COM Corner:
ActiveX Documents, Part 1

by Steve Teixeira

It's funny, but OLE developers used to be concerned with linking and embedding things. After all, OLE used to stand for Object Linking and Embedding. Linking and embedding isn't something we think about very often these days. In fact, we tend to take these things for granted, thanks to the pervasiveness of ActiveX controls. Heck, we don't even call ourselves OLE developers anymore, we're COM developers now. With this instalment of *COM Corner*, we're going to go back to our roots and have some fun with embedding as we learn about creating an ActiveX Document server in Delphi 4.

ActiveX Documents

ActiveX Documents are the logical extension of OLE 2.0 Document Objects. You may recall that OLE 2.0 enables a document server to be embedded in a client application. As a part of the embedding process, the document server can take control of some or all of the client area of the client and optionally merge its own menus and toolbars with those of the client. The classic example of this is the proverbial Excel spreadsheet embedded within a Word document. ActiveX Documents extended this concept by formalizing the means by which servers and clients communicate with one another and providing the ability for ActiveX Documents to be served over the web using Internet Explorer as the client. This article will provide you with a technical description of ActiveX Documents and a basic framework for creating ActiveX Document servers based on the Delphi ActiveX framework, usually known as *DAX*.

Put plainly, an ActiveX Document server is just an Automation server than supports a number of specific interfaces. Table 1 lists the interfaces that a server must implement in order to be an ActiveX document server.

Abstractly speaking, the ActiveX Document architecture is made up of frames, documents and views. The frame is the 'socket' provided by the container application in which the ActiveX Document resides. The document is the server data being manipulated in the container. The item represents a specific view of the document data. If you've done any MFC programming in the past, you might recognize the ActiveX Document architecture as being similar to the MFC document/view architecture. Tying the abstract architecture to the COM interfaces mentioned above, the `IOleDocument` interface represents the document, while the `IOleDocumentView` interface represents a view on a particular document. The frame and associated container logic is represented by various interfaces on the client side. This includes, in particular, `IOleInPlaceFrame`, `IOleInPlaceSite`, and `IOleContainer`.

Delphi Implementation

Implementing an ActiveX Document from scratch would be a pretty tall order, considering the number of required interfaces and the complexity of the implementation of those interfaces. Being a lazy programmer at heart, I really wasn't eager to dive in and start implementing a bunch of large interfaces. And if you're an ActiveX propeller-head like me, you may have noticed that the interface requirements for ActiveX Documents are very similar to those of

➤ *Table 1: ActiveX Document interfaces.*

Interface	Description
IPersistStorage	Enables the use of OLE structured storage as a persistence mechanism for server.
IPersistFile	Enables the use of OLE compound files as a persistence mechanism for server.
IOleObject	The principal interface by which the embedded object communicates with the container.
IDataObject	Defines data transfer capabilities and data format.
IOleInPlaceObject	Manages the activation and deactivation of in-place objects, and determines how much of the in-place object should be visible.
IOleInPlaceActiveObject	Provides the communications channel between the in-place object and the client application that contains the embedded object.
IOleDocument	Provides information to containers on ActiveX Document's ability to create views of its data.
IOleDocumentView	Enables container to communicate with various views of ActiveX Document object.
IOleCommandTarget	OPTIONAL. Enables objects and containers to dispatch commands to one another.
IPrint	OPTIONAL. Enables ActiveX Documents to support programmatic printing.

ActiveX controls. In fact, all of the required interfaces for ActiveX Documents are implemented by DAX's TActiveXControl class, save for the IOleDocument and IOleDocumentView classes. Therefore, as you will see, I was able to create a new class, which I call TActiveXDocument, that descends from TActiveXControl and encapsulates an ActiveX Document. Listing 1 shows the AxDocs unit, which contains the TActiveXDocument class and its corresponding class factory, TActiveXDocumentFactory.

Before explaining the meat of the TActiveXDocument class, I feel compelled to apologize for the nasty little hack I use to obtain access to the private FOleInPlaceSite pointer found in the ancestor TActiveXControl class. Since the designer of TActiveXControl never intended it to be used as a base class for ActiveX Documents, he or she didn't know the GetInPlaceSite and SetInPlaceSite methods of IOleDocumentView would be implemented on a descendant of this class and so chose to keep the FOleInPlaceSite field private. I get at this private data by determining the instance size of immediate ancestor of TActiveXControl and adding the correct number of bytes to the offset of the FOleInPlaceSite field. A neat trick, yes, but not exactly exemplary object oriented technique.

Another point of interest in the TActiveXDocument class is the ObjQueryInterface method, which prevents an IOleLink pointer from being returned to the caller. If a container finds that an object supports IOleLink, it will assume the ActiveX Document is linked rather than embedded.

The IOleDocument implementation for TActiveXDocument is rather straightforward because this is a simple ActiveX Document that supports only one view. The IOleDocumentView implementation is also fairly simple, as you can see by the small amount of code used to implement each method. Most notable are the Show and UIActivate methods, which make a call to the InPlaceActivate helper function found in TActiveXControl that handles the complexities of in-place activation and UI-activation of an OLE object.

I have created a special TActiveXDocumentFactory class factory object, mostly to handle the extra registry entries needed for ActiveX Documents. By overriding the factory's Update Registry method, I can do any special registry processing for the server that will occur when the server's DllRegisterServer and DllUnregisterServer exports are called.

Armed with this base class, I can now implement an example ActiveX Document. Thanks to the fact that TActiveXDocument and its ancestors do the majority of the work for me, the implementation for my example ActiveX Document is relatively small, and it is shown in Listing 2.

I actually created this unit by using the Automation Wizard to create a new Automation object and massaging the code by hand a little bit. In particular, I changed the ancestor of TDelphiAxDoc from TAutoObject to TActiveXDocument, and modified the code that creates the class factory so that it is appropriate for TActiveXDocumentFactory. This example simply uses a TMemo as the ActiveX Document.

After the application is compiled and registered, it is ready to test. To test this server, I will embed it in a Visual Basic 6 application using VB's OLE control. After dropping an OLE control on a VB form, the Insert Object dialog is invoked as shown in Figure 1.

After selecting Delphi ActiveX Document Object in the Insert Object dialog, the object is inserted in the OLE control, and can be manipulated as shown in Figure 2. Note that to run the precompiled demo program on the disk you will need MSVBVM60.DLL, which you can get from http://pcworld/fileworld/ along with much other useful stuff.

Summary

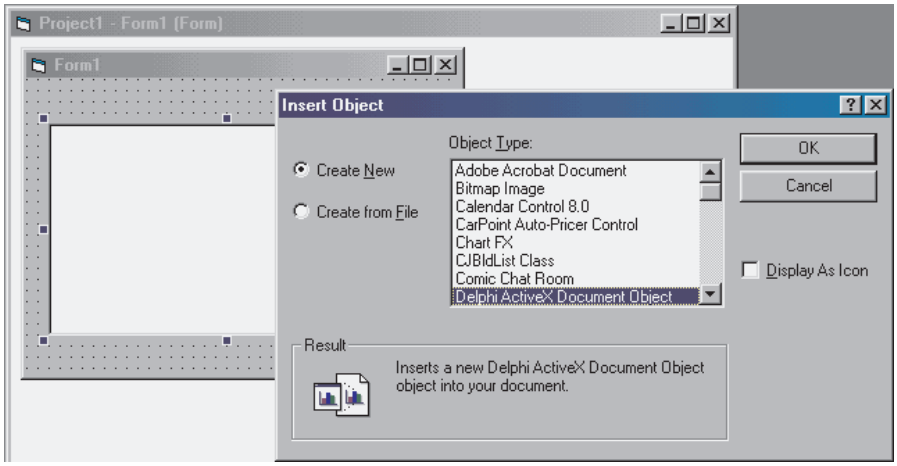
This article introduced you to the basics of ActiveX Documents and a server implementation in Delphi. In next month's issue, I will extend this example even further by demonstrating advanced ActiveX Document concepts such as file

➤ *Facing page: Listing 1*

```
unit Main;
interface
uses
  ComObj, ActiveX, AxDocs, DAXDoc_TLB;
type
  TDelphiAxDoc = class(TActiveXDocument, IDelphiAxDoc)
  protected
  end;
implementation
uses
  ComServ, StdCtrls;
initialization
  TActiveXDocumentFactory.Create(ComServer, TDelphiAxDoc, TMemo,
    Class_DelphiAxDoc, 0, 131473, tmApartment);
end.
```

➤ *Above: Listing 2*

➤ *Below: Figure 1*

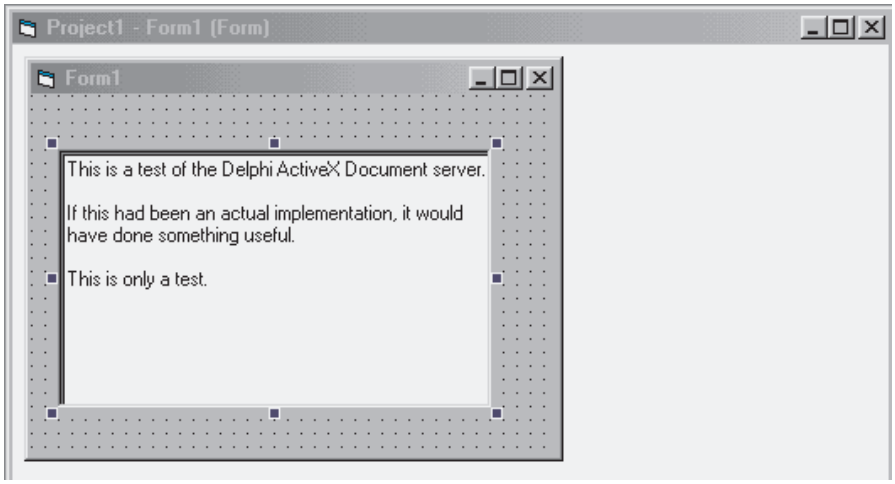


```
unit AxDocs;
interface
uses Windows, ComObj, ActiveX, AxCtrls, Controls;
type
  TActiveXDocument = class(TActiveXControl, IOleDocument,
    IOleDocumentView)
  private
    function GetAncestorValueByField(FieldNum: Cardinal):
      Cardinal;
    procedure SetAncestorValueByField(FieldNum, Value:
      Cardinal);
    function GetOleInPlaceSite: IOleInPlaceSite;
    procedure SetOleInPlaceSite(const Value:
      IOleInPlaceSite);
  protected
    function CreateView(Site: IOleInPlaceSite; Stream:
      IStream; rsrvd: DWORD;
      out View: IOleDocumentView):HResult; stdcall;
    function GetDocMiscStatus(var Status: DWORD):HResult;
      stdcall;
    function EnumViews(out Enum: IEnumOleDocumentViews;
      out View: IOleDocumentView):HResult; stdcall;
    function SetInPlaceSite(Site: IOleInPlaceSite):HResult;
      stdcall;
    function GetInPlaceSite(
      out Site: IOleInPlaceSite):HResult; stdcall;
    function GetDocument(out P: IUnknown):HResult; stdcall;
    function SetRect(const View: TRECT):HResult; stdcall;
    function GetRect(var View: TRECT):HResult; stdcall;
    function SetRectComplex(const View, HScroll, VScroll,
      SizeBox):HResult; stdcall;
    function Show(fShow: BOOL):HResult; stdcall;
    function UIActivate(fUIActivate: BOOL):HResult; stdcall;
    function Open:HResult; stdcall;
    function CloseView(dwReserved: DWORD):HResult; stdcall;
    function SaveViewState(pstm: IStream):HResult; stdcall;
    function ApplyViewState(pstm: IStream):HResult; stdcall;
    function Clone(NewSite: IOleInPlaceSite; out NewView:
      IOleDocumentView):HResult; stdcall;
  public
    function ObjQueryInterface(const IID: TGUID; out Obj):
      HResult; override;
    property OleInPlaceSite: IOleInPlaceSite
      read GetOleInPlaceSite write SetOleInPlaceSite;
  end;
  TActiveXDocClass = class of TActiveXDocument;
  TActiveXDocumentFactory = class(TActiveXControlFactory)
    constructor Create(ComServer: TComServerObject;
      ActiveXDocClass: TActiveXDocClass; WinControlClass:
      TWinControlClass; const ClassID: TGUID;
      ToolboxBitmapID, MiscStatus: Integer;
      ThreadingModel: TThreadingModel);
    procedure UpdateRegistry(Register: Boolean); override;
  end;
implementation
uses ComServ;
function TActiveXDocument.ObjQueryInterface(
  const IID: TGUID; out Obj): HResult;
begin
  // Must stub out IOleLink, or container will assume this
  // is a linked object rather than an embedded object.
  if IsEqualGuid(IID, IOleLink) then
    Result := E_NOINTERFACE
  else
    Result := inherited ObjQueryInterface(IID, Obj);
end;
function TActiveXDocument.GetOleInPlaceSite :
  IOleInPlaceSite;
begin
  // Work around fact that FOleInPlaceSite is private in
  // TActiveXControl. only guaranteed to work in Delphi 4
  Result := IOleInPlaceSite(GetAncestorValueByField(9));
end;
procedure TActiveXDocument.SetOleInPlaceSite(
  const Value: IOleInPlaceSite);
begin
  // Work around fact that FOleInPlaceSite is private...
  SetAncestorValueByField(9, Cardinal(Value));
end;
function TActiveXDocument.GetAncestorValueByField(
  FieldNum: Cardinal): Cardinal;
var ParentInstanceSize, Ofs: Cardinal;
begin
  // Nasty hack: returns value of a field in ancestor class,
  // assuming given field and all prior fields are 4 bytes
  ParentInstanceSize :=
    ClassParent.ClassParent.InstanceSize;
  Ofs := ParentInstanceSize+((FieldNum-1)*4);
  asm
    mov eax, Self
    add eax, Ofs
    mov eax, dword ptr [eax]
    mov @Result, eax
  end;
end;
                {** LISTING CONTINUES ON NEXT PAGE...  **}
```

```
procedure TActiveXDocument.SetAncestorValueByField(FieldNum,
  Value: Cardinal);
var ParentInstanceSize, Ofs: Cardinal;
begin
  // Nasty hack... (as before)
  ParentInstanceSize :=
    ClassParent.ClassParent.InstanceSize;
  Ofs := ParentInstanceSize + ((FieldNum - 1) * 4);
  asm
    mov eax, Self
    add eax, Ofs
    mov ecx, Value
    mov dword ptr [eax], ecx
  end;
end;
function TActiveXDocument.CreateView(Site: IOleInPlaceSite;
  Stream: IStream; rsrvd: DWORD;
  out View: IOleDocumentView): HResult;
var OleDocView: IOleDocumentView;
begin
  Result := S_OK;
  try
    if View = nil then begin
      Result := E_POINTER;
      Exit;
    end;
    OleDocView := Self as IOleDocumentView;
    if (OleInPlaceSite=nil) or (OleDocView=nil) then begin
      Result := E_FAIL;
      Exit;
    end;
    if Site <> nil then
      OleDocView.SetInPlaceSite(Site);
    if Stream <> nil then
      OleDocView.ApplyViewState(Stream);
    View := OleDocView;
  except
    Result := E_FAIL;
  end;
end;
function TActiveXDocument.EnumViews(out Enum:
  IEnumOleDocumentViews; out View: IOleDocumentView):
  HResult;
begin
  Result := S_OK;
  try
    View := Self as IOleDocumentView;
  except
    Result := E_FAIL;
  end;
end;
function TActiveXDocument.GetDocMiscStatus(
  var Status: DWORD): HResult;
begin
  Status := 8 {DOCMISC_NOFILESUPPORT};
  Result := S_OK;
end;
function TActiveXDocument.ApplyViewState(pstm: IStream):
  HResult;
begin
  Result := E_NOTIMPL;
end;
function TActiveXDocument.Clone(NewSite: IOleInPlaceSite;
  out NewView: IOleDocumentView): HResult;
begin
  Result := E_NOTIMPL;
end;
function TActiveXDocument.CloseView(dwReserved: DWORD):
  HResult;
begin
  Result := S_OK;
  try
    Show(False);
    SetInPlaceSite(nil);
  except
    Result := E_UNEXPECTED;
  end;
end;
function TActiveXDocument.GetDocument(out P: IUnknown):
  HResult;
begin
  Result := S_OK;
  try
    P := Self as IUnknown;
  except
    Result := E_FAIL;
  end;
end;
function TActiveXDocument.GetInPlaceSite(
  out Site: IOleInPlaceSite): HResult;
begin
  Result := S_OK;
  try
    Site := OleInPlaceSite;
  except
    Result := E_FAIL;
  end;
end;
function TActiveXDocument.GetRect(var View: TRECT): HResult;
begin
  Result := S_OK;
  try
    View := Control.BoundsRect;
  except
    Result := E_UNEXPECTED;
  end;
end;
function TActiveXDocument.Open: HResult;
begin
  Result := E_NOTIMPL;
end;
function TActiveXDocument.SaveViewState(pstm: IStream):
  HResult;
begin
  Result := E_NOTIMPL;
end;
function TActiveXDocument.SetInPlaceSite(
  Site: IOleInPlaceSite): HResult;
begin
  Result := S_OK;
  try
    if OleInPlaceSite <> nil then
      Result := InPlaceDeactivate;
    if Result <> S_OK then
      Exit;
    if Site <> nil then
      OleInPlaceSite := Site;
  except
    Result := E_UNEXPECTED;
  end;
end;
function TActiveXDocument.SetRect(const View: TRECT):
  HResult;
begin
  // Implement using TActiveXControl's
  // IOleInPlaceObject.SetObjectRects impl
  Result := SetObjectRects(View, View);
end;
function TActiveXDocument.SetRectComplex(const View;
  const HScroll; const VScroll; const SizeBox): HResult;
begin
  Result := E_NOTIMPL;
end;
function TActiveXDocument.Show(fShow: BOOL): HResult;
begin
  try
    if fShow then
      Result := InPlaceActivate(False)
    else begin
      Result := UIActivate(False);
      Control.Visible := False;
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;
function TActiveXDocument.UIActivate(fUIActivate: BOOL):
  HResult;
begin
  Result := S_OK;
  try
    if FUIActivate then begin
      if OleInPlaceSite <> nil then
        InPlaceActivate(True)
      else
        Result := E_UNEXPECTED;
    end else
      UIDeactivate;
  except
    Result := E_UNEXPECTED;
  end;
end;
constructor TActiveXDocumentFactory.Create(ComServer:
  TComServerObject; ActiveXDocClass: TActiveXDocClass;
  WinControlClass: TWinControlClass; const ClassID: TGUID;
  ToolboxBitmapID, MiscStatus: Integer; ThreadingModel:
  TThreadingModel);
begin
  inherited Create(ComServer, ActiveXDocClass,
    WinControlClass, ClassId, ToolboxBitmapID, '',
    MiscStatus, ThreadingModel);
end;
procedure TActiveXDocumentFactory.UpdateRegistry(
  Register: Boolean);
var ClassKey: string;
begin
  ClassKey := 'CLSID\' + GUIDToString(ClassID) + '\';
  if Register then begin
    inherited UpdateRegistry(Register);
    CreateRegKey(ClassKey + 'DocObject', '', '8');
    CreateRegKey(ClassKey + 'Programmable', '', '');
    CreateRegKey(ClassKey + 'Insertable', '', '');
  end else begin
    DeleteRegKey('DocObject');
    DeleteRegKey('Programmable');
    DeleteRegKey('Insertable');
    inherited UpdateRegistry(Register);
  end;
end;
end.
```

associations, menu and toolbar merging, and web delivery of ActiveX Documents. Until then, I hope you enjoy this rediscovery of old-school embedding.

Steve Teixeira is the Director of Software Development at DeVries Data Systems, a software consulting and training firm. Send your comments, questions, or article ideas to Steve by email at steve@dvdata.com



➤ *Right: Figure 2*