COM Corner:
New COM Features In Delphi 5

by Steve Teixeira

It's that special time of the year again, when we all wait with anticipation for the postman to come knocking with the latest copy of Delphi in hand. Once we get it installed, we immediately fire it up to answer the question playing over and over again in our minds, 'What's in this new version for *me*?' Well, if you happen to be a COM developer, make yourself comfortable because we're about to discuss what's new for COM developers in Delphi 5.

Active Server Objects

Delphi 5 introduces a new wizard that enables easy creation of Active Server Objects, which are COM objects designed to be used in the context of Active Server Pages (ASPs).

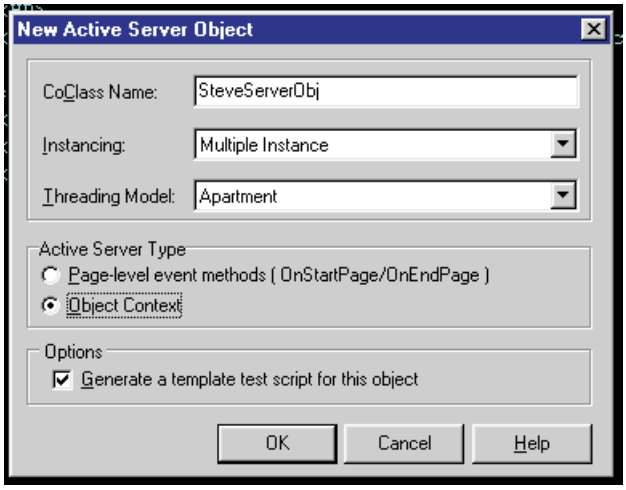
If you're unfamiliar with ASPs, they provide a means for combining some mix of HTML, VBScript, JScript, Automation objects and Java classes in order to generate dynamic web content. In contrast to regular static web pages (HTML files), which are simply passed through by the web server to the client, ASP files are interpreted by the web server before content is sent to the client. As a part of this interpretation, the server will execute script within the document and create instances of objects specified in the script. The scripts and object typically generate dynamic HTML content, which can then be passed on to the web client along with any plain old HTML contained in the ASP document. The advantage of the ASP model is that scripts are interpreted by the server prior to the document being passed to web clients, so content can be dynamic and scripts do not need to be run on the clients.

While ASPs can work with any Automation object, Automation objects that are aware of the context in which they are executing will integrate much more snugly. There are two ways that an Active Server Object can make itself aware that it is executing within the context of an ASP. The older technique, designed for Internet Information Server (IIS) 3 and 4, involves the Automation Object exposing methods called `OnStartPage` and `OnEndPage`. The current technique, used with IIS 5, involves calling the `GetObjectContext` COM API function to obtain the current execution context.

If you are designing objects for IIS 3 or 4, `OnStartPage` is called after the web server loads the ASP and an instance of the Automation server has been created, while `OnEndPage` is called just before the web server is through processing the page. The execution context is passed as a parameter to `OnStartPage` method in the form of an `IUnknown` that supports the `IScriptingContext` interface. These methods are implemented automatically in the code generated by the wizard for Active Server Objects and the framework behind the wizard.

➤ *Listing 1*

```
<HTML>
<BODY>
<TITLE> Testing Delphi ASP </TITLE>
<CENTER>
<H3>You should see the results of your Delphi Active Server method below</H3>
</CENTER>
<HR>
<% Set DelphiASPObj = Server.CreateObject("SteveASO.SteveServerObj")
   DelphiASPObj.{Insert Method name here}
%>
<HR>
</BODY>
</HTML>
```



➤ *Figure 1*

For IIS 5, Active Server Objects are loading within the context of Microsoft Transaction Server (MTS) and therefore objects are able to call `GetObjectContext` at any time to obtain information about the execution context. If you're unfamiliar with MTS, you may wish to take a look at the May and June 1999 issues, where I discuss this technology in detail.

When you're ready to create a new Active Server Object, you may do so by clicking `File | New...` and selecting the `Active Server Object` item from the `ActiveX` page of the `New Items` dialog. You will then be presented with the `New Active Server Object` dialog, which is shown in Figure 1.

Most of this dialog should be familiar to you, as it is based on the Automation Object wizard. Different in this dialog, however, are the

options for selecting the server type and choosing whether to create a test ASP document. The server type options enable you to select whether you wish the object to function as an IIS 3/4 object or an IIS 5 object, as described above. If you choose to create a test ASP, a text file will be created for you that looks very much like the one shown in Listing 1.

You'll notice the bit of VBScript in the middle of this document that creates the instance of the object, followed by a line of code that shows how to call a method on the Automation server. To demonstrate, I will add a method to the object called `DoIt` that emits some text to the web client. I do this by adding the method to the interface in the type library editor, and filling in the implementation for the stub generated in the source file. Listing 2 shows the complete source for the unit containing my Active Server Object.

You might be asking yourself, 'Hey, where is all that `Request` and `Response` stuff coming from in the `DoIt` method?' These are properties of the ancestor class whose values are obtained using the object context I mentioned earlier. This particular object supports IIS 5 and therefore descends from `TASPMTSObject`, which is defined in the `AspTlb` unit as in Listing 3.

IIS 3/4 objects will descend from a similar object called `TASPObject`, which is defined in `AspTlb` as in Listing 4.

Each of these classes encapsulate the details of how to obtain instances of the `IRequest`, `IResponse`, `ISessionObject`, `IServer`, and `IApplicationObject` so that you can focus on working with these interfaces to make your Active Server Object do whatever your heart desires. Each of these interfaces are documented in the Microsoft Developer Network, which can be accessed online at http://msdn.microsoft.com.

I call the `DoIt` method from my ASP by changing the line:

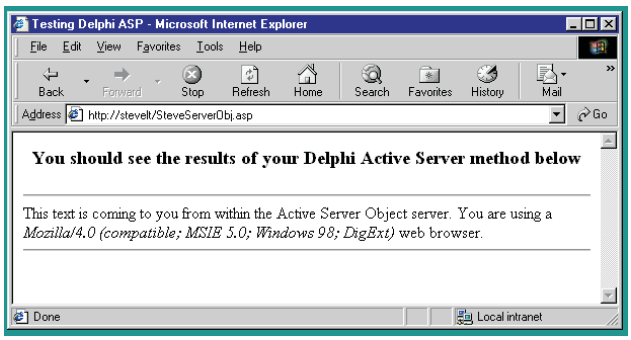
```
DelphiASPObj.{
  Insert Method
  name here}
```

to read:

```
DelphiASPObj.DoIt
```

The result is shown in Figure 2.

➤ *Figure 2*




```
unit Main;
interface
uses
  ComObj, ActiveX, AspTlb, SteveASO_TLB, StdVcl;
type
  TSteveServerObj = class(TASPMTSObject, ISteveServerObj)
  protected
    procedure DoIt; safecall;
  end;
implementation
uses
  ComServ;
procedure TSteveServerObj.DoIt;
var UserAgent: string;
begin
  UserAgent := Request.ServerVariables.Item['HTTP_USER_AGENT'];
  Response.Write('This text is coming to you from within the Active Server ' +
    'Object server.  You are using a <i>' + UserAgent + '</i> web browser.');
end;
initialization
  TAutoObjectFactory.Create(ComServer, TSteveServerObj, Class_SteveServerObj,
    ciMultiInstance, tmApartment);
end.
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

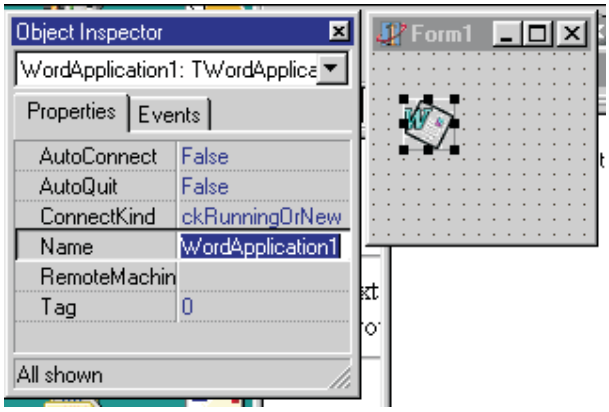
```
TASPMTSObject = class(TAutoObject)
private
  function GetApplication: IApplicationObject;
  function GetRequest: IRequest;
  function GetResponse: IResponse;
  function GetServer: IServer;
  function GetSession: ISessionObject;
public
  property Request: IRequest read GetRequest;
  property Response: IResponse read GetResponse;
  property Session: ISessionObject read GetSession;
  property Server: IServer read GetServer;
  property Application: IApplicationObject read GetApplication;
end;
```

```
TASPObject = class(TAutoObject, IASPObject)
private
  FScriptingContext: IScriptingContext;
  function GetApplication: IApplicationObject;
  function GetRequest: IRequest;
  function GetResponse: IResponse;
  function GetScriptingContext: IScriptingContext;
  function GetServer: IServer;
  function GetSession: ISessionObject;
public
  procedure OnStartPage(AScriptingContext: IUnknown); safecall;
  procedure OnEndPage; safecall;
  property ScriptingContext: IScriptingContext read GetScriptingContext;
  property Request: IRequest read GetRequest;
  property Response: IResponse read GetResponse;
  property Session: ISessionObject read GetSession;
  property Server: IServer read GetServer;
  property Application: IApplicationObject read GetApplication;
end;
```

➤ *Listing 4*

COM Server Components

Another new feature in Delphi 5 is the ability to generate a `TComponent` wrapper for an Automation server. This is done by the type library wrapper generator, which creates a descendant of the new `TOleServer` class with properties and methods that manipulate the properties and methods of the Automation server that it encapsulates. The best example of this technology is the collection of items found on the `Servers` tab of the Component Palette. The source for these components is in the \Delphi5\OCX\Servers subdirectory. Figure 3 shows the act of



➤ *Figure 3*

editing the properties of a `TWordApplication` component.

Looking at Figure 3 you may notice that properties of the Word Application object are not present in the Object Inspector. This is because the published properties are `IFDEF`ed out by default. In order to see the properties, you would need to rebuild the components from source using the `LIVE_SERVER_AT_DESIGN_TIME` conditional define. Of course, the properties are left out by default for a good reason; they don't work correctly.

Let me preface my explanation by laying my cards on the table and telling you that my company produces a suite of components that are engineered to make Automation with MS Office applications easier. It's important for me to tell you, however, that this doesn't taint my opinion of the components on the `Server` tab, but rather it has given me the unique perspective of knowing first-hand the problems that needed to be solved for such components to work correctly and therefore the ability to see easily what problems were not fully addressed.

The problem with the approach of using a generic wrapper creator to encapsulate Automation servers is it does not take into account interdependencies amongst properties. For example, on some arbitrary Automation server, Property A might not be able to be read until Function B is called. Or Property C might not be available until Property D is set. And of course, no properties or methods can be invoked unless the Automation server is activated. A generic wrapper generator can never know about such interdependencies, and therefore would not be able to generate a working set of published properties for the component wrapper.

Another issue that arises from viewing an Automation server's properties at design-time is that the property read and write methods are implemented simply to get and set the Automation server's properties. This can introduce some confusion because there is no way to distinguish whether one wants to have the properties from the component 'pushed' to the Automation server or the component property values 'pulled' from the Automation server.

In short, there are a lot of technical hurdles associated with publishing Automation server properties, so Borland took the prudent approach of `IFDEF`ing them out. If you are working with a server that you know will not experience any problems stemming from the issues I just mentioned, then you can compile with the condition define as I mentioned.

Properties aside, the COM server components do serve a very useful purpose of managing Automation server instantiation and lifetime, which is always code I'd rather not write by hand. Simply drop the component and you know that server lifetime will be properly managed.

One additional nice new addition associated with the COM server components, however, is the automatic handling of events and routing of Automation events to Object Pascal events. This is made possible by the `TServerEventDispatch` class found in the new `OleServer` unit and the instance of this class maintained by `TOleServer`. All of the COM event stuff happens behind the scenes, and it is surfaced at the component level very nicely by a single method called `InvokeEvent`, which is overridden by the type library wrapper generator to check for the proper dispids, unpack parameters from an array of variants, and call to the Object Pascal event.

Type Library Updates Dialog

One new product feature that tends to leave me scratching my head is the type library updates dialog. You can enable this dialog by selecting the option on the `Type Library` page of the `Environment Options` dialog. When enabled, you are presented upon refresh of the type library editor with a dialog explaining what changed in the type library since the last refresh and giving you the option to selectively not implement certain interface methods. This dialog is shown in Figure 4.

That's all jim-dandy, except for the fact that if you choose to employ the dialog by de-selecting any of the items in the checked listbox, your code will not compile. The reason for this, of course, is that all methods of an interface must be present in a class that implements the interface. It's not possible to selectively choose not to implement certain methods of an interface. Interfaces, being a contract of supported functionality, are an all-or-nothing proposition. As long as you know about the quirks of this feature going into the game, you won't be confused when your code stops compiling.

Porting Gotchas

In addition to new features, there were a few changes to the COM support between Delphi 4 and 5 that could cause you some grief as you try to port over your applications.

HResult

In Delphi 4, `HResult` was a 32-bit *unsigned* integer. The definition has changed in Delphi 5 such that `HResult` is now a 32-bit *signed* integer. The most likely problem this will cause for you are some constant or expression out of bounds errors and warnings that you will need to address by typecasting.

Changed Interfaces

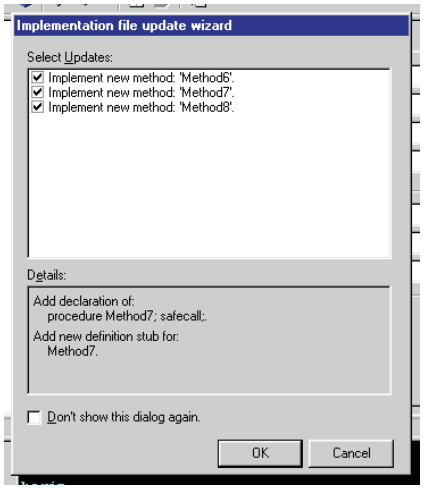
There were also some changes to a few interfaces defined in the ActiveX unit that may cause your application not to compile any longer. In particular, the definition of the `IEnumVARIANT` interface has change rather significantly. Most notable in this regard is parameter two of the next method, which has changed from an untyped out parameter to an `OleVariant` passed by reference. Ironically, the Delphi 4 definition is the more accurate, as the SDK documentation specifies that parameter two is a pointer to an array of `VARIANT` structs. If you are implementing this interface, trying to stuff this array into the single `OleVariant` var parameter will be a fun exercise in typecasting. If you'd rather not deal with the problem, you can instead implement the `IEnumVARIANT_D4` interface, which maintains the old definition.

The `ReadMultiple` and `WriteMultiple` method of the `IPropertyStorage` interface have also changed, so watch out for these. Finally, the `IStream` interface no longer descends directly from `IUnknown`, but now descends from an intermediate ancestor called `ISequentialStream`.

Summary

That about sizes it up for new COM development features found in Delphi 5. While Delphi 5 doesn't make the large leaps forward in progress that Delphi 3 and 4 seemed to make in the COM arena, there is enough there to continue to be useful and make the game interesting. I hope you have the opportunity to toy with, work with, and enjoy these features sometime soon, and we will likely be discussing finer points of these features in more detail over the coming months. Until then, happy COM.

Steve Teixeira is the VP of Software Development at DeVries Data Systems, a Silicon Valley-based consulting firm. You can reach Steve at steve@dvdata.com.



➤ *Figure 4*