10/24/2019 • 24 minutes to read

COM: Handle Late-bound Events within Visual Basic Using an ATL Bridge

Carlo Randone

This article assumes you�re familiar with Visual Basic and ATL

Level of Difficulty 1 2 3

Download the code for this article: Connpoints.exe (99KB)

Browse the code for this article at **Code Center**: COMBRIDGE_EVENT

SUMMARYSince a Visual Basic client doesn't handle events directly from late-bound COM objects, it needs some way to capture all the events and parameters launched by any COM object server instantiated at runtime and not known at design time. This article explains how to build a bridge component that does just that. The bridge component transmits the intercepted event data back to the Visual Basic client using another supporting COM object that is capable of holding event data and attributes. The Visual Basic client receives the notification from the bridge and extracts all the information relative to the event from the supporting object.

Visual Basic® 6.0 is a good tool for creating, assembling, and testing COM components. However, it doesn't directly handle events from late-bound COM objects. Although you can't access this event data automatically from the language, you can write a component that receives the data and relays it to your program. In this article, I'll describe a bridge component I've written that intercepts event data from a late-bound COM object and transfers the information back to the Visual Basic client. But first I'll present a little background information on Visual Basic and the handling of COM events.

Visual Basic 6.0 can automatically handle asynchronous notifications from COM objects through events if the COM server uses the connection point method for event notification. Visual Basic even makes it easy to develop applications that can handle these notifications. For example, to intercept the events launched by a COM component named comsrvcls, you can write in Visual Basic:

Copy

```
Dim WithEvents serverobj As comsrvcls
```

Visual Basic reads the component type library and automatically creates an event-handling procedure such as:

Copy

Copy

```
Sub serverobj_eventname(eventarguments)



End Sub
```

For early-bound COM objects, Visual Basic can use the server component type library to determine the events that the server can launch and their arguments.

A similar mechanism is available if the server is an ActiveX® control as well as a simple COM component. All you need to do is add the ActiveX control to the project, place it on a form, and select the events you want to intercept in the Procedure box. In the case of ActiveX controls, the server component (the control) always runs in-process inside the client container, so the overhead is relatively low.

As you can see, in these two cases (COM servers using connection points and early-bound ActiveX controls) handling events is easy and the overhead is minimal. But what about handling events from late-bound COM objects? Let's look at two possible event-handling solutions.

The VBControlExtender Solution

Visual Basic can receive the events from runtime-instantiated, late-bound ActiveX controls. For example, the code in Figure 1 for the Click event handler of the cmd1 button shows how the Microsoft Access Calendar control (msacal70.ocx) is dynamically instantiated. Visual Basic captures all the events launched by the Calendar control, even if the ProgID of the control (MSCAL.Calendar) is not known until runtime.

As you can see in Figure 1 , the VBControlExtender mechanism is powerful and interesting, but you need to take into account the overhead of this solution. Typically a VBControlExtender variable is used for dynamically added controls, and this performs late binding on the control. Therefore, the overhead involved here is typical of late binding.

The VBControlExtender construct is not applicable to non-ActiveX controls, where you can enable the handling of events in Visual Basic using the WithEvent directive as you saw previously. In other words, you cannot write statements like the following because the Visual Basic syntax does not support it.

Copy

```
Dim WithEvents serverobj As Object
```

As a consequence, Visual Basic cannot directly handle the events launched by COM objects known only at runtime and instantiated with a CreateObject.

A Windows Script Host Solution

You could use Windows® Script Host (WSH) to handle events. WSH is a language-independent scripting host for 32-bit Windows that enables scripts to be executed directly on the Windows desktop or command console without the need to embed those scripts in an HTML document. (You can learn about WSH at http://msdn.microsoft.com/scripting/windowshost/default.htm .) WSH is interesting in this event-handling scenario because it offers a flexible method for the interception of events launched by late-bound COM servers. Take a look at this code from lbscript.vbs:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
Dim serverobj



Set serverobj = WScript.CreateObject("comsrv.comsrvcls", "myob-
ject_")



msgbox "Make the call..."



serverobj.test2(10)




'______________________



sub myobject_event2(v1, v2)
```

```
    msgbox "Incoming Event. v1 = " & CStr(v1) & " v2 = " &
CStr(v2)



end sub



'______________________
```

In this example, comsrv.comsrvcls is the ProgID of a COM object that, after the test2 call, issues event2 with two arguments. It then activates the subroutine myobject_event2(v1, v2) that receives the two parameters issued by the event.

In the following call, WSH connects the object's outgoing interface to the script file after creating the object.

```
WScript.CreateObject (strProgID [,strPrefix])
```

When the object fires an event, WSH calls a subroutine whose name consists of strPrefix and the event name. (Warning: some releases of the documentation for WSH leave out the parameter strPrefix.)

But WSH cannot be the panacea for every situation for many reasons, including performance, environment, context, and so on. So if you don't want to use a WSH solution, and you want Visual Basic to capture events issued by late-bound COM objects where the ProgID is known only at runtime and the object is instantiated through CreateObject, you have another choice: the bridge component I'm about to describe.

The Bridge Solution

Let's look at the development of two components logically inserted between the client and any COM server object instantiated by the client through late binding. The components are the bridge, and a supporting component used by the bridge and the client to transfer data relative to the intercepted event back to the client. The bridge is developed as an ActiveX Template Library (ATL) project in Visual C++®, while the supporting component for the transfer of event information is created in Visual Basic as an ActiveX DLL project. I chose the language and component type for the sake of simplicity and to demonstrate the deep level of integration you can achieve using Visual Basic and Visual C++ together.

For download with this article, you will find four projects that create a coherent framework. The first is COMSRV, a simple COM component. It is used as an example of a COM object instantiated by the Visual Basic client and is used in late binding. The events that this component launches must be intercepted through connection points.

Next is COMBRIDGE, the main component in this scenario. It exposes methods that a Visual Basic client can use to enable or disable the capture of events on a server object. The bridge sends the captured events back to the client using the supporting COMEVENT component. It would have been impossible to achieve the same functionality using Visual Basic.

COMEVENT is the Visual Basic ActiveX DLL that is used by the bridge to send details about the intercepted events to the Visual Basic client and VBCLIENT is the Visual Basic-based demo app that uses the bridge component to capture all the events launched by the late-bound server object that is instantiated at runtime.

In a typical scenario, the four components interact in the following way:

1. VBCLIENT instantiates the COMBRIDGE bridge object that is referenced in early binding and declared using the WithEvents statement in order to intercept its events.
2. VBCLIENT instantiates the COMEVENT object that will hold all the information about the events that the bridge intercepts. The COMEVENT object is created by the Visual Basic client, populated by the bridge, and then read and reset by the Visual Basic client.
3. In late binding, the client instantiates the COMSRV object (whose ProgID is known only at runtime) using CreateObject. Therefore, it is impossible to declare it using WithEvents.
4. The client tells the bridge to be ready to intercept all the events launched by the COMSRV using the startmonitoring method. The client transfers the pointer to the COMSRV server object and the pointer to the supporting object COMEVENT as parameters. From now on, the bridge listens to all the events the server launches.
5. The client invokes one of the methods supported by the server component. The name of the method to call and its arguments are known only at runtime, so the call is issued as a CallByName.
6. After the call, the server sends an asynchronous notification to the components in a position to receive its calls. Specifically, the Sink interface, activated by the bridge, intercepts the call (see step 4 in this list).
7. The bridge receives the event and all of its arguments from the server in the Sink interface.

8. The bridge populates the COMEVENT object with all the information and arguments from the intercepted event and sends a signaling event to the VBCLIENT client. The signaling event has only one argument, the COMEVENT object that has already been populated with information about the event.
9. VBCLIENT receives a notification from the bridge in the Sink interface. It can then examine the COMEVENT object and extract all information relative to the intercepted event.
10. When VBCLIENT wants to stop receiving notifications about the events launched by the server, it can call the stopmonitoring method exposed by the bridge. The bridge sends an Unadvise call that interrupts the connection between the bridge sink interface and the server object connection point.

To see how this works using the demo files provided with this article, build the project using the instructions in the readme.txt available with the download. Once you have built the projects, you can launch the client and use the buttons on the interface to check the interactions between the components and the reception of the events. Theoretically, COMBRIDGE and COMEVENT will not need to be compiled again since they act as a reusable library. The client and the server components can vary, depending on your specific needs.

Developing the Components

Let's now examine how the components were built, paying particular attention to the bridge, which is the real engine of the architecture. The projects described in this article were developed on Microsoft Windows NT® 4.0 Service Pack 4 in Microsoft Visual Studio® 6.0 Service Pack 3.

To make things easier, I did not add the code to handle errors. To use these tools in the real world, you need to handle errors and evaluate all the return values of the calls to functions and methods.

COMSRV

COMSRV.DLL is an in-process COM server that exposes two methods and launches two possible events to the container. I developed the component in C++ with the ATL COM AppWizard using the menu option Insert | New ATL Object | Simple Object. The component class is called comsrvcls. The test1 and test2 methods are defined in IDL, as shown in Figure 2 .

The methods simulate an activity that lasts about three seconds, followed by a notification through the firing of an event. The event1 and event2 events are defined on the dispinterface _IcomsrvclsEvents. This is the definition of the event interface in IDL:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
dispinterface _IcomsrvclsEvents

{

    properties:

    methods:

    [id(1), helpstring("method event1")]

        HRESULT event1();
    [id(2), helpstring("method event2")]



        HRESULT event2([in] long v1, [in] long v2);



};
```

Using the Implement Connection Point Wizard, I implemented the corresponding Fire methods of the events inside the CProxy_IcomsrvclsEvents class (derived from IConnectionPointImpl). The Implement Connection Point Wizard does the following:

- Creates the header file that contains a template for the event objects.
- Includes the header file, adding it to your COM object header file.
- Adds the event proxy class to the class inheritance list of your component.
- Adds an entry to the class connection point map.

The component is a simple example of a server that uses connection points to signal an event to a container, as you can see in **Figure 3**. The server object that fires an event is called the source and the client interface that actually implements the event interface in order to catch the event is called the sink.

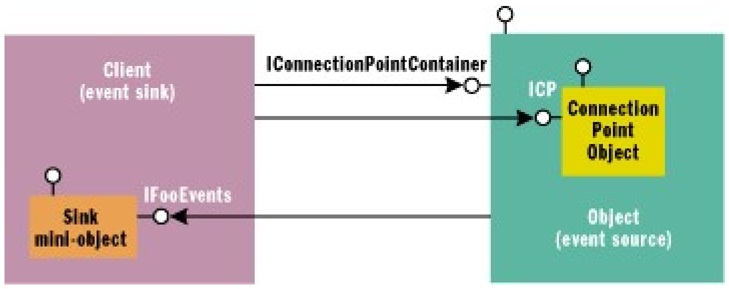


Figure 3 Connection Points and Sink Interface

Figure 4 and **Figure 5** show the server class structure and list the files contained in the project. The server component is registered using the ProgID comsrv.comsrvcls. It is used by VBCLIENT, which instantiates it through a CreateObject and activates its methods using a CallByName.

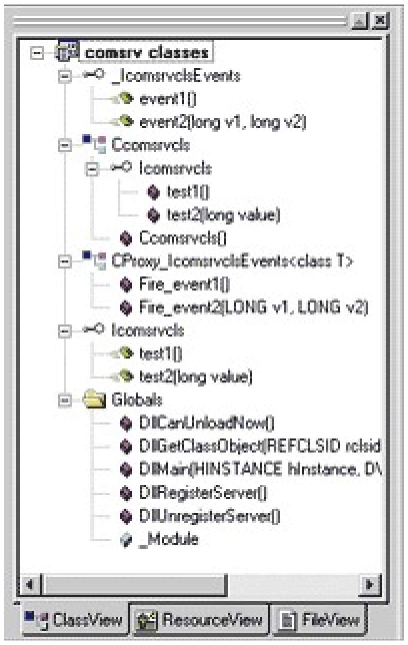


Figure 4 COMSRV Class Structure

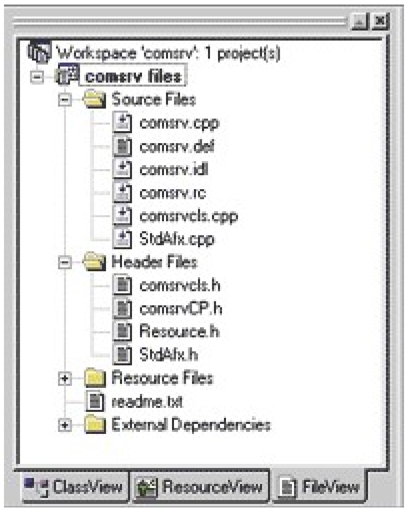
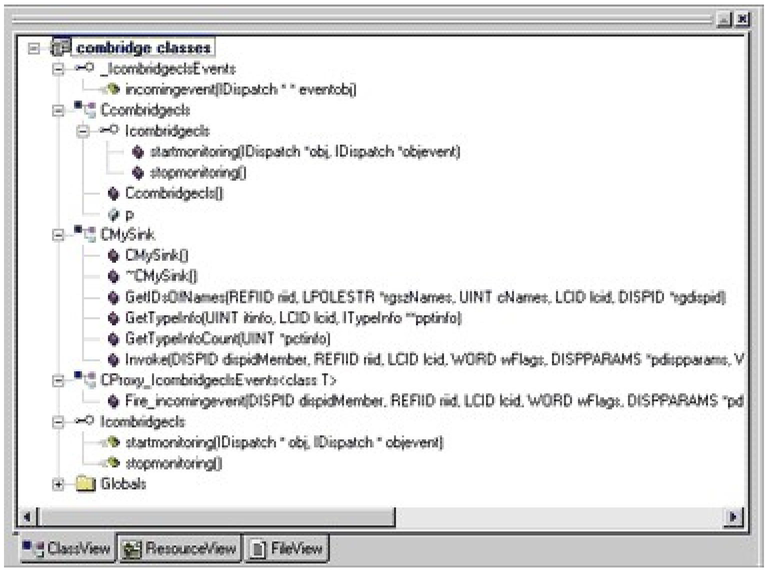


Figure 5 The COMSRV Project

COMBRIDGE

COMBRIDGE.DLL is the main component of the system. This component intercepts all the events of a server object and redirects them to the client.

Figure 6 shows the class structure of the bridge component. I added an ATL Simple Object called combridgecls to the ATL COM project COMBRIDGE. The object is the class component of the server bridge. This object needs the support of connection points in order to implement the mechanism that sends the events to the client.



```
combridge classes
    _IcombridgeclsEvents
        incomingevent(IDispatch * * eventobj)
    Ccombridgecls
        Icombridgecls
            startmonitoring(IDispatch *obj, IDispatch *objevent)
            stopmonitoring()
        Ccombridgecls()
        p
    CMySink
        CMySink()
        ~CMySink()
        GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames, UINT cNames, LCID lcid, DISPID *rgdispid)
        GetTypeInfo(UINT itinfo, LCID lcid, ITypeInfo **pptinfo)
        GetTypeInfoCount(UINT *pctinfo)
        Invoke(DISPID dispidMember, REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS *pdispparams, V
    CProxy_IcombridgeclsEvents<class T>
        Fire_incomingevent(DISPID dispidMember, REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS *pd
    Icombridgecls
        startmonitoring(IDispatch * obj, IDispatch * objevent)
        stopmonitoring()
    Globals
```

Figure 6 COMBRIDGE Class Structure

In the C++ Ccombridgecls class, I defined two methods, startmonitoring and stopmonitoring, that the component exposes to the client:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
interface Icombridgecls : IDispatch



    {
        [id(1), helpstring("method startmonitoring")]

            HRESULT startmonitoring([in] IDispatch *obj,

                                    [in] IDispatch

                                      *objevent);

        [id(2), helpstring("method stopmonitoring")]

            HRESULT stopmonitoring();

    };
```

The startmonitoring method accepts two pointers, obj and objevent, as input. The client uses these pointers to communicate with the bridge. The obj pointer is the server to monitor, and the objevent pointer is the COMEVENT object that transfers data about the intercepted events.

The bridge is able to send the incomingevent event to its container in order to notify the container that an event coming from the server has been intercepted like so:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
dispinterface _IcombridgeclsEvents

{

    properties:

    methods:

    [id(1), helpstring("method incomingevent")]

        HRESULT incomingevent([out] IDispatch

                                   **eventobj);
};
```

The most interesting aspect about the bridge is the method it uses to capture all the events that the server launches. First, I created a new CMySink class (in the files MySink.cpp and MySink.h, as you can see in **Figure 7**) that is used as a custom sink interface to receive events from the server.

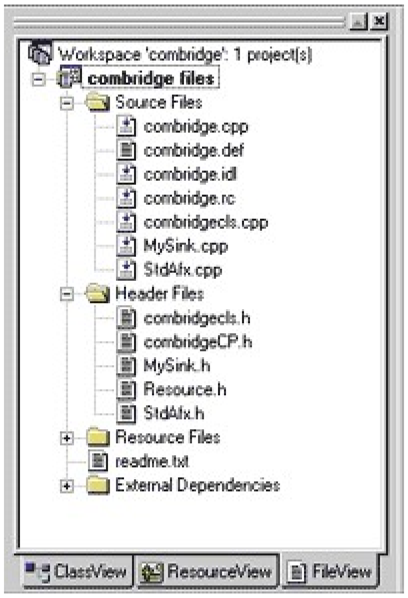


Figure 7 The COMBRIDGE Project

In ATL, you can implement a sink dispinterface in two ways: using an ATL implementation of IDispatch or using an ATL implementation of IUnknown. If you go with the first approach, you need to make a dual interface implementation with a type library. With the second approach, you need to separately handle the DISPIDs yourself. For further details, see the Knowledge Base article Q181277, "AtlSink Uses ATL to Create a Dispinterface Sink ". When you create a sink following this second approach, you do not need a type library nor do you need to provide support in the IDL code of the client (receiver) for the methods (event-receivers) of the sink interface. The bridge component follows the second approach.

The CMySink class is derived and declared in Figure 8 . If the server component whose events you want to capture is already known (from early binding), you could define a sink interface as follows:

Copy

Copy

Copy

Copy

```
BEGIN_COM_MAP(CDispatchSink)

   COM_INTERFACE_ENTRY(IDispatch)

   COM_INTERFACE_ENTRY_IID(DIID_DsomeEvents, IDispatch)

END_COM_MAP()
```

(or you could use the macro SINK_ENTRY_EX, as explained in Knowledge Base article Q194179, "AtlEvnt.exe Creates ATL Sinks Using IDispEventImpl ").

The first parameter for the COM_INTERFACE_ENTRY_IID would be the dispinterface IID. In this situation, however, it is impossible to define the IID since the server component (and its dispinterface) is not known until runtime. Therefore, I used the macro COM_INTERFACE_ENTRY_FUNC_BLIND instead. The COM_INTERFACE_ENTRY_FUNC_BLIND macro is defined in the file ATLCOM.H located in the Program Files\Microsoft Visual Studio\VC98\ATL\Include\ directory of the Visual C++ 6.0 development environment installation:

Copy

Copy

Copy

Copy

```
#define COM_INTERFACE_ENTRY_FUNC_BLIND(dw, func)\

        {NULL, \

        dw, \

        func},
```

When you use this macro, querying for any interface other than IUnknown will always result in a call to func, regardless of the interface ID. This macro is similar to the COM_INTERFACE_ENTRY_FUNC macro. Instead of filtering out IIDs, however, this macro blindly calls a specified function during interface requests.

In this project, the COM_INTERFACE_ENTRY_FUNC_BLIND macro calls the CallThisForEveryQI function that is implemented in the file MySink.cpp, as shown in Figure 9 . At runtime the CallThisForEveryQI function returns the pointer to the current sink interface contained in pMySink. The pointer pMySink is initialized in the startmonitoring method of the Ccombridgecls class, as you will see shortly. The startmonitoring method binds the sink interface to the connection points found on the server object, so that any server event can be captured by the Invoke method implemented in the CMySink class.

The Invoke method of CMySink calls the Fire_incomingevent method, which is implemented in the CProxy_IcombridgeclsEvents class derived from IConnectionPointImpl, and sends it all the parameters. The Fire_incomingevent code changes from the code generated by the Implement Connection Point Wizard so that it can receive the parameters issued by the Invoke, as you'll find in the SDK documentation and in Figure 10 . The Fire_incomingevent function uses these parameters to correctly populate a COMEVENT object to be sent to the client (for this article, VBCLIENT.EXE).

Through smart pointers, the bridge project can access the component class having a ProgID equal to comevent.comeventcls. The #import directive inserted in the StdAfx.h file allows access to the object model of the COMEVENT object.

Copy

```
#import "..\\comevent\\comevent.dll" no_namespace
```

Figure 11 shows the complete code of the Fire_incomingevent function as it is implemented in the bridge. Notice that the address of the COMEVENT object passed to the startmonitoring method is assigned to the variable _comeventclsPtr pmyevent, a smart pointer for the interface _comeventcls, using the following instruction. (Since the COMEVENT component is written in Visual Basic, there is an underscore at the beginning of _comeventclsPtr.)

Copy

```
pmyevent = pIDispatchobjevent;
```

The following call sends the dispid of the intercepted event to the COMEVENT object.

Copy

```
pmyevent->PuteventID(dispidMember);
```

This is the first step in populating the COMEVENT object that the bridge uses to transfer the information about the intercepted event to the client.

In the following loop, all the intercepted event arguments are loaded in the arguments collections exposed by the COMEVENT component.

Copy

```
for (i=ncArgs; i>=1; i–) { ... }
```

Finally, you need to change the body of the standard loop created by the Connection Point Wizard in order to tell the client that the parameter sent is an object:

Copy

Copy

Copy

```
for (nConnectionIndex = 0;

     nConnectionIndex < nConnections;

     nConnectionIndex++) { ... }
```

That's why the parameter is set as follows

Copy

```
pvars[0].vt = VT_DISPATCH | VT_BYREF;
```

and the value is set like so:

Copy

```
pvars[0].ppdispVal = &pIDispatchobjevent;
```

According to the Automation Platform SDK documentation, if you specify the type VT_DISPATCH | VT_BYREF, you have defined a pointer to a pointer to an object. The pointer to the object is stored in the location referred to by ppdispVal. You will find a description of the COMEVENT component (developed in Visual Basic) later in the article.

I still need to show you how the startmonitoring method of the Ccombridgecls class of the bridge associates the bridge sink interface with the server connection points. Figure 12 shows the code of the startmonitoring and stopmonitoring methods. To enable the reception of events in the startmonitoring method, you need to do the following:

1. Get a pointer (pCPContainer) to the connection point container (IConnectionPointContainer).
   - Call EnumConnectionPoints to enumerate all the connection points that the server supports and obtain an LPCONNECTIONPOINT lpCPT pointer to the connection point.
     - Advise the connection point that the event sink is ready to receive events and pass a pointer to the IUnknown interface of the event sink (pUnkDispatchSink). Although not shown, here would be a good place to check the returned HRESULT for CONNECT_E_CANNOTCONNECT so that some form of reporting can be done.
       - When the sink no longer wants to receive events, call the Unadvise method and pass it the cookie that you got from the call to Advise.

The behavior of the stopmonitoring method is as follows. The Advise method establishes a connection between the connection point object and the client sink. The first parameter, pUnkDispatchSink, is the IUnknown pointer to the client's sink that wants to receive calls for the outgoing interface managed by this connection point. The client sink receives outgoing calls from the connection point. The second parameter, &dwCookie, is a pointer to a returned token that uniquely identifies this connection. The caller later uses this token to delete the connection by passing it to the IConnectionPoint::Unadvise method. If the connection is not successfully established, this value is zero.

COMEVENT

COMEVENT is a Visual Basic ActiveX DLL that is used by the bridge to transfer the data about intercepted events to the client. As mentioned earlier, I could have developed this component (that implements a collection) in Visual C++ as well, but I chose Visual Basic for simplicity and to demonstrate Visual Basic and Visual C++ integration. The files that are part of this component's project are shown in **Figure 13**. The COMEVENT object structure is described in Figure 14 .

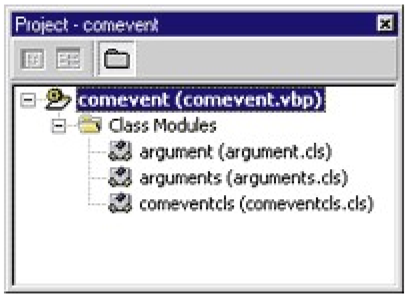


Figure 13 Class Module Files

The comevent.comeventcls component exposes an eventID property that stores the dispid of the intercepted event and contains a collection, named arguments, that is (predictably) a collection of objects of type argument. The implementation of classes that expose collections is described in "Using Properties and Collections to Create Object Models " in the MSDN® Library.

The Add method is called by the bridge component from the Fire_incomingevent method; it populates the collection arguments of the COMEVENT object. The first parameter identifies a value for the argument, while the second identifies the type. An object argument is then instantiated in the Add method and its properties are assigned. Each argument object has two properties: argvalue and argtype. The complete code for the Add method is shown here.

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

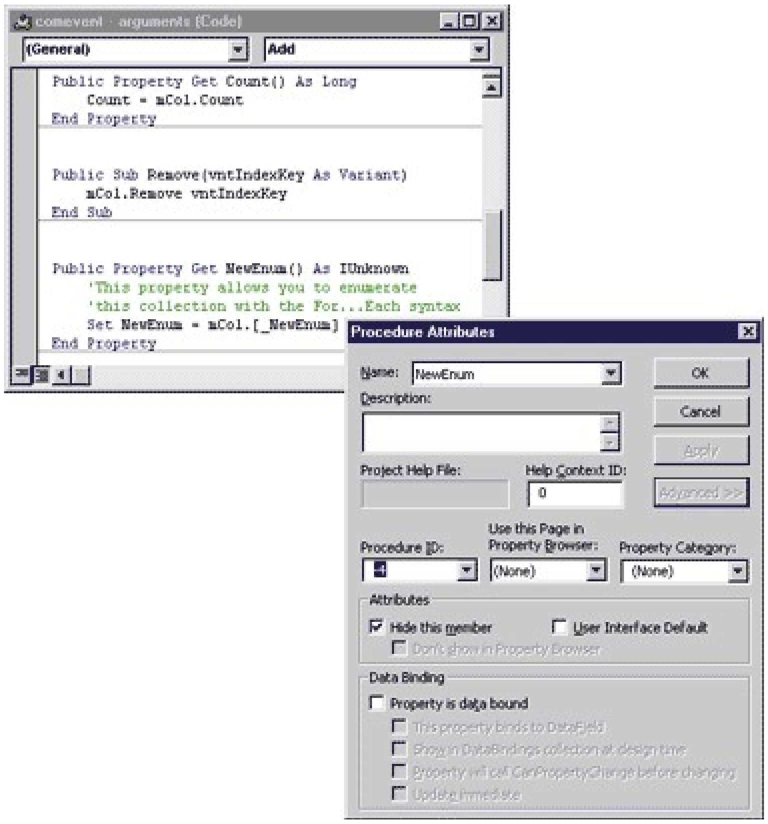
Copy

Copy

```
Public Function Add(ByVal argvalue As Variant,

                    ByVal argtype As Long)

    'Create a new object

    Dim objNewMember As argument

    Set objNewMember = New argument
    'Set the properties passed into the method

    objNewMember.argvalue = argvalue

    objNewMember.argtype = argtype



    mCol.Add objNewMember



Set objNewMember = Nothing



End Function
```

The client can choose to reset the collection received by the bridge to make it ready for the next time. It resets the collection by using the Remove method.

In the class module arguments (implemented in arguments.cls) you can find the member function NewEnum (shown in **Figure 15**) that allows you to iterate the collection using the For Each...Next syntax. NewEnum returns the IUnknown interface of an enumerator object that the For Each�Next loop can use to iterate over the items in a collection. It should be hidden in the type library, so it must have a Procedure ID value of -4 to work with the For Each�Next loop.



```
Public Property Get Count() As Long
    Count = mCol.Count
End Property


Public Sub Remove(vntIndexKey As Variant)
    mCol.Remove vntIndexKey
End Sub


Public Property Get NewEnum() As IUnknown
    'This property allows you to enumerate
    'this collection with the For...Each syntax
    Set NewEnum = mCol.[_NewEnum]
End Property
```

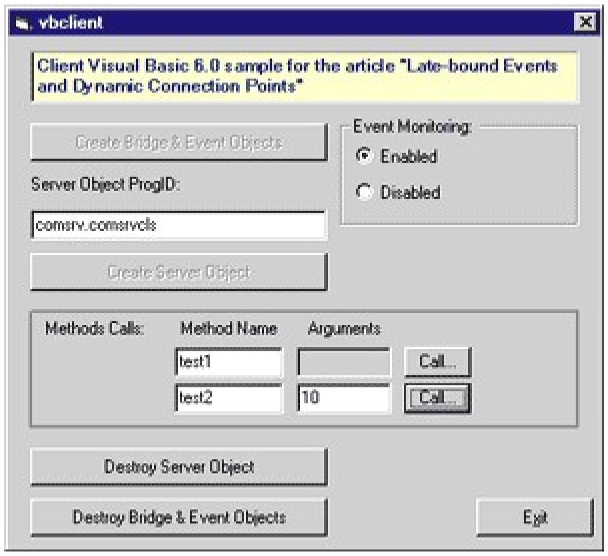
Name: NewEnum
Description:
Project Help File:
Help Context ID: 0
Procedure ID: -4
Use this Page in Property Browser: (None)
Property Category: (None)
[✓] Hide this member
[ ] User Interface Default
[ ] Don't show in Property Browser
[ ] Property is data bound
[ ] This property binds to DataField
[ ] Show in DataBindings collection at design time
[ ] Property will call CanPropertyChange before changing
[ ] Update immediate

Figure 15 Procedure Attributes

In COM, the DISPID_NEWENUM constant has a value of -4, and it can be specified as the procedure ID for NewEnum by opening the Procedure Attributes dialog box from the Tools menu when the code window is open (as shown in **Figure 15**). By selecting NewEnum and pressing the Advanced button, you can type -4 into the Procedure ID box. To ensure that this method is not seen in the type library, you have to check "Hide this member" in the Procedure Attributes dialog box.

VBCLIENT

VBCLIENT is a Visual Basic client that uses the bridge component (and the COMEVENT component) to intercept the events of a server component whose ProgID is known only at runtime. **Figure 16** shows the user interface of the VBCLIENT application.


vbclient

Client Visual Basic 6.0 sample for the article "Late-bound Events and Dynamic Connection Points"

Create Bridge & Event Objects

Server Object ProgID: comsrv.comsrvcls

Create Server Object

Event Monitoring: (● Enabled) ○ Disabled

Methods Calls:	Method Name	Arguments	
	test1		Call...
	test2	10	Call...

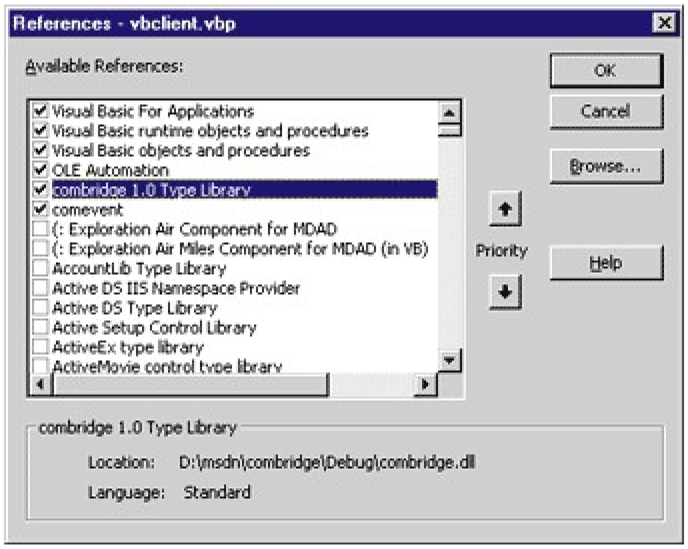
Destroy Server Object

Destroy Bridge & Event Objects

Exit

Figure 16 VBCLIENT User Interface

In the Visual Basic client project References, I selected the type libraries of the bridge and COMEVENT components (see **Figure 17**).


References - vbclient.vbp

Available References:

- [✓] Visual Basic For Applications
- [✓] Visual Basic runtime objects and procedures
- [✓] Visual Basic objects and procedures
- [✓] OLE Automation
- [✓] combridge 1.0 Type Library
- [✓] comevent
- [ ] (: Exploration Air Component for MDAD
- [ ] (: Exploration Air Miles Component for MDAD (in VB)
- [ ] AccountLib Type Library
- [ ] Active DS IIS Namespace Provider
- [ ] Active DS Type Library
- [ ] Active Setup Control Library
- [ ] ActiveEx type library
- [ ] ActiveMovie control type library

OK | Cancel | Browse... | Priority | Help

combridge 1.0 Type Library

Location: D:\msdn\combridge\Debug\combridge.dll

Language: Standard

Figure 17 References to COMBRIDGE and COMEVENT

As a consequence, in VBCLIENT you can declare a bridge object (bridgeobj), and an event object (eventobj), like this:

Copy

Copy

Copy

```
Dim WithEvents bridgeobj As _
```

```
     COMBRIDGELib.combridgecls

Dim eventobj As comevent.comeventcls
```

The server object is known only at runtime, so it must be declared using a Dim statement like this:

```
Dim serverobj As Object
```

By declaring the object bridgeobj with the WithEvents statement, the event handler is defined like this:

```
Private Sub bridgeobj_incomingevent _     (eventobj As Object)
```

The event handler collects calls sent by the Invoke method implemented in the Fire_incomingevent method of the bridge component. The eventobj parameter represents a pointer to a COMEVENT object that transfers the intercepted event information to the client.

If you click on the Create Bridge & Event Objects button in the client application, the application instantiates the bridge and the event components like this:

Copy

Copy

```
Set bridgeobj = New COMBRIDGELib.combridgecls


Set eventobj = New comevent.comeventcls
```

The server object is instantiated using a CreateObject based on a ProgID redefinable at runtime:

Copy

```
Set serverobj = CreateObject(txtServerProgID.Text)
```

The startmonitoring method, available through the bridge object, enables the monitoring:

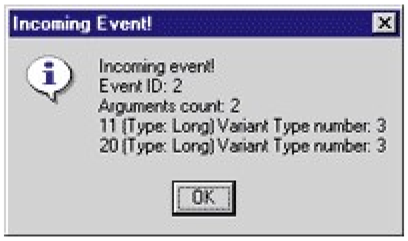
Copy

```
bridgeobj.startmonitoring serverobj, eventobj
```

You pass it the server object and the comeventcls event object pointers.

You can enable or disable the monitoring (calling the startmonitoring and stopmonitoring methods) using the two radio buttons in the Event monitoring frame in the client application.

When Visual Basic receives each event on the bridge object, it calls the function shown in Figure 18 . In this callback function, the event object collection is analyzed and the user receives a feedback message, as shown in **Figure 19**. In a different context, of course, the receipt of this event could cause different responses, most likely responses that are more complex (but not blocking, as handling events rarely generates blocking situations).


Incoming Event!

Incoming event!
Event ID: 2
Arguments count: 2
11 (Type: Long) Variant Type number: 3
20 (Type: Long) Variant Type number: 3

Figure 19 Receiving a Server Event

Going One Step Further

The project described in this article obviously cannot be considered a finished product, but it is a way of delving deeply into some aspects of the COM technology and investigating the potential interactions between Visual Basic and Visual C++. Using this article as a starting point, you can implement components or solutions that allow you to solve problems you might run into while you are developing applications using Visual Basic and Visual C++ together.

When is it useful to intercept events that are known only at runtime? One possible answer is that in some situations, different components can launch events whose handling requires similar business logic. In such cases, the dynamic handling of late-bound events can be a really useful strategy.

Late binding in general, and the handling of events like those described here, always produces overhead. (According to Knowledge Base article Q245115 , early binding is at least twice as fast as late binding.) This aspect cannot be ignored. In some instances it could be helpful to evaluate alternative solutions—for example, those based on direct calls to callback functions. (See "Events vs. Callbacks " in the MSDN Library, and The Visual Programmer column in the April 1999 issue of *MSJ*.)

The Visual C++ 6.0 COMBRIDGE component handles events that came from in-process (.dll) COM servers. To extend the bridge to also handle events from out-of-process (.exe) local and remote COM servers, four additional ATL macros are required in the COM map of the CMySink class (MySink.h file). This tells COM to use the standard marshaler for the connection between the connection points of the server and the sink interface on the client. The COM map of the CMySink class needs to be modified as follows:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
BEGIN_COM_MAP(CMySink)

    COM_INTERFACE_ENTRY(IDispatch)

    COM_INTERFACE_ENTRY_NOINTERFACE(IMarshal)

    COM_INTERFACE_ENTRY_NOINTERFACE(IdentityUnmarshal)

    COM_INTERFACE_ENTRY_NOINTERFACE(IStdMarshalInfo)

    COM_INTERFACE_ENTRY_NOINTERFACE(IExternalConnection)

    COM_INTERFACE_ENTRY_FUNC_BLIND(0, CallThisForEveryQI)
```

```
END_COM_MAP()
```

The COM_INTERFACE_ENTRY_NOINTERFACE macros declare that the sink does not implement a custom marshaler, and that it is the responsibility of the COM subsystem to handle the interprocess communications of the events. The COM_INTERFACE_ENTRY_NOINTERFACE macro, in fact, returns E_NOINTERFACE and terminates COM map processing when the specified interface (IMarshal and some other related interfaces, in this case) is queried for. Without these four additional macros, the function CallThisForEveryQI is reached by the marshaling-related QueryInterfaces call, that CallThisForEveryQI, as it is written, is unable to handle.

Conclusion

The technique described in this article allows a Visual Basic client to intercept events launched by late-bound COM objects instantiated and known only at runtime. This solution was implemented with both Visual Basic and Visual C++ with the use of the ATL library for the development of an intermediate bridge component. The bridge component I developed can be used by any Visual Basic client and completely hides from the client the internal mechanism used to intercept the events. The code is available for download from the link at the top of this article.

For related articles see:
"SAMPLE:AtlSink Uses ATL to Create a Dispinterface Sink "
"Keeping an Eye on Your Browser by Monitoring Internet Explorer 4.0 Events "
"Events vs. Callbacks "

Carlo Randone *works as an MCT/MCSD for ExecuTrain Italia SpA (http://www.executrain.it). He is a developer, trainer, and consultant in Visual Basic, Visual InterDev, Visual C++, COM, and DCOM. He enjoys collecting documentation and hardware pieces related to the historical development in IT. Carlo can be reached at carlo.randone@executrain.it , or carlornd@libero.it .*

From the March 2001 issue of MSDN Magazine