COM Corner: CoPourri

by Steve Teixeira

This month we're going to dig into some COM programming topics inspired by the questions and ideas I've received from you over the months in my email inbox.

Incomplete Definition

Question: I'm using the type library editor to define a new interface. One of the methods of this interface includes a parameter of a COM interface type that isn't supported by default in the type library editor. How can I complete this definition?

Before I explain how to define such a method, it's important to understand why the type library editor behaves the way it does. If you create a new method in the type library editor and take a look at the types available in the `Type` column of the `Parameters` page (Figure 1), you will see a number of interfaces, including `IDataBroker`, `IDispatch`, `IEnumVARIANT`, `IFont`, `IPicture`, `IProvider`, `IStrings`, and `IUnknown`. Why are these the only interfaces available? What makes them so special? They're not special, really, they just happen to be types that are defined in type libraries that are used by this type library. By default, a Delphi type library automatically uses the Borland Standard VCL type library and the OLE Automation type library. You can configure which type libraries are used by your type library by selecting the root node in the tree view in the left pane of the type library editor and choosing the `Uses` tab in the page control in the right pane (Figure 2). The types contained in the type libraries used by your type library will automatically become available in the drop down list shown in Figure 1.

Armed with this knowledge, you've probably figured out that if the interface you wish to use as the method parameter in question is defined in a type library, you can simply use that type library: problem solved. But what if the interface isn't defined in a type library? There are certainly quite a few COM interfaces that are defined by the SDK only in header or IDL files and are not found in type libraries. If this is the case, the best course is to define the method parameter as being of type `IUnknown`. This `IUnknown` can be `QueryInterfaced` in your method implementation for the specific interface type you wish to work with. You should also be sure to document this method parameter as an `IUnknown` that must support the appropriate interface. The code in Listing 1 gives an example of how such a method could be implemented.

Data Exchange

Question: I wish to exchange a block of binary data between an Automation client and server. I understand that COM doesn't support exchange of raw pointers, so how can I accomplish this task?

```
procedure TSomeClass.SomeMethod(
  SomeParam: IUnknown);
var
  Intf: ISomeComInterface;
begin
  Intf := SomeParam
            as ISomeComInterface;
  // remainder of method
implementation
end;
```

➤ *Listing 1*

The easiest way to exchange binary data between Automation clients and servers is to use `safearrays` of bytes. Delphi encapsulates `safearrays` nicely in `OleVariants`. The admittedly contrived example code in Listings 2 and 3 show client and server units that use memo text to demonstrate how to transfer binary data using `safearrays` of bytes.

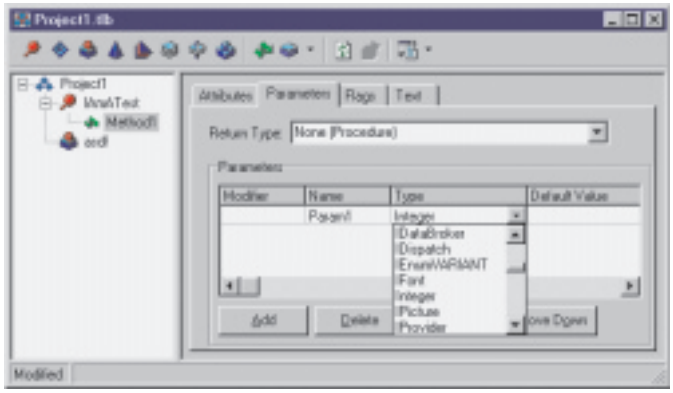
Misfiring Events

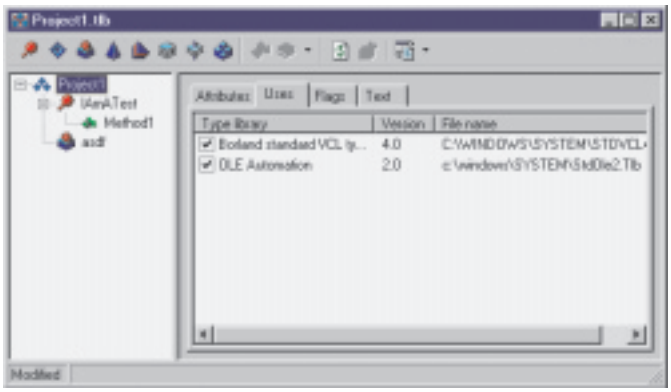
Question: I'm employing the technique you mentioned in the November 1998 issue for surfacing events on objects in my Automation server. I'm connecting to my object from multiple clients, but my events only fire back to one client. I need the events to fire back to all clients. What's a developer to do?

The example code from the November 1998 issue, as well as the events created by Delphi's ActiveX Control Wizard, only supports firing events back to a single client. In order to fire events back to multiple clients, you must write code that enumerates over each advised connection and calls the appropriate method on the sink.

➤ *Below: Figure 1*

➤ *Right: Figure 2*





```
unit ServObj;
interface
uses ComObj, ActiveX, Server_TLB;
type
  TBinaryData = class(TAutoObject, IBinaryData)
  protected
    function Get_Data: OleVariant; safecall;
    procedure Set_Data(Value: OleVariant); safecall;
  end;
implementation
uses ComServ, ServMain;
function TBinaryData.Get_Data: OleVariant;
var
  P: Pointer;
  L: Integer;
begin
  // Move data from memo into array
  L := Length(MainForm.Memo.Text);
  Result := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(Result);
  try
    Move(MainForm.Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(Result);
  end;
end;
procedure TBinaryData.Set_Data(Value: OleVariant);
var
  P: Pointer;
  L: Integer;
  S: string;
begin
  // Move data from array into memo
  L := VarArrayHighBound(Value, 1) -
    VarArrayLowBound(Value, 1) + 1;
  SetLength(S, L);
  P := VarArrayLock(Value);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(Value);
  end;
  MainForm.Memo.Text := S;
end;
initialization
  TAutoObjectFactory.Create(ComServer, TBinaryData,
    Class_BinaryData, ciSingleInstance, tmApartment);
end.
```

➤ *Listing 2*

This can be done by modifying the example from November 1998.

To support multiple client connections on a connection point, we must pass the `ckMulti` in the `Kind` parameter of `TConnectionPoints.CreateConnectionPoint`. This method is called from the Automation object's `Initialize` method:

```
FConnectionPoints.
  CreateConnectionPoint(
  AutoFactory.EventIID,
  ckMulti, EventConnect);
```

Before connections can be enumerated, we need to obtain a reference to our `IConnectionPointContainer`. From the `IConnectionPointContainer`, we can obtain the `IConnectionPoint` representing the outgoing interface, and using the `IConnectionPoint.EnumConnections` method, we can obtain an `IEnumConnections` interface that can be used to enumerate the connections. All of this logic is encapsulated into the method in Listing 4.

After the enumerator interface has been obtained, calling the sink for each client is just a matter of iterating over each connection. This logic is demonstrated in Listing 5, which fires the `OnTextChanged` event.

Finally, to enable clients to connect to a single active instance of the Automation object, we must call the `RegisterActiveObject` COM API function. This function accepts as parameters an `IUnknown` for the

➤ *Listing 4*

```
function TServerWithEvents.GetConnectionEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID, CP));
  CP.EnumConnections(Result);
end;
```

➤ *Listing 3*

```
unit CliMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, ExtCtrls, Server_TLB;
type
  TMainForm = class(TForm)
    Memo: TMemo;
    Panel1: TPanel;
    SetButton: TButton;
    GetButton: TButton;
    OpenButton: TButton;
    OpenDialog: TOpenDialog;
    procedure OpenButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SetButtonClick(Sender: TObject);
    procedure GetButtonClick(Sender: TObject);
  private
    FServer: IBinaryData;
  end;
var MainForm: TMainForm;
implementation
{$R *.DFM}
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FServer := CoBinaryData.Create;
end;
procedure TMainForm.OpenButtonClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    Memo.Lines.LoadFromFile(OpenDialog.FileName);
end;
procedure TMainForm.SetButtonClick(Sender: TObject);
var
  P: Pointer;
  L: Integer;
  V: OleVariant;
begin
  // Send memo data to server
  L := Length(Memo.Text);
  V := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(V);
  try
    Move(Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(V);
  end;
  FServer.Data := V;
end;
procedure TMainForm.GetButtonClick(Sender: TObject);
var
  P: Pointer;
  L: Integer;
  S: string;
  V: OleVariant;
begin
  // Get server's memo data
  V := FServer.Data;
  L := VarArrayHighBound(V, 1) - VarArrayLowBound(V, 1) + 1;
  SetLength(S, L);
  P := VarArrayLock(V);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(V);
  end;
  Memo.Text := S;
end;
end.
```

object, the CLSID of the object, a flag indicating whether the registration is strong (the server should be AddRefd) or weak (do not AddRef the server), and a handle that is returned by reference.

```
RegisterActiveObject(
  Self as IUnknown,
  Class_ServerWithEvents,
  ACTIVEOBJECT_WEAK,
  FObjRegHandle);
```

Listing 6 shows the complete source code for the ServAuto unit, which ties all of these tidbits together.

On the client side, a small adjustment enables clients to connect to an active instance if it is already running. This is accomplished using the GetActiveObject COM API function as shown in Listing 7.

Steve Teixeira is the Vice President of Software Development at DeVries Data Systems, a Silicon Valley consulting firm. Send your feedback, ideas, or questions to steve@dvdata.com

➤ *Listing 5*

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then (ConnectData.pUnk as
        IServerWithEventsEvents).OnTextChanged(
        (Sender as TMemo).Text);
  end;
end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
var ActiveObj: IUnknown;
begin
  // Get active object if it's available,
  // or create anew if not
  GetActiveObject(Class_ServerWithEvents, nil, ActiveObj);
  if ActiveObj <> nil then
    FServer := ActiveObj as IServerWithEvents
  else
    FServer := CoServerWithEvents.Create;
  FEventSink := TEventSink.Create(Self);
  InterfaceConnect(FServer, IServerWithEventsEvents,
   FEventSink, FCookie);
end;
```

➤ *Listing 6*

```
unit ServAuto;
interface
uses ComObj, ActiveX, AxCtrls, Server_TLB;
type
  TServerWithEvents = class(TAutoObject,
    IConnectionPointContainer, IServerWithEvents)
  private
    FConnectionPoints: TConnectionPoints;
    FObjRegHandle: Integer;
    procedure MemoChange(Sender: TObject);
  protected
    procedure AddText(const NewText: WideString); safecall;
    procedure Clear; safecall;
    function GetConnectionEnumerator: IEnumConnections;
    property ConnectionPoints: TConnectionPoints read
     FConnectionPoints implements IConnectionPointContainer;
  public
    destructor Destroy; override;
    procedure Initialize; override;
  end;
implementation
uses Windows, ComServ, ServMain, SysUtils, StdCtrls;
destructor TServerWithEvents.Destroy;
begin
  inherited Destroy;
  // Make sure I'm removed from ROT
  RevokeActiveObject(FObjRegHandle, nil);
end;
procedure TServerWithEvents.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(
      AutoFactory.EventIID, ckMulti, EventConnect);
  // Route main form memo's OnChange event to MemoChange method:
  MainForm.Memo.OnChange := MemoChange;
  // Register this object with COM's Running Object Table
  // (ROT) so other clients can connect to this instance.
  RegisterActiveObject(Self as IUnknown,
    Class_ServerWithEvents, ACTIVEOBJECT_WEAK,
    FObjRegHandle);
end;
procedure TServerWithEvents.Clear;
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  MainForm.Memo.Lines.Clear;
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as
        IServerWithEventsEvents).OnClear;
  end;
end;
procedure TServerWithEvents.AddText(const NewText:
  WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;
procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then (ConnectData.pUnk as
        IServerWithEventsEvents).OnTextChanged(
        (Sender as TMemo).Text);
  end;
end;
function TServerWithEvents.GetConnectionEnumerator:
  IEnumConnections;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer,
    Container));
  OleCheck(Container.FindConnectionPoint(
    AutoFactory.EventIID, CP));
  CP.EnumConnections(Result);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TServerWithEvents,
    Class_ServerWithEvents, ciMultiInstance, tmApartment);
end.
```

➤ *Listing 7*