is the one method you must override in your CScriptEngine-derived class. Its purpose is to provide information to a scripting engine about your application's automation objects, specifically those, and only those, named using the CScriptEngine::AddNamedItem() method.

Through OnGetItemInfo(), the script engine asks for a pointer to the item's IUnknown interface, a pointer to an ITypeInfo interface describing the object, or pointers to both. IUnknown and ITypeInfo are COM interfaces. If you're not familiar with them, don't worry. In the code described below, you'll see how to obtain them.

The implementation of CScriptTestScriptEngine::OnGetItemInfo() appears below with comments and without most of the error-handling code.

```
HRESULT CScriptTestScriptEngine::OnGetItemInfo(
  /* [in] */ LPCOLESTR pstrName,
  /* [in] */ DWORD dwReturnMask,
  /* [out]*/ IUnknown** ppUnknownItem,
  /* [out]*/ ITypeInfo** ppTypeInfo)
{
```

Let's begin with OnGetItemInfo()'s input parameters and output value.

When calling the OnGetItemInfo() method, the script engine asks for information about a particular object by name, the same name specified in one of the calls to the CScriptEngine::AddNamedItem() method. The engine specifies the object's name

through the pstrName parameter, which is always an OLE string.

The script engine uses the next parameter, dwReturnMask, to indicate whether it wants the object's IUnknown pointer or a pointer to an ITypeInfo interface describing the object. The last two parameters, ppUnknownItem and ppTypeInfo, are pointers to where you'll store the resulting IUnknown and ITypeInfo pointers.

Finally, the return value is an HRESULT initialized here to S_OK to indicate all is well.

```
HRESULT hr = S_OK;
```

Next, the wide-character-string-case-insensitive-comparison (yikes!) function wcsicmp()determines if the scripting engine is asking for information about ScriptTest's "Document" object. It does the comparison against the constant L"Document" where the L forces the string to be an OLE character string.

```
// if the script engine wants the Document object ...
  if (wcsicmp(pstrName, L"Document") == 0) {
```

Your implementation of OnGetItemInfo() should contain one if statement for each top-level object named using CScriptEngine::AddNamedItem(). Of course, for lots of top-level objects you might want to use another method, such as a lookup table, instead of a bunch of if-then-else blocks.

With the object in question identified, our next task is to determine what information the scripting engine wants by examining the dwReturnMask parameter. We'll check dwReturnMask against SCRIPTINFO_IUNKNOWN, a constant defined by the ActiveX Scripting Specification. If the result is TRUE, the scripting engine is asking for a pointer to the object's IUnknown interface.

```
    // if the script engine wants the object's IUnknown pointer ...
    if (dwReturnMask & SCRIPTINFO_IUNKNOWN) {
```

For objects whose class is derived from MFC's CCmdTarget class, we can simply call CCmdTarget::GetIDispatch() to obtain a pointer to the object's IDispatch interface and then cast that pointer to a pointer to an IUnknown. That's safe because IDispatch, like all good COM interfaces, is derived from IUnknown:

```
// return the document's IDispatch interface as its IUnknown
      *ppUnknownItem = (IUnknown*)m_pDoc->GetIDispatch(TRUE);
```

Make sure you pass TRUE to GetIDispatch(). A value of TRUE tells GetIDispatch() to increment the object's reference count. The script engine will decrement the count once it's done using the pointer.

Next we'll check the dwReturnMask parameter against SCRIPTINFO_ITYPEINFO. If the result is TRUE, the engine asks for a pointer to an ITypeInfo interface describing the object's COM object class (coclass):

```
    // if the script engine wants the object's ITypeInfo pointer ...
    if (dwReturnMask & SCRIPTINFO_ITYPEINFO) {
```

If your object generates automation events, and you want your scripts to handle those events, you must return a pointer to an ITypeInfo; otherwise the script engine won't know how to connect to your object's events. See the sidebar "Obtaining Your Object's ITypeInfo Interface" on the next page, for more details on how to get an object's ITypeInfo.

If your object doesn't generate events, or you don't care to handle them from a script, you can safely return NULL in the *ppTypeInfo parameter.

```
      // CScriptTestDoc doesn't support events,
      // so just return a NULL pointer
      *ppTypeInfo = NULL;
    }
  } else {
```

You'll remember that near the top of OnGetItemInfo() we had an if statement that compared the input name pstrName against L"Document". Well, here we are in the else clause, and in it we set the return value hr to E_UNEXPECTED to indicate the script engine has asked for an unexpected item.

```
    // the script engine asked for an unknown item
    hr = E_UNEXPECTED;
  }
  return hr;
}
```

That's it. The CScriptEngine class creates, initializes, and runs an ActiveX scripting engine and gives that engine additional information about your automation objects by overriding the CScriptEngine::OnGetItemInfo() method.

Obtaining Your Object's ITypeInfo Interface

Chances are, if you do anything more sophisticated than ScriptTest, you'll want to supply the scripting engine with an ITypeInfo interface for at least those objects that generate events. Here's one method that's worked well for me:

```
// get the object's IUnknown pointer
CCmdTarget* pMyObject = ...;
IUnknown* pUnknown = (IUnknown*)pMyObject->GetIDispatch(FALSE);

// QueryInterface() to get IprovideClassInfo
IProvideClassInfo* pProvideClassInfo = NULL;
Hr              =              pUnknown->QueryInterface(IID_IprovideClassInfo,
(void**)&pProvideClassInfo);

// if the object supports IprovideClassInfo ...
if (SUCCEEDED(hr) && pProvideClassInfo != NULL) {
  hr = pProvideClassInfo->GetClassInfo(ppTypeInfo);
} else {
  // no luck, just return a NULL
  *ppTypeInfo = NULL;
}
```

One other way to get an object's ITypeInfo is to load the object's type library (typelib) by calling ::LoadTypeLib() or ::LoadRegTypeLib(), then calling ItypeLib::GetTypeInfoOfGuid(). GetTypeInfoOfGuid() requires the class identifier (CLSID) or (GUID) of the requested type information. You should specify the (CLSID) of your object's coclass to ensure that you get the correct ITypeInfo.

Finally, a word of warning: You might be tempted, as I was, to call your object's IDispatch::GetTypeInfo() method to obtain the object's ITypeInfo interface. You'll get an ITypeInfo for sure, but it'll be the wrong one. Specifically, you'll get the ITypeInfo for your object's automation methods and properties, not the one describing your object's COM object class (coclass).

What you might want to do next

Add scripting to your document class. If you really want to integrate ActiveX scripting into your application, you'll want to integrate it directly into your application's document object. That entails:

storing the script and its type in CString variables within your document object
serializing (saving and restoring) the script and its type from within your document's Serialize() method
running the script from your document's OnOpenDocument() method
displaying the script through a form view, assuming your application has a Multiple

Document Interface
Unlike the CScriptDialog::OnOK() method, which closes and releases the script engine immediately after the script runs, your document's should allow the script engine to continue to run so it can process events generated by your application.

"The richer your application's support for automation, the more useful scripting will be to your users."

Fill out your application's automation object model. The richer your application's support for automation, the more useful scripting will be to your users. One of the most accessible examples of automation object models (the list of objects, methods, properties, and events supported by an application) is the object model for Microsoft Internet Explorer, which you can access at http://www.microsoft.com/intdev/sdk/docs/scriptom/.

Extend your applications

I hope you find ActiveX scripting as useful in your MFC applications as I have in mine, and with the CScriptEngine class I hope you'll find ActiveX Scripting easy to implement as well. Most importantly, your users will benefit by being able to write scripts that extend the usefulness of your applications

References

ActiveX scripting

If you'd like to understand the details behind ActiveX Scripting, check out the specification at http://www.microsoft.com/intdev/sdk/docs/olescrpt/axscript.htm. Don Box has written an excellent article in the February 1997 edition of Microsoft Interactive Developer—"Say Goodbye to Macro Envy with Active Scripting"—that describes ActiveX Scripting from a COM perspective.
VBScript and JScript

Microsoft's VBScript Web site, located at http://www.microsoft.com/vbscript, is full of information about VBScript, including information on hosting VBScript in your applications and how VBScript relates to Visual Basic and Visual Basic for Applications.

The Microsoft JScript site, located at http://www.microsoft.com/jscript, has

information similar to the VBScript site's, including language documentation.
Automation

Microsoft has published a number of good Knowledge Base and MSDN articles and samples relate to Visual C++ and Automation. Below are just a few found while searching for "Automation" in the Visual C++ 4.0 and MSDN documentation:

"SAMPLE: Simple OLE Automation Object Sample" (Article ID: Q107081) explains the basics of creating OLE automation objects.
"SAMPLE: MFCDISP: Replacing MFC IDispatch implementation" (Article ID: Q140616) explains how to replace MFC's IDispatch implementation with one that uses a typelib.
"VB Automation of Visual C++ Server Using OBJ1.OBJ2.prop Syntax" (Article ID: Q137343) explains how to expose nested automation objects within your Visual C++ applications.
"SAMPLE: MFCARRAY: Using Safe Arrays in MFC Automation" (Article ID: Q140202) demonstrates the use of safe arrays to pass information between an automation client and server.
"How Visual Basic Automation Statements Map to OLE Calls" (Article ID: Q122288) explains how Visual Basic's CreateObject() and GetObject() functions are implemented in terms of OLE and COM functions. Note that VBScript and JScript do not support CreateObject() or GetObject() functions. Instead, you can add similar functions through your application's automation objects.