New COM Features In Delphi 4: An Overview

by Steve Teixeira

Delphi 3 brought an unbelievable number of new COM features to the party including language support for interfaces and IUnknown, one step ActiveX control creation, ActiveForms and web deployment. In one version Delphi went from nominal support for COM technologies to perhaps the best COM development tool around *[Biased? Steve? Nah! Ed]*. With such an awesome opening act, the question is: what could we do for an encore? The answer: Delphi 4. Delphi 4 extends Delphi's reach into the world of COM with new features such as language enhancements, threading support, MTS integration and framework improvements designed to make your life easier.

Language Enhancements

Only one new COM-related language enhancement was added for this release, but what a powerful feature it is. New to Delphi 4 is the implements directive, which enables you to delegate the implementation of interface methods to another class or interface. This technique is sometimes called *implementation by delegation*. Implements is used as the last directive on a property of class or interface type as shown in Listing 1.

The use of implements in the Listing 1 example instructs the compiler to look to the Foo property for the methods that implement the IFoo interface. The type of the property must be a class that contains IFoo methods or an interface of type IFoo or a descendant of IFoo. You can also provide a comma-delimited list of interfaces following the implements directive, in which case the type of the property must contain the methods to implement the multiple interfaces.

The implements directive buys you two key advantages in your development. First, it allows you to perform aggregation in a no-hassle manner (if you're not familiar with aggregation, it's a COM concept pertaining to the combination of multiple classes so that they appear as one). Second, it allows you to defer the consumption of resources necessary to implement an interface until it is absolutely necessary. For example, say there was an interface whose implementation requires allocation of a 1Mb bitmap, but clients seldom require that interface. You probably wouldn't want to implement that interface all the time 'just in case' because that would be a waste of resources. Using implements, you could create the class to implement the interface on demand in the property read method.

One final twist on implements is that the inner class does not need to implement all of the methods of the specified interface(s). You can split the implementation of the methods between the inner and outer classes as shown in the Listing 2 code.

Threading Model Support

Every COM object operates in a particular threading model that dictates how an object can be manipulated in a multithreaded environment. When a COM server is registered, each of the COM objects contained in that server should register the threading model they support. For COM objects written in Delphi, the threading model chosen in the Automation, ActiveX control, or COM object wizards dictates how a control is registered. The common COM threading models include the following.

In *single* threading the entire COM server runs on a single thread.

Second up is *apartment* threading, also known as Single Threaded Apartment (STA). Each COM

➤ *Listing 1*

```
type
  TSomeClass = class(TInterfacedObject, IFoo)
    // stuff
    function GetFoo: TFoo;
    property Foo: TFoo read GetFoo imlements IFoo;
    // stuff
  end;
type
  TSomeClass = class(TInterfacedObject, IFoo)
    // stuff
    function GetFoo: TFoo;
    property Foo: TFoo read GetFoo imlements IFoo;
    // stuff
  end;
```

➤ *Listing 2*

```
type
  IBar = interface
    ['{87218660-10D7-11D2-AE5C-00A024E3867F}']
    procedure M1;
    procedure M2;
  end;
  TBar = class
    procedure M1;  //  TFoo.IBar.M1 is here!
  end;
  TFoo = class(TInterfacedObject, IBar)
  private
    FBar: TBar;
  protected
    procedure M2;  // TFoo.IBar.M2 is here!
    property Bar: TBar read FBar implements IBar;
  end;
```

object executes within the context of its own thread, and multiple instances of the same type of COM object can execute within separate threads. Because of this, thread synchronization objects must protect data that is shared between object instances (such as global variables) when appropriate.

In *free* threading, also known as Multithreaded Apartment (MTA), a client can call a method of an object on any thread at any time. This means that the COM object must protect even its own instance data from simultaneous access by multiple threads.

Finally, the *both* threading model specifies that a COM object supports both apartment and free threading.

Keep in mind that merely selecting the desired threading model in the wizard doesn't guarantee that your COM object will be safe for that threading model. You must write the code to ensure that your COM servers operate correctly for the threading model you wish to support. This most often includes using thread synchronization objects to protect access to global or instance data in your COM objects.

To aid you in your quest for thread safety, the `ComObj`, `ComServ`, and `Classes` units have been retrofitted to function properly in a multi-threaded environment. For example, the COM class creation, destruction, and reference counting logic, as well as the VCL streaming mechanism, are now all thread safe.

COM Object Wizard

Delphi 4 now provides a COM object wizard to assist you in the creation of simple (that is, non-Automation) COM objects. Found on the ActiveX page of the `File | New` dialog, this is a relatively straightforward wizard which allows you to specify the name, implemented interfaces, threading model and description for a COM object. Once these items are specified and you select OK, the wizard will generate the proper source code. This can be a big time-saver for creating plain old COM servers like shell extensions or specialized in-process servers.

The COM object wizard also provides the option of whether or not to include a type library with your COM object. If you choose to include a type library, then you design your interfaces interactively in the type library editor in a manner similar to Automation objects. Without a type library, you simply tell the wizard what predefined interfaces you intend to implement and then implement the methods of those interfaces in the generated code.

MTS

Delphi 4 Client/Server Suite contains new wizards which enable you to create special Automation objects and Remote DataModules that are designed to function efficiently within the Microsoft Transaction Server (MTS) environment. Briefly, MTS is a host environment for ActiveX servers that enables you to build distributed components that can take advantage of services such as transaction processing, security, and resource sharing.

The new MTS wizards create objects which implement the `IObjectControl` interface, which is required of MTS objects wishing to support context-specific activation or deactivation activity or participate in object pooling. Also, Delphi MTS objects maintain a property called `ObjectContext` which holds the `IObjectContext` pointer that defines the MTS object's current context.

MTS is an enormous topic worthy of many articles on its own, so I won't go into the nitty-gritty details at this time. Watch this spot in future issues for articles on MTS.

Automation Object Event Support

The Delphi 4 Automation object wizard now provides an option to generate the code necessary to support an outgoing events interface on the Automation object. Having the wizard generate event support code for you is only half the battle, though. You must also define the events in the type library editor and write the code necessary to fire the events at the appropriate time. But it is generally all straightforward work.

To demonstrate, you can create an Automation object which fires an event back to the clients when a method is called in several simple steps. Firstly, select `Automation` object from the ActiveX page of the `File | New` dialog. Secondly, give the object a name in the wizard (say, `TestObject`), and check the `Generate Event Support Code` checkbox. Click `OK`. The type library editor will now show two interfaces: one called `ITestObject`, which is the default dispatch interface for your Automation object, and one called `ITestObjectEvents`, which is the outgoing events interface for the object. Thirdly, add a method with no parameters to the `ITestObject` interface called `Foo`. Next, add a method with no parameters to the `ITestObjectEvents` interface called `OnFoo`. Finally, refresh the type library, and, in the implementation for the `TTestObject.Foo` method, add the line of code necessary to fire the `OnFoo` event:

```
procedure TTestObject.Foo;
begin
  // Foo implementation code goes here
  if FEvents <> nil then FEvents.OnFoo;  // Fire event
end;
```

➤ *Listing 3*

That's all there is to it!

Delphi ActiveX Framework Enhancements

Finally, a couple of minor improvements were made to the Delphi ActiveX control framework (DAX) that make it a little easier to write controls for use on the world wide web. In particular, the `IPersistPropertyBag` interface has been implemented to enable you to set properties for your ActiveX controls using plain text in your HTML pages. Listing 4 shows some

```
<OBJECT
  classid="clsid:FFCB1548-1111-11D2-AAB6-00C04FA370E8"
  codebase="file://c:/temp/ButtonXControl1.ocx"#version=1,0,0,0
  width=350
  height=250
  align=center
  hspace=0
  vspace=0
>
<param name = "Caption" value = "Hello world">
</OBJECT>
```

➤ *Listing 4*

sample HTML code that sets `Caption` property for an ActiveX control written in Delphi 4.

In addition to property bag support, Delphi 4 ActiveX controls now implement the `IObjectSafety` interface in order to indicate to web browsers that a web deployed ActiveX control is safe to be embedded in an HTML page.

One important note on this is that the default implementation of this interface always claims that your control is safe, even if you implement your control to do things which are considered unsafe, such as writing to the registry. It's up to you to implement this interface differently if your control doesn't meet the criterion for a safe control as defined in the ActiveX spec.

Last but not least, DAX now provides a partial implementation for the `IDataObject` interface so that Delphi 4 ActiveX controls function correctly in MS Office products such as Word, Excel, and PowerPoint. This is necessary because, unlike most containers, these Office products do their painting of controls via metafiles in the `IDataObject.GetData` method rather than through the traditional `IViewObject.Draw` method.

Conclusion

That about covers all the new COM features found in Inprise's latest release of Borland Delphi. This was kind of the 30,000 foot high-level overview that gave you a taste of everything. In future articles, I will drill down into some of the individual topics covered in this article to provide more in-depth insight and techniques. Until then, enjoy your new COM development tool!

Steve Teixeira is an R&D Engineer at Inprise Corporation where he works on the Borland Delphi and Borland C++Builder products. He is also the co-author of *Delphi 4 Developer's Guide* from SAMS Publishing.

Got a Delphi COM question that you think would make a good article? Email Steve at:
steixeira@inprise.com

For Delphi News

Check the

Developers Review

website at

www.itecuk.com