# *COM Corner:* CoPourri

**by Steve Teixeira**

***Potpourri***
*Pronunciation: "pO-pu-'rE*
*Function: noun*
*Etymology: French pot pourri, literally, rotten pot*
*Date: 1749*
*1: a mixture of flowers, herbs, and spices that is usually kept in a jar and used for scent*
*2: a miscellaneous collection: MEDLEY*

***CoPourri***
*Pronunciation: "kO-pu-'rE*
*Function: noun*
*Etymology: English COM, French pourri, literally, COM pot*
*Date: 1999*
*1: A miscellaneous collection of COM tips derived from the questions of COM developers and readers of The Delphi Magazine*
*2: COM tips that smell nice*

I get a lot of COM questions from readers of this magazine and visitors to Borland's newsgroups. Often, there are so many that I can't always respond to every query, but I try to make up for it by keeping track of questions and answering them in this column. Many of these questions don't merit an entire column, so I like to periodically set out a fresh batch of CoPourri and address several issues in a single column. I hope you enjoy the collection I have set out this month, as I cover how to tell whether an ActiveX control is in run or design mode, how to implement multiple `IDispatches` on a single automation object, and how to use the Running Object Table (ROT).

## The Running Man

Determining whether a component is operating in run or design mode is something that VCL component developers take for granted. As you probably know this can be accomplished simply by checking the `ComponentState` set property for the `csDesigning` member. What about us ActiveX control developers? Is it possible for us to easily determine whether an ActiveX control is currently operating in run or in design mode? The answer is yes, but it's not quite so straightforward.

The first step on the path to achieving this ActiveX version of self awareness is to obtain some interface for the COM object that implements the container for the ActiveX control. The most convenient container interface accessible to you from within an ActiveX control is `IOleClientSite`, which can be obtained by `QueryInterfacing` yourself for `IOleObject` and then calling the `IOleObject.GetClientSite` method. Once you have this interface into the container object, you can `QueryInterface` it for `IAmbientDispatch`. `IAmbientDispatch` is a dispinterface that provides you with all kinds of great information on the control's current operating environment, and is defined in the `AxCtrls` unit as shown in Listing 1.

In particular, we are interested in the `UserMode` property. When this property is `True`, the control is in run mode, and when `False`, the control is in design mode. The `IsControlRunning` function in Listing 2 consolidates these steps.

Note that this method is called differently depending on whether you are using it from an ActiveX control or an ActiveForm. From an ActiveX control, you simply pass `Self` to `IsControlRunning`, since the `IOleObject` implementation lives in the `TActiveXControl` class:

```
if IsControlRunning(Self) then
  ... // from ActiveX control
```

However, the `IOleObject` implementation for an ActiveForm resides in the ActiveForm's `ActiveFormControl` property. Therefore, the call from an ActiveForm would look more like this:

```
if IsControlRunning(
  ActiveFormControl) then
    ... // from an ActiveForm
```

As a simple example, I added the `IsControlRunning` function to an ActiveForm project, and put these lines in the `PaintEvent` method for my ActiveForm:

```
if IsControlRunning(
  ActiveFormControl) then
  Color := clBlue
else
  Color := clRed;
```

➤ *Listing 1*

```
IAmbientDispatch = dispinterface
  ['{00020400-0000-0000-C000-000000000046}']
  property BackColor: Integer dispid DISPID_AMBIENT_BACKCOLOR;
  property DisplayName: WideString dispid DISPID_AMBIENT_DISPLAYNAME;
  property Font: IFontDisp dispid DISPID_AMBIENT_FONT;
  property ForeColor: Integer dispid DISPID_AMBIENT_FORECOLOR;
  property LocaleID: Integer dispid DISPID_AMBIENT_LOCALEID;
  property MessageReflect: WordBool dispid DISPID_AMBIENT_MESSAGEREFLECT;
  property ScaleUnits: WideString dispid DISPID_AMBIENT_SCALEUNITS;
  property TextAlign: Smallint dispid DISPID_AMBIENT_TEXTALIGN;
  property UserMode: WordBool dispid DISPID_AMBIENT_USERMODE;
  property UIDead: WordBool dispid DISPID_AMBIENT_UIDEAD;
  property ShowGrabHandles: WordBool dispid DISPID_AMBIENT_SHOWGRABHANDLES;
  property ShowHatching: WordBool dispid DISPID_AMBIENT_SHOWHATCHING;
  property DisplayAsDefault: WordBool dispid DISPID_AMBIENT_DISPLAYASDEFAULT;
  property SupportsMnemonics: WordBool dispid DISPID_AMBIENT_SUPPORTSMNEMONICS;
  property AutoClip: WordBool dispid DISPID_AMBIENT_AUTOCLIP;
end;
```

The idea here is that the form should appear blue when the ActiveForm is in run mode and red when the ActiveForm is in design mode. Figures 1 and 2 demonstrate the effectiveness of this technique.

One gotcha with this approach is that it cannot be used from within the `Initialize` and `InitializeControl` methods, because they are executed before the client site is property initialized. Therefore, you'll have to call `IsControlRunning` in a method that is called after the client site is initialized, as in my example.

## Multiple IDispatches?

One of the whole points of COM is the ability to surface multiple interfaces from a single object. This enables object implementers to provide multiple well-defined means for manipulating an object's properties and behavior. We already know that multiple interfaces on a single object work fine, because every automation object you create supports at least `IUnknown` and `IDispatch`. However, what happens when you wish to support more than one `IDispatch` interface on a single object?

Well, when you're working with early-bound interfaces, this kind of thing works swimmingly: your compiler generates vtables for all of the interfaces in your object, your clients query for those interfaces to receive the vtables, your clients call via those vtables, things just seem to go according to plan! However, throw a little late binding into the mix and suddenly things are not going quite so swell as before.

➤ *Listing 2*

```
function IsControlRunning(Control: IUnknown): Boolean;
var
  OleObj: IOleObject;
  Site: IOleClientSite;
begin
  Result := True;
  // Get control's IOleObject pointer.  From that, get container's
  // IOleClientSite.  From that, get IAmbientDispatch.
  if (Control.QueryInterface(IOleObject, OleObj) = S_OK) and
    (OleObj.GetClientSite(Site) = S_OK) and (Site <> nil) then
    Result := (Site as IAmbientDispatch).UserMode;
end;
```

➤ *Below: Figure 1*
*Bottom: Figure 2*


```
[
  uuid(94ED6F09-5C0D-40BA-BC06-0069D77719AF),
  version(1.0),
  helpstring("Project1 Library")
]
library Project1
{
  importlib("StdOle2.Tlb");
  importlib("STDVCL40.DLL");
  [
    uuid(7E34A054-0931-405D-8D72-11F01C3AFD55),
    version(1.0),
    helpstring("Dispatch interface for MultiObj Object"),
    dual,
    oleautomation
  ]
   interface IMultiObj: IDispatch
  {
    [id(0x00000001)]
    HRESULT _stdcall SomeMethod( void );
  };
  [
    uuid(D407D6A0-88D2-4BBF-8B1E-EE44C3C7CA8C),
    version(1.0),
    dual,
    oleautomation
  ]
   interface ISecondIntf: IDispatch
  {
    [id(0x00000001)]
    HRESULT _stdcall OtherMethod( void );
  };
  [
    uuid(CAB19A4E-73ED-4D3A-83AB-01AFE4BD3394),
    version(1.0),
    helpstring("MultiObj Object")
  ]
  coclass MultiObj
  {
    [default] interface IMultiObj;
    interface ISecondIntf;
  };
};
```

➤ *Listing 3*

```
procedure TMultiObj.OtherMethod;
begin
  ShowMessage('OtherMethod was called');
end;
procedure TMultiObj.SomeMethod;
begin
  ShowMessage('SomeMethod was called');
end;
```

➤ *Listing 4*

To illustrate, imagine a Delphi Automation object defined as shown in Listing 3 (copied from the type library editor).

As you can see, the coclass is called `MultiObj`, and it supports two `IDispatch` descendants, `IMultiObj` and `ISecondIntf`. These interfaces each have one method, `SomeMethod` and `OtherMethod`, respectively. The implementation for these methods simply shows a dialog, as you can see in Listing 4

In creating a client project for this Automation server, a few lines of code on a button click will suffice:

```
procedure TForm1.Button1Click(
  Sender: TObject);
var
  V: OleVariant;
begin
  V := CoMultiObj.Create;
  V.SomeMethod;
end;
```

As you can see, this client uses early binding to call the method of the `IMultiObj` interface of the Automation server. The results of clicking the button are shown in Figure 3.

Modifying the client to call a method of the method of the `ISecondIntf` interface requires only changing the line which calls `SomeMethod` to call `OtherMethod`:

```
  V.OtherMethod;
```

After making the change, the result of compiling, running, and clicking the button is shown in Figure 4.

This simple experiment illustrates the fact that you can only call methods of one of the `IDispatch` interfaces through late binding. The problem here is that COM only supports one late-bound `IDispatch` interface per object. This can definitely make for some developer confusion, and I'm certain this is one of the things COM guru Don Box had in mind when he referred to the 'general cruftiness of all things `IDispatch`'.

Unfortunately, there isn't a way to defeat this problem. However, it is possible to control which interface you wish to make accessible via early binding. You do this by marking the interface you wish to be early-bindable as the 'default' interface of the coclass in the type library editor.

➤ *Figure 3*



➤ *Figure 4*

## Brain ROT

A common question among Delphi COM developers is, 'Why can't I get to my running Delphi Automation server using the `GetActiveObject` API?'

The answer to this is: 'Because Delphi applications do not automatically add themselves to the list that `GetActiveObject` looks into'. That list is known as the Running Object Table, or ROT for short. Applications can register COM objects into the ROT using the `RegisterClassObject` API. Once that is done, the registered COM object will be accessible to other processes on the same machine using the `GetActiveObject` API.

Listing 5 shows a simple Automation server that registers itself with the ROT. Notice that the object also cleans up after itself by calling `RevokeActiveObject` prior to its destruction. The client code for connecting to this server is shown in Listing 6. Figure 5 shows the server application running alongside a couple of clients.

```
unit SrvMain;
interface
uses ComObj, ActiveX, Srv_TLB, StdVcl;
type
  TRegObj = class(TAutoObject, IRegObj)
  private
    FRegCookie: Integer;
  protected
    procedure AddString(const Value: WideString); safecall;
  public
    destructor Destroy; override;
    procedure Initialize; override;
  end;
implementation
uses ComServ, Dialogs, SrvU;
{ TRegObj }
destructor TRegObj.Destroy;
begin
  RevokeActiveObject(FRegCookie, nil);
  inherited Destroy;
end;
procedure TRegObj.Initialize;
begin
  inherited Initialize;
  OleCheck(RegisterActiveObject(Self, CLASS_RegObj,
ACTIVEOBJECT_WEAK, FRegCookie));
end;
procedure TRegObj.AddString(const Value: WideString);
begin
  MainForm.Memo.Lines.Add(Value);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TRegObj,
    Class_RegObj, ciMultiInstance, tmApartment);
end.
```

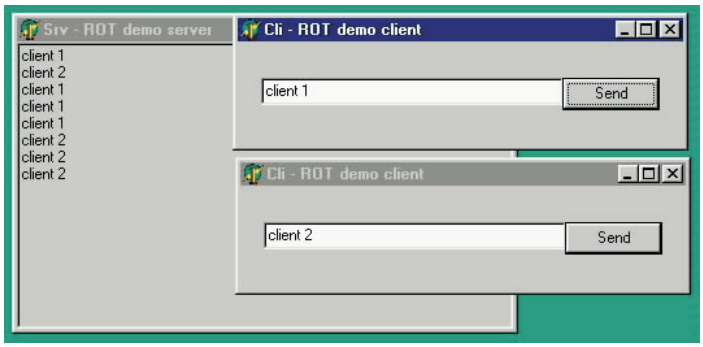
➤ *Listing 5*

I hope you enjoyed this month's CoPourri. Keep sending me those COM questions, and I'll pick the best and publish the answers in *COM Corner*.

Steve Teixeira is VP of software development of DeVries Data Systems (www.dvdata.com), an interactive architect firm based in Silicon Valley. You can email Steve at steve@dvdata.com. Thanks to Merriam-Webster's WWWebster Dictionary at www.m-w.com for the nice definition of potpourri.

```
procedure TMainForm.FormCreate(Sender: TObject);
var Unk: IUnknown;
begin
  if Succeeded(GetActiveObject(CLASS_RegObj, nil, Unk)) and (Unk <> nil) then
    FSrv := Unk as IRegObj
  else
    FSrv := CoRegObj.Create;
end;
```

➤ *Listing 6*




➤ *Figure 5*