Chapter Eleven
In-Process Object Handlers and Servers

Follow this silver watch with your eyes. Back and forth it moves. Moving...moving...you are feeling sleepy. You are feeling hungry. When I count to three you will forget about any lengthy chapter introduction and awake starving for information on these handler things. One. Two. Two and a half...

The Structure of In-Process Modules

Style and structure are the essence of a book; great ideas are hogwash.[1]

Vladimir Nabokov (1899-1977)
Russian-American novelist

In all truth, both in-process servers and object handlers are structurally identical. The idea that the two are somehow mystically different *is* hogwash. Both are DLLs that export DllGetClassObject and DllCanUnloadNow like any other DLL component as described in Chapter 4. Both must implement a class factory and keep track of how many objects it has created such that it can provide the proper unloading mechanism. Both implement the same interfaces on their objects, IOleObject, IDataObject, IPersistStorage, IViewObject, and IOleCache, such that they appear as an embedded object from a container's point of view as shown in Figure 11-1. The container remains safely ignorant of who or what implements the embedded object.

Figure 11-1: The structure of any in-process embedded object component, be it a handler, the default handler, or an in-process server.

Indeed, the only thing separating embedded objects (from in-process server and object handlers) from those of a more generic component object (in some other DLL) is what interfaces the object supports. To be a component object you can choose to implement whatever interfaces you want, because you define how that object can be used. For embedded objects, the OLE 2.0 specifications for compound documents dictate how the object will be used, and thus you don't have total freedom for choosing your interfaces.

So just what is the difference between an object handler and an in-process server? It lies in the completeness of implementation of their respective objects. It's the same as the difference between the lowest-price economy car and the most expensive European luxury sports card–both have the same basic car structure: a body, a chassis, four wheels, a steering column, an engine and transmission, and some seats inside. In order to be the most basic lowest-priced car on the market you have to be extraordinarily careful about what you put in this car such that no part is any more expensive than necessary. Likewise in order to be the most luxurious you can spare no expense whatsoever. It is always a fact that there is a better car than the economy model and that there is no better or more complete car than the top luxury automobile.

An object handler is like the economy car: as small as possible, as inexpensive as possible (to load into memory, that is), and contains the minimal amount of features possible to qualify as a car. In other words, the handler implements as little as possible to get into the market, only overriding specific

1The Concise Columbia Dictionary of Quotations is licensed from Columbia University Press Copyright © 1987, 1989, 1990 by Columbia University Press. Third printing with emendations 1990. All rights reserved.

member functions of specific interfaces and delegating all others to the default handler (which may ultimately end up in a local server). An in-process server is like the luxury model where there is no higher power to which to delegate requests on its interfaces: the in-process handler implements is all.

I can tell by the look on your face that you are starting to feel the same way about these object DLLs as you would if you could only choose between a cheap $4500 car and the $1,000,000 car. Relax, you have more choices because we *do* want to sell you a car at a little over what you think you can afford, like any good dealer. While the basic and most minimal handler is at one end of the spectrum and the complete "buck-stops-here" in-process server is at the other, there are many many choices in between. Like there are many more models of cars to choose from between the two extremes there are many choices for DLL object implementations. Every small feature you add to the minimal handler brings you one step towards luxury, and somewhere in the middle the picture gets very fuzzy as to whether you are truly a handler or an in-process server as shown in Figure 11-2. Anywhere between the two extremes, you are still a car (off the low end you cease to be useful as a car; off the high end you are dabbling in unauthorized experimental sciences).

Figure 11-2: Between the minimalist handler and the complete in-process server lie many possibilities for handler/servers, all of which still have the same structure.

There is one more point to all this which I call the luxury tax. At some point in car prices the tax kicks in: suddenly you're paying and extra 10%. This is not a gradual change but rather a very sudden one. In the context of handlers and in-process servers, there comes a time when a handler is overqualified to be a handler and must then be called an in-process server. The dividing line is whether or not the handler still depends on the existence of a Local Server. In other words, when the handler no longer delegates any features or calls to a higher model (that is the default handler) that would require a Local Server, that handler is now a qualified in-process server.

That is the difference: a handler is designed to work in conjunction with a local server whereas an in-process server operations exclusively of a local server. Handlers are designed to be smaller on disk and therefore faster to load and only provide a few basic features. The handler can cut corners all it wants, and if a container wants something more then it will require the local server. An in-process server is meant to be larger and to completely implement the object such that a local server is never necessary and a container can get from the server whatever it wants.

This is not to say that you cannot take advantage of the default handler even from an in-process server. In both the server and handler cases we will want to use the services of the default handler, primarily those features that implement caching and those that will implement member functions of specific interfaces based on information we put in the registration database. The whole trick is to know exactly when the default handler will try to launch a Local Server to fulfill a request which is the topic of "Delegating to the Default Handler" below. But before we get into the implementation details we need to look at the pros and cons of DLL-based objects.

Three! Wait, don't tell me you just slept through all that![1]

Why Use a Handler?

There are two main reasons for implementing an object handler to work with your local server: speed and portability. First, and object handler can generally satisfy most requests a container might make on an object such as drawing an object on a specific device or making a copy of the object in another IStorage. Object handlers may also be capable of reloading a linked file and providing an updated

1If you didn't get what Three was for, reread the opening paragraph of this chapter.

presentation to the container. Object handlers do not, however, provide any sort of editing facilities for the object itself: it might be able to play a sound, for example, but cannot provide the user interface and functionality to change the sound. Therefore the handler has much less code than might be present in a server, because the most code in an application is usually due to editing features. Imagine how small a word-processor would be if all it could do was read and display text, but never edit it! That's the idea of a handler, so the speed advantage comes from the fact that the handler is a small DLL optimized to perform specific actions such as drawing and rending presentations. Because it's a DLL it loads much faster into a container's process space than launching another EXE, and because there's no LRPC involved all calls to the handler are much faster than to a local server. In addition, since a handler (at minimum) knows how to draw the object from its native data, there is no reason to cache a metafile or bitmap for the object, thus making the container's compound files smaller (saving at least 2K per object). In general, therefore, the existence of a handler greatly improves the performance of an object and its container.

The second benefit is a little less tangible. What I mean by portability is explained through a scenario where one user on one machine has created an embedded object of class X using the local server for class X and has saved that object in some container's file. In this case there will be a cached screen presentation in the object's IStorage along with its native data. Let's also say that there is a cached presentation for a PostScript printer as well.

Now this document is sent to other users who do not have the local server for class X on their machines because they have no use to edit the object, only to view and print it. Since there are cached presentations for this object in the container's file, these users can open those files and view or print the object. However, this assumes that the cached presentations are compatible with the output devices these users want to send them to. If they have only 16-color displays and the objects depend on 256 colors, the screen display can be pretty ugly. If their printer is only a dot-matrix, then the only possible presentation to send to such a printer is the one for the screen since we know that PostScript data sent to a dot-matrix printer does not get you anywhere. In either case, the output quality is poor.

There are two solutions to this. First we could give these secondary users copies of the local server, but why would they want to pay for extra copies just to print? You wouldn't want to let them freely copy your server just for this purpose either. So that solution is just a bad idea all around. The second is much better: provide an object handler that can render a presentation for an object's data reasonably well on any display or printer and allow your customers to freely distribute this handler. Since it's a small piece of code that can only be used to display and print objects, it's not like you are letting people freely copy your full application. Furthermore, since a handler is generally very small, it lends very well to being put on a floppy disk along with a document such that the receiver of the document has the necessary tools for optimized output. This big win for you is that your objects always show up nicely regardless of where they're created and where they're displayed. That's the portability.

The bottom line is that object handlers improve object performance and can be confidently licensed for free distribution to optimize output wherever the object happens to travel. Since handlers do not include editing capabilities (which would qualify it for luxury tax), people still have incentive to purchase your full server.

Just as an aside, you can include some free advertising in an object handler that would encourage people to buy your full product much like shareware includes annoying messages that continually suggest registration. A handler will typically just pass IOleObject::DoVerb to the default handler which will try to launch the local server. If that fails, it's a good time to pop up a message that says "Since you'd really like to edit this object, why don't you call this toll-free number right now and have your credit card handy so we can get you this product in your hands by tomorrow morning for the incredibly low-low price of $149.95!"

Why Use an In-Process Server?

An in-process server first of all provides all the same benefits as an in-process handler except loading time will generally be slower since the DLL will be larger. Nonetheless, there is no LRPC and no need for cached presentations, so all the other speed benefits still apply. You may also want to license an in-process server DLL for redistribution if you see fit, and you can, of course, have it pop up registration or purchase notices as much as you want.

In all, the primary benefit of an in-process server is that all the speed and portability advantages of a handler and all the editing capabilities of a local server are stored under one roof, that is, inside a single disk entity. Truly one-stop shopping for and embedded object. Such a DLL is a great choice for control-like objects using in-place activation. They are also great choices for objects that can be editing inside a modal dialog-box type of user interface[1] because they will look like part of the container application instead of a separate application. In any case where your user interface for editing is simple, consider an in-process server.

Why Not?

The biggest reason to avoid DLLs is summed up in one word: limited interoperability. OK, so I can't count. Nevertheless, when you choose to write any part of an OLE 2.0 embedded object in a DLL you are first of all limiting yourself to use only from OLE 2.0 containers (since an OLE 1.0 container can't make any sense of an OLE 2.0 DLL) and are limiting yourself to only work with containers that were written for the same 16- or 32-bit environment as yourself. That is, if you write a 16-bit handler (native for Windows 3.1) you will not work with a 32-bit application under Windows NT or under Win32s on Windows 3.1. The converse is true as well where only 32-bit DLLs work with 32-bit containers. The only solutions to these problems for the time being is to write multiple versions of your DLLs: one for OLE 1.0,[2] one for OLE 2.0, and both in 16- and 32-bit versions. Yep, that's a pain, but that's part of the price we must still pay as the operating systems evolve.

The other issues that may put DLLs out of your reach are various technical implementation issues. DLLs, since they have no message loop themselves, have a serious problem handling things like keyboard accelerators or MDI interfaces that would normally require changes to a message loop. Overall, there are simply a number of things that you cannot do from a DLL, and if you must have one of them then a DLL is not for you. In implementing the Polyline example for this chapter I ran into such problems. I originally planned to make the Polyline In-Process Server look pretty much like Schmoo when it opened an object for editing, including menus and so forth, but the lack of accelerators meant menus were only marginally useful. So I was forced to come up with a different user interface altogether based on a dialog box. While this works well, it is different, and that difference may be reason enough for you to avoid an in-process server yourself.

The other technical issue of an in-process server specifically (but not a handler) is that since there is nothing that can ever run stand-alone (like a local server EXE can) there is no possibility to provide linked objects. You cannot run a DLL by itself, so how do you create files, especially when the embedded object user interface we saw in Chapter 10 eliminates most vestiges of "file" from the server? If your data is potentially large, then an in-process server is not be your best choice–users may balk at their container files growing outrageously large because you don't let them link.

Delegating to the Default Handler

We programmers generally do not like to make extra work for ourselves–if the code has already been

1As examples, two OLE 1.0 servers from Microsoft, Note-It and Word-Art appeared as model dialog boxes in the container application–they looked to be part of the container and not a separate application.

2See the OLE 1.0 Programmer's Reference from Microsoft Press.

written somewhere then use it. I'm not saying we're lazy, just that we like to be as resourceful as possible to find reusable code. That's why a language like C++ was invented in the first place and why OLE 2.0 has the code-reuse mechanism of aggregation.

In writing handlers and in-process servers we will aggregate on the default handler in order to reuse quite a lot of its functionality, just like Patron has been freeloading its presentation display and caching functionality for a number of chapters now. Aggregation in our case here will mean calling the function OleCreateDefaultHandler, getting a number of interface pointers to that 'default handler object' and delegating interface calls to that object whenever we have no need to implement it ourselves.

There is nothing special or tricky here. When we implemented a local server in Chapter 10 we were doing exactly the same thing, only from the other side. For example, we implemented IOleObject::EnumVerbs by returning ResultFromScode(OLE_S_USEREG) which forces the default handler to implement the function using the verbs for our CLSID in the registration database. This worked because the container always calls the handler first, and if the handler sees a running server it asks the server. If the server says "just use the registration database instead" then the default handler does just that. When we implement a handler or in-process server now, we get the call from the container first and we delegate it to the default handler object. That object in turn checks to see if a local server is running, and if not, just uses the registration database to fulfill the request. For a function like IOleObject::EnumVerbs, the default handler does the same thing when the server is running and returns OLE_S_USEREG as when the server is not running at all, and that's what we can exploit.

But then the question is "how do I know when the default handler will implement something and how to I know when it will try to launch a local server?" There is no simple answer. Instead we have to look at each member function of the interfaces we will use in the default handler. The next sections list all the function in IOleObject, IDataObject, IPersistStorage, IViewObject, and IOleCache and what the default handler (and the cache, which sits underneath the default handler) will do with them. There are only a few specific instances in which the default handler or cache will attempt to launch a local server. If you implement a handler you will want to use this functionality because you need to work with the local server to provide the full object implementation. From an in-process server, however, there is no local server to execute so you will want to make sure you avoid those calls completely.

The tables in the following sections form a guide to implementing handlers and in-process servers starting with "Implementing an Object Handler" below. As we'll see, there are some cases where we can just expose the default handler's interface as our own (like IOleCache), some cases where we delegate some or most of an interface's calls (like IOleObject), and other cases where we don't delegate them at all (as with IViewObject). IOleCache is not listed in a table below because it's an all-or-nothing: either you implement it completely without delegation or you don't implement it at all.

Except where noted, all of the actions in the following sections assume that a local server is not running. Those that will launch the server are boldfaced.

IOleObject

There are only two member functions in IOleObject that will always attempt to run the LocalServer and delegate directly to its IOleObject:: DoVerb and Update. All others either have minimal implementations or simply return an HRESULT as shown in Table 11-1. Note also that the default handler saves the information from Advise, Unadvise, EnumAdvise, SetClientSite, GetClientSite, and SetHostNames such that when and if it launches a local server it can forward that information on.

Table 11-1: Actions for the IOleObject interface on a non-running object.

Member Function	Action

Advise	Calls CreateOleAdviseHolder if one has not yet been created. In either case delegates to IOleAdviseHolder::Advise which does not run the server in any case.
Close	Returns NOERROR
DoVerb	**Runs and delegates to the server.**
EnumAdvise	Delegates to IOleAdviseHolder. Does not run the server.
EnumVerbs	Creates and enumeration based on the verb entries for the CLSID in the registration database and returns NOERROR.
GetClientSite	Returns the last IOleClientSite seen in SetClientSite and NOERROR.
GetClipboardData	Returns OLE_E_NOTRUNNING
GetExtent	Attempts to locate the requested aspect in the cache and returns the size of that presentation if available. Otherwise returns NOERROR.
GetMiscStatus	Retrieves the value from the CLSID's MiscStatus entries in the registration database and returns NOERROR.
GetMoniker	Calls IOleClientSite::GetMoniker if SetClientSite has been called with a valid IOleClientSite pointer. Otherwise returns E_UNSPEC.
GetUserClassID	Returns the CLSID passed to OleCreateDefaultHandler and NOERROR.
GetUserType	Retrieves a string from the CLSID's AuxUserType entries in the registration database and returns NOERROR.
InitFromData	Returns OLE_E_NOTRUNNING
IsUpToDate	Returns OLE_E_NOTRUNNING
SetClientSite	Saves the IOleClientSite pointer in an internal variable and returns NOERROR.
SetColorScheme	Returns OLE_E_NOTRUNNING
SetExtent	Returns OLE_E_NOTRUNNING
SetHostNames	Stores the strings in atoms and returns NOERROR.
SetMoniker	Returns NOERROR.
Unadvise	Delegates to IOleAdviseHolder. Will not run the server.
Update	**Runs and delegates to the server.**

IDataObject

The default handler's implementation of IDataObject in OLE 2.0 depends on what's available in the cache. Only the implementation of IDataObject::GetData and IDataObject::GetDataHere will run the server and delegate. What really happens is that the default handler delegates to the IDataObject of the cache which then runs the server for the two functions above. The default handler's IDataObject implementation is described in Table 11-2 where those of the cache are given in Table 11-3.

Table 11-2: Actions for the IDataObject interface on a non-running object.

Member Function	Action
DAdvise	Delegates to cache which returns OLE_E_ADVISENOTSUPPORTED
DUnadvise	Delegates to cache which returns OLE_E_NOCONNECTION
EnumDAdvise	Delegates to cache which returns OLE_E_ADVISENOTSUPPORTED
EnumFormatEtc	Creates an enumerator based on the CLSID's entries under DataFormats\ GetSet in the registration database.
GetCanonicalFormatEtc	Returns OLE_E_NOTRUNNING
GetData	**Delegates to the cache.**
GetDataHere	**Delegates to the cache**
QueryGetData	Returns OLE_E_NOTRUNNING
SetData	Returns OLE_E_NOTRUNNING

Table 11-3: Actions for the cache's IDataObject interface for a non-running object.

Member Function	Action
DAdvise	Returns OLE_E_ADVISENOTSUPPORTED
DUnadvise	Returns OLE_E_NOCONNECTION
EnumDAdvise	Returns OLE_E_ADVISENOTSUPPORTED
EnumFormatEtc	Returns E_NOTIMPL
GetCanonicalFormatEtc	Returns E_NOTIMPL
GetData	**Attempts to find the data in the cache. If not, it attempts to run the server and retrieve the data from there. Otherwise it returns OLE_E_NOTRUNNING..**
GetDataHere	**Same as for GetData.**
QueryGetData	Returns OLE_E_NOTRUNNING
SetData	Returns OLE_E_NOTRUNNING

IPersistStorage (on the Cache)

In no case will the cache run the server to implement IPersistStorage. The cache deals exclusively with cached presentations and does not affect the object's native data. If you want to use the cache to store, say, iconic presentations, you always want to call the default IPersistStorage members from your own IPersistStorage after manipulating your native data, with the exception of GetClassID which is pointless to delegate.

Table 11-4: Actions for the cache's IPersistStorage interface for a non-running object. The default handler delegates all calls to the cache.

Member Function	Action

GetClassID	Returns the CLSID passed to CreateDefaultHandler and NOERROR.
IsDirty	Returns S_OK if the cache's IAdviseSink has seen an OnViewChange, otherwise S_FALSE.
InitNew	Returns NOERROR but saves and AddRef's the IStorage.
Load	Loads information describing what presentations area available in the cache. No data is actually loaded until required in the cache's IDataObject implementation. Return value may contain an error code. This happens regardless of the running state of a server.
Save	Saves any presentations that have changed since the call to Load as well as an information block describing what is cached. Return value may contain an error code. This happens regardless of the running state of a server.
SaveCompleted	Releases and replaces any held IStorage pointers as necessary and returns NOERROR.
HandsOffStorage	Releases any held IStorage pointers and returns NOERROR.

IViewObject

In most cases a handler or in-process server will implement all of IViewObject for at least some display aspects. For others, like DVASPECT_ICON, you can delegate to the cache through the default handler which will try to perform the action on a presentation in the cache. The default handler's IViewObject never runs the local server. If you implement a member function for an aspect you need not call the default.

Table 11-5: Actions for the cache's IViewObject interface for a non-running object.

Member Function	Action
Draw	Attempts to draw using a presentation from the cache, otherwise returns OLE_E_BLANK.
GetColorSet	Tries to determine the color set from the metafile or bitmap in the cache. Returns OLE_E_BLANK if there is no presentation otherwise returns NOERROR or S_FALSE depending on success of the function.
Freeze	Adds the aspect to an internal list that affects the behavior of Draw and returns NOERROR if successful, OLE_E_BLANK if not. Returns VIEW_S_ALREADY_FROZEN if this is a repeat request.
Unfreeze	Removes an entry from the internal list of frozen aspects and frees any duplicate presentation. Returns OLE_E_NOCONNECTION if the aspect was not frozen, otherwise returns NOERROR.
SetAdvise	Saves the IAdviseSink such that the cache will call its OnViewChange when the cache itself is notified from the server through the cache's own IAdviseSink. Returns NOERROR.
GetAdvise	Returns the last IAdviseSink from SetAdvise and returns NOERROR.

Implementing an Object Handler

This section deals with implementation details for a basic handler using CHAP11\HSCHMOO (Handler for Schmoo) as an example. It will not provide as detailed of a step-by-step process as in the last two chapters simply because most of the interfaces we need to implement in a handler we've already discussed. In addition, we've already seen (way back in Chapter 4) how to make an object in a DLL, including the class factory. Nevertheless, the list below outlines the steps to create the *minimal* an object handler. Anything else you do above this is a bonus:

1. Implement DllGetClassObject, DllCanUnloadNow, your class factory, and your basic object with an IUnknown interface. You may or may not want to support aggregation and there is no requirement to do so. In all ways a handler is structured exactly like a component object DLL.

2. Store the path of the handler under your CLSID in the registration database under the key "InprocHandler." In Chapter 10 we stored "ole2.dll" here since we should always have some entry and ole2.dll is the default handler.

3. When creating your object, create a default handler object and obtain pointers to its IOleObject, IPersistStorage, and IViewObject interfaces

4. Delegate QueryInterface calls for any interface you are not implementing to the default handler, including IDataObject and IOleCache.

5. Implement all of IPersistStorage but still call the default handler anyway to insure maintenance of the cache.

6. Implement IOleObject::GetExtent, delegating all IOleObject calls to the default handler. GetExtent may not even be important for your type of data in which case you may still delegate.

7. Implement all of IViewObject for supported aspects and otherwise delegate to the default handler.

I will not bother to discuss steps 1 and 2 as they should already be a familiar process for you–the code for step 3 in HSchmoo is in HSCHMOO.CPP where IClassFactory::CreateInstance creates a C++ object of the class CFigure that holds all the necessary interfaces. The CFigure class is defined in HSCHMOO.H and shown below:

```
class __far CFigure : public IUnknown
    {
    //Make any contained interfaces your friends so they can get at privates
    friend class CImpIOleObject;
    friend class CImpIViewObject;
    friend class CImpIPersistStorage;
    friend class CImpIDataObject;
    friend class CImpIAdviseSink;

    protected:
        ULONG           m_cRef;         //Object reference count.
        LPUNKNOWN       m_punkOuter;    //Controlling Unknown for aggregation
        LPFNDESTROYED   m_pfnDestroy;   //Function to call on closure.

        POLYLINEDATA    m_pl;           //Our actual data.
        UINT            m_cf;           //Object clipboard format.
        CLSID               m_clsID;        //Current CLSID

        //These are default handler interfaces we use
        LPUNKNOWN           m_pDefIUnknown;
        LPOLEOBJECT         m_pDefIOleObject;
        LPVIEWOBJECT        m_pDefIViewObject;
        LPPERSISTSTORAGE    m_pDefIPersistStorage;
        LPDATAOBJECT        m_pDefIDataObject;

        //Implemented interfaces
        LPOLEOBJECT         m_pIOleObject;
        LPVIEWOBJECT        m_pIViewObject;
        LPPERSISTSTORAGE    m_pIPersistStorage;
        LPADVISESINK        m_pIAdviseSink;

        //Advise sink we get in IViewObject
        LPADVISESINK        m_pIAdvSinkView;
        DWORD               m_dwAdviseFlags;
        DWORD               m_dwAdviseAspects;
        DWORD               m_dwFrozenAspects;

        //Copies of frozen aspects
        POLYLINEDATA        m_plContent;
        POLYLINEDATA        m_plThumbnail;


    protected:
        void      Draw(HDC, LPRECT, DWORD, DVTARGETDEVICE FAR *, HDC,
LPPOLYLINEDATA);
        void      PointScale(LPRECT, LPPOINT, BOOL);

    public:
        CFigure(LPUNKNOWN, LPFNDESTROYED, HINSTANCE);
        ~CFigure(void);

        BOOL      FInit(void);

        //Non-delegating object IUnknown
        STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);
    };

typedef CFigure FAR * LPCFigure;
```

Note that HSCHMOO.H also contains definitions of all the CImpI* classes that are friends of CFigure.

These are like all the other interface implementations we've already seen so I won't repeat them here. Most of the variables in CFigure are set initially to NULL or zero except for *m_punkOuter* and *m_pfnDestroy* which are set from the parameters to the constructor, *m_clsID* which is set to CLSID_Schmoo2Figure, and *m_cf* which is set to the return value from RegisterClipboardFormat on the string "Polyline Figure." This clipboard format matches that used in all Schmoo applications. In addition, the CFigure destructor will call Release on any interface that we're using and call delete for any interface we implement, as normal cleanup goes. You can see these in FIGURE.CPP.

Now let's concentrate specifically on the rest of CFigure inside the handler and what it must do. Working through these steps works best if you already have created a compound document in some container that has an object of the class you want to handle.

Obtain a Default Handler IUnknown

When initializing the handler object you will want to obtain at least an IUnknown pointer to a default handler object set up for your CLSID so you can take advantage of the many services provided by OLE2.DLL. To obtain the pointer call OleCreateDefaultHandler, passing your CLSID and a pointer to your object's IUnknown (because the default handler must be aggregated). You want to pass your CLSID so that the default handler can implement various functions using the entries in the registration database under that CLSID. You must pass your IUnknown as the controlling unknown because we are aggregating the default handler.

Once you have this IUnknown you should QueryInterface for IOleObject, IPersistStorage, and IViewObject pointers (and also perhaps IDataObject) to which you can later delegate. This is much more efficient than calling QueryInterface, delegating the function, and calling Release every time you need to delegate. In HSchmoo this is handled in CFigure::FInit, called from IClassFactory::CreateInstance. FInit first allocated the interface implementations for this object, then calls OleCreateDefaultHandler followed by a series of QueryInterface calls:

```
BOOL CFigure::FInit(void)
    {
    LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;
    HRESULT         hr;
    DWORD           dwConn;
    FORMATETC       fe;

    if (NULL!=m_punkOuter)
        pIUnknown=m_punkOuter;

    //First create our interfaces.
    m_pIOleObject=new CImpIOleObject(this, pIUnknown);

    if (NULL==m_pIOleObject)
        return FALSE;

    m_pIViewObject=new CImpIViewObject(this, pIUnknown);

    if (NULL==m_pIViewObject)
        return FALSE;
```

```
m_pIPersistStorage=new CImpIPersistStorage(this, pIUnknown);

if (NULL==m_pIPersistStorage)
   return FALSE;

m_pIAdviseSink=new CImpIAdviseSink(this, pIUnknown);

if (NULL==m_pIAdviseSink)
   return FALSE;

/*
 * Get an IUnknown on the default handler, passing pIUnknown
 * as the controlling unknown.
 */
hr=OleCreateDefaultHandler(CLSID_Schmoo2Figure, pIUnknown, IID_IUnknown
   , (LPLPVOID)&m_pDefIUnknown);

if (FAILED(hr))
   return FALSE;

//Now try to get other interfaces to which we delegate
hr=m_pDefIUnknown->QueryInterface(IID_IOleObject
   , (LPLPVOID)&m_pDefIOleObject);

if (FAILED(hr))
   return FALSE;

hr=m_pDefIUnknown->QueryInterface(IID_IViewObject
   , (LPLPVOID)&m_pDefIViewObject);

if (FAILED(hr))
   return FALSE;

hr=m_pDefIUnknown->QueryInterface(IID_IDataObject
   , (LPLPVOID)&m_pDefIDataObject);

if (FAILED(hr))
   return FALSE;

hr=m_pDefIUnknown->QueryInterface(IID_IPersistStorage
   , (LPLPVOID)&m_pDefIPersistStorage);

if (FAILED(hr))
   return FALSE;

//Set up an advise on native data so we can keep in sync
```

```
SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL);
m_pDefIDataObject->DAdvise(&fe, 0, m_pIAdviseSink, &dwConn);

return TRUE;
}
```

Everything about this code should look pretty familiar by now, except for that part at the end which sets up a data advise with the IDataObject interface in the default handler. This connection is used to synchronize the handler and a local server when one is launched as discussed below in "Synchronized Swimming with your Local Server." Note that I don't save the connection key from DAdvise because I won't need to DUnadvise until I release the *m_pDefIDataObject* which will terminate the connection for me.

Expose Default Handler Interfaces in QueryInterface

You want to hold on to the default handler's IOleObject, IViewObject, and IPersistStorage pointers for delegation, but why hold on to its IUnknown? Remember that when we aggregate on any object as described at the end of Chapter 4 we are required to first of all ask for an IUnknown pointer when creating the object–OleCreateDefaulHandler eventually calls IClassFactory::CreateInstance with a non-NULL *punkOuter* which requires that *riid* passed to CreateInstance is IID_IUnknown. Second, as the outer object controlling the aggregation, we are required to delegate QueryInterface calls to the aggregated object when we ourselves do not implement that interface. We can see this in CFigure::QueryInterface:

```
STDMETHODIMP CFigure::QueryInterface(REFIID riid, LPVOID FAR *ppv)
    {
    *ppv=NULL;

    /*
     * The only calls we get here for IUnknown are either in a non-aggregated
     * case or when we're created in an aggregation, so in either we always
     * return our IUnknown for IID_IUnknown.
     */
    if (IsEqualIID(riid, IID_IUnknown))
        *ppv=(LPVOID)this;

    if (IsEqualIID(riid, IID_IPersist) || IsEqualIID(riid, IID_IPersistStorage))
        *ppv=(LPVOID)m_pIPersistStorage;

    if (IsEqualIID(riid, IID_IOleObject))
        *ppv=(LPVOID)m_pIOleObject;

    if (IsEqualIID(riid, IID_IViewObject))
        *ppv=(LPVOID)m_pIViewObject;

    //AddRef any interface we'll return.
    if (NULL!=*ppv)
        {
```

```
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    //Otherwise see if the default handler knows it.
    return m_pDefIUnknown->QueryInterface(riid, ppv);
    }


STDMETHODIMP_(ULONG) CFigure::AddRef(void)
    {
    return ++m_cRef;
    }


STDMETHODIMP_(ULONG) CFigure::Release(void)
    {
    ULONG       cRefT;

    cRefT=--m_cRef;

    if (0==m_cRef)
        delete this;

    return cRefT;
    }
```

Since we are implementing IPersistStorage, IOleObject, and IViewObject ourselves, we return our interface pointers from QueryInterface (as well as CFigure's IUnknown). If we were asked for some other interface, we hand off the request to the default handler with that last line in the function. So when we're asked, for example, for IDataObject or IOleCache, the handler will return its interfaces directly, which means that as far as our container (the user of this object) is concerned, those interfaces belong to us. Again, the container does no know that we are aggregating the default handler and just sees all the necessary embedded object interfaces as if we were implementing them. Through this simple act of delegating QueryInterface we expose default handler interfaces as our own.

This exposure is the entire reason why we had to give a controlling unknown to OleCreateDefaultHandler: any QueryInterface call to a pointer returned from the default handler, with the exception of IUnknown, will enter CFigure::QueryInterface allowing us the chance to show our interface implementation. We trust that the default handler's IUnknown will not delegate the controlling unknown as is required.

Note also that since CFigure is the embedded object, and not an interface implementation, it destroys itself when the reference count is zero as shown above in CFigure::Release.

If you want to compile and test something at this point you can omit the code in the example above that instantiates our interface implementations as well as that which returns their pointers from QueryInterface–then your object's QueryInterface delegates *all* requests to the default handler, essentially making your handler nothing more than a reference counter for the default handler object.

But what you can do is verify that your DllGetClassObject and class factory are working correctly and that all other operations of your embedded object class in a container are normal, that is, you should not see any difference between having your handler registered as the "InProcHandler" and having OLE2.DLL listed there. Your handler is just a simple layer between the container and the default handler.

Implement IPersistStorage

What gets exciting now is the opportunity to start changing specific pieces of each interface as suits our purposes. IPersistStorage is the best place to start because it's always called first in an embedded object scenario and is one that we can implement without any other impact on the operation of the container or default handler.

When you want to override any member function of an interface you must have entry points for the entire interface in your code. What I recommend is that you start off with a template interface implementation where each function simply delegates to the default handler's interface, as shown below for IPersistStorage (NOTE: This is not part of HSchmoo, just an example). Note that this assumes that the interface implementation has a controlling unknown to which it delegates all IUnknown members (*m_punkOuter*) and access to the default IPersistStorage pointer, in this case that is *m_pObj->m_pDefIPersistStorage* where *m_pObj* is a CFigure pointer:

```
STDMETHODIMP CImpIPersistStorage::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   return m_punkOuter->QueryInterface(riid, ppv);
   }

STDMETHODIMP_(ULONG) CImpIPersistStorage::AddRef(void)
   {
   ++m_cRef;
   return m_punkOuter->AddRef();
   }

STDMETHODIMP_(ULONG) CImpIPersistStorage::Release(void)
   {
   --m_cRef;
   return m_punkOuter->Release();
   }

STDMETHODIMP CImpIPersistStorage::GetClassID(LPCLSID pClsID)
   {
   return m_pObj->m_pDefIPersistStorage->GetClassID(pClsID);
   }

STDMETHODIMP CImpIPersistStorage::IsDirty(void)
   {
   return m_pObj->m_pDefIPersistStorage->IsDirty();
   }

STDMETHODIMP CImpIPersistStorage::InitNew(LPSTORAGE pIStorage)
```

```
    {
    return m_pObj->m_pDefIPersistStorage->InitNew(pIStorage);
    }

STDMETHODIMP CImpIPersistStorage::Load(LPSTORAGE pIStorage)
    {
    return m_pObj->m_pDefIPersistStorage->Load(pIStorage);
    }

STDMETHODIMP CImpIPersistStorage::Save(LPSTORAGE pIStorage, BOOL fSameAsLoad)
    {
    return m_pObj->m_pDefIPersistStorage->Save(pIStorage, fSameAsLoad);
    }

STDMETHODIMP CImpIPersistStorage::SaveCompleted(LPSTORAGE pIStorage)
    {
    return m_pObj->m_pDefIPersistStorage->SaveCompleted(pIStorage);
    }

STDMETHODIMP CImpIPersistStorage::HandsOffStorage(void)
    {
    return m_pObj->m_pDefIPersistStorage->HandsOffStorage();
    }
```

You can now integrate this "pass-through" interface implementation into your code and compile and test again. Since you are not doing anything but calling the default handler there should be no effect on the container or other operation of the embedded object.

Now you can pick and choose which member functions to override meaning which functions do you want to change in any way. In most handlers this means overriding at least GetClassID, InitNew, Load, and Save but not necessarily IsDirty, SaveCompleted, and HandsOffStorage. The code shown below is taken from HSchmoo's IPersistStorage implementation which, being a minimal and simple handler, overrides the four crucial functions and passes the remainder through:

```
STDMETHODIMP CImpIPersistStorage::GetClassID(LPCLSID pClsID)
    {
    *pClsID=m_pObj->m_clsID;
    return NOERROR;
    }

STDMETHODIMP CImpIPersistStorage::IsDirty(void)
    {
    /*
     * Since we don't edit, we have no idea if this data is dirty.
     * Delegate to the default handler in case it wants to ask the server.
     */
    return m_pObj->m_pDefIPersistStorage->IsDirty();
    }
```

```
STDMETHODIMP CImpIPersistStorage::InitNew(LPSTORAGE pIStorage)
   {
   //Good time to initilize our data
   m_pObj->m_pl.wVerMaj=VERSIONMAJOR;
   m_pObj->m_pl.wVerMin=VERSIONMINOR;
   m_pObj->m_pl.cPoints=0;
   m_pObj->m_pl.rgbBackground=GetSysColor(COLOR_WINDOW);
   m_pObj->m_pl.rgbLine=GetSysColor(COLOR_WINDOWTEXT);
   m_pObj->m_pl.iLineStyle=PS_SOLID;

   //Make sure these aren't filled with trash.
   _fmemcpy(&m_pObj->m_plContent,   &m_pObj->m_pl, CBPOLYLINEDATA);
   _fmemcpy(&m_pObj->m_plThumbnail, &m_pObj->m_pl, CBPOLYLINEDATA);

   m_pObj->m_pDefIPersistStorage->InitNew(pIStorage);
   return NOERROR;
   }


STDMETHODIMP CImpIPersistStorage::Load(LPSTORAGE pIStorage)
   {
   POLYLINEDATA    pl;
   ULONG           cb;
   LPSTREAM        pIStream;
   HRESULT         hr;

   if (NULL==pIStorage)
      return ResultFromScode(STG_E_INVALIDPOINTER);

   //Open the CONTENTS stream
   hr=pIStorage->OpenStream("CONTENTS", 0, STGM_DIRECT | STGM_READ
      | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

   if (FAILED(hr))
      return ResultFromScode(STG_E_READFAULT);

   //Read all the data into the POLYLINEDATA structure.
   hr=pIStream->Read((LPVOID)&pl, CBPOLYLINEDATA, &cb);
   pIStream->Release();

   if (CBPOLYLINEDATA!=cb)
      return ResultFromScode(STG_E_READFAULT);

   //Copy into the actual object now.
   _fmemcpy(&m_pObj->m_pl, &pl, CBPOLYLINEDATA);
```

```
    m_pObj->m_pDefIPersistStorage->Load(pIStorage);
    return NOERROR;
    }


STDMETHODIMP CImpIPersistStorage::Save(LPSTORAGE pIStorage, BOOL fSameAsLoad)
    {
    ULONG           cb;
    LPSTREAM        pIStream;
    HRESULT         hr;

    if (NULL==pIStorage)
        return ResultFromScode(STG_E_INVALIDPOINTER);

    /*
     * If the server is running, don't do the save ourselves since
     * we'd end up writing the storage twice with possible conflicts.
     */
    if (OleIsRunning(m_pObj->m_pDefIOleObject))
        return m_pObj->m_pDefIPersistStorage->Save(pIStorage, fSameAsLoad);

    //Rewrite the entire stream
    WriteClassStg(pIStorage, m_pObj->m_clsID);
    WriteFmtUserTypeStg(pIStorage, m_pObj->m_cf, "Polyline Figure");

    hr=pIStorage->CreateStream("CONTENTS", STGM_DIRECT | STGM_CREATE
        | STGM_WRITE | STGM_SHARE_EXCLUSIVE, 0, 0, &pIStream);

    if (FAILED(hr))
        return ResultFromScode(STG_E_WRITEFAULT);

    hr=pIStream->Write((LPVOID)&m_pObj->m_pl, CBPOLYLINEDATA, &cb);
    pIStream->Release();

    m_pObj->m_pDefIPersistStorage->Save(pIStorage, fSameAsLoad);

    return (SUCCEEDED(hr) && CBPOLYLINEDATA==cb) ?
        NOERROR : ResultFromScode(STG_E_WRITEFAULT);
    }


STDMETHODIMP CImpIPersistStorage::SaveCompleted(LPSTORAGE pIStorage)
    {
    return m_pObj->m_pDefIPersistStorage->SaveCompleted(pIStorage);
    }
```

```
STDMETHODIMP CImpIPersistStorage::HandsOffStorage(void)
    {
    return m_pObj->m_pDefIPersistStorage->HandsOffStorage();
    }
```

First of all, we have no reason here to override SaveCompleted and HandsOffStorage because we don't hold the IStorage from InitNew or Load outside the scope of those functions. Therefore these are pass-throughs. Likewise we don't override IsDirty because as a handler, we don't edit the data so how should we know when it's dirty? We could implement this ourselves because we do have a data advise set up on the default handler's IDataObject–which is ultimately the local server's when it's running–which will tell us when things change. But why bother complicating our lives? If the server is running, let the default handler ask it about being dirty. If the server is not running, then it can't be dirty and the default handler will return S_FALSE. So we just pass IsDirty on through.

GetClassID is somewhat similar to IsDirty in that we could just pass it through to the default handler which will return the CLSID we passed to OleCreateDefaultHandler. This will always be the same as the CLSID we know in this handler, so why not save an extra call and just implement this function ourselves? It really doesn't matter either way.

The interesting stuff is now contained in InitNew, Load, and Save where we either set the native data in the embedded object to default values, load it from an IStorage, or save it to an IStorage. InitNew is a great place for data initialization and is no better or worse than doing the same thing in the object's constructor or initializer–I like it here because it keeps all manipulation of that data in the same locality. Either we'll get a call to InitNew or Load when this object is first created, so either way we'll initialize the data, and we all like initialized data, right? Of course, InitNew will be called when a container uses the Insert Object dialog, and being a handler, we'll see this call first.

As a handler we have to implement IPersistStorage::Load if we want to implement anything else like IViewObject::Draw. We need something to draw! And Load is the way we get it. As long as we're implementing Load, we should also implement save because it will allow container's to make copies of the object were servicing even if the local server is not present. If you read Chapter 9 you'll know that the container copies embedded object by creating a new IStorage and asking the object to IPersistStorage::Save into that IStorage. If we don't implement Save then that functionality is seriously impaired, because the default handler's implementation of Save knows nothing of our object's native data.

So why then do I still call the default handler's InitNew, Load, and Save? Henry Jones Sr. would remind us that "There is more in the diary than just the map." To which we respond, "OK dad, tell me."[1] Well, there is more in the storage than just our object–there's cached presentations. If the container has obtained an IOleCache pointer and has called IOleCache::SetData, then somewhere down in the bowels of the default handler that presentation is waiting to be serialized to the storage. If we never called InitNew, Load, and Save from our own implementation, even when we implement these functions ourselves, then the cache will never have a chance to work with its presentations. So unless you want to implement your own cache, which is not something I recommend, you must give the cache the opportunity to do its job by calling the default InitNew, Load, and Save. Note, however, that the return value of these functions depends on your implementation and not that of the default handler, so you can

1From Indiana Jones and the Last Crusade. (c)1989 LucasFilm Ltd.

ignore the return code from the defaults.[2]

NOTE: If a handler implements IViewObject::Draw for any given aspect such that it never delegates Draw to the default handler, then the cache will *never* contain a presentation for that aspect. Your handler is entirely responsible for generating presentations for that aspect, and if a compound document is taken to another machine where not even your handler is present then the object will appear blank in the container. This is another reason why you should license handlers for free redistribution.

Finally I wanted to explain that little use of **OleIsRunning** (OLE2.H) in Save. The OleIsRunning function returns a BOOL telling us if the server for this CLSID is running or not for the object pointed to in the only parameter to the function (which *must* be an IOleObject pointer, mind you). If the server is running, then we the handler completely delegate to the default handler, which will save the cache and call the server's IPersistStorage::Save. What this does is insure that we don't save our data twice (a waste of time) and that what we save in the handler does not conflict with what the server decides to save. This is especially important if the server is incrementally accessing the storage and has already written some changes there, changes which a Save in the handler might obliterate. We want to avoid conflicts with the running server, so OleIsRunning is just what we need here.

I will point out again that you can override functions one at a time then compile and test that one function. To test InitNew you need to use the Insert Object dialog from a container. While you're doing this you might as well create a new object in the server and save it in the container's compound file, then close the file and reload it. This will generate a call to Load which you can insure will properly load your data without the server running at all. In the container you can copy the object to the clipboard which should generate a call to Save allowing you to test that out as well. You should also try editing the object in the server again and closing the server down which will call Save with the server still running, allowing you to test your use of OleIsRunning.

Implement IOleObject::GetExtent

I have some bad news: the minimal handler has to override the GetExtent function in IOleObject which means you have to provide pass-throughs for the other twenty members. What a pain. We never said life was fair. But you can use HSchmoo's IOLEOBJ.CPP as a template for doing this if you have structured your code similar to mine.

GetExtent is a required override because a container may depend on this function to determine the size of its site in which it displays your object when loading a compound document. The entry for GetExtent in Table 11-1 mentioned that for this function the default handler tries to determine the size of the data based on presentations in the cache. But as we saw in discussing IPersistStorage just now, there is generally no cached presentations for those aspects for which you implement IViewObject::Draw, and so the default handler cannot implement GetExtent for those aspects. In HSchmoo we implement IViewObject::Draw for DVASPECT_CONTENT and DVASPECT_THUMBNAIL and so we must provide an implementation of GetExtent for both those presentations as well:

```
STDMETHODIMP CImpIOleObject::GetExtent(DWORD dwAspect, LPSIZEL pszl)
    {
    SIZEL       szl;
    LPRECT      prc;
```

2While this may not seem like a great practice, it's more friendly to succeed even if the cache fails because you still have the potential to draw a presentation from your native data regardless of what happened in the cache. A sophisticated handler may detect and remember that the cache fails such that it does not later rely on the cache from a function like IViewObject::Draw. That level of sophistication is outside the scope of this discussion.

```
/*
 * We can answer for CONTENT/THUMBNAIL, but try the server for others.
 * In addition, always delegate is the server is running since
 * it has a window to define the size.
 */
if (!((DVASPECT_CONTENT | DVASPECT_THUMBNAIL) & dwAspect)
   || OleIsRunning(m_pObj->m_pDefIOleObject))
   return m_pObj->m_pDefIOleObject->GetExtent(dwAspect, pszl);

/*
 * The size is in the rc field of the POLYLINEDATA structure
 * which we now have to convert to HIMETRIC.
 */
prc=&m_pObj->m_pl->rc;
SETSIZEL(szl, prc->right-prc->left, prc->bottom-prc->top);
XformSizeInPixelsToHimetric(NULL, &szl, pszl);
return NOERROR;
}
```

The POLYLINEDATA structure that makes up the object's data in HSchmoo just so happens to contain a rectangle which we can use here to calculate the size of the object in HIMETRIC units (as GetExtent must return). In addition, notice that we delegate this call to the default handler not only when it's called with an aspect we don't draw ourselves but also when the server is running for *any* aspect. This is because a running server, since it generally implements SetExtent as well, has a better idea of how large the object currently is whereas this handler only knows what's in our storage. You may also choose, if desired, to implement SetExtent in your handler to record what the container says is the size (then calling the default anyway) and return those extents from GetExtent.

You can again compile and test your GetExtent implementation and to verify that everything else you pass through in IOleObject is correct.

Implement IViewObject

Now comes the real reason why we generally have object handlers in the first place: to optimize drawing for particular devices. Your implementation of IPersistStorage loads the data to draw and your IOleObject::GetExtent lets the container know what sort of rectangle you generally like to be drawn in (although the container may have its own rather individualistic ideas about your size, in which case you must obey). So now we can implement IViewObject::Draw.

If you implement IViewObject::Draw you also need to implement IViewObject::Freeze, Unfreeze, SetAdvise, and GetAdvise for the same aspects. This makes IViewObject the most complicated interfaces in your handler as you can see from the extent of code shown below in Listing 11-1.

IVIEWOBJ.CPP

[Constructor, destructor, IUnknown members omitted]

```
/*
```

```
 * CImpIViewObject::Draw
 *
 * Purpose:
 *  Draws the object on the given hDC specifically for the requested
 *  aspect, device, and within the appropriate bounds.
 *
 * Parameters:
 *  dwAspect        DWORD aspect to draw.
 *  lindex          LONG index of the piece to draw.
 *  pvAspect        LPVOID for extra information, always NULL.
 *  ptd             DVTARGETDEVICE FAR * containing device information.
 *  hICDev          HDC containing the IC for the device.
 *  hDC             HDC on which to draw.
 *  pRectBounds     LPRECTL describing the rectangle in which to draw.
 *  pRectWBounds    LPRECTL describing the placement rectangle if part
 *                  of what you draw is another metafile.
 *  pfnContinue     Function to call periodically during long repaints.
 *  dwContinue      DWORD extra information to pass to the pfnContinue.
 */

STDMETHODIMP CImpIViewObject::Draw(DWORD dwAspect, LONG lindex
    , void FAR * pvAspect, DVTARGETDEVICE FAR * ptd, HDC hICDev
    , HDC hDC, const LPRECTL pRectBounds, const LPRECTL pRectWBounds
    , BOOL (CALLBACK * pfnContinue) (DWORD), DWORD dwContinue)
    {
    RECT            rc;
    POLYLINEDATA    pl;
    LPPOLYLINEDATA  ppl=&m_pObj->m_pl;

    RECTFROMRECTL(rc, *pRectBounds);

    //Delegate iconic and printed representations.
    if (!((DVASPECT_CONTENT | DVASPECT_THUMBNAIL) & dwAspect))
        {
        return m_pObj->m_pDefIViewObject->Draw(dwAspect, lindex
            , pvAspect, ptd, hICDev, hDC, pRectBounds, pRectWBounds
            , pfnContinue, dwContinue);
        }



    /*
     * If we're asked to draw a frozen aspect, use the data from
     * a copy we made in IViewObject::Freeze.  Otherwise use the
     * current data.
     */
    if (dwAspect & m_pObj->m_dwFrozenAspects)
```

```
        {
        //Point to the data to actually use.
        if (DVASPECT_CONTENT==dwAspect)
            ppl=&m_pObj->m_plContent;
        else
            ppl=&m_pObj->m_plThumbnail;
        }


    //Make a copy so we can modify it (and not worry about threads)
    _fmemcpy(&pl, ppl, CBPOLYLINEDATA);

    /*
     * If we're going to a printer, check if it's color capable.
     * if not, then use black on white for this figure.
     */
    if (NULL!=hICDev)
        {
        if (GetDeviceCaps(hICDev, NUMCOLORS) <= 2)
            {
            pl.rgbBackground=RGB(255, 255, 255);
            pl.rgbLine=RGB(0, 0, 0);
            }
        }

    m_pObj->Draw(hDC, &rc, dwAspect, ptd, hICDev, &pl);
    return NOERROR;
    }



/*
 * CImpIViewObject::GetColorSet
 * Not implemented
 */

STDMETHODIMP CImpIViewObject::GetColorSet(DWORD dwDrawAspect, LONG lindex
    , LPVOID pvAspect, DVTARGETDEVICE FAR * ptd
    , HDC hICDev, LPLOGPALETTE FAR * ppColorSet)
    {
    return ResultFromScode(S_FALSE);
    }



/*
 * CImpIViewObject::Freeze
 *
 * Purpose:
 *  Freezes the view of a particular aspect such that data changes do
 *  no affect the view.
 *
 * Parameters:
 *  dwAspect        DWORD aspect to freeze.
 *  lindex          LONG piece index under consideration.
 *  pvAspect        LPVOID for further information, always NULL.
 *  pdwFreeze       LPDWORD in which to return the key.
 */

STDMETHODIMP CImpIViewObject::Freeze(DWORD dwAspect, LONG lindex
    , LPVOID pvAspect, LPDWORD pdwFreeze)
    {
    //Delegate any aspect we don't handle.
    if (!((DVASPECT_CONTENT | DVASPECT_THUMBNAIL) & dwAspect))
        {
        return m_pObj->m_pDefIViewObject->Freeze(dwAspect, lindex
            , pvAspect, pdwFreeze);
        }

    if (dwAspect & m_pObj->m_dwFrozenAspects)
        {
        *pdwFreeze=dwAspect + FREEZE_KEY_OFFSET;
        return ResultFromScode(VIEW_S_ALREADY_FROZEN);
        }

    m_pObj->m_dwFrozenAspects |= dwAspect;



    /*
     * For whatever aspects become frozen, make a copy of the data.
     * Later when drawing, if such a frozen aspect is requested, we'll
     * draw from this data rather than from our current data.
     */
    if (DVASPECT_CONTENT==dwAspect)
        _fmemcpy(&m_pObj->m_plContent, &m_pObj->m_pl, CBPOLYLINEDATA);

    if (DVASPECT_THUMBNAIL==dwAspect)
        _fmemcpy(&m_pObj->m_plThumbnail, &m_pObj->m_pl, CBPOLYLINEDATA);

    if (NULL!=pdwFreeze)
        *pdwFreeze=dwAspect + FREEZE_KEY_OFFSET;

    return NOERROR;
    }
```

```
/*
 * CImpIViewObject::Unfreeze
 *
 * Purpose:
 *  Thaws an aspect frozen in ::Freeze.  We expect that a container
 *  will redraw us after freezing if necessary, so we don't send
 *  any sort of notification here.
 *
 * Parameters:
 *  dwFreeze        DWORD key returned from ::Freeze.
 */

STDMETHODIMP CImpIViewObject::Unfreeze(DWORD dwFreeze)
    {
    DWORD       dwAspect=dwFreeze - FREEZE_KEY_OFFSET;

    //Delegate any aspect we don't handle.
    if (!((DVASPECT_CONTENT | DVASPECT_THUMBNAIL) & dwAspect))
        return m_pObj->m_pDefIViewObject->Unfreeze(dwFreeze);

    //The aspect to unfreeze is in the key.
    m_pObj->m_dwFrozenAspects &= ~(dwAspect);

    /*
     * Since we always kept our current data up to date, we don't
     * have to do anything thing here like requesting data again.
     * Because we removed dwAspect from m_dwFrozenAspects, Draw
     * will again use the current data.
     */

    return NOERROR;
    }


/*
 * CImpIViewObject::SetAdvise
 *
 * Purpose:
 *  Provides an advise sink to the view object enabling notifications
 *  for a specific aspect.
 *
 * Parameters:
 *  dwAspects       DWORD describing the aspects of interest.
 *  dwAdvf          DWORD containing advise flags.
 *  pIAdviseSink    LPADVISESINK to notify.
 */

STDMETHODIMP CImpIViewObject::SetAdvise(DWORD dwAspects, DWORD dwAdvf
    , LPADVISESINK pIAdviseSink)
    {
    //Pass anything we don't support on through
    if (!((DVASPECT_CONTENT | DVASPECT_THUMBNAIL) & dwAspects))
        m_pObj->m_pDefIViewObject->SetAdvise(dwAspects, dwAdvf, pIAdviseSink);

    if (NULL!=m_pObj->m_pIAdvSinkView)
        m_pObj->m_pIAdvSinkView->Release();

    m_pObj->m_dwAdviseAspects=dwAspects;
    m_pObj->m_dwAdviseFlags=dwAdvf;

    m_pObj->m_pIAdvSinkView=pIAdviseSink;

    if (NULL!=m_pObj->m_pIAdvSinkView)
        m_pObj->m_pIAdvSinkView->AddRef();

    return NOERROR;
    }


/*
 * CImpIViewObject::GetAdvise
 *
 * Purpose:
 *  Returns the last known IAdviseSink seen by ::SetAdvise.
 *
 * Parameters:
 *  pdwAspects      LPDWORD in which to store the last requested aspects.
 *  pdwAdvf         LPDWORD in which to store the last requested flags.
 *  ppIAdvSink      LPADVISESINK FAR * in which to store the IAdviseSink.
 */

STDMETHODIMP CImpIViewObject::GetAdvise(LPDWORD pdwAspects
    , LPDWORD pdwAdvf, LPADVISESINK FAR* ppAdvSink)
    {
    if (NULL==m_pObj->m_pIAdvSink)
        return ResultFromScode(OLE_E_NOCONNECTION);

    if (NULL==ppAdvSink)
        return ResultFromScode(E_INVALIDARG);
    else
```

```
        {
        *ppAdvSink=m_pObj->m_pIAdvSinkView;
        m_pObj->m_pIAdvSinkView->AddRef();
        }

    if (NULL!=pdwAspects)
        *pdwAspects=m_pObj->m_dwAdviseAspects;

    if (NULL!=pdwAdvf)
        *pdwAdvf=m_pObj->m_dwAdviseFlags;

    return NOERROR;
    }
```

Listing 11-1: Implementation of the IViewObject interface in HSchmoo.

Let's look first at the simpler member functions before we jump into Draw. First of all, GetColorSet is unimportant for this handler (and Schmoo as well) so we just return S_FALSE to say we don't have anything for the caller. Next, SetAdvise and GetAdvise handle a container's IAdviseSink pointer to which we must send OnViewChange notifications when any data change occurs in our object such that we would need to repaint. We delegate both these calls to the default handler for DVASPECT_ICON and DVASPECT_DOCPRINT since we rely on the cache to handle those aspects for us in all other parts of the handler. That leaves us in SetAdvise to save the advise aspects, the flags, and the IAdviseSink pointer to which we'll later send notifications as described below in "Synchronized Swimming with your Local Server." We need to hold all of these parameter such that we can return them through GetAdvise as shown. Remember to AddRef the IAdviseSink pointer when you save a copy (and Release it before overwriting it) as well as to AddRef it when returning a copy from GetAdvise.

NOTE: SetAdvise may be called with a NULL IAdviseSink pointer! This means that the container is terminating the connection. Be sure you don't attempt to AddRef it without checking for NULL. (I tell you this because I forgot and had a few extra UAEs to deal with!)

Freeze and Unfreeze (which as I said before might have better been named Thaw) are a pair that the container uses to control when a presentation is allowed to change. Note that a change in the presentation does not mean that you freeze your underlying data because a freeze only affects one aspect. In HSchmoo's case, a freeze on DVASPECT_CONTENT cannot freeze the data because DVASPECT_THUMBNAIL is drawn from the same data and it is not frozen. Therefore we must make a snapshot of the frozen data such that when we're asked to draw that aspect we use the frozen copy instead of the current data, thus still allowing the current data to change as desired. This also allows IPersistStorage::Save to write the current data without having to consider a frozen view aspect, which should not affect storage in any way.

Your implementation of Freeze must somehow remember that the aspect is frozen and make a snapshot of the data. HSchmoo's code above ORs the new aspect in with a list of currently frozen aspects in CFigure's *m_dwFrozenAspects*, then snapshots the current data into either CFigure's *m_plContent* or *m_plThumbnail* structures depending on the aspect. Draw will later use all of this to determine exactly which data to present. In addition, Freeze must return some sort of key that can be later passed to Unfreeze–a good key is the aspect plus some random number to make the number meaningless to the caller. For example, I use FREEZE_KEY_OFFSET which I define in HSCHMOO.H as 0x0723.[1] When this key is later passed to Unfreeze we subtract the offset to yield an aspect and remove that aspect from *m_dwFrozenAspects*. When Draw is subsequently called (which we assume will happen since a container that thaws a view object will generally want to update immediately) we will see that the aspect is not frozen and therefore draw from the current data.

Note also that Freeze you should first check to see if the requested aspect is already frozen and return VIEW_S_ALREADY_FROZEN if so but must still return a key which the caller can later pass to Unfreeze. VIEW_S_ALREADY_FROZEN, having the _S_, is not an error but just avoids unnecessary repetition.

This brings us finally to Draw which, in general either calls CFigure::Draw (FIGURE.CPP) for DVASPECT_CONTENT and DVASPECT_THUMBNAIL or the default handler's IViewObject::Draw for any other aspect. The latter check happens first so we can ignore those cases.

For content and thumbnail aspects we now have to see if they are frozen, and if so we use the data we copied in Freeze instead of the current data. In the code above, Draw sets a pointer *ppl* initially to the current data and later, if the aspect is frozen, points *ppl* instead to the snapshot of that aspect. This way Draw can then copy whatever *ppl* points to into a local POLYLINEDATA structure such that we can do a few device optimizations. HSchmoo doesn't do anything fancy with devices but does provide an example of rending differently for a printer than for the screen. If the parameter *hICDev* to Draw is non-NULL that means we are going to a device other than the screen. If that is true, then we check for the number of colors the device supports. If it's a black and white device with only two colors, then we force the background color of the rendering to be white and the line color to be white which avoids potentially ugly dithering or large black blocks on the printer. I can't say if this is the best thing to do, but hey, it's just an example.

IViewObject::Draw then calls CFigure::Draw with the local POLYLINEDATA structure that we potentially modified. If might be a frozen aspect in which case the right thing is drawn, and it may be modified for a printer. In any case CFigure::Draw does its thing. I won't show that function here as it's a load of GDI calls that add nothing to our insights about handlers. I do want to point out one bug I had when writing it, however. This code was originally in Schmoo for the WM_PAINT handling of the Polyline window. Because the Polyline was always in its own window, it's client area always started at (0,0). So assuming that I did not have the code in place to handle cases where the upper-left was not (0,0). So when I first compiled HSchmoo with the IViewObject implementation it continually drew in the upper-left corner of the container instead of in the container's site. Not good. I had to make sure that HSchmoo's implementation of the drawing code would work at any rectangle.

After implementing IViewObject you now have a handler that can load, save, and display or print your object's data to whatever device without requiring a local server. After you have your drawing code debugged, you may want to try creating a compound document and copying it to another machine where you can open it again with and without your handler installed in the registration database. This will give you an indication of what will happen without the handler and what is possible with it there and in the absence of a local server.

Synchronized Swimming with your Local Server

So now everything looks great and is less filling too, until you activate the object and start making changes in the server. Wait a minute! The changes you make in the server are not reflected in the container like they were before! What's going on? Well, you have a handler, and whenever the container calls OleDraw or IViewObject::Draw it's going to the handler and the handler doesn't know

1If you can figure out where I got this number, I congradulate you on your resourcefulness. But don't expect any prizes.

about the changes you've been making. So how do we keep the handler and the running server in sync?

When we implemented a server in Chapter 10 we sent OnDataChange notifications to all advise sinks that came into our IDataObject implementation (through IDataAdviseHolder, of course). Where do these notifications go? Simple answer is that they go to any advise sink with a connection to the running object. By default, the only advise sink with a connection to the running server is the cache, which will ask the server through IDataObject::GetData for an updated presentation of whatever aspects are in the cache.

As a handler we would also like to know when the data changes such that we can update our data in preparation for the next time the container calls our IViewObject::Draw. In addition, since we have the container's IAdviseSink pointer as passed to IViewObject::SetAdvise we must also inform the container that we have new data to draw. For this reason we need our own implementation of IAdviseSink::OnDataChange (but NOT OnViewChange, since we don't call the default handler's IViewObject::SetAdvise ourselves).

This is a little tricky and had me beating my head into the wall a few times. First of all, IAdviseSink is conceptually part of a different object separate from the embedded object. What this means in practice is that your object's QueryInterface does *not* know about IAdviseSink and that your IAdviseSink::QueryInterface does *not* know any of the object interfaces. I'll admit that I'm doing it a little sleazy here: my handler's advise sink is managed by CFigure but CFigure::QueryInterface and CImpIAdviseSink::QueryInterface do not intermix. In other words, my advise sink does not delegate QueryInterface to CFigure, although it still delegates AddRef and Release since the advise sink's lifetime is still contained within CFigure's lifetime as you can see in the code below:

```
STDMETHODIMP CImpIAdviseSink::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IAdviseSink))
      {
      *ppv=(LPVOID)this;
      AddRef();
      return NOERROR;
      }

   return ResultFromScode(E_NOINTERFACE);
   }


STDMETHODIMP_(ULONG) CImpIAdviseSink::AddRef(void)
   {
   ++m_cRef;
   return m_punkOuter->AddRef();
   }

STDMETHODIMP_(ULONG) CImpIAdviseSink::Release(void)
   {
   --m_cRef;
   return m_punkOuter->Release();
   }
```

This advise sink is, by the way, exactly the one I previously passed to the default handler's IDataObject::DAdvise in CFigure::FInit, asking for notifications on the registered clipboard format of "Polyline Figure." Note also that when I called DAdvise I did not specify ADVF_NODATA. Therefore whenever a running Schmoo server calls OnDataChange, HSchmoo's advise sink will see an OnDataChange call where the STGMEDIUM passed to this call contains the most recent data from Schmoo:

```
/*
 * IAdviseSink::OnDataChange
 *
 * Purpose:
 *  Tells us that things changed in the server.  We asked for data
 *  on the advise so we can copy it from here into our own structure
 *  such that on the next OnViewChange we can repaint with it.
 */

STDMETHODIMP_(void) CImpIAdviseSink::OnDataChange(LPFORMATETC pFE
    , LPSTGMEDIUM pSTM)
    {
    //Get the new data first, then notify the container to repaint.
    if ((pFE->cfFormat==m_pObj->m_cf) && (TYMED_HGLOBAL & pSTM->tymed))
        {
        LPPOLYLINEDATA      ppl;

        ppl=(LPPOLYLINEDATA)GlobalLock(pSTM->hGlobal);
        _fmemcpy(&m_pObj->m_pl, ppl, CBPOLYLINEDATA);
        GlobalUnlock(pSTM->hGlobal);

        /*
         * Now tell the container that the view changed, but only
         * if the view is not frozen.'
         */
        if (pFE->dwAspect & m_pObj->m_dwAdviseAspects
            && !(pFE->dwAspect & m_pObj->m_dwFrozenAspects))
            {
            //Pass this on to the container.
            if (NULL!=m_pObj->m_pIAdvSinkView)
                {
                m_pObj->m_pIAdvSinkView->OnViewChange(pFE->dwAspect
                    , pFE->lindex);
                }
            }
        }

    return;
    }
```

When we get the OnDataChange, we then check that the data is actually what we asked for (a good defensive measure) then copy it into our current data. But that is not all we're responsible for, because we have a pointer to the container's advise sink that has been patiently waiting for a call to its OnViewChange. Therefore in OnDataChange we check to see if the aspect that changed (in *pFE->dwAspect*) is frozen, and if not we call the container's IAdviseSink::OnViewChange. This in turn will cause the container to repaint and will call IViewObject::Draw in our handler which then draws this current data.

Simple enough? It looks that way, but I want to let you know a few things that reared their ugly heads while I was writing this code. The problem I had to solve was how to get a copy of the most recent server data into the process space of the container and handler. I thought first that when I received IViewObject::Draw I could reload the data from the IStorage passed to InitNew and Load, but that presupposed that the server was actively writing changes to the storage. Alas, it was not so with Schmoo, so I thought, "why not ask the server to save the data for me before I repaint so I can reload it?" Well, that's slow for one thing, but the bigger problem was that when you received an OnDataChange you're in an asynchronous call (as notifications go). What I tried was to send OnViewChange to the container which turned around and called IViewObject::Draw, still within the asynchronous call.[1] That meant that a call to the server's IPersistStorage::Save failed with RPC_E_CANTCALLOUT_INASYNCCALL. So that didn't work. So sooner or later I had to let go the idea of reloading from the storage I saw in IPersistStorage and opted instead for the correct method of having data send through OnDataChange, since you also get the same error code if you attempt to call IDataObject::GetData from within an IViewObject member.

The other problem I had, which was really a mistaken assumption, was that I was calling the container's OnViewChange from within my OnViewChange after I tried calling the default handler's IViewObject::SetAdvise on those aspects that I wanted to support myself. Heck, I mean the cache sends out OnViewChange notifications when the server sends OnDataChange, right? True, it does, but it send OnViewChange before any OnDataChange. So what I was doing was causing the container to repaint with my old data before I was given the new data in OnDataChange. Once I understood how these notifications were coming through I had to make sure that my handler controlled the timings of OnViewChange notifications to the container from *within* my OnDataChange implementation.

The final note I wanted to make was that you want to make optimal use the storage mediums when communicating between server and handler across OnDataChange. For small data structures it's best for the data to be exchanged in TYMED_HGLOBAL, as Schmoo and HSchmoo do. For larger data, take advantage of TYMED_ISTORAGE or TYMED_ISTREAM such that very little data has to be copied when crossing the process boundary. TYMED_ISTORAGE is a great choice because it can allow incremental access in both pieces of code. Note that if you use this method the server must open the IStorage with the appropriate permissions such that the handler can also access that storage.

Christmas Bonuses

After following through the last sections you will have a complete basic handler fully functional to render your object on a variety of displays without depending on the cache or a local server. Anything else you do now is just one more added benefit to your handler like giving it an extra paycheck at the end of the year.

There are many possibilities for improvement. You might want to implement IDataObject::GetData and GetDataHere, for example, to further reduce the need to launch a local server. After all, you already know how to draw your data in the handler as well as how to save it to an IStorage, and so you'd

1You could ask all containers to yield between OnViewChange and a repaint, but that is ludicrous and would place an absurd burden on containers.

only need to add a little code to draw into a metafile or bitmap and to save into a different IStorage. You might also think of ways to reduce your dependence on the cache, possibly managing it all yourself. You might also consider adding additional features through a custom interface since you are a DLL and can provide one without the need for custom marshaling–these new interfaces can do whatever you like. You may also implement a "Play" verb in OleObject::DoVerb if your object type supports that sort of concept, again greatly improving performance and reducing your need for a server.

But eventually you cross the line. You give the handler too much extra pay and it decides to go our and buy a yacht for which it has to pay the luxury tax. At that point it ceases to rely on a local server for anything by implementing all of IDataObject and all of IOleObject, thus providing full editing services. The handler is now qualified to be an in-process server. But before you are *confirmed* as such there are a number of points to consider. The remainder of this chapter looks specifically at those issues.

Notes on Implementing an In-Process Server

In Chapter 10 we saw how to implement a complete local server and in the last section we saw how to implement a handler in a DLL. Now we can bring to the two together into a single in-process server DLL. Since we've seen most of the implementation already, this section will highlight specific issues that I faced when modifying the Polyline object DLL to be an in-process server as well as the component object it has been up to this point. It will not be an implementation guide because you only need to follow the guide for the handler in the previous section, then add more code to fill out the interfaces such that the default handler never attempts to launch a local server. In essence you go to a car dealer and keep adding more options until that bare-bones economy car becomes your high-performance dream machine. Note that you also don't need to worry about synchronizing with a running local server since there will never be one.

I want to point out up front that modifications made to Polyline to be an in-process DLL has not affected its usefulness to Component Schmoo. In fact, this chapter's version of CoSchmoo required no changes to use this chapter's version of Polyline except that it refers to it using CLSID_Polyline11 instead of CLSID_Polyline6 (which was the last revision of Polyline). If I was not using chapter numbers on the CLSID, CoSchmoo would have required no changes at all! What this shows is that support for compound documents *does not interfere with the general operation of the object as a component object*. While container's can now use Polyline as an in-process server for embedded objects, CoSchmoo can still use the exact same DLL and the exact same objects as a component object with a custom interface. That's the beauty of the QueryInterface mechanism: an object like Polyline can support both types of users without requiring either one to hard-code specific information. CoSchmoo doesn't know anything about Polyline's IOleObject interface and never asks for it. Any container will remain safely ignorant about the IPolyline interface. All because of the almighty QueryInterface.

To serve as a basis for our discussion the important modifications and additions to Polyline (from various source files) are shown in Listing 11-2. There is considerable code that is not shown in this listing because it is identical to that we've shown already for the handler in the previous section.

POLYLINE.H

```
...

#define PROP_SELECTOR   "Selector"
#define PROP_OFFSET     "Offset"
```

```
BOOL __export FAR PASCAL PolyDlgProc(HWND, UINT, WPARAM, LPARAM);

class __far CPolyline : public IUnknown
    {
   ...

    friend BOOL __export FAR PASCAL PolyDlgProc(HWND, UINT, WPARAM, LPARAM);
    friend class CImpIOleObject;
    friend class CImpIViewObject;

    protected:
        ...

        //These are default handler interfaces we use
        LPUNKNOWN           m_pDefIUnknown;
        LPOLEOBJECT         m_pDefIOleObject;
        LPVIEWOBJECT        m_pDefIViewObject;
        LPPERSISTSTORAGE    m_pDefIPersistStorage;
        LPDATAOBJECT        m_pDefIDataObject;

        //Implemented and used interfaces
        LPOLEOBJECT         m_pIOleObject;          //Implemented
        LPOLEADVISEHOLDER   m_pIOleAdviseHolder;    //Used

        LPOLECLIENTSITE     m_pIOleClientSite;      //Used

        LPVIEWOBJECT        m_pIViewObject;         //Implemented
        LPADVISESINK        m_pIAdviseSink;         //Used
        DWORD               m_dwFrozenAspects;      //From IViewObject::Freeze
        DWORD               m_dwAdviseAspects;      //From IViewObject::SetAdvise
        DWORD               m_dwAdviseFlags;        //From IViewObject::SetAdvise

        POLYLINEDATA        m_plContent;            //For freezing
        POLYLINEDATA        m_plThumbnail;          //For freezing

        HWND                m_hDlg;                 //Editing window

        ...
    };
```

[Also added codes for CPolyline::SendAdvise and a class CImpIOleObject to POLYLINE.H, not shown here.]

RESOURCE.H

```
[Other definitions omitted]

#define IDS_CLOSECAPTION    2
#define IDS_CLOSEPROMPT     3
#define IDS_POLYLINEMAX     3

#define IDR_ICON            1

#define IDD_EDITDIALOG      1

#define ID_UNDO             100
#define ID_COLORBACK        101
#define ID_COLORLINE        102
#define ID_GROUPCOLORS      103
#define ID_GROUPPREVIEW     104
#define ID_GROUPSTYLES      105
#define ID_GROUPFIGURE      106
#define ID_POLYLINERECT     107
#define ID_POLYLINE         108

#define ID_LINEMIN          200
#define ID_LINESOLID        200     //(ID_LINEMIN+PS_SOLID)
#define ID_LINEDASH         201     //(ID_LINEMIN+PS_DASH)
#define ID_LINEDOT          202     //(ID_LINEMIN+PS_DOT)
#define ID_LINEDASHDOT      203     //(ID_LINEMIN+PS_DASHDOT)
#define ID_LINEDASHDOTDOT   204     //(ID_LINEMIN+PS_DASHDOTDOT)
```

POLYLINE.RC

```
[Other resources omitted]

//This is the default for iconic representations.
IDR_ICON        ICON    polyline.ico


//This dialog is used to edit the Polyline
IDD_EDITDIALOG DIALOG 6, 18, 258, 152
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Polyline"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "", ID_POLYLINERECT, "Static", SS_BLACKFRAME, 8, 12, 134, 13
    PUSHBUTTON      "&Close", IDOK, 178, 6, 56, 14
    PUSHBUTTON      "&Undo", ID_UNDO, 178, 24, 56, 14
    PUSHBUTTON      "&Background...", ID_COLORBACK, 178, 54, 56, 14
    PUSHBUTTON      "&Line...", ID_COLORLINE, 178, 72, 56, 14
    GROUPBOX        "Colors", ID_UNDO, 158, 42, 94, 48
    GROUPBOX        "Figure", ID_GROUPFIGURE, 2, 0, 146, 148
    GROUPBOX        "Line Styles", ID_GROUPSTYLES, 158, 94, 94, 54
    CONTROL         "&Solid", ID_LINESOLID, "Button",
                    BS_AUTORADIOBUTTON | WS_GROUP, 166, 106, 32, 10
    CONTROL         "&Dash", ID_LINEDASH, "Button", BS_AUTORADIOBUTTON, 216,
                    106, 32, 10
    CONTROL         "Da&sh-Dot-Dot", ID_LINEDASHDOTDOT, "Button",
                    BS_AUTORADIOBUTTON, 166, 134, 80, 10
    CONTROL         "D&ot", ID_LINEDOT, "Button", BS_AUTORADIOBUTTON, 216,
                    120, 32, 10
    CONTROL         "D&ash-Dot", ID_LINEDASHDOT, "Button",
                    BS_AUTORADIOBUTTON, 166, 120, 48, 10
END


STRINGTABLE
    BEGIN
     IDS_STORAGEFORMAT,      "Polyline Figure"
     IDS_USERTYPE,           "Polyline Figure"
     IDS_CLOSECAPTION,       "Polyline"
     IDS_CLOSEPROMPT,        "Object has changed.  Do you wish to update it?"
    END

...
```

POLYLINE.CPP

```
STDMETHODIMP CPolyline::QueryInterface(REFIID riid, LPVOID FAR *ppv)
    {
    *ppv=NULL;

    [Cases for IUnknown, IPersist(Storage), IDataObject, unmodified]

    if (IsEqualIID(riid, IID_IOleObject))
        *ppv=(LPVOID)m_pIOleObject;

    if (IsEqualIID(riid, IID_IViewObject))
        *ppv=(LPVOID)m_pIViewObject;

    //Use the default handler's cache.
    if (IsEqualIID(riid, IID_IOleCache))
        return m_pDefIUnknown->QueryInterface(riid, ppv);
    //AddRef any interface we'll return.
    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }


[AddRef, Release, and RectConvertMappings unmodified]


STDMETHODIMP CPolyline::DataSet(LPPOLYLINEDATA pplIn, BOOL fSizeToData
    , BOOL fNotify)
    {
    [Unmodified code to integrate data...]

    //Notify containers
    SendAdvise(OBJECTCODE_DATACHANGED);
    return NOERROR;
    }


[Rendering code unmodified]
[CPolyline::SendAdvise identical to Schmoo's in Chapter 10]
```

POLYWIN.CPP

```
...

LRESULT __export FAR PASCAL PolylineWndProc(HWND hWnd, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    ...
    case WM_LBUTTONDOWN:
        [Adds a point to the Polyline and repaints]
        ppl->SendAdvise(OBJECTCODE_DATACHANGED);
        break;

    ...
    }
```

[CPolyline::Draw modified to take a rectangle and POLYLINEDATA structure, but is still just a lot of GDI]

```
/*
 * PolyDlgProc
 *
 * Purpose:
 *  Dialog procedure for a window in which to display the Polyline
 *  for editing.  This pretty much handles all editing functionality
 *  for the embedded object.
 */

BOOL __export FAR PASCAL PolyDlgProc(HWND hDlg, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    LPCPolyline     ppl=NULL;
    WORD            w1, w2;
    HWND            hWnd;
    RECT            rc;
    POINT           pt;
    UINT            uID, uTemp;
    UINT            cx, cy;

    w1=(WORD)GetProp(hDlg, PROP_SELECTOR);
    w2=(WORD)GetProp(hDlg, PROP_OFFSET);

    ppl=(LPCPolyline)MAKELP(w1, w2);

    switch (iMsg)
        {
        case WM_INITDIALOG:
            ppl=(LPCPolyline)lParam;
            ppl->m_hDlg=hDlg;

            SetProp(hDlg, PROP_SELECTOR, (HANDLE)SELECTOROF(ppl));
            SetProp(hDlg, PROP_OFFSET,   (HANDLE)OFFSETOF(ppl));

            //Center the dialog on the screen
            cx=GetSystemMetrics(SM_CXSCREEN);
            cy=GetSystemMetrics(SM_CYSCREEN);
            GetWindowRect(hDlg, &rc);
            SetWindowPos(hDlg, NULL, (cx-(rc.right-rc.left))/2
                , (cy-(rc.bottom-rc.top))/2, 0, 0, SWP_NOZORDER | SWP_NOSIZE);

            //Create the Polyline to exactly cover the static rectangle.
            hWnd=GetDlgItem(hDlg, ID_POLYLINERECT);
            GetWindowRect(hWnd, &rc);
            SETPOINT(pt, rc.left, rc.top);
            ScreenToClient(hDlg, &pt);

            //Set the polyline just within the black frame
            SetRect(&rc, pt.x, pt.y, pt.x+(rc.right-rc.left)
                , pt.y+(rc.bottom-rc.top));
            InflateRect(&rc, -1, -1);

            //Try to create the window.
            ppl->m_pIPolyline->Init(hDlg, &rc, WS_CHILD | WS_VISIBLE, ID_POLYLINE);

            //Set the initial line style radiobutton.
            ppl->m_pIPolyline->LineStyleGet(&uTemp);
            CheckRadioButton(hDlg, ID_LINESOLID, ID_LINEDASHDOTDOT
                , uTemp+ID_LINEMIN);

            ppl->SendAdvise(OBJECTCODE_SHOWOBJECT);
            ppl->SendAdvise(OBJECTCODE_SHOWWINDOW);
            return TRUE;


        case WM_COMMAND:
            uID=LOWORD(wParam);

            switch (uID)
                {
                case IDOK:
                    RemoveProp(hDlg, PROP_SELECTOR);
                    RemoveProp(hDlg, PROP_OFFSET);

                    if (NULL!=ppl)
                        {
                        ppl->SendAdvise(OBJECTCODE_HIDEWINDOW);
                        ppl->SendAdvise(OBJECTCODE_CLOSED);
                        ppl->m_hDlg=NULL;
                        }

                    EndDialog(hDlg, TRUE);
                    break;

                case ID_UNDO:
                    if (NULL!=ppl)
                        ppl->m_pIPolyline->Undo();
                    break;
                case ID_COLORLINE:
                case ID_COLORBACK:
                    if (NULL!=ppl)
                        {
                        UINT            i;
                        COLORREF        rgColors[16];
                        CHOOSECOLOR     cc;

                        //Invoke the color chooser for either color
                        uTemp=(ID_COLORBACK==uID)
                            ? POLYLINECOLOR_BACKGROUND : POLYLINECOLOR_LINE;

                        for (i=0; i<16; i++)
                            rgColors[i]=RGB(0, 0, i*16);

                        _fmemset(&cc, 0, sizeof(CHOOSECOLOR));
                        cc.lStructSize=sizeof(CHOOSECOLOR);
                        cc.lpCustColors=rgColors;
                        cc.hwndOwner=hDlg;
                        cc.Flags=CC_RGBINIT;
                        ppl->m_pIPolyline->ColorGet(uTemp, &cc.rgbResult);

                        if (ChooseColor(&cc))
                            {
                            //rgColor is just some COLORREF pointer
                            ppl->m_pIPolyline->ColorSet(uTemp, cc.rgbResult
                                , rgColors);
                            }
                        }
                    break;

                case ID_LINESOLID:
                case ID_LINEDASH:
                case ID_LINEDOT:
                case ID_LINEDASHDOT:
                case ID_LINEDASHDOTDOT:
                    if (NULL!=ppl)
                        ppl->m_pIPolyline->LineStyleSet(uID-ID_LINEMIN, &uTemp);

                    break;
                }
            break;

        case WM_CLOSE:
            //Close just like we hit the "Close" button.
            SendCommand(hDlg, IDOK, 0, 0);
            break;
          }
       return FALSE;
       }
```

IOLEOBJ.CPP

```
...


STDMETHODIMP CImpIOleObject::DoVerb(LONG iVerb, LPMSG pMSG
   , LPOLECLIENTSITE pActiveSite, LONG lIndex, HWND hWndParent
   , LPCRECT pRectPos)
   {
   switch (iVerb)
      {
      case OLEIVERB_HIDE:
         if (NULL!=m_pObj->m_hDlg)
            ShowWindow(m_pObj->m_hDlg, SW_HIDE);
         break;

      case OLEIVERB_PRIMARY:
      case OLEIVERB_OPEN:
      case OLEIVERB_SHOW:
         //If we already have a window, just make sure it's showing
         if (NULL!=m_pObj->m_hDlg)
            {
            ShowWindow(m_pObj->m_hDlg, SW_NORMAL);
            SetFocus(m_pObj->m_hDlg);
            return NOERROR;
            }

         //This stores the dialog handle in m_pObj.
         DialogBoxParam(hgInst, MAKEINTRESOURCE(IDD_EDITDIALOG)
            , hWndParent, PolyDlgProc, (LPARAM)m_pObj);
         break;

      default:
         return ResultFromScode(OLEOBJ_S_INVALIDVERB);
      }

   return NOERROR;
   }
```

[All others are standard implementations as we have seen before.]

Listing 11-2: Important modifications and additions to Polyline to be an In-Process Server.

Many of the modifications to Polyline are minor as most of the code to support embedded objects comes in the form of additions. Both types of changes are detailed below.

- The CPolyline class now manages additional interface pointers as well as pointers into the default handler *m_pDef<Interface>* exactly like the CFigure class in HSchmoo. In fact, much of Polyline's code was taken from HSchmoo and so has not been shown again in Listing11-2. It also maintains the window handle of the dialog we use to implement IOleObject::DoVerb. These additional variables are initialized to NULL in the CPolyline constructor and set to their real values in CPolyline::FInit through QueryInterface on an IUnknown as returned from OleCreateDefaultHandler. This is also identical to that shown earlier for HSchmoo.
- RESOURCE.H now contains many more identifiers for additions to POLYLINE.RC: an icon (for the default icon), some strings we use in UI, and a dialog box used for editing the figure
- The function SendAdvise was added to CPolyline that takes a notification code and calls the appropriate member function in IOleClientSite, IOleAdviseHolder, or IDataAdviseHolder. The most frequent notification that required a number of changed in Polyline is OnDataChange (generated by calling CPolyline::SendAdvise with OBJECTCODE_DATACHANGED). Any member function in IPolyline (IPOLYLIN.CPP) that changes data, such as LineStyleSet and ColorSet, have been modified to send data changes, as has CPolyline::DataSet itself. In addition, we send this code when the user clicks in the Polyline window thus adding a point to the figure.
- Member functions of IPersistStorage (IPERSTOR.CPP, not shown) now, in addition to their normal operation, call the default handler's IPersistStorage interface to allow the cache to do what caches do.
- Polyline's IDataObject now implements GetDataHere for CF_EMBEDDEDOBJECT and can now delegate the implementation of EnumFormatEtc to the default handler, thereby eliminating all the cumbersome enumerator code.
- An implementation of IViewObject was added that is virtually identical to the one for HSchmoo shown in Listing 11-1 and therefore is not shown here.
- CPolyline's QueryInterface now includes the additional interfaces (IViewObject and IOleObject) that it implements and also returns the default handler's IOleCache interface, but no others.
- Most of the implementation of IOleObject is identical to ones we have already seen with the exception of DoVerb which does oddly invokes a dialog box. PolyDlgProc in POLYWIN.CPP is the dialog procedure for the one DoVerb invokes.

These last two changes are the most important ones. Let me first explain the handling of QueryInterface on the object. When we implemented a handler we were selectively overriding just those interfaces we needed to augment, passing everything else to the default handler. If we did not know about an interface in QueryInterface we just passed it to the default handler's IUnknown::QueryInterface. We had no idea what interfaces were requested or returned.

For an in-process server you no longer want to delegate QueryInterface to the default handler for just any interface. The simple reason is that OleCreate functions will fail for your CLSID if there is no local server registered. Therefore you cannot be inserted as a new object in a container. But of course there is no local server because we're an in-process server! What happens is that the default handler supports more interfaces than we want to show from a default handler, including some OLE 2.0 internal ones. OleCreate will QueryInterface for one of these internal interfaces through which it attempts to run the *local* server. This will, of course, fail because there is no local server, and so OleCreate returns REGDB_E_CLASSNOTREG. This is not at all what you would expect because you have your InProcServer key registered and that's all you need. Figuring this out was one of those cases that make you seriously consider becoming a

So the bottom line is that you have to exercise more control over what interfaces you expose because as an in-process server you are totally responsible for what is and is not implemented. In Polyline we only pass QueryInterface on to the default handler for IOleCache, which essentially means that we are directly using the cache's IOleCache. We do use other interfaces in the handler but we do not expose them directly, instead filtering calls through our own implementations first. Note that passing QueryInterface through like this is no more effective than doing a QueryInterface in CPolyline::FInit like we do for all the other defaults, storing that pointer in a variable like *m_pDefIOleCache*, and returning that pointer here.

That leaves us with Polyline's implementation of IOleObject::DoVerb which looks much like any other except that instead of just showing an already existing but hidden window like we did in Schmoo we have to create the whole editing user interface right here.

The question of user interface is a very important one to answer when you are designing an in-process server. Since you cannot be run stand-alone, it's not a good idea to show a window that looks like other stand-alone applications. When I first began this chapter's modifications on Polyline I tried to duplicate Schmoo's user interface with a frame window, a GizmoBar, menus, and the like. But there are specific technical difficulties with creating this sort of window from within a DLL, the most important of which is that you have no message loop to call your own. Therefore you can have no keyboard accelerators and you cannot use an MDI interface (which is absurd from an in-process server anyhow). Keystrokes like Alt+Backspace, which we used in Schmoo for Undo, are meaningless, as are those for clipboard operations since in general, in-process servers don't need to mess with the clipboard since they don't run stand-alone (but there is nothing stopping you, of course).

The better user interface for an in-process server is that of a modal dialog box which therefore makes your object look very much like a part of the container application. Since you are invoking this dialog from a DLL that has already been loaded, such a window will appear very fast after the end-user double-clicks the object in the container. This further reinforces the idea that the dialog belongs the container. Dialog boxes also give you a lot of support, such as keyboard mnemonics on controls, that you would otherwise not get without your own message loop. As you can see from the code in Listing 12-2, it didn't take much to implement such a dialog in Polyline that provides all the editing capabilities of the full Schmoo application, just through the dialog-box interface shown in Figure 12-3.

§

Figure 12-3: Polyline's dialog-box user interface for editing an embedded object.

Since the dialog is declared with DS_MODALFRAME and because we call DialogBoxParam in DoVerb with the hWnd of the container as the parent, this dialog box is modal to the container. Note also that we *center* the dialog box on the *screen* instead of letting it be placed relative to the container window as is the norm. This is because the default placement will typically cause the dialog box to cover the container's site, and usually the first thing an end-user is going to do in this context is move the dialog. Placing it at the center of the screen will generally keep the site visible and your server more usable.

Other than that, the dialog processes commands like any other, changing the Polyline's line style or invoking the Choose Color dialog to change background and line colors. It also sends the appropriate notifications when closing the dialog as we would when closing a document window in a server application. Note that we don't need to call IOleClientSite::SaveObject because we aren't unloading the server, we're just closing the dialog. We had to do this in an EXE server because the application would generally shut down when the user interface went away and that would mean the data is lost as well. But since this is a DLL, we're going to stay in memory along with the object's most current data, data that will be used in subsequent calls to IViewObject::Draw and the like. When the container wants to actually save us they'll call OleSave which will call our IPersistStorage::Save in which we'll save our current data. Therefore no SaveObject call is necessary.

Summary

In-process servers and in-process handlers are both structurally identical as they both export the standard component object functions of DllGetClassObject and DllCanUnloadNow. Furthermore, to work with embedded objects, they serve objects with the IOleObject, IDataObject, IViewObject, IPersistStorage, and IOleCache interfaces. Therefore a container application is not cognizant of what sort of DLL module is in use for a specific object.

Handlers and DLL servers only differ in the extent to which they implement the interfaces on the embedded objects. Handlers, in general, implement as little as possible to remain small, lightweight modules that can provide specific optimizations for an object. For example, they can render data more specifically for devices because handlers, like in-process servers but unlike local servers, can implement IViewObject and therefore have direct access to the hDC on which the container wants the object displayed. Therefore the handler can obtain information about the device and optimize its output accordingly. As a simple example of this the HSCHMOO example in this chapter, a handler for Schmoo, checks to see if the printer is color-capable, and if not, renders its data in black and white only.

In-process modules also increase performance since they eliminate much of the need to run and communicate via LRPC to a local server as well as cutting out the middlemen. Since object handlers can, through IViewObject, provide for all rendering necessities, they can eliminate the need for a local server to render metafile or bitmaps as well as the need for OLE 2.0 to cache those same presentations. The reduced dependence on the cache can greatly reduce the amount of space an object requires in a container's compound file. So not only do you benefit from increased performance, so does your container.

As handlers implement only select interface member functions they usually rely on the presence of an local server to fulfill all requests on the object, especially those for editing. Handlers are designed to still allows for display, optimized printing, and copying of objects without the local server anywhere in sight.

In-process servers, on the other hand, implement enough interfaces and member functions to completely eliminate the need for a local server. That means that they provide for all drawing, all data transfer, all storage management, and all editing facilities. The latter requirement, editing, means that the in-process server will need to display windows of some sort from within their implementation of IOleObject::DoVerb. However, there are some technical details that make a window shown in a DLL different from one shown from an EXE, most notably the lack of a message loop you control. Therefore a dialog-box user interface is more applicable to in-process servers than a normal overlapped application-type window.

There are also a few interoperability issues to note when considering to use a DLL module. Since DLLs written for a 16-bit world, like Windows 3.1, cannot be loaded into a 32-bit process, as under Windows NT, you may need to provide DLLs for both or limit yourself to what container's can work with you. In addition, DLLs written for OLE 2.0 can only be used with containers written for OLE 2.0 and therefore you lose compatibility with OLE 1.0 containers which OLE 2.0 local servers keep.

Certainly there are trade-offs: either suffer in interoperability, suffer in performance, or suffer the burden of maintaining multiple versions of your code to suit the various environments you want to support.

Nonetheless, this chapter shows how to implement a basic handler as well as details the modifications made to the Polyline object, last seen in Chapter 6, to make it an in-process server complete with editing facilities.