Chapter Ten
Compound Documents & Embedded Object Servers (EXE)

COO-KIES!
Cookie Monster

I mentioned in Chapter 9 that embedded objects are like a batch of cookies in a cookie jar, and that chapter discussed how to create the cookie jar. That leaves us here with the problem of baking cookies. At least some of us I'm sure have at one time or another got out all the flour, sugar, eggs, butter, shortening, baking soda, salt, and vanilla (and maybe an obscene amounts of chocolate chips) to mix up and bake a batch of cookies. (OK, I'll admit that once or twice I didn't bother to bake them.) We later remove them from the oven, let them cool, and put them into the cookie jar.

Baking cookies shows exactly what's involved in creating an OLE 2.0 server for embedded objects. As bakers, we first create the cookies, by mixing the dough, then manage the cookies by baking, cooling, and storing them. As bakers we are cookie servers. With embedded objects, a server application must also create objects and then manage those objects. Object creation, as we saw in Chapter 4, is the exact responsibility of a class factory. Object management is the exact responsibility of everything else in the application. As we saw in previous chapters, a C++ object is a very convenient way to manage a Windows Object, but you can accomplish the same thing in a language like C with well-designed data structures.

This chapter deals exclusively with EXE servers, or Local Servers, that implement the embedded object parts of the compound document picture discussed in Chapter 9. Any server components that are implemented in DLLs, that is, in-process servers and object handlers, are covered in the next chapter as they have their own structures and their own unique issues. That leaves us here to discuss the specific structure of a server application and where to implement certain interfaces. Using Schmoo as an example, this is followed by a step-by-step guide to adding basic server features in the form of a class factory and the objects that it creates.

Because OLE 2.0 defines such a rich compound document technology, we will not be able to cover ever possibility for embedded objects, leaving some functions of some interfaces to your own exploration. There are so many recipes for objects that we'd go crazy trying to look at them all at once. We need to start with the OLE equivalent of your basic sugar cookie. That will be enough to satisfy a container as well as a furry-blue-ball-eyed maniac from Sesame Street.

The Structure of a Server Application

In "Compound Document Mechanisms" in Chapter 9 we saw that a server, in order to fit the compound document standard, must provide objects that implement the IOleObject, IDataObject, and IPersistStorage interfaces as shown in Figure 10-1. In addition, the server must provide a class factory that implements the IClassFactory interface, as that class factory is responsible for creating the server's objects.

Figure 10-1: The server side of compound documents.

The three object interfaces each have their own specific purposes in the compound document picture. IPersistStorage is the interface through which the object is told about its IStorage in the container's document through which the object can load and save its native data. IDataObject is the interface through which the handler, generally OLE2.DLL, asks for presentations to cache, such as a metafile or a bitmap. Handlers may also ask the data object for an advisory connection such that they know when the data has changed in the server such that the cache may need updating. Containers as well may ask the data object for an advisory connection or they might request a copy of specific data other than a graphic format.

IOleObject is then the interface that pretty much handles all other parts of the standard. If you read the entirety of Chapter 9 you will have seen where containers use a good number of the member functions in this interface. For the most part, this interface provides containers will a wide variety of operations that the object supports, the most important of which is the action of executing a verb.

Becoming an embedding server will therefore affect your application on a number of levels, just like container modifications affected that class of applications. Server applications have levels I describe as Application, Document, and Object as shown in Figure 10-2. The application is the agent which loads and saves documents, and those documents may provide one or more objects. For example, a spreadsheet application can open a specific worksheet document, and in that document are a wide variety of cell ranges where each cell range is a potential object. Simpler applications, like drawing packages (of which Schmoo is a low-tech example) have one object (the drawing) per document.

Figure 10-2: The structure of a server application.

The application level is where we'll implement the class factory for the server. When that class factory creates an object, it will first have to create a document in which is created an object. The document is really the agent that presents objects to the outside world. Those objects need not be tightly integrated with what your application may already have as the data editor. Instead, the things presented as objects to a container use the editor code just like any other part of your application. In Schmoo we'll implement objects called Figures (in a C++ class called CFigure) which implement the compound document interfaces by *using*, instead of modifying, Schmoo's CPolyline implementation. What this means is that you need not modify what you see as 'objects' that already exists in your application as our compound document objects will, for the most part, provide another wrapper around what you already have.

Before diving into implementation, let's first look at a few issues that require a little thought.

Linking Support and Mini- vs. Full-Servers

While support for embedded objects is only half of the overall linking and embedding picture, some applications will have no need to support linking. Such servers are called mini-servers and can only run within the context of embedding, that is, the only way to run the server is to create an object it manipulates through a container's Insert Object dialog (and later, by activating the object again). Those that support linking and embedding are called full-servers

Mini-servers work well for small visual data, such as font effects or a simple drawing. Since they only support embedding, mini-servers should not require a great deal of storage since each object from them will take up space in the container's document. Simple drawings, for example, can be stored in a metafile of a few hundred bytes. On the other hand, a 24-bit DIB may take megabytes. A server whose native data is small can therefore be either a mini-server or a full-server but those that can potentially generate very large data should always be full servers. Give the end-user the choice of whether to link or embed when storage space is a factor.

Since mini-servers cannot run by themselves, there is no opportunity for it to load, edit, and same any sort of file. Since they cannot generate files, there is nothing to which anyone can link, and therefore mini-servers cannot support linking. Full-servers on the other hand, since they can run stand-alone, have some concept of files and therefore can provide links. Most existing applications that you might consider making a server probably read and write their own files already and should thus become full-servers. If, however, you are designing a new type of editable object, a mini-server may be the best route.

Version Numbers

If you ever plan to release a new version of your application, which just about all of us do, you will want to version number all instances of data that might be stored on disk. This includes the data you store as an embedded object, since that data ultimately ends up in a container's disk file. A later version of your server may be asked to load and edit an old version of your data that's been sitting undisturbed in a container's compound document for years. By having a version number in this embedded object your new server can convert the old data into the new data. Typically the first few bytes in an object's storage should contain a version number since that will be the first thing you want to read before attempting to load data. Schmoo, for example, uses the version number to know how many bytes to read, as later versions of its storage contain more than older ones.[1]

Step-By-Step Embedding Servers

The remainder of this chapter will follow modifications I made to Schmoo for it to become an embedded object server. Unlike container applications, there is very little user interface to contend with in a server; instead, most of the work is functional and deals with rending and exchanging data. In other words, the server has to define how a cookie looks, how it tastes, the list of necessary ingredients, and how to mix together all those ingredients. However, it does not have to worry about cookie jars or counters, kitchens, and houses in which such a cookie jar would sit. Its only concerns are the cookies.

These steps assume that you already have some sort of application working that you now wish to make an embedded object server. If you are writing a new application that you want to make a server, I recommend reviewing these steps, then writing the application without embedding support, then returning here to go through the step-by-step process. This approach will let you first concentrate on your application's specifics with knowledge about how embedding will affect various areas.

About a third of the steps in this implementation deal with the class factory portion of a server, that is, what's necessary to launch the server and create an object. The steps dealing with the object are primarily concerned with the three object interfaces and a small bit of user interface. The final steps are concerned with additional features important to full-server as well as some MDI and SDI differences. The steps, like those for the container in Chapter 9 if you've read them, are designed such that you can compile and test something after each step. Again I strongly recommend that you test your implementations of the early steps before moving on as the later steps use the earlier ones as a foundation. What you'll see is that by leaving out much of the implementation at the beginning you naturally simulate a number of failure conditions.

1. Call SetMessageQueue(96) (Windows 3.1 only), OleBuildVersion, OleInitialize, and OleUninitialize as described in "The New Application for Windows Objects" in Chapter 4.

2. Create compound document entries in the registration database.

3. Implement a class factory and register it when you detect "-Embedding" on the command line during startup. Include a shutdown mechanism as described in Chapter 4 under "Implementing a Component Object and Server."

4. Implement an object with IUnknown but no other interfaces and modify other code in your application to create and manage this object.

1I am referring to my OLE 1.0 version of Schmoo (called Schmoo).

5. Implement the IPersistStorage interface for the object. Much of this code can use any existing function that read and write your data to IStorage objects.

6. Implement the IDataObject interface for the object. If you have already written code to handle the clipboard through a data object, this implementation will be quite straightforward.

7. Implement the IOleObject interface for the object. This includes code to execute object verbs. This is the largest of all interfaces and has various functions that need not be implemented at all and others that are downright trivial.

8. Modify your server's user interface when editing an embedded object to eliminate or disable various menu commands as well as changing the window's caption bar to reflect the embedding state. This also affects the your implementation of File Save As and File Close.

9. Send notifications of data change, closure, saving, and renaming at various times in the object's life as well as when it's closed.

10. (Full-Servers) Augment your server's clipboard code to provide CF_EMBEDSOURCE and CF_OBJECTDESCRIPTOR such that you could copy from your server and paste an embedded object into a container.

11. (Optional, Full-Servers) Provide alternate user interface for MDI applications which adds another shutdown condition.

After following these steps in your own application, you'll end up with an embedding server fully usable by OLE 1.0 and OLE 2.0 containers, including any you implemented in Chapter 9.

Call Initialization Functions at Startup and Shutdown

As mentioned in Chapter 4 and implemented in containers in Chapter 9, all applications that use Windows Objects must call SetMessageQueue(96) (Windows 3.1 only), OleBuildVersion, and OleInitialize on startup and OleUnintialize (and possibly OleFlushClipboard) on shutdown. As servers we not only use storage objects but also advise sinks and objects in our container.

I want to again point out again that OleInitialize is necessary to work with compound documents as opposed to CoInitialize. If you have been using CoInitialize to this point, switch now to OleInitialize. You can also compile and run to verify that these are all being called at the right time.

Create Registration Database Entries

In Chapter 4 we created a number of basic registration database entries such that CoGetClassObject could locate and load the DLL or EXE that serviced an object of a particular CLSID. These entries are still required, but for compound documents there are a number of additions. The first task for embedding servers is therefore to obtain a CLSID for its use. For Schmoo we'll use {0021107-000-0000-C000-000000000046} which is defined in INC\BOOKGUID.H as CLSID_Schmoo2Figure. We use this because all the compound files that Schmoo generates from Chapter 5 and beyond are already marked with this CLSID. We'll just use it now for our object CLSID as well.

Let's now review those entries from Chapter 4 where *{classID}* in this context is your CLSID value spelled out as in "{0021107-000-0000-C000-000000000046}". The first set of entries went under the key <*ProgID*> where <*ProgID*> is a short name without spaces or punctuation, like an OLE 1.0 class name (\ means HKEY_CLASSES_ROOT):[1]

```
\
    <ProgID> = <Main Descriptive Object Name>
        CLSID = <class ID>
```

1A good technique to follow to avoid ProgID conflicts as well as create a unique identifier is to use *<App><VersionNumber><ObjectType>* as in Schmoo2Figure. The application name can also identify your company for even less possibility of conflicts.

where *<Descriptive Object Name>* is a user-readable name that will appear in the Insert Object dialog box in container applications and other user interface areas. Besides this basic entry, we also had entries under CLSID (shown for a server EXE):

```
\
   CLSID
      <class ID> = <Main Descriptive Object Name>
         LocalServer = <Path to server EXE>
```

For this chapter's version of Schmoo, the first set of entries (for ProgID) contains the following:

```
\
   Schmoo2Figure = Schmoo 2.0 Figure (Chap 10)
      CLSID = {0021107-000-0000-C000-000000000046}
      Insertable
      protocol
         StdFileEditing
            server=<Path to schmoo.exe>
            verb
               0 = &Edit
```

The new entry with the key "Insertable" and no value marks this object as one that can appear in a container's Insert Object dialog. Only those classes registered with Insertable will be shown in that dialog. The other entries under protocol\StdFileEditing are for OLE 1.0 compatibility; they are, in fact, the same entries used by OLE 1.0 servers. By this small amount of registration an OLE 2.0 server is usable from an OLE 1.0 container.

The second set of entries under CLSID grows tremendously for compound document servers:

```
\
   CLSID
      {0021107-000-0000-C000-000000000046}= Schmoo 2.0 Figure (Chap 10)
         LocalServer = <Path to schmoo.exe>
         ProgID = Schmoo2Figure
         Insertable
         DefaultIcon = <Path to schmoo.exe>,0
         verb
            0 = &Edit,0,2
            -1 = Show,0,0
            -2 = Open,0,0
            -3 = Hide,0,1
         AuxUserType
            2 = Schmoo 2.0
            3 = Schmoo 2.0 from Chapter 10
         MiscStatus = 0
            1 = 1
         DataFormats
            GetSet
               0 = Polyline Figure,1,1,3
```

1 = Embed Source,1,8,1
2 = 3,1,32,1
3 = 2,1,16,1

Obviously there is a lot of new stuff here, so let's take them on one at a time, skipping the LocalServer entry that we already understand.

Key	Subkeys and Values
ProgID	Provides the name of the key under which other entries may be found. This value must match the ProgID registered elsewhere.
Insertable	Marks the object as one that can appear in a container's Insert Object dialog. This is redundant with the Insertable that appears under the other entries for the server, and both should always appear together.
DefaultIcon	A path to a module and an index of the icon in that module to use by default when the user checks "Display as Icon" in the Insert Object and Paste Special dialogs of a container. If no entry is given here or the path is invalid then the dialogs will show a default icon (a sheet of paper with an edge folded down, your standard 'document' type icon).
AuxUserType	This key has no value itself but instead has subkeys of the format *<form number>* = *<string>* where *<form number>* is either a 2 or a 3 (but never a 1 for reasons only known to the gods[1]) and *<string>* is some user-readable name of the object. Form number 2 should always be a short name that describes the type of the object, as in "Schmoo 2.0". Try to keep this string under 10 characters. Form number 3 is a longer application name like "Schmoo 2.0 from Chapter 10" which is used specifically in the OLE2UI Paste Special dialog.
MiscStatus	Can have one value called the "default" status and can optionally have subkeys where each has the form *<aspect>* = *<status>* where *<aspect>* is a DVASPECT value and *<status>* is an integer flag. The status values come from the OLEMISC enumeration in OLE2.H and can be any combination of the values in the enumeration. In the example above, "1 = 1" marks DVASPECT_CONTENT with the flag OLEMISC_RECOMPOSEONRESIZE which means the object would like to redraw itself if a content presentation is resized in a container.

I left descriptions of **verb** and **DataFormats** off the table above because they are somewhat detailed. The purpose of registering these, as well as many of the others, is such that OLE 2.0 and container application can determine specific capabilities of an object without having that object running. For example, when a container loads an object it wants to display the list of verbs the user can invoke on that object, and only when the user does select a verb is the object put into the running state. It would be silly to run the object just to determine the verbs, since the user may look at the verbs and decide not to invoke them at all. In addition, a good number of functions in the IOleObject interface return the same information as you register here. By registering these values you can trivialize the implementation of these member functions as we will see later.

1Actually number 1 is always the string after the *<class ID>* in this set of entries, and so we don't need to store it here.

The "verb" key itself has no value. It only has any number of subkeys below it where each subkey and value is of the format *<verb number> = <name>,<menu flags>,<verb flags>*. Verb numbers are those defined with OLEIVERB prefixes OLE2.H as well as any server-defined values. Those with negative numbers are pre-defined verbs such as OLEIVERB_HIDE (-3), OLEIVERB_OPEN (-2), and OLEIVERB_SHOW (-1) where those with zero or positive numbers are defined by the application (number 0 is defined as OLEIVERB_PRIMARY as well).

Each verb number key has three components in its value: verb name, menu flags, and verb flags. First, the name of the verb is a string which will appear in a container's menus. The OLE 2.0 user interface recommends that all verbs numbered zero and above include a mnemonic character in these names, that is, one character prefixed with an ampersand as in "&Edit." Next, the "menu flags" is a combination of various MF_* values from windows.h. Most often this is MF_STRING | MF_ENABLE | MF_UNCHECKED which translates into a zero (hence the entries shown above). Generally this is used to control enabling of the menu items, so don't register some values that make no sense, like MF_BITMAP or MF_OWNERDRAW. Finally, the "verb flags" is a combinations of values from the OLEVERBATTRIB enumeration in OLE2.H. In OLE 2.0 this enumeration contains two bits: OLEVERBATTRIB_NEVERDIRTIES (value of 1) indicates that executing this verb on an object does not have the possibility of modifying the data, and OLEVERBATTRIB_ONCONTAINERMENU (value of 2) indicates whether a container should show this verb in its menus. According to the OLE 2.0 specification, Open, Show, and Hide should never use this latter flag, but you should still register them with either a 0 or 1 value for the verb flags. For Schmoo, Open and Show allow modifications whereas Hide does not (of course not!). The other verb, Edit, can be shown on container menus and also has the possibility of modification. A verb like Play (on a sound object) would probably have a value of 3 here since it appears on menus but doesn't allow modification.

DataFormats is another special entry that describes all the different clipboard formats that an object of this class can exchange. **DataFormats** can have one or more subkeys where the first is key 0, the second key 1, and so on. These values only determine the priority of the formats and have no meaning otherwise, so the format with the highest fidelity (such as your application-specific format) should have key 0, and better graphic formats like a metafile should have a lower-values key number than a bitmap.

Each value for the subkeys has four components: format, aspect, medium, and flag. First is the number of the clipboard format if there is a CF_* entry for it in windows.h, or the string name of the format if it's registered. In Schmoo's registration above, the first two entries have registered formats of "Polyline Figure" and "Embed Source" (the latter is an IStorage with the objects data) whereas the last two have values of 3 (CF_METAFILEPICT) and 2 (CF_BITMAP). The next component, the aspect, is the combination of any DVASPECT values where -1 means "all aspects." All the sample entries are registered with a value of 1 for DVASPECT_CONTENT. If, for example, we could also provide CF_METAFILEPICT with DVASPECT_ICON (defined as 4) then the value would be 5 (DVASPECT_ICON | DVASPECT_CONTENT). The third component contains all the types of storage mediums in which the server can render the format. This value can be any combination from the TYMED enumeration in DVOBJ.H. For "Polyline Figure," for example, we have a 1 which is TYMED_HGLOBAL. "Embed Source" is 8, or TYMED_ISTORAGE, CF_METAFILEPICT is 32 for TYMED_MFPICT, and CF_BITMAP is 16 for TYMED_GDI. If we could render something in two mediums like TYMED_ISTORAGE | TYMED_ISTREAM the value would be 12. Finally, the "flag" for each format is a combination of DATADIR values from the (what else) DATADIR enumeration in DVOBJ.H. If you can ask the object for the format, that's a "Get" direction and has the value 1 (DATADIR_GET). If you can send the object the format but not retrieve it, use 2 (DATADIR_SET). If the object can exchange the format in both direction (as with "Polyline Figure") use the combined value of 3.

A primary benefit of this registration is to again prevent OLE 2.0 from having to run your application to learn such information when a container wants to see if your object supports exchange using a specific format. In fact, OLE2.DLL as the default handler will implement IDataObject::EnumFormatEtc using this information when that call is made from a container, must less bother than launching your server. Even if you are running, you can still tell OLE 2.0 to do implement the enumerator for you based on these entries by returning OLE_S_USEREG from this function. Making these entries is sure easier than implementing an enumerator, but you still have to take this latter route if you need to dynamically change what's enumerated. A rare but still possible occasion.

With all of your entries in the registration database you will now appear in a container's Insert Object dialog. If you select your server's name in the dialog (and hit OK) then OLE 2.0 will find the path in the registration database and launch your application. Of course, nothing else will work, but you can at least verify that this part of the process is working. So now let's see what we should do on startup when we're launched like this.

Implement and Register a Class Factory

In Chapter 4 we learned how to implement a class factory in a simple EXE server called EKoala. The "Koala" objects that this class factory created were pretty simple: they had only the one IPersist interface. We learned how to register the class factory from an EXE server by calling the CoRegisterClassObject function in COMPOBJ.DLL as well as how to provide an unloading mechanism that would insure the server shut down properly when there were no more objects to server.

An embedding server in the compound document scenario is really the same thing as a very simple server like EKoala. The difference is *solely* in what sort of object the server provides. In the case of EKoala we had an object with one functional interface and no user interface. An embedded object, on the other hand, has at least three interfaces, IOleObject, IPersistStorage, and IDataObject, and does require some user interface through which the end user can make changes to that object. But since the difference is in the *object*, most parts of the *server* that deal with the class factory are the same as before. We can see these difference in changes made to Schmoo on the "application" (or server) level as shown in Listing 10-1. These changes include modifications to SCHMOO.H and SCHMOO.CPP as well as the addition of two new files, SCHMOOLE.H and ICLASSF.CPP.

The C++ object we use to implement our embedded objects is called CFigure which we'll see in the next section. The class factory is thus called CFigureClassFactory. Since most of this latter class is the same as other class factories we've seen the code listing does no show its definition (the only additions to the class itself are a pointer variable to the CSchmooFrame called *m_pFR* and a BOOL member called *m_fCreated*) or any of its implementation outside of its CreateInstance member. In addition, the listing does not show parts of SCHMOO.CPP that are identical to other server code we've seen, such as the global variables and ObjectDestroyed function we use to manage provide the unloading mechanism.

SCHMOO.H

[Other lines omitted]

```
class __far CSchmooFrame : public CFrame
    {
    friend class CFigureClassFactory;
    friend class CFigure;   //For UI purposes.
    private:
        ...

        BOOL            m_fEmbedding;       //-Embedding on command line?
        DWORD           m_dwRegCO;          //From CoRegisterClassObject
        LPCLASSFACTORY  m_pIClassFactory;

    protected:
        ...

        virtual void      ParseCommandLine(void);

        ...

    public:
        ...

        virtual void      UpdateEmbeddingUI(BOOL, LPCDocument, LPCSTR, LPCSTR);
    };
```

SCHMOO.CPP

```
[Other code omitted]
/*
 * CSchmooFrame::CSchmooFrame
 * CSchmooFrame::~CSchmooFrame
 */

CSchmooFrame::CSchmooFrame(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    : CFrame(hInst, hInstPrev, pszCmdLine, nCmdShow)
    {
    ...
    char    szTemp[256];

    m_fInitialized=FALSE;

    m_fEmbedding=FALSE;
    //THis function is in OLE2UI
    ParseCmdLine(m_pszCmdLine, &m_fEmbedding, szTemp);
    m_dwRegCO=0;
    m_pIClassFactory=NULL;
    return;
    }

CSchmooFrame::~CSchmooFrame(void)
    {
    UINT            i;

    //Opposite of CoRegisterClassObject, takes class factory ref to 1
    if (0L!=m_dwRegCO)
        CoRevokeClassObject(m_dwRegCO);

    //This should be the last ::Release, which frees the class factory.
    if (NULL!=m_pIClassFactory)
        m_pIClassFactory->Release();

    ...

    return;
    }


/*
 * CSchmooFrame::FInit
 *
 * Purpose:
 *  Call OleInitialize then calling down into the base class
 *  initialization.
 */

BOOL CSchmooFrame::FInit(LPFRAMEINIT pFI)
    {
    HRESULT     hr;

    [OleInitialize et. all omitted from listing]
    ...

    if (m_fEmbedding)
        {
        m_pIClassFactory=new CFigureClassFactory(this);

        if (NULL==m_pIClassFactory)
            return FALSE;

        //Since we hold on to this, we should AddRef it.
        m_pIClassFactory->AddRef();
        hr=CoRegisterClassObject(CLSID_SchmooFigure
            , (LPUNKNOWN)m_pIClassFactory, CLSCTX_LOCAL_SERVER
            , REGCLS_SINGLEUSE, &m_dwRegCO);

        if (FAILED(hr))
            return FALSE;
        }

    return CFrame::FInit(pFI);
    }



/*
 * CSchmooFrame::FPreShowInit
 *
 * Purpose:
 *  Called from FInit before intially showing the window.  We do whatever
 *  else we want here, modifying m_nCmdShow as necessary which affects
 *  ShowWindow in FInit.
 */

BOOL CSchmooFrame::FPreShowInit(void)
    {
    [Other code omitted]

    //Save the window handle for shutdown if necessary.
    g_hWnd=m_hWnd;

    //If we're -Embedding, don't show the window initially.
    if (m_fEmbedding)
        m_nCmdShow=SW_HIDE;

    return TRUE;
    }



/*
 * CSchmooFrame::ParseCommandLine
 *
 * Purpose:
 *  Allows the application to parse the command line and take action
 *  after the window has possibly been shown.  For a compound document
 *  server we need to just make sure that if -Embedding is there that
 *  we take no file action.  FPreShowInit has already handled the
 *  window visibility.
 */
```

```
void CSchmooFrame::ParseCommandLine(void)
    {
    //If -Embedding was there, prevent any  attempt at loading a file.
    if (m_fEmbedding)
        return;

    CFrame::ParseCommandLine();
    return;
    }
```

ICLASSF.CPP

```
[Other code omitted]

CFigureClassFactory::CFigureClassFactory(LPCSchmooFrame pFR)
    {
    m_cRef=0L;
    m_pFR=pFR;
    m_fCreated=FALSE;
    return;
    }

    ...


/*
 * CFigureClassFactory::CreateInstance
 *
 * Purpose:
 *  Instantiates a Figure object that supports embedding.
 */

STDMETHODIMP CFigureClassFactory::CreateInstance(LPUNKNOWN punkOuter
    , REFIID riid, LPVOID FAR *ppvObj)
    {
    LPCSchmooDoc        pDoc;
    HRESULT             hr;

    *ppvObj=NULL;

    //Great idea to protect yourself from multiple creates here.
    if (m_fCreated)
        return ResultFromScode(E_UNEXPECTED);
    m_fCreated=TRUE;
    hr=ResultFromScode(E_OUTOFMEMORY);

    //We don't support aggregation
    if (NULL!=punkOuter)
        return ResultFromScode(CLASS_E_NOAGGREGATION);

    //Try creating a new document, which creates the object.
    pDoc=(LPCSchmooDoc)m_pFR->m_pCL->NewDocument(TRUE, m_pFR->m_pAdv);

    if (NULL==pDoc)
        {
        //This will cause shutdown as the object count will go to zero.
        g_cObj++;
        ObjectDestroyed();
        return hr;
        }

    //Insure the document is untitled, then get the requested interface.
    pDoc->ULoad(TRUE, NULL);
    pDoc->m_pFigure->FrameSet(m_pFR);
    hr=pDoc->m_pFigure->QueryInterface(riid, ppvObj);

    //Closing the document will destroy the object and cause shutdown.
    if (FAILED(hr))
        {
        m_pFR->m_pCL->CloseDocument(pDoc);
        return hr;
        }

    return NOERROR;
    }
```

Listing 10-1: Changes and additions to Schmoo on the "application" or "server" level to support embedded objects.

The following two sections explain what all of this is for with the exception of the function UpdateEmbeddingUI which will be covered in "Modify the Server's User Interface" later on.

The Class Factory for Embedded Objects

There are only a few differences between the class factory shown here and those we've seen for component objects. First is that here the class factory's constructor is given a pointer to the application's frame structure. We need this in Schmoo because IClassFactory::CreateInstance will ask the frame and its client window to create a new document in which will be a new object. This is somewhat different than before but is necessary since the object were creating here involves user interface, and therefore we need some way to create a window in which to display that object.

The second difference is that in our implementation of IClassFactory::CreateInstance we remember if this class factory has already created one object through the variable *m_fCreated*. I have this here because we're going to register this class factory for single use with REGCLS_SINGLEUSE, but that flag only determines whether or not COMPOBJ.DLL will launch another instance of your EXE from CoGetClassObject. If someone obtains a pointer to our class factory through a single CoGetClassObject call, they could potentially call our CreateInstance more than once. For simplicity we only want to service a single embedded object from one instance of Schmoo, and so we prevent the outside world from asking us for more. There are a few complications in servicing multiple embedded objects (mostly with user interface) that are covered in the last section of this chapter "MDI Servers, User Interface, and Shutdown."

You'll also notice that if the value of *punkOuter* passed to CreateInstance is non-NULL, we fail with E_NOAGGREGATION. To allow aggregation or not is up to you, but I see little point in supporting it here so I fail if someone attempts to aggregate. This, of course, makes the actual object implementation rather atheistic as it never acknowledges the existence of a higher unknown.

If you are adding code to an application you can implement all the parts of the class factory as shown here for Schmoo except for the part of IClassFactory::CreateInstance that instantiates an object, because you don't have an object yet. For now, start shutdown as appropriate (Schmoo does this by calling ObjectDestroyed) and return ResultFromSCode(E_OUTOFMEMORY) which is a good simulation of what might happen if real object creation fails. Note that in such a failure situation your server should free itself from memory by exiting WinMain one way or another.

Startup with -Embedding

Back in Chapter 4 I mentioned that when a server is launched to service an object COMPOBJ.DLL will send "-Embedding" on the command line as an indication. Things are no difference for an embedded object server, so Schmoo detects the presence of the flag in it's function frame constructor and sets it's *m_fEmbedding* flag appropriately using the ParseCmdLine function in the OLE2UI library:

```
ParseCmdLine(m_pszCmdLine, &m_fEmbedding, szTemp);
```

ParseCmdLine checks for -Embedding as well as /Embedding and removes all whitespace from the command line before parsing. Checking for both - and / variations and ignoring whitespace is essential to work with OLE 1.0 containers who may send such variations.

This flag is later used in CSchmooFrame::PreShowInit and CSchmooFrame::ParseCommandLine which are called in this order (both are overrides of default implementation in CLASSLIB's CFrame). PreShowInit first of all saves the window handle in *g_hWnd* so the ObjectDestroyed function can post it a WM_CLOSE for shutdown. In addition, PreShowInit checks if *m_fEmbedding* is set, and if so, forces the initial ShowWindow parameter to SW_HIDE irrespective of its original value. This insures that the server's main window, and thus all other child windows within it, remain hidden until the server is specifically told to show those windows. This is part of the compound document contract: the server may be asked to silently load an object and provide an updated rendering (say a metafile) of the object's data. By silently I mean that the server is launched, asked for the rendering, and immediately closed without ever having shown itself. If the server is launched and later asked to edit the object, then it can show itself as we will see below in "Implement IDataObject and IOleObject Interfaces."

The ParseCommandLine function is called from CLASSLIB's CFrame::FInit function at the end of all other initialization with the purpose of loading any document that was listed on the command line. We need to override this function such that if *m_fEmbedding* is set we do not attempt to load anything. Otherwise we would attempt to load a file called "-Embedding" which will invariably fail and throw up a terribly confusing message box that says "Cannot load file." Now there's something I'd call user friendly especially if this was all happening in the context of the Insert Object dialog in a container application.

The other place we use *m_fEmbedding* is in CSchmooFrame::Finit where it determines whether or not we create and register a class factory. If the end-user launches Schmoo from the shell then -Embedding is not present and so we don't need to worry about servicing an embedded object at all. Therefore we have no need for the class factory. In addition, Schmoo uses REGCLS_SINGLEUSE such that OLE 2.0 will launch another instance for each embedded object the end user wants to manipulate.

NOTE: MDI applications that can service multiple objects in a single instance of the application may want to create and register a class factory even when -Embedding does not appear on the command line. See "MDI Servers, User Interface, and Shutdown" below for more details.

The bottom line is that handling -Embedding involves four steps:

1. On startup look for -Embedding on the command line and set a flag to indicate its presence or absence.
2. If your flag is TRUE, create and register your class factory. Otherwise, only register the class factory if you can handle multiple objects.
3. If your flag is TRUE, do not initially show your application window. Otherwise obey the nCmdShow parameter passed to WinMain.
4. If your flag is TRUE, prevent the rest of your application's code from trying to load "-Embedding" as an initial file.

When you're all done adding the class factory and placing the code to handle -Embedding, you can compile and test first of all that you detect -Embedding properly and register your class factory as well when appropriate. An easy way to test this is to run your program using Program (or File) Manager's File Run command which allows you to add command-line arguments. You can toss -Embedding there and see what happens. If all goes well you'll have you application in memory but hidden, in which case you have no way to close the thing unless you purge it with a tool like WPS.EXE (in the OLE 2.0 SDK). But hey, you know that it works.

You can also test your unloading mechanism by using Insert Object in a container. In this situation you'll again be launched with -Embedding (for real this time), and if you've testing your code with a fake -Embedding you'll know that your initialization and class factory registration works fine. So after you're launched you should see a call to IClassFactory::CreateInstance. Since you failed to create an object and you have no other locks or objects, your shutdown should kick in and purge your app from memory.

Implement an Initial Object with IUnknown

Having a class factory is pretty pointless unless you have some object for it to create, so the next step is to begin implementing your embedded object. I say "begin implementing" because it will be a little less painful to implement the object in pieces rather than attempt to implement the entire thing at once. So let's implement an object with just the IUnknown interface such that we can compile and test that the object is indeed created and that we close it properly.

As mentioned before, Schmoo's object is implemented using a C++ class called CFigure with the parts of its implementation important for this discussion as shown in Listing 10-2. SCHMOOLE.H contains the definition of CFigure as well as interface implementation classes for IPersistStorage, IDataObject, and IOleObject which I have omitted from the listing. FIGURE.CPP contains the implementation of CFigure. All of the extra functions in CFigure and the use of many of the class's variables will be shown in the context of the other interfaces that use them so only those relevant to our immediate discussion are shown here.

SCHMOOLE.H

```
/*
 * SCHMOOLE.H
 * Include file containing all compound document related definitions.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

[Class factory-related parts shown in Listing 10-1]

//FIGURE.CPP
//This is what the class factory creates

#define CFORMATETCGET   5   //Private, Embed Source, Obj Descript, MF, Bmp.

class __far CFigure : public IUnknown
    {
    friend class CImpIPersistStorage;
    friend class CImpIDataObject;
    friend class CImpIOleObject;

    protected:
        ULONG               m_cRef;
        LPCSchmooFrame      m_pFR;          //Frame (for UI manipulation)
        LPCSchmooDoc        m_pDoc;         //What holds the real polyline
        LPCPolyline         m_pPL;          //Copy of m_pDoc->m_pPL

        BOOL                m_fEmbedded;    //TRUE on IOleObject::SetHostNames

        LPFNDESTROYED       m_pfnDestroy;   //Function to call on closure.
        LPPERSISTSTORAGE    m_pIPersistStorage;

        //Things for IDataObject
        LPDATAOBJECT        m_pIDataObject;         //Implemented
        LPDATAADVISEHOLDER  m_pIDataAdviseHolder;   //Used

        UINT                m_cf;                   //Copy of pDoc->m_cf
        ULONG               m_cfeGet;
        FORMATETC           m_rgfeGet[CFORMATETCGET];
        //Things for IOleObject
        LPOLEOBJECT         m_pIOleObject;          //Implemented
        LPOLEADVISEHOLDER   m_pIOleAdviseHolder;    //Used
        LPOLECLIENTSITE     m_pIOleClientSite;      //Used


    public:
        CFigure(LPFNDESTROYED, LPCSchmooDoc);
        ~CFigure(void);

        //Non-delegating IUnknown:  we don't support aggregation here.
        STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        BOOL FInit(void);
        void FrameSet(LPCSchmooFrame);
        BOOL FIsDirty(void);
        BOOL FIsEmbedded(void);
        void SendAdvise(UINT);
    };

typedef CFigure * LPCFigure;

[Remainder of file relevant to interfaces]
```

FIGURE.CPP

```
/*
 * FIGURE.CPP
 * Implementation of the CFigure object for Schmoo.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "schmoo.h"

/*
 * CFigure::CFigure
 * CFigure::~CFigure
 */

CFigure::CFigure(LPFNDESTROYED pfnDestroy, LPCSchmooDoc pDoc)
  {
  m_cRef=0;
```

```
    m_pfnDestroy=pfnDestroy;

    m_pFR=NULL;     //We get this later through FrameSet.
    m_pDoc=pDoc;
    m_pPL=pDoc->m_pPL;

    m_fEmbedded=FALSE;

    //NULL any contained interfaces initially.
    m_pIPersistStorage=NULL;
    m_pIDataObject=NULL;
    m_pIDataAdviseHolder=NULL;
    m_pIOleObject=NULL;
    m_pIOleAdviseHolder=NULL;
    m_pIOleClientSite=NULL;

    m_cf=pDoc->m_cf;

    //These are for IDataObject::QueryGetData
    m_cfeGet=CFORMATETCGET;

    SETDefFormatEtc(m_rgfeGet[0], pDoc->m_cf, TYMED_HGLOBAL);
    SETDefFormatEtc(m_rgfeGet[1], pDoc->m_cfEmbedSource, TYMED_ISTORAGE);
    SETDefFormatEtc(m_rgfeGet[2], pDoc->m_cfObjectDescriptor, TYMED_HGLOBAL);
    SETDefFormatEtc(m_rgfeGet[3], CF_METAFILEPICT, TYMED_MFPICT);
    SETDefFormatEtc(m_rgfeGet[4], CF_BITMAP, TYMED_GDI);

    return;
    }


CFigure::~CFigure(void)
    {
    //Free contained interfaces.
    if (NULL!=m_pIOleObject)
        delete m_pIOleObject;

    if (NULL!=m_pIDataObject)
        delete m_pIDataObject;

    if (NULL!=m_pIPersistStorage)
        delete m_pIPersistStorage;

    return;
    }
```

```
/*
 * CFigure::QueryInterface
 * CFigure::AddRef
 * CFigure::Release
 */

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

    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_IDataObject))
        *ppv=(LPVOID)m_pIDataObject;

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

    //AddRef any interface we'll return.
    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }

    return ResultFromScode(E_NOINTERFACE);
    }


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


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

    cRefT=--m_cRef;
    if (0==m_cRef)
        {
        if (NULL!=m_pfnDestroy)
            (*m_pfnDestroy)();

        delete this;
        }

    return cRefT;
    }



/*
 * CFigure::FInit
 *
 * Purpose:
 *  Performs any intiailization of a CFigure that's prone to failure
 *  that we also use internally before exposing the object outside.
 */

BOOL CFigure::FInit(void)
    {
    //Allocate contained interfaces.
    m_pIPersistStorage=new CImpIPersistStorage(this, (LPUNKNOWN)this);

    if (NULL==m_pIPersistStorage)
        return FALSE;

    m_pIDataObject=new CImpIDataObject(this, (LPUNKNOWN)this);

    if (NULL==m_pIDataObject)
        return FALSE;

    m_pIOleObject=new CImpIOleObject(this, (LPUNKNOWN)this);

    if (NULL==m_pIOleObject)
        return FALSE;

    return TRUE;
    }



/*
 * CFigure::FrameSet
 *
 * Purpose:
 *  Provides the compound document object with access to the frame
 *  of this application for UI purposes.
 */

void CFigure::FrameSet(LPCSchmooFrame pFR)
    {
    m_pFR=pFR;
    return;
    }
```

Listing 10-2: The implementation of Schmoo's Figure object.

As you can see, the figure itself has a non-delegating IUnknown implementation to which all the interfaces will delegate. I mentioned before that this object does not support aggregation which manifests itself through the fact that the CFigure::FInit always passes itself (the *this* pointer) as the IUnknown to which the interfaces delegate. In previous examples that did support aggregation, this might have been the outer unknown had there been one.[1]

Most of what's shown in this listing makes little sense until we see exactly where a figure object is created and manipulated. In Listing 10-1 you can see that the CreateInstance function of our class factory doesn't actually create an object, it creates a document window (CSchmooDoc) just as if the user had selected File New from Schmoo's menu. CSchmooDoc has been modified for this chapter to have a variable called *m_pFigure* which is a pointer to a single CFigure object it manages. This gives a one to one correspondence between a CFigure object we server as an embedded object and the CPolyline object that Schmoo uses to display and edit its data. So by creating a new document, which calls new CSchmooDoc and CSchmooDoc::FInit (DOCUMENT.CPP) we create a new figure object:

```
CSchmooDoc::CSchmooDoc(HINSTANCE hInst)
    : CDocument(hInst)
    {
    ...

    m_pFigure=NULL;

    return;
    }


CSchmooDoc::~CSchmooDoc(void)
    {
    ...

    CoDisconnectObject((LPUNKNOWN)m_pFigure, 0L);
```

1Note also that I didn't bother to change the interface implementations themselves; other code we've already implemented always passed the object pointer and the controlling unknown to the interface implementation, where the controlling unknown was either the object or a real outer unknown. I decided not to change the interface implementations here just to remain consistent with all other implementations. I've often heard that readers prefer consistency above sheer elegance, so here ya go.

```
    if (NULL!=m_pFigure)
       m_pFigure->Release();   //So it can start shutdown if necessary.
    ...

    return;
    }

BOOL CSchmooDoc::FInit(LPDOCUMENTINIT pDI)
    {
    [Other initialization]

    m_pFigure=new CFigure(ObjectDestroyed, this);

    if (NULL==m_pFigure)
       return FALSE;

    /*
     * These two lines are 1) so we can ::Release on document closure
     * and 2) if we fail FInit then we'll destroy the document which
     * calls ::Release which will call ObjectDestroyed which will
     * decrement g_cObj and shut down if necessary.
     */
    m_pFigure->AddRef();
    g_cObj++;

    if (!m_pFigure->FInit())
       return FALSE;

    return TRUE;
    }
```

I may seem strange that I play some tricks with reference counting here on the figure object, but I do it to centralize all the shutdown code into CFigure::Release (and ultimately ObjectDestroyed). When we create the figure using new it has a reference count of zero (as set in CFigure::CFigure) so if creation succeeds we should AddRef it to a reference count of one and increment the global object count which we do in CSchmooDoc::FInit instead of the class factory's CreateInstance. Let's say now that the call to m_pFigure->FInit() shown above fails, and CSchmooDoc::FInit returns FALSE which eventually calls the document's destructor. Here, instead of calling delete m_pFigure it calls m_pFigure->Release which will reduce the reference count to zero which will call ObjectDestroyed. This function will decrement the global object count to zero, see that there are no locks and that the frame window is still valid, and will then post a WM_CLOSE that shuts the whole application down and purges it from memory. In this way I keep all the shutdown code in one place and one place only.

You probably also noticed that funny **CoDisconnectObject** call on m_pFigure. This function insures that just before we release the object which might shut the application down that there are no external connections to it. As we make this call in the document's destructor, we are on a non-stop one-way flight to oblivion as far as this object is concerned, so we have to tell OLE 2.0 that after this there is nothing and to make sure that no other agent tries to get at this object. CoDisconnectObject is just the ticket although it seems rather brutal. In any case, you must call this function when closing the document that holds your object. It's the only way to remove all other reference counts on your object.

Again, the rest of the CFigure implementation supports the other interfaces we need on this object: IPersistStorage, IDataObject, and IOleObject, which are the subjects of the next three sections. But seeing as you have an IUnknown implementation here already, you have, of course, a Windows Object. You can now again test your server in the context of a container's Insert Object dialog. This time, instead of failing inside CreateInstance, you can actually instantiate an object and return successfully. You will probably see a number of QueryInterface calls on this object asking for the other three expected interfaces. Since we don't have those interfaces yet, things won't be working right and your server may get left in memory. So it's time to start adding those interfaces.

Implement the IPersistStorage Interface

The best choice for the first interface to implement on an embedded object is IPersistStorage, exactly the IPersistStorage interface we discussed in Chapter 5 with all the same contractual obligations. Through this interface the embedded object is told to initialize its storage, save or load itself from a storage, and to get its grubby hands off the storage when the container wants to perform a File Save As with the object still open.[1]

For all intents and purposes, there is nothing special about the implementation of this interface that we don't already know, but it's worthwhile to point out that this interface is only used for embedded objects; linked objects have no use for it. In any case, to be a server of embedded objects you must have an implementation of IPersistStorage like the one shown for Schmoo in Listing 10-3.

```
/*
 * CFigure::FIsDirty
 *
 * Purpose:
 *  Checks if the document is dirty.  This can be called from
 *  IPersistStorage::IsDirty which doesn't have access to CSchmooDoc.
 */

BOOL CFigure::FIsDirty(void)
    {
    return m_pDoc->m_fDirty;
    }
```

IPERSTOR.CPP

```
/*
 * IPERSTOR.CPP
 *
 * Implementation of the IPersistStorage interface that we expose on the
 * CSchmooFigure compound document object.  This ties into the functionality
 * of CPolyline.
```

1Patron as a container does not keep objects running when performing a File Save or Save As and so it does not call IPersistStorage::HandsOffStorage.

```
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "schmoo.h"

/*
 * CImpIPersistStorage:CImpIPersistStorage
 * CImpIPersistStorage::~CImpIPersistStorage
 */

CImpIPersistStorage::CImpIPersistStorage(LPCFigure pObj, LPUNKNOWN punkOuter)
    {
    m_cRef=0;
    m_pObj=pObj;
    m_punkOuter=punkOuter;
    return;
    }

CImpIPersistStorage::~CImpIPersistStorage(void)
    {
    return;
    }



/*
 * CImpIPersistStorage::QueryInterface
 * CImpIPersistStorage::AddRef
 * CImpIPersistStorage::Release
 */

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();
    }


/*
 * CImpIPersistStorage::GetClassID
 *
 * Purpose:
 *  Returns the CLSID of the object represented by this interface.
 */

STDMETHODIMP CImpIPersistStorage::GetClassID(LPCLSID pClsID)
    {
    *pClsID=CLSID_SchmooFigure;
    return NOERROR;
    }


/*
 * CImpIPersistStorage::IsDirty
 *
 * Purpose:
 *  Tells the caller if we have made changes to this object since
 *  it was loaded or initialized new.
 */

STDMETHODIMP CImpIPersistStorage::IsDirty(void)
    {
    return ResultFromScode(m_pObj->FIsDirty() ? S_OK : S_FALSE);
    }


STDMETHODIMP CImpIPersistStorage::InitNew(LPSTORAGE pIStorage)
    {
    return NOERROR;
    }


STDMETHODIMP CImpIPersistStorage::Load(LPSTORAGE pIStorage)
    {
    LONG    lRet;

    lRet=m_pObj->m_pPL->ReadFromStorage(pIStorage);

    if (lRet >= 0)
        return NOERROR;
    return ResultFromScode(STG_E_READFAULT);
    }



STDMETHODIMP CImpIPersistStorage::Save(LPSTORAGE pIStorage, BOOL fSameAsLoad)
    {
    LONG    lRet;

    //fSameAsLoad is unimportant since we don't hold on to it.
    lRet=m_pObj->m_pPL->WriteToStorage(pIStorage, VERSIONCURRENT);

    if (lRet >= 0)
        return NOERROR;

    return ResultFromScode(STG_E_WRITEFAULT);
    }



STDMETHODIMP CImpIPersistStorage::SaveCompleted(LPSTORAGE pIStorage)
    {
    return NOERROR;
    }

STDMETHODIMP CImpIPersistStorage::HandsOffStorage(void)
    {
    return NOERROR;
    }
```

Listing 10-3: Schmoo's IPersistStorage implementation.

There are only a few interesting features to point out in this small piece of code. First, the FIsDirty function in the CFigure class exists solely to implement IPersistStorage::IsDirty because of C++ access restrictions. CFigure is marked as a friend of CSchmooDoc which holds the dirty flag in a protected member *m_fDirty*, so only CFigure functions can access it. Since CImpIPersistStorage is only a friend of CFigure and not of CSchmooDoc, the interface has to call the object which in turn can look up the dirty state. You might wonder why this IPersistStorage implementation does not just call the public CSchmooDoc:FIsDirty function. The reason is that due to the notifications that we'll be sending out from this server, an embedded object is never considered dirty as far as user-interface is concerned, and CSchmooDoc:FIsDirty is a function for user interface. Later we'll modify this function to always return "not dirty" when we're editing an embedded object. IPersistStorage::IsDirty is not so much a user interface thing as it is a way for a container to know if it needs to ask us to save before closing the object which does not necessarily involve user interface (see IOleObject::Close).

I also wanted to point out that if you already have functions in place to read and write your data to an IStorage the implementations of IPersistStorage::Load and IPersistStorage::Save become trivial. In Schmoo's case the CPolyline object implements the functions ReadFromStorage and WriteToStorage which do exactly what Load and Save are designed to do. This shows how the implementation of our embedded object, CFigure, is *using* the other code that exists in Schmoo already. Instead of modifying CPolyline to be an embedded object, we're adding a completely new object that implements its interfaces using functionality already present in other parts of the application. Therefore we do not have to change CPolyline in any way to support embeddings; rather we keep all changes in CFigure. This is an approach I highly recommend because it greatly reduces the intrusion of compound document code into other code that is not specific to compound documents. This separate allows either code to change without affecting much of the other.

Once you have an implementation of IPersistStorage and you again launch your server through a container's Insert Object dialog, you will now see your object created and initially asked for the IOleObject interface. If you fail that you'll be asked for IUnknown then IPersistStorage. Once you return that interface you'll see a call to IPersistStorage::InitNew which tells you that you're being used from Insert Object. But for now we cannot test Load, Save, SaveCompleted, or HandsOffStorage since we don't have a complete object and since the container will never get to the point of asking us to save. But you can at least make sure this all compiles and that your server still runs (its also a good idea to see if the server runs stand-alone as well, if you allow it).

Implement the IDataObject Interface

We saw in Chapter 9 how the default handler, OLE2.DLL, maintains a presentation cache in the container's storage. But somehow OLE2.DLL must obtain the presentations to cache in the first place. For that reason, all embedded objects must implement an IDataObject interface that at minimum can provide CF_METAFILEPICT or CF_BITMAP (or CF_DIB) through IDataObject::GetData and IDataObject::QueryInterface. At least one of these formats is required to maintain some sort of basic cache for the object.

Besides presentations, an object will generally want to support its native clipboard format through both GetData and SetData. A container that knows more about you object may ask you for your data or may give you some data to integrate into yourself. If you don't foresee any reason why someone would want to GetData or SetData your private format, then you have no reason to support it in this implementation of IDataObject (you will, however, still want to support it on the clipboard through the IDataObject you use in such operations, but that is a different object and a different IDataObject than the one on the embedded object). On case where you will have to support at least GetData on a native format is when you have your own object handler such that the object handler can synchronize its copy of an object's with the server's. But that is a topic for Chapter 11.

Besides graphic and native formats you should also support the CF_EMBEDSOURCE format (which is the registered string "Embed Source") which is an IStorage containing exactly the same data you would write to an IStorage in IPersistStorage. In fact, Schmoo's implementation of IDataObject, shown in Listing 10-4 along with parts of CSchmooDoc that are used by the IDataObject implementation, ultimately uses the same CPolyline::WriteToStorage to implement IDataObject::GetData and GetDataHere as it uses to implement IPersistStorage::Save.

IDATAOBJ.CPP

```
/*
 * IDATAOBJ.CPP
 * Implementation of the IDataObject interface.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */
```

```
#include "schmoo.h"

/*
 * CImpIDataObject::CImpIDataObject
 * CImpIDataObject::~CImpIDataObject
 */

CImpIDataObject::CImpIDataObject(LPCFigure pObj, LPUNKNOWN punkOuter)
    {
    m_cRef=0;
    m_pObj=pObj;
    m_punkOuter=punkOuter;
    return;
    }

CImpIDataObject::~CImpIDataObject(void)
    {
    return;
    }


/*
 * CImpIDataObject::QueryInterface
 * CImpIDataObject::AddRef
 * CImpIDataObject::Release
 */

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

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

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

```
/*
 * CImpIDataObject::GetData
 *
 * Purpose:
 *  Retrieves data described by a specific FormatEtc into a StgMedium
 *  allocated by this function.  Used like GetClipboardData.
 */

STDMETHODIMP CImpIDataObject::GetData(LPFORMATETC pFE, LPSTGMEDIUM pSTM)
    {
    UINT            cf=pFE->cfFormat;
    BOOL            fRet=FALSE;

    //Another part of us already knows if the format is good.
    if (NOERROR!=QueryGetData(pFE))
        return ResultFromScode(DATA_E_FORMATETC);

    if (CF_METAFILEPICT==cf || CF_BITMAP==cf || m_pObj->m_cf==cf)
        {
        if (CF_METAFILEPICT==cf)
            {
            pSTM->tymed=TYMED_MFPICT;
            }
        else
            pSTM->tymed=TYMED_HGLOBAL;

        pSTM->pUnkForRelease=NULL;
        pSTM->hGlobal=m_pObj->m_pDoc->RenderFormat(cf);
        fRet=(NULL!=pSTM->hGlobal);
        }
    else
        fRet=m_pObj->m_pDoc->FRenderMedium(cf, pSTM);

    return fRet ? NOERROR : ResultFromScode(DATA_E_FORMATETC);
    }


/*
 * CImpIDataObject::GetDataHere
 *
 * Purpose:
 *  Renders the specific FormatEtc into caller-allocated medium
 *  provided in pSTM.
 */
```

```
STDMETHODIMP CImpIDataObject::GetDataHere(LPFORMATETC pFE, LPSTGMEDIUM pSTM)
    {
    UINT        cf;
    LONG        lRet;

    /*
     * The only reasonable time this is called is for CF_EMBEDSOURCE
     * and TYMED_ISTORAGE (and later for CF_LINKSOURCE).  This means
     * the same as IPersistStorage::Save.
     */

    cf=RegisterClipboardFormat(CF_EMBEDSOURCE);

    //Aspect is unimportant to us here, as is lindex and ptd.
    if (cf==pFE->cfFormat && (TYMED_ISTORAGE & pFE->tymed))
        {
        //We have an IStorage we can write into.
        pSTM->tymed=TYMED_ISTORAGE;
        pSTM->pUnkForRelease=NULL;
        lRet=m_pObj->m_pPL->WriteToStorage(pSTM->pstg, VERSIONCURRENT);

        if (lRet >= 0)
            return NOERROR;

        return ResultFromScode(STG_E_WRITEFAULT);
        }

    return ResultFromScode(DATA_E_FORMATETC);
    }


/*
 * CImpIDataObject::QueryGetData
 *
 * Purpose:
 *  Tests if a call to ::GetData with this FormatEtc will provide
 *  any rendering; used like IsClipboardFormatAvailable.
 */

STDMETHODIMP CImpIDataObject::QueryGetData(LPFORMATETC pFE)
    {
    UINT            cf=pFE->cfFormat;
    UINT            i;

    //Check the aspects we support.
    if (!(DVASPECT_CONTENT & pFE->dwAspect))
```

```
        return ResultFromScode(S_FALSE);

    for (i=0; i < m_pObj->m_cfeGet; i++)
        {
        if (pFE->cfFormat==m_pObj->m_rgfeGet[i].cfFormat
            && pFE->tymed & m_pObj->m_rgfeGet[i].tymed)
            {
            return NOERROR;
            }
        }

    return ResultFromScode(S_FALSE);
    }



STDMETHODIMP CImpIDataObject::GetCanonicalFormatEtc(LPFORMATETC pFEIn
    , LPFORMATETC pFEOut)
    {
    return ResultFromScode(DATA_S_SAMEFORMATETC);
    }



/*
 * CImpIDataObject::SetData
 *
 * Purpose:
 *  Places data described by a FormatEtc and living in a StgMedium
 *  into the object.  The object may be responsible to clean up the
 *  StgMedium before exiting.
 */

STDMETHODIMP CImpIDataObject::SetData(LPFORMATETC pFE, STGMEDIUM FAR *pSTM
    , BOOL fRelease)
    {
    LONG            lRet;

    /*
     * Data can only come from global memory containing a POLYLINEDATA
     * structure that we send to the Polyline's DataSetMem.
     */
    if ((pFE->cfFormat!=m_pObj->m_cf) || !(DVASPECT_CONTENT & pFE->dwAspect)
        || (TYMED_HGLOBAL!=pSTM->tymed))
        return ResultFromScode(DATA_E_FORMATETC);

    lRet=m_pObj->m_pPL->DataSetMem(pSTM->hGlobal, FALSE, TRUE, TRUE);
```

```
   if (fRelease)
      ReleaseStgMedium(pSTM);

   return (POLYLINE_E_NONE==lRet) ?
      NOERROR : ResultFromScode(DATA_E_FORMATETC);
   }


STDMETHODIMP CImpIDataObject::EnumFormatEtc(DWORD dwDir
   , LPENUMFORMATETC FAR *ppEnum)
   {
   return ResultFromScode(OLE_S_USEREG);
   }

STDMETHODIMP CImpIDataObject::DAdvise(LPFORMATETC pFE, DWORD dwFlags
   , LPADVISESINK pIAdviseSink, LPDWORD pdwConn)
   {
   HRESULT         hr;

   if (NULL==m_pObj->m_pIDataAdviseHolder)
      {
      hr=CreateDataAdviseHolder(&m_pObj->m_pIDataAdviseHolder);

      if (FAILED(hr))
         return ResultFromScode(E_OUTOFMEMORY);
      }

   hr=m_pObj->m_pIDataAdviseHolder->Advise((LPDATAOBJECT)this, pFE
      , dwFlags, pIAdviseSink, pdwConn);

   return hr;
   }


STDMETHODIMP CImpIDataObject::DUnadvise(DWORD dwConn)
   {
   if (NULL==m_pObj->m_pIDataAdviseHolder)
      return ResultFromScode(E_FAIL);

   return m_pObj->m_pIDataAdviseHolder->Unadvise(dwConn);
   }



STDMETHODIMP CImpIDataObject::EnumDAdvise(LPENUMSTATDATA FAR *ppEnum)
   {
```

```
   if (NULL==m_pObj->m_pIDataAdviseHolder)
       return ResultFromScode(E_FAIL);

   return m_pObj->m_pIDataAdviseHolder->EnumAdvise(ppEnum);
   }
```

DOCUMENT.CPP

```
...

CSchmooDoc::CSchmooDoc(HINSTANCE hInst)
   : CDocument(hInst)
   {
   [Other initialization omitted]

   //Registers clipboard formats for embedded objects.
   m_cfEmbedSource=RegisterClipboardFormat(CF_EMBEDSOURCE);
   m_cfObjectDescriptor=RegisterClipboardFormat(CF_OBJECTDESCRIPTOR);

   [Other code omitted]
   return;
   }


/*
 * CSchmooDoc::FRenderMedium
 *
 * Purpose:
 *  Like RenderFormat, this function creates a specific data format
 *  based on the cf parameter.  Unlike RenderFormat, we store the
 *  result in a STGMEDIUM in case it has a medium other than TYMED_HGLOBAL.
 *  For conveniece we'll centralize all compound document formats here,
 *  hGlobal or not.
 *
 * Parameters:
 *  cf              UINT clipboard format of interest.
 *  pSTM            LSTGMEDIUM to fill.  We only fill the union and tymed.
 *
 * Return Value:
 *  BOOL            TRUE if we could render the format, FALSE otherwise.
 */

BOOL CSchmooDoc::FRenderMedium(UINT cf, LPSTGMEDIUM pSTM)
   {
   if (NULL==pSTM)
```

```
    return FALSE;

if (cf==m_cfEmbedSource)
    {
    pSTM->pstg=OleStdCreateStorageOnHGlobal(NULL, TRUE, STGM_DIRECT
        | STGM_READWRITE | STGM_SHARE_EXCLUSIVE);

    if (NULL==pSTM->pstg)
        return FALSE;

    //Now save the data to the storage.
    WriteClassStg(pSTM->pstg, CLSID_SchmooFigure);
    WriteFmtUserTypeStg(pSTM->pstg, m_cf, PSZ(IDS_CLIPBOARDFORMAT));

    if (POLYLINE_E_NONE!=m_pPL->WriteToStorage(pSTM->pstg, VERSIONCURRENT))
        {
        pSTM->pstg->Release();
        return FALSE;
        }

    pSTM->tymed=TYMED_ISTORAGE;
    return TRUE;
    }

if (cf==m_cfObjectDescriptor)
    {
    SIZEL   szl, szlT;
    POINTL  ptl;
    RECT    rc;

    m_pPL->SizeGet(&rc);
    SETSIZEL(szlT, rc.right, rc.bottom);
    XformSizeInPixelsToHimetric(NULL, &szlT, &szl);

    SETPOINTL(ptl, 0, 0);

    pSTM->hGlobal=OleStdGetObjectDescriptorData(CLSID_SchmooFigure
        , DVASPECT_CONTENT, szl, ptl, OLEMISC_RECOMPOSEONRESIZE
        , PSZ(IDS_OBJECTDESCRIPTION), NULL);

    pSTM->tymed=TYMED_HGLOBAL;
    return (NULL!=pSTM->hGlobal);
    }

return FALSE;
}
```

Listing 10-4: Implementation of the IDataObject interface for Schmoo's CFigure class.

First note that the function CSchmooDoc::FRenderFormat, which generates bitmaps, metafiles, and native data, is unchanged from previous versions of Schmoo. Again, the implementation of this interface, specifically IDataObject::GetData, is *using* the existing code instead of modifying it. Likewise the implementation of IDataObject::SetData uses CPolyline::DataSet to support native data imports.

However, we do have to add a little more code to CSchmooDoc, first in the constructor to register the OLE 2.0 clipboard formats of CF_EMBEDSOURCE and CF_OBJECTDESCRIPTOR and second by adding the FRenderMedium function to create those formats. Creating CF_OBJECTDESCRIPTOR is a matter of calling the OLE2UI function OleStdGetObjectDescriptorData which will conveniently allocate and fill the structure for us.

CF_EMBEDSOURCE is a little more interesting, however. This format, again, is an IStorage containing the object's native data exactly like that from IPersistStorage::Save. To create the IStorage, Schmoo uses the OLE2UI function OleStdCreateIStorageOnHGlobal which internally calls CreateILockBytesOnHGlobal and StgCreateDocfileOnILockBytes. Schmoo uses a memory IStorage because it knows its data is small, using somewhere on the order of 2K for the entire storage. If you have much larger data, then I strongly encourage you weigh the tradeoffs between a memory IStorage and a disk based IStorage, because at some point a memory IStorage will turn begin using virtual memory which is on the disk anyway, and dealing with the disk through both IStorage and swapping layers will be much slower than just going to a disk-based IStorage directly. But regardless of what you do with storage, notice that in Schmoo, once we obtain the IStorage, we call CPolyline::WriteToStorage exactly as we called it from IPersistStorage::Save. Furthermore, our implementation of IDataObject::GetDataHere does exactly the same thing, calling CPolyline::WriteToStorage on whatever IStorage is provided by the caller.

So why do I make such a big point out of matching IPersistStorage::Save and IDataObject::GetData on CF_EMBEDSOURCE? Because the data you render for CF_EMBEDSOURCE may be used by a container application to create a new object of your class. When that container later activates that object, it will launch your server and ask your IPersistStorage::Load to read from that very same storage. Since IPersistStorage::Save and IPersistStorage::Load must be compatible, so must your rendering of CF_EMBEDSOURCE be compatible as well.

That leaves us with the easy stuff in IDataObject, such as QueryGetData which is implemented here by cycling through an array of FORMATETCs kept in CFigure. GetCanonicalFormatEtc is its typically trivial self, and the three amigos DAdvise, DUnadvise, and EnumDAdvise are implemented using the DataAdviseHolder provided by OLE 2.0. So now you're saying "the only member function left is EnumFormatEtc, and you have to implement another object for that, right?" Well, not any more, because now we can use the FORMATETC enumerator implemented in the default handler, OLE2.DLL, which will enumerate whatever formats we list under our CLSID\DataFormats\GetSet in the registration database. All we have to do to obtain this freebie implementation is return OLE_S_USEREG. What really happens is that when something over in the container's process calls IDataObject::EnumFormatEtc the call first reaches the object handler. If this is the default handler (generally the case, even when custom handlers are present), it first attempts to call EnumFormatEtc on a running server. If the server is not running, or it is running but returns OLE_S_USEREG, the default handler will provide the enumerator based on registration database entries. If you want control and wish to implement the enumerator yourself, go right ahead. I won't stop you. This simple return value is just the quickest and easiest way to provide this necessary functionality.

The best you can do after implementing this interface is to make sure everything compiles, because a container will not make use of this interface unless it can also make use of IOleObject which is next on our list to implement. If you really wanted to test your interface, you can, of course, write a simple object user like DATAUSER in Chapter 6 that will call CoCreateInstance on your CLSID and ask for IDataObject. That works perfectly well, since this object you're implementing looks exactly like any other component object with the IDataObject interface. In fact, you could make only minor changes to DATAUSER to turn it into a quick and dirty test application for what you do with IDataObject here.

Implement the IOleObject Interface

The IOleObject interface is the big prize winner of all OLE 2.0 interfaces for having the most member functions: 21 excluding those in IUnknown. In some ways it looks like the interface from hell, a dumping ground for every function that just didn't seem to have any better home. Intimidating? You bet! Is it a problem? Not really. For the most part, many of the member functions in this interface either have trivial or optional implementations, resulting in about 15 you really have to implement in one way or another of which 12 are trivial or use defaults from the registration database. The remaining functions are entirely optional for embedded objects, and some are not even useful to embedded objects whatsoever.

All 21 member functions and their responsibilities are shown in Table 10-1 with Schmoo's CImpIOleObject implementation shown in Listing 10-5. The definition of CImpIOleObject can be found in SCHMOOLE.H and is pretty much the same old boring sort of interface implementation that we've seen for all the others. IOleObject is really the main interface of an embedded object as it forms the bulk of the functions presented to a container application. If you've read the entirety of Chapter 9 you will have already seen where many of these functions are called.

Table 10-1: The IOleObject Interface

IOleObject Member	Description
Requires real programming:	
SetHostNames	Provides the object with the name of its container application and the name of the document in which the object lives. On this call the object changes its user interface to reflect is embedded state. This function is only called on embedded objects.
Close	Instructs the object to close itself, possibly saving in the process and possibly asking the end-user to verify the save. This will also destroy the object which might cause server shutdown.
DoVerb	Executes an action on the object which may include hiding or showing the object's editing window.
Trivial implementations:	
SetClientSite	Provides the object with an IOleClientSite pointer to its container site. This is the only way in which the object can obtain any pointer to the container, and this will be the first container pointer it sees. The object must hold on to this pointer (meaning AddRef it as well) in order to send notifications later on.
GetClientSite	Returns the last IOleClientSite pointer seen in SetClientSite.

Update	Insures that the object is up to date. Embedded objects are always up to date.
IsUpToDate	Checks if the object is up to date for which the answer is always yes for embedded objets.
GetExtent	Retrieves the current size of the object. This is implemented by simply copying your object's horizontal and vertical dimensions (in HIMETRIC) into a structure.
Advise	Provides an IAdviseSink which the object notifies through OnSave, OnClose, and OnRename. This function can be delegated to a holder available from CreateOleAdviseHolder.
Unadvise	Terminates a connection from Advise which can also be delegated to an OleAdviseHolder.
EnumAdvise	Enumerates the connections made through Advise and can be delegated to an OleAdviseHolder.
GetUserClassID	Returns the CLSID of the object users thinks they're editing, even if we're not the real server for that class.

Default implementations using the registration database:

EnumVerbs	Returns an enumerator for OLEIVERB values. Object can return OLE_S_USEREG for a default implementation.
GetUserType	Returns a pointer to a user-readable string which can be taken from the AuxUserType entries in the registration database.
GetMiscStatus	Returns a set of OLEMISC flags for the given aspect, which can be taken from the registered values under MiscStatus. The most common values are OLEMISC_RECOMPOSEONRESIZE which indicates that the object would like to redraw its presentation if scaled, and OLEMISC_ONLYICONIC which indicates that there is no useful view of this object other than an iconic one.

Completely optional:

SetExtent	Instructs the object to change its size usually to match the size of the object in the container's view.
InitFromData	Provides the object with an IDataObject pointer from which it can initialize itself, essentially performing a paste into the object. This is provided in addition to IDataObject::SetData as it does not require the caller to know server-specific clipboard formats.
GetClipboardData	Asks the object for an IDataObject pointer which would be exactly what the server would place on the clipboard for this object.
SetColorScheme	Asks the object for a color palette it would prefer to use.

Completely unimportant for embedded objects:

SetMoniker	Provides the object with a name in a moniker. This is only used in linking scenarios as described in later chapters.
GetMoniker	Asks the object for a moniker describing itself with or without information about its container as well.

Schmoo's implementation of these members is contained in IOLEOBJ.CPP as shown in Listing 10-5. Note that the function in the listing are grouped according to the table above but the actual source code on the sample disk has no such organization as it instead follows the order of the functions as defined in the interface.

IOLEOBJ.CPP

```
/*
 * IOLEOBJ.CPP
 * Implementation of the IOleObject interface for Polyline.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "schmoo.h"

/*
 * CImpIOleObject::CImpIOleObject
 * CImpIOleObject::~CImpIOleObject
 */

CImpIOleObject::CImpIOleObject(LPCFigure pObj, LPUNKNOWN punkOuter)
    {
    m_cRef=0;
    m_pObj=pObj;
    m_punkOuter=punkOuter;
    return;
    }

CImpIOleObject::~CImpIOleObject(void)
    {
    return;
    }



/*
```

```
 * CImpIOleObject::QueryInterface
 * CImpIOleObject::AddRef
 * CImpIOleObject::Release
 */

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

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

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



/*
 * CImpIOleObject::SetHostNames
 *
 * Purpose:
 *  Provides the object with names of the container application and the
 *  object in the container to use in object user interface.
 *
 * Parameters:
 *  pszApp          LPCSTR of the container application.
 *  pszObj          LPCSTR of some name that is useful in window titles.
 */

STDMETHODIMP CImpIOleObject::SetHostNames(LPCSTR pszApp, LPCSTR pszObj)
    {
    m_pObj->m_fEmbedded=TRUE;
    m_pObj->m_pFR->UpdateEmbeddingUI(TRUE, m_pObj->m_pDoc, pszApp, pszObj);
    return NOERROR;
    }



/*
 * CImpIOleObject::Close
 *
 * Purpose:
 *  Forces the object to close down its user interface and unload.
 *
 * Parameters:
 *  dwSaveOption    DWORD describing the circumstances under which the
 *                  object is being saved and closed.
 */

STDMETHODIMP CImpIOleObject::Close(DWORD dwSaveOption)
    {
    HWND        hWnd;
    BOOL        fSave=FALSE;

    hWnd=m_pObj->m_pDoc->Window();

    if (OLECLOSE_SAVEIFDIRTY==dwSaveOption && m_pObj->FIsDirty())
        fSave=TRUE;

    if (OLECLOSE_PROMPTSAVE==dwSaveOption && m_pObj->FIsDirty())
        {
        char        szTemp[80];
        char        szTitle[20];
        HINSTANCE   hInst;
        UINT        uRet;

       #ifdef WIN32
        hInst=(HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE);
       #else
        hInst=(HINSTANCE)GetWindowWord(hWnd, GWW_HINSTANCE);
       #endif

        LoadString(hInst, IDS_CAPTION, szTitle, sizeof(szTitle));
        LoadString(hInst, IDS_MISCCLOSEPROMPT, szTemp, sizeof(szTemp));
        MessageBox(hWnd, szTemp, szTitle, MB_YESNOCANCEL);

        if (IDCANCEL==uRet)
            return ResultFromScode(OLE_E_PROMPTSAVECANCELLED);

        if (IDYES==uRet)
            fSave=TRUE;
        }

    if (fSave)
        {
        m_pObj->SendAdvise(OBJECTCODE_SAVEOBJECT);
        m_pObj->SendAdvise(OBJECTCODE_SAVED);
        }

    //We get directly here on OLECLOSE_NOSAVE.
    PostMessage(hWnd, WM_CLOSE, 0, 0L);
    return NOERROR;
    }



/*
 * CImpIOleObject::DoVerb
 *
 * Purpose:
 *  Executes an object-defined action.
 *
 * Parameters:
 *  iVerb           LONG index of the verb to execute.
 *  pMSG            LPMSG describing the event causing the activation.
 *  pActiveSite     LPOLECLIENTSITE to the site involved.
 *  lIndex          LONG the piece on which execution is happening.
 *  hWndParent      HWND of the window in which the object can play in-place.
 *  pRectPos        LPRECT of the object in hWndParent where the object
 *                  can play in-place if desired.
 */

STDMETHODIMP CImpIOleObject::DoVerb(LONG iVerb, LPMSG pMSG
    , LPOLECLIENTSITE pActiveSite, LONG lIndex, HWND hWndParent
    , LPCRECT pRectPos)
    {
    HWND            hWnd, hWndT;

    //Find the upper most window
    hWndT=GetParent(m_pObj->m_pDoc->Window());

    while (NULL!=hWndT)
        {
        hWnd=hWndT;
        hWndT=GetParent(hWndT);
        }

    switch (iVerb)
        {
        case OLEIVERB_HIDE:
            ShowWindow(hWnd, SW_HIDE);
            m_pObj->SendAdvise(OBJECTCODE_HIDEWINDOW);
            break;
        case OLEIVERB_PRIMARY:
        case OLEIVERB_OPEN:
        case OLEIVERB_SHOW:
            ShowWindow(hWnd, SW_SHOWNORMAL);
            SetFocus(hWnd);
            m_pObj->SendAdvise(OBJECTCODE_SHOWOBJECT);
            m_pObj->SendAdvise(OBJECTCODE_SHOWWINDOW);
            break;

        default:
            return ResultFromScode(OLEOBJ_S_INVALIDVERB);
        }

    return NOERROR;
    }


/*
 * CImpIOleObject::SetClientSite
 * CImpIOleObject::GetClientSite
 *
 * Stores or retrieves the container's IOleClientSite pointer.
 */

STDMETHODIMP CImpIOleObject::SetClientSite(LPOLECLIENTSITE pIOleClientSite)
    {
    if (NULL!=m_pObj->m_pIOleClientSite)
        m_pObj->m_pIOleClientSite->Release();

    m_pObj->m_pIOleClientSite=pIOleClientSite;
    m_pObj->m_pIOleClientSite->AddRef();
    return NOERROR;
    }

STDMETHODIMP CImpIOleObject::GetClientSite(LPOLECLIENTSITE FAR * ppSite)
    {
    //Be sure to AddRef the new pointer you are giving away.
    *ppSite=m_pObj->m_pIOleClientSite;
    m_pObj->m_pIOleClientSite->AddRef();

    return ResultFromScode(E_NOTIMPL);
    }


STDMETHODIMP CImpIOleObject::Update(void)
    {
```

```
    //We're always updated since we don't contain.
    return NOERROR;
    }



STDMETHODIMP CImpIOleObject::IsUpToDate(void)
    {
    //We're always updated since we don't contain.
    return NOERROR;
    }



STDMETHODIMP CImpIOleObject::GetUserClassID(LPCLSID pClsID)
    {
    *pClsID=CLSID_SchmooFigure;
    return NOERROR;
    }



/*
 * CImpIOleObject::GetExtent
 *
 * Purpose:
 *  Retrieves the size of the object in HIMETRIC units.
 *
 * Parameters:
 *  dwAspect        DWORD of the aspect requested
 *  pszl            LPSIZEL into which to store the size.
 */

STDMETHODIMP CImpIOleObject::GetExtent(DWORD dwAspect, LPSIZEL pszl)
    {
    RECT            rc;
    SIZEL           szl;

    if (!(DVASPECT_CONTENT & dwAspect))
        return ResultFromScode(E_FAIL);

    m_pObj->m_pPL->RectGet(&rc);
    szl.cx=rc.right-rc.left;
    szl.cy=rc.bottom-rc.top;

    XformSizeInPixelsToHimetric(NULL, &szl, pszl);
    return NOERROR;
    }
```

```
STDMETHODIMP CImpIOleObject::Advise(LPADVISESINK pIAdviseSink, LPDWORD pdwConn)
    {
    if (NULL==m_pObj->m_pIOleAdviseHolder)
        {
        HRESULT     hr;

        hr=CreateOleAdviseHolder(&m_pObj->m_pIOleAdviseHolder);

        if (FAILED(hr))
            return hr;
        }

    return m_pObj->m_pIOleAdviseHolder->Advise(pIAdviseSink, pdwConn);
    }


STDMETHODIMP CImpIOleObject::Unadvise(DWORD dwConn)
    {
    if (NULL!=m_pObj->m_pIOleAdviseHolder)
        return m_pObj->m_pIOleAdviseHolder->Unadvise(dwConn);

    return ResultFromScode(E_FAIL);
    }


STDMETHODIMP CImpIOleObject::EnumAdvise(LPENUMSTATDATA FAR * ppEnum)
    {
    if (NULL!=m_pObj->m_pIOleAdviseHolder)
        return m_pObj->m_pIOleAdviseHolder->EnumAdvise(ppEnum);

    return ResultFromScode(E_FAIL);
    }


STDMETHODIMP CImpIOleObject::EnumVerbs(LPENUMOLEVERB FAR * ppEnum)
    {
    //Trivial implementation if you fill the regDB.
    return ResultFromScode(OLE_S_USEREG);
    }


STDMETHODIMP CImpIOleObject::GetUserType(DWORD dwForm, LPSTR FAR * ppszType)
    {
    return ResultFromScode(OLE_S_USEREG);
    }
```

```
STDMETHODIMP CImpIOleObject::GetMiscStatus(DWORD dwAspect, LPDWORD pdwStatus)
    {
    return ResultFromScode(OLE_S_USEREG);
    }



/*
 * CImpIOleObject::InitFromData
 *
 * Purpose:
 *  Initializes the object from the contents of a data object.
 *
 * Parameters:
 *  pIDataObject    LPDATAOBJECT containing the data.
 *  fCreation       BOOL indicating if this is part of a new creation.
 *                  If FALSE, the container is trying to paste here.
 *  dwReserved      DWORD reserved.
 */

STDMETHODIMP CImpIOleObject::InitFromData(LPDATAOBJECT pIDataObject
    , BOOL fCreation, DWORD dwReserved)
    {
    BOOL    fRet;

    /*
     * If we get a data object here, try to paste from it.  If you've
     * written clipboard code already, this is a snap.  We don't really
     * care about fCreation or not since pasting in us blasts away
     * whatever is already here.
     */
    fRet=m_pObj->m_pDoc->FPasteFromData(pIDataObject);
    return fRet ? NOERROR : ResultFromScode(E_FAIL);
    }



/*
 * CImpIOleObject::GetClipboardData
 *
 * Purpose:
 *  Returns an IDataObject pointer to the caller representing what would
 *  be on the clipboard if the server did an Edit/Copy using OleSetClipboard.
 *
 * Parameters:
 *  dwReserved      DWORD reserved.
 *  ppIDataObj      LPDATAOBJECT FAR * into which to store the pointer.
 */

STDMETHODIMP CImpIOleObject::GetClipboardData(DWORD dwReserved
    , LPDATAOBJECT FAR * ppIDataObj)
    {
    /*
     * Again, if you have a function to create a data object for the
     * clipboard, this is a simple implementation.  The one we have
     * does all the compound document formats already.
     */
    *ppIDataObj=m_pObj->m_pDoc->TransferObjectCreate();
    return (NULL!=*ppIDataObj) ? NOERROR : ResultFromScode(E_FAIL);
    }



/*
 * CImpIOleObject::SetExtent
 *
 * Purpose:
 *  Sets the size of the object in HIMETRIC units.
 *
 * Parameters:
 *  dwAspect        DWORD of the aspect affected.
 *  pszl            LPSIZEL containing the new size.
 */

STDMETHODIMP CImpIOleObject::SetExtent(DWORD dwAspect, LPSIZEL pszl)
    {
    RECT            rc;
    SIZEL           szl;

    if (!(DVASPECT_CONTENT & dwAspect))
        return ResultFromScode(E_FAIL);

    XformSizeInHimetricToPixels(NULL, pszl, &szl);

    //This resizes the window to match the container's size.
    SetRect(&rc, 0, 0, (int)szl.cx, (int)szl.cy);
    m_pObj->m_pPL->SizeSet(&rc, TRUE);

    return NOERROR;
    }


STDMETHODIMP CImpIOleObject::SetColorScheme(LPLOGPALETTE pLP)
```

```
   {
   return ResultFromScode(E_NOTIMPL);
   }



STDMETHODIMP CImpIOleObject::SetMoniker(DWORD dwWhich, LPMONIKER pmk)
   {
   return ResultFromScode(E_NOTIMPL);
   }

STDMETHODIMP CImpIOleObject::GetMoniker(DWORD dwAssign, DWORD dwWhich
   , LPMONIKER FAR * ppmk)
   {
   return ResultFromScode(E_NOTIMPL);
   }
```

Listing 10-5: Implementation of the IOleObject interface for Schmoo's CFigure class.

You'll note first that unimplemented functions return E_NOTIMPL and those that use registration database defaults return OLE_S_USEREG. The interesting member functions of the other groups that have at least some implementation in the code above are discussed in the following three sections.

Trivial Functions

Let's look at the simple implementations first, because we'll need some of the information from these functions to implement the more complex ones. The functions in this set are SetClientSite, GetClientSite, Update, IsUpToDate, GetExtent, GetUserClassID, and the triumvirate of Advise, Unadvise, and EnumAdvise.

SetClientSite illustrates one of the rare times when an object is handed the first pointer to some other object on a silver platter. This is, in fact, the only way through which the object gets an IOleClientSite pointer to the container's site object. You must hold on to this pointer for the lifetime of your object which means to save it in a variable somewhere. And holding on to an interface pointer means what? **AddRef** it, of course. In addition, on the off chance that you get multiple calls to SetClientSite, Release whatever pointer you are currently holding before overwriting it with the new one. While I haven't seen this actually happen, it's very possible and making sure you release before you overwrite is good defensive programming.

GetClientSite is the direct sibling of SetClientSite which only needs to copy the last IOleClientSite pointer from SetClientSite into the out parameter **ppSite*. In addition, you are a function that is returning a new copy of a pointer, so be sure to AddRef the IOleClientSite again.

Update and **IsUpToDate** are a pair that a container can use to make sure that the presentation it has in its cache matches the current state of the object. IsUpToDate asks "are you current" whereas Update tells you "make yourself current." I mentioned in earlier in Table 10-1 that embedded objects are always up to date. This is because embedded objects must always notify the container's IAdviseSink (which is actually in the handler) when data changes and this mechanism always keeps the cache updated. These functions really exist to support linked objects where the current visible state of the object may not, in fact, reflect the current contents of another file, so we may have to launch some application to load the file and give us a new presentation. We'll see these again in Chapter 12 when we deal with linking.

GetExtent asks the object "how big is this aspect" by asking the object to fill a SIZEL structure with the horizontal and vertical dimensions of the object in HIMETRIC units, sensitive to the requested aspect. These extents are in absolute units, that is, the vertical value is not negative as if you were dealing in the MM_HIMETRIC *mapping mode*. Since there is no hDC anywhere in sight, there is no conception of a mapping mode in this function. Schmoo implements this by retrieving the rectangle of the current Polyline window in pixels and using the OLE2UI function XformSizeInPixelsToHimetric to convert the values before returning.

GetUserClassID is a rather odd fellow that is used to support object conversion and emulation through a container's Change Type dialog which we'll see in Chapter 14. In the conversion/emulation scenario your server and object may be editing an object of a different CLSID than your usual one (but only if you registered yourself as being able to work with that CLSID). In such a situation, the CLSID the user thinks they are working on is not the default CLSID of your server, so this function gives the container (and OLE 2.0) a way of knowing what the object actually is. Since we are not going to look at conversion and emulation until Chapter 14, implementation here is simple: just fill the out parameter **pClsID* with your server's CLSID.

The last three functions dealing with advising can be implemented in the same fashion that we implemented their counterparts in IDataObject: use an "advise holder." A container will call Advise with its IAdviseSink pointer in order to get OnSave, OnRename, and OnClose notifications from the object. Usually these are of interest to the object handler to support linking, but we really don't see that from the server side of things. Therefore we have to ignorantly squirrel away ever IAdviseSink we see in IOleObject::Advise, which is why we have the convenience of the advise holder to reduce the implementation down to simple steps:

1. In Advise, if you do no already have an advise holder, call CreateOleAdviseHolder which returns (in an out-parameter) a pointer to an IOleAdviseHolder interface which has a reference count of one.

2. In Advise, Unadvise, and EnumAdvise, if you have an IOleAdviseHolder pointer available, delegate the call directly to its member functions of the same name.

3. When destroying the object, remember to call IOleAdviseHolder::Release to free the holder created in Advise.

Once you have implemented SetClientSite, the three IOleObject advising functions, and the three advising functions in IDataObject, you have three pointers, to IOleClientSite, IOleAdviseHolder, and IDataAdviseHolder, through which you send various notifications and requests to the container. Through these notifications we allow the container to update its cache and presentations, to save the object when appropriate, and to control its user interface properly.

Required Functions

In this set we'll find DoVerb, Close, and SetHostNames the three most important members of IOleObject (in that order). First and most important is DoVerb which asks an object to execute some action. Without do verb there would be no such thing as activation so it is really the crux of compound documents. As we'll see later, it is through this function that the whole process of In-Place Activation begins.

As discussed before, **DoVerb** is what takes an object from the loaded state into the running or activated state or to transition between visible and hidden states of the object's editing window. This function receives a number of parameters, the first of which, *iVerb*, is the number of the verb to execute which may either be a server-defined verb or a pre-defined verb.[1] Each pre-defined verb has a specific implementation:

OLEIVERB_SHOW	Call ShowWindow(hWnd, SW_SHOWNORMAL) where hWnd is your main application window (for single-object servers) or the document window containing the object (if you are an MDI server). In the latter case you may have to show the frame window if it is not yet visible itself. After ShowWindow, call SetFocus(hWnd) followed by calls to IOleClientSite::ShowObject and IOleClientSite::OnShowWindow(TRUE) where the IOleClientSite pointer is the one you save from SetClientSite, not the one passed to this function (which is for in-place activation). Schmoo again handles these two calls through CFigure::SendAdvise as you can see in the previous listing.
OLEIVERB_OPEN	Outside of in-place activation this has the same semantics as OLEIVERB_SHOW.
OLEIVERB_HIDE	Call ShowWindow(hWnd, SW_HIDE) where hWnd is your main application window (if you are editing one object only) or your document window (if you are editing multiple objects in multiple documents). Follow this with a call to IOleClientSite::OnShowWindow(FALSE) where the IOleClientSite is the one you saved in SetClientSite.

Outside of these three verbs (which are values -1, -2, and -3) which you must implement you will see verbs with values of zero and higher and possibly some negative ones less than -3. For any verb index that makes no sense, return OLE_E_INVALIDVERB. Otherwise, the exact meaning of the verb is something only you know in which case you perform whatever action is appropriate. For Schmoo, its single verb "Edit" (which is also the primary verb of index zero or OLEIVERB_PRIMARY) only needs to show the window containing an editable Polyline figure. In this case our primary verb is exactly the same as OLEIVERB_SHOW. Other types of objects like a sound that have a Play verb would only play the sound and not actually show any windows here at all, nor would it call anything in IOleClientSite.

The other parameters to DoVerb provide more information to the object that it can use to modify its behavior. *lpMsg* tells the object what message actually caused the DoVerb call from the container, such as WM_LBUTTONDBLCLK, which is mostly important for a type of in-place object we call "inside-out" that is a topic of a later chapter. The IOleClientSite pointer *pActiveSite* is again used in other special cases which are not yet important. *lindex* is always 0 in OLE 2.0 meaning "the entire object" and is reserved for future enhancements to OLE. Finally, *hWndParent* and *pRectPos* are useful to objects like video that wish to play in the context of the container without having to implement full in-place activation. The object is allowed to call GetDC(*hWndParent*) and draw within *pRectPos* (which is in client coordinates of *hWndParent*) during verb execution. The object is not, however, allowed to draw in the container window outside the scope of DoVerb. Even if you don't want to draw in the container's window, you can still use *pRectPos* to position your output or editing window relative to the object in the container.

The next most important member of IOleObject is **Close** that moves an object from the running state into the loaded state, that is, moves the object in the direction opposite DoVerb. This is also called when the container either closes the document in which this object lives (meaning we're going back to

1Not all pre-defined verbs are shown here since some are only relevant to in-place activation.

passive, not just loaded) or that the user has deleted the object from the container altogether, taking us far beyond passive to just plain non-existent. In any case, the object is generally required to execute two steps:

1. Close the document that's showing this object. If there are no other objects being editing in this instance of your application, this should start the shutdown process. As we saw before, closing a Schmoo document will destroy the object which will call ObjectDestroyed which will begin shutdown if there are no more objects.

2. When you destroy the window that displays the object, inform the container that the object is no longer visible by calling IOleClientSite::OnShowWindow(FALSE). This visually shows the end user that the object is reverting back to the loaded state.

Now there's a reason why I said that these four steps are "generally" required: the *dwSaveOption* parameter to this function which may be one of three values:

OLECLOSE_SAVEIFDIRTY	If you are not dirty, save yourself, then close.
OLECLOSE_NOSAVE	Just close
OLECLOSE_PROMPTSAVE	Display a message box with a message on the order of "This object has been changed. Do you wish to update <*container document*> before closing?"[1] with Yes, No, and Cancel buttons. If the user presses Yes, save yourself and close. If they press No, then just close. On Cancel, return OLE_E_PROMPTSAVECANCELLED without doing anything else. The <*container document*> string is passed in IOleObject::SetHostNames as we'll see in a later section.

The process of saving yourself has two steps which must be executes before steps 1 and 2 above (which is why these are numbered i and ii):

i. If your object has been modified, call IOleClientSite::SaveObject. The IOleClientSite pointer through which you call is the most recent one passed to SetClientSite. If you have no pointer, you cannot make the call, of course, and any lack of functionality is the container's fault. In Schmoo this is accomplished by calling the our own CFigure::SendAdvise with OBJECTCODE_SAVEDOBJECT. SendAdvise is just a handy function that verifies that we have a interface pointer through which to send a particular notification and so centralizes such pointer validation, simplifying the rest of the code.

ii. If you called IOleClientSite::SaveObject, call IAdviseSink::OnSave for every IAdviseSink you've seen in IOleObject::Advise. If you are using an advise holder, call IOleAdviseHolder::SendOnSave. Again, Schmoo handles this through CFigure::SendAdvise with OBJECTCODE_SAVED.

Finally, **SetHostNames** informs the object that it's being embedded in a container and is a signal to the server to show the appropriate user interface for embeddings. At this point in our development it's not necessary to fully implement this function, so we'll come back to this below in "Modify the Server's User Interface" after we take a short look at the optional functions in IOleObject.

Optional Functions

The four functions in this group, SetExtent, InitFromData, GetClipboardData, and SetColorScheme are not required for standard operation of compound documents, and so you can implement them as suits

1I could find no standard for this in any OLE 2.0 documentation. This message is a slight modification from OLE 1.0 guidelines where the word 'closing' is used in place of 'proceeding.'

your fancy.

SetExtent adds a nice touch to the interaction between a container and server and so I do recommend that you implement it. A container will call this function when it resizes an object in one of its documents. If it's appropriate for your server, you can use this to change the size of the object in your editing window. Schmoo, for example, scales the Polyline window (and the document window in which it lives) such that its as close to the size of the container's site as possible (and within reason). SetExtent works best for graphical objects but not so well for text or table objects where scale is much less important than the textual or numerical data. SetExtent is also sensitive to the display aspect which is passed in the *dwAspect* parameter.

InitFromData allows a container to do one of two things: either paste into your object directly or to provide initial data upon creation. This function is passed a pointer to an IDataObject which you can use to retrieve the data and a flag *fCreation* which indicates the scenario in which this function is being called. If *fCreation* is FALSE, then you should integrate the data in the data object with your current data as if Edit/Paste was performed in the server itself. If *fCreation* is true, then the container is attempting to create a new instance of your object based on a selection in the container which is described by the data object. For example, a spreadsheet application may pass a range of selected cells to a new chart object in a format that the chart object registered for the Set direction under DataFormats\GetSet in the registration database. Schmoo happens to treat both cases identically by passing the data object to CSchmooDoc::FPasteFromData. Again, if you have not read through Chapters 7 and 8, I highly recommend that you make a function that pastes data from any arbitrary data object such as those you can get from the clipboard, from drag-drop, or from a function like InitFromData.

GetClipboardData goes the other direction from InitFromData, asking the object for an IDataObject pointer that is identical to what the server would place on the clipboard if the user did Edit/Copy. This allows a caller to get a snapshot of the object as opposed to the IDataObject interface on the object itself which always reflects the most recent data. Therefore if you implement this function you have to return an IDataObject pointer for an object whose data will not change. Schmoo, in implementing OLE 2.0 clipboard handling in chapter 7, has this handy function CSchmooDoc::TransferObjectCreate which does the job for us.

Finally, **SetColorScheme** provides the object with the container's recommended palette. The object may choose to ignore this without any dire consequences, but if you can, try to use the colors provided.

But it Still Doesn't Work

After implementing and compiling this mammoth interface, you have almost all of the server side complete such that running Insert Object from a container will launch your application, obtain your class factory, create an object, and fire off calls like IPersistStorage::InitNew, IOleObject::SetClientSite, IOleObject::GetExtent, IOleObject::SetHostNames, and IOleObject::DoVerb such that your window will appear ready for editing. If you have implemented IOleObject::Close complete with calling IOleClientSite::SaveObject then when you close your application you will see a call to IPersistStorage::Save. When you activate the object from the container (with a quick double-click) you will see a call to IPersistStorage::Load followed by the same calls to IOleObject as before. You should at this point again be visible with the previously saved data ready for editing.

What is not happening is that there is no presentation in the container, or that whatever presentation is there is not being updated when you make changes in the server as OLE 2.0 servers should. In addition, when you closed your application you probably got a prompt that says "document has changed, do you want to save," and if you say yes, you get a File Save dialog. Well, that's not part of the OLE 2.0 interface for embedded object servers. In fact, there's nothing else to tell you that you are working with an embedded object as opposed to an untitled file. To solve both these problems we have to modify the application for embedded object UI as well as to round out the notifications we send to the container.

Modify the Server's User Interface

I will admit it. I seriously loathe writing user interface code because it's the one place that you cannot be the least bit wrong without someone noticing. In addition, user interface specifications seldom identify who is responsible for doing what and sometimes never articulate all possible cases. I guess that's why they're called "guidelines." The situation where you execute IOleObject::DoVerb and show a server window in which the user can edit the object is one such case that is not very well defined. Neither the OLE 2.0 Design Specifications nor the OLE 2.0 SDK documentation show what an embedded object server is supposed to look like as both documents concentrate solely on in-place activation. Such is the life of us in the trenches, shooting in the dark at an unknown and unseen target.

So what I'm describing here is not something I could call "official" but is more pieced together from what I've seen other applications doing. That's how we get standards in the first place.

All of the changes listed below should take place when your IOleObject::SetHostNames is called, because that function tells you a) that you are an embedded object and b) the names of the container application and container document that you need to make these changes. SetHostNames is always called before DoVerb, so these changes should be in effect before you show an editing window:

1. Remove the File New, Open, Close, and Save commands from your menus as well as any toolbar buttons that invoke the same commands.

2. Change File "Save As..." to "Save Copy As..." You can usually keep the same command identifier for this modified item. You may also want to remove any toolbar button for this function, but that's up to you. Save Copy As essentially creates an Export function that does not remember the filename after the copy is written. In addition, if you have an Import function, like Schmoo, you can leave that on the menu and toolbar. If you have a status line you may also want to change the message displayed for this item (which Schmoo does not do, mind you).

3. Change File "Exit" to "Exit and return to <*container document*>" where <*container document*> is the string pointed to by the *pszObj* parameter of SetHostNames. Again, you may want to change any toolbar and status line UI to accomodate this. (I know that it seems silly that the parameter containing the document name is called *pszObj*, or *pszContainerObj* in the OLE 2.0 documentation, but hey, it's only software, call it anything you like).

4. Change your title bar to read "<*object type*> in <*container document*>" where <*object type*> is the user-readable name of your object like "Schmoo 2.0 Figure" and <*container document*> is again the *pszObj* parameter from SetHostNames. If you are an SDI application or you are MDI but the document is maximized, then this string appears in the main application window's title bar prefixed with "<*application name*> - ". If you are an MDI application without a maximized document window, the frame caption remains the same and this string appears in the document's title bar.[1]

Schmoo affects these changes by calling CSchmooFrame::UpdateEmbeddingUI because the frame controls the menus and the GizmoBar. This is why our CFigure class needed to have a pointer to the frame which it received through CFigure::FrameSet. UpdateEmbeddingUI is actually capable of switching between an embedding state and a non-embedding state in case I ever decided to allow it to service multiple objects as well as other non-object documents as described in "MDI Servers, User Interface, and Shutdown."

1MDI automatically handles the maximized document case by concatenating the frame window's caption with " - " and the document's caption.

```
void CSchmooFrame::UpdateEmbeddingUI(BOOL fEmbedding, LPCDocument pDoc
   , LPCSTR pszApp, LPCSTR pszObj)
   {
   HMENU         hMenu;
   char          szTemp[256];

   //First let's play with the File menu.
   hMenu=m_phMenu[0];

   //Remove or add the File New, Open, and Save items
   if (fEmbedding)
      {
      DeleteMenu(m_phMenu[0], IDM_FILENEW,   MF_BYCOMMAND);
      DeleteMenu(m_phMenu[0], IDM_FILEOPEN,  MF_BYCOMMAND);
      DeleteMenu(m_phMenu[0], IDM_FILECLOSE, MF_BYCOMMAND);
      DeleteMenu(m_phMenu[0], IDM_FILESAVE,  MF_BYCOMMAND);

      //Save As->Save Copy As
      ModifyMenu(m_phMenu[0], IDM_FILESAVEAS, MF_BYCOMMAND, IDM_FILESAVEAS
         , PSZ(IDS_SAVECOPYAS));

      }
   else
      {
      InsertMenu(m_phMenu[0], 0, MF_BYPOSITION, IDM_FILENEW,   PSZ(IDS_NEW));
      InsertMenu(m_phMenu[0], 1, MF_BYPOSITION, IDM_FILEOPEN,  PSZ(IDS_OPEN));
      InsertMenu(m_phMenu[0], 2, MF_BYPOSITION, IDM_FILESAVE,  PSZ(IDS_SAVE));
      InsertMenu(m_phMenu[0], 3, MF_BYPOSITION, IDM_FILECLOSE, PSZ(IDS_SAVE));

      //Save Copy As->Save As
      ModifyMenu(m_phMenu[0], IDM_FILESAVEAS, MF_BYCOMMAND, IDM_FILESAVEAS
         , PSZ(IDS_SAVEAS));
      }

   //Change "Exit" to "Exit & Return to xx" or vice-versa for SDI
   if (fEmbedding)
      wsprintf(szTemp, PSZ(IDS_EXITANDRETURN), (LPSTR)pszObj);
   else
      lstrcpy(szTemp, PSZ(IDS_EXIT));

   ModifyMenu(m_phMenu[0], IDM_FILEEXIT, MF_STRING, IDM_FILEEXIT, szTemp);
   DrawMenuBar(m_hWnd);

   //Now let's pzlay with the gizmobar.
   m_pGB->Show(IDM_FILENEW,   !fEmbedding);
   m_pGB->Show(IDM_FILEOPEN,  !fEmbedding);
```

```
 m_pGB->Show(IDM_FILECLOSE, !fEmbedding);
 m_pGB->Show(IDM_FILESAVE,  !fEmbedding);

 //Enable what's left appropriately.
 UpdateGizmos();

 //Now play with the title bar.

 //IDS_EMBEDDINGCAPTION is MDI/SDI sensitive in SCHMOO.RC.
 wsprintf(szTemp, PSZ(IDS_EMBEDDINGCAPTION), pszObj);

 /*
  * Remember that in MDI situations that Windows takes care of
  * the frame window caption bar when the document is maximized.
  */
#ifdef MDI
 SetWindowText(pDoc->Window(), szTemp);
#else
 SetWindowText(m_hWnd, szTemp);
#endif

 return;
 }
```

When Schmoo is in the embedding state it appears as shown in Figure 10-3. Note that the Import... command is still on the File menu and toolbar.

§

Figure 10-3: Schmoo sporting its embedded object user interface.

Since we modified how certain menu items appear, we also need to modify how those command behave. First of all, to change Save As to Save Copy As you can either implement a new function or modify your existing save function. In either case Save Copy As is the same as a Save As except that you don't use the filename as the "active document" or anything to that effect. In other words, you write the file and forget it, not changing any other part of your user interface to reflect that file. It's just a way for the user to make a disk copy of the object if they so choose.

In Schmoo we can just modify CSchmooDoc::USave such that a Save Copy As does not make the document "clean" like a normal Save As would and that we don't store the filename in the document's structure or change the caption bar. USave determines that we're in an embedding state by calling CFigure::FIsEmbedded (which returns the value of CFigure's *m_fEmbedded* flag, set to TRUE in IOleObject::SetHostNames). You can see these changes in CHAP10\SCHMOO\DOCUMENT.CPP, which is not shown here.

We also changed File "Exit" to "Exit and return to <*container document*>". By itself there's no big change to the process of closing a document and closing the application. But when running normally, Schmoo always checks to see if the document is dirty when closing the document before exiting the application. If so, then it asks the user if they want to save the document to a file and so forth. Since saving the object as a file makes no sense in the embedding case (we removed File Save altogether) we need to prevent this prompt. So we modify CSchmooDoc::FDirtyGet to return FALSE if we're in the embedded state which effectively prevents prompting:

```
BOOL CSchmooDoc::FDirtyGet(void)
   {
   if (m_pFigure->FIsEmbedded())
      return FALSE;

   return m_fDirty;
   }
```

Now you are probably asking, "What if the object really is dirty? How do we make sure the container saves the object before we destroy it?" We need to tell the container to save the object when we're closing the document holding the object by calling IOleClientSite::SaveObject which is done in the CSchmooDoc destructor:

```
CSchmooDoc::~CSchmooDoc(void)
   {
   m_pFigure->SendAdvise(OBJECTCODE_SAVEOBJECT);

   ...
   }
```

And since I know you're getting sick of this CFigure::SetAdvise function, it's about time we looked at it and notifications in general. If you want, you can compile and test your application here to see that your user interface is set appropriately when SetHostNames is called and that your Save Copy As functionality works correctly. However, you won't work correctly with the container until you tell it everything that's going on.

Send Notifications

As far as a server is concerned I classify a number of different interface function calls as "notifications" although they are not all notifications in the strictest sense. True notifications are only those call to an IAdviseSink interface because they are all asynchronous. All others as completely synchronous, such as IOleClientSite::SaveObject. Nevertheless, I find it very convenient to lump these all together as 'notifications' in an embedding server. That enables us to make one function for our object which will send the right notification to the right interface if in fact. That function is CFigure::SendAdvise as shown below:

```
//SCHMOOLE.H
//Codes for CFigure::SendAdvise
//......Code.........................Method called in CFigureSendAdvise...
#define OBJECTCODE_SAVED          0  //IOleAdviseHolder::SendOnSave
#define OBJECTCODE_CLOSED         1  //IOleAdviseHolder::SendOnClose
#define OBJECTCODE_RENAMED        2  //IOleAdviseHolder::SendOnRename
#define OBJECTCODE_SAVEOBJECT     3  //IOleClientSite::SaveObject
#define OBJECTCODE_DATACHANGED    4  //IDataAdviseHolder::SendOnDataChange
#define OBJECTCODE_SHOWWINDOW     5  //IOleClientSite::OnShowWindow(TRUE)
#define OBJECTCODE_HIDEWINDOW     6  //IOleClientSite::OnShowWindow(FALSE)
#define OBJECTCODE_SHOWOBJECT     7  //IOleClientSite::ShowObject
```

```
/*
 * CFigure::SendAdvise
 *
 * Purpose:
 *  Calls the appropriate IOleClientSite or IAdviseSink member function
 *  for various events such as closure, renaming, saving, etc.
 */

void CFigure::SendAdvise(UINT uCode)
   {
   switch (uCode)
      {
      case OBJECTCODE_SAVED:
         if (NULL!=m_pIOleAdviseHolder)
            m_pIOleAdviseHolder->SendOnSave();
         break;

      case OBJECTCODE_CLOSED:
         if (NULL!=m_pIOleAdviseHolder)
            m_pIOleAdviseHolder->SendOnClose();

         break;

      case OBJECTCODE_RENAMED:
         //Call IOleAdviseHolder::SendOnRename (later)

         break;

      case OBJECTCODE_SAVEOBJECT:
         if (m_pDoc->m_fDirty && NULL!=m_pIOleClientSite)
            m_pIOleClientSite->SaveObject();

         break;

      case OBJECTCODE_DATACHANGED:
         //No flags are necessary here.
         if (NULL!=m_pIDataAdviseHolder)
            m_pIDataAdviseHolder->SendOnDataChange(m_pIDataObject, 0, 0);

         break;

      case OBJECTCODE_SHOWWINDOW:
         if (NULL!=m_pIOleClientSite)
            m_pIOleClientSite->OnShowWindow(TRUE);
```

```
        break;

    case OBJECTCODE_HIDEWINDOW:
        if (NULL!=m_pIOleClientSite)
            m_pIOleClientSite->OnShowWindow(FALSE);

        break;

    case OBJECTCODE_SHOWOBJECT:
        if (NULL!=m_pIOleClientSite)
            m_pIOleClientSite->ShowObject();

        break;
    }

return;
}
```

What this function really does for us is eliminate the need to check for NULL pointers anywhere else and to distill all 'notifications' down to one function and one parameter. We then don't have to remember which interface out of IAdviseSink, IOleClientSite, IOleAdviseHolder, and IDataAdviseHolder we use to send which notification to the container.

We've seen a number of places already in this chapter where we send various notifications and so we can collect them all into one list in Table 10-2. The only ones we haven't covered so far are all those for closing a document and that for data changes.

Table 10-2: When to Send Notifications from an Embedding Server

Event	Notifications (in the order shown)
Closing a document	IOleClientSite::SaveObject
(CSchmooDoc::~CSchmooDoc)	IOleClientSite::OnShowWindow(FALSE);
	IOleAdviseHolder::SendOnClose
	(all of this before CoDisconnectObject)
Data changes	IDataAdviseHolder::SendOnDataChange
(CSchmooDoc::FDirtySet)	
IOleObject::Close (if saving)	IOleClientSite::SaveObject
	IOleAdviseHolder::SendOnSave
IOleObject::DoVerb (HIDE)	IOleClientSite::OnShowWindow(FALSE)
IOleObject::DoVerb (SHOW)	IOleClientSite::ShowObject
(includes any verb that shows)	IOleClientSite::OnShowWindow(TRUE)

You'll notice that we don't use or implement OBJECTCODE_RENAMED anywhere because it's only used for linking and requires a moniker which we won't add until Chapter 13. In addition, you'll notice that the server never calls IAdviseSink::OnViewChange because a local server never implements IViewObject. Instead, the object handler watched IAdviseSink::OnDataChange and generates OnViewChange notification from that.

And now, as the stork would say, "Congratulations! You're a mother!" Well, at least a mother of an embedded object server that is fully functional with a container application. At this point you should be able to run Insert Object from a container to launch your server, make changes the object and see those changes reflected in a shaded object site. If changes are not being reflected in the container than either it's displaying a different aspect than the one you are changing, or when your IDataObject::GetData is asked for CF_METAFILEPICT or CF_BITMAP you're returning the wrong STGMEDIUM. In developing Schmoo I tore my hair out for a while trying to figure out why the container was not reflecting changes, and it was because the *tymed* I was storing in GetData's STGMEDIUM was TYMED_HGLOBAL instead of TYMED_MFPICT for the CF_METAFILEPICT format. Subtle, but ever so important.

If the container site is not shading itself, you may not be calling IOleClientSite::OnShowWindow at the appropriate times, or the container itself may be at fault. To determine if you are doing it correctly, try your server with a dependable container such as Chapter 9's version of Patron and the samples shipped with the OLE 2.0 SDK.

When you close your server you should see no prompts asking to save unless you get OLECLOSE_PROMPTSAVE in IOleObject::Close. In this case, and in the case when you delete the running object from a container, your server should be completely purged from memory. If not, then your shutdown conditions are not being met and so you are not closing your main window and exiting WinMain. If you are shutting down completely, you should see your object in a container document, and double-clicking on that object should again launch your server but this time you are asked to reload that saved object and edit that data instead.

The last two sections in this chapter deal briefly with the clipboard and MDI servers. The first is mostly a consideration for a full-server but is optional with a mini-server. The second is, of course, only important for full servers that want to service multiple embedded objects possibly with other non-embedded documents open.

(Full-Servers) Add OLE 2.0 Clipboard Formats

If you read through Chapter 9 you will already know that one way a container might create an object is to paste it from the clipboard or a drag-drop operation using the OleCreateFromData function (if you didn't know this, now you do). For this to happen something has to put embedded object data up on the clipboard in the first place, and that's left to a server application, regardless of whether its running to service an embedded object or running stand-alone.

Embedded object data is made of two formats: CF_EMBEDSOURCE and CF_OBJECTDESCRIPTOR which are exactly the same two formats we supported in the embedded object's implementation of IDataObject. The difference is that here we also need to provide these formats in the data object we place on the clipboard, whenever we so happen to do Edit/Copy or Edit/Cut. A mini-server may not have any clipboard user-interface which is why such a feature is generally for full-servers.

In Schmoo this means a modification to CSchmooDoc::TransferObjectCreate which is used in calling OleSetClipboard, in sourcing a drag-drop operation, and as we saw in this chapter, from IOleObject::GetClipboardData. Creating the two formats we're adding here is done in CSchmooDoc::FRenderMedium, exactly the function we used in implementing the object's

IDataObject::GetData function. Again, these formats should be placed in the data object after your private data but before any presentations.

I want to mention that when you call OleSetClipboard with a data object that contains CF_EMBEDSOURCE, OLE2.DLL will create the OLE 1.0 formats of "Native" and "OwnerLink" and place those on the clipboard as well. This allows OLE 1.0 containers to also paste in your object, a fact of which you remain blissfully ignorant.

(Optional) MDI Servers, User Interface, and Shutdown

If you would like to support multiple objects in one instance of your application through an MDI interface you have a few additional considerations and differences from what we've covered in this chapter.

1. Only full-servers can use MDI as mini-servers are always single-object servers.

2. Register your class factory with REGCLS_MULTIPLEUSE and remove any code in IClassFactory::CreateInstance that prevents creation of multiple objects.

3. When modifying your user interface on IOleObject::SetHostNames, save the container's application and document string with your document (for #4 below). Do not remove File New and Open but still remove File Close and Save and still modify the Save As and Exit items as described before. File New and Open just create new file-based documents as they always have which does not interfere with the document holding the embedded object.

4. If you have multiple documents open you need to switch your user interface between the embedded and non-embedded states as you change document windows. That means that switching to an embedded object document installs the embedded object UI, and switching to a normal document reinstates your normal UI. Because you will be switching the UI, you should hold on to the strings from IOleObject::SetHostNames because this function will not be called again.

5. Do not shut down when the last embedded object document is closed if the user has at *any time in the life of this application* used File New of File Open. Invoking either function passes control of the application to the user. Generally this means you should set a "user control" flag to FALSE initially and set it to TRUE on any File New or Open. When you test for shutdown in a function like ObjectDestroyed, don't close if this flag is TRUE. You should also make sure that you only close when the *last* object is closed, that is, your object count is truly zero.

6. Hiding an object through IOleObject::DoVerb(OLEIVERB_HIDE) should only hide its document window unless that's the only object in which case you also hide the frame window. If another object is created before the existing one is shown again, then you must show the frame window and the new document but not the existing document. In addition, whenever the visible document is closed and there is still a hidden document, hide the frame window again such that the whole server is in the state expected by the container that sent OLEIVERB_HIDE.

NOTE: As of the first draft these additions are not in the MDI version of Schmoo. I've not yet decided if I necessarily want to encourage MDI applications anymore since general policy is not to do it if you can avoid it...I'll come back to resolve this later.

Summary

A server application's support for the compound document standard means adding code to expose a class factory that creates embedded object. These objects support the IPersistStorage, IDataObject, and IOleObject interfaces which make them embeddable in any container application that is also written to its part of the compound document standard. Neither server nor embedded object need any specific knowledge about any container and will work transparently with any OLE 1.0 or OLE 2.0 container application.

There are two kinds of server applications: mini-servers and full-servers. The former can only manipulate one object per instance of the application and does not support linking. It may have a dialog-box type user interface as well. Full-servers are those applications that can also run stand alone and save and load their own files, and thus can support multiple objects per instance and providing linking support as discussed later in Chapter 13. The possible features for both types are a little different, but the implementation is the same.

This chapter focuses on the compound document server application (EXE) and how to implement its class factory and the embedded objects that the class factory creates. This chapter gives detailed step-by-step instructions to adding the necesary pieces including application initialization and shutdown, registration database entries, the class factory and shutdown mechanisms, the embedded object with IUnknown, the IPersistStorage, IDataObject, and IOleObject interfaces, embedded object user interface, and notifications that comminicate events and requests to the container. In addition, a server may want to provide additional clipboard formats in clipboard and drag-drop operations, and may want to support multiple objects as an MDI application. Both these latter features have additional requirements that are not required for a mini-server or a single-object server. More advanced server features are left for Chapter 14.