Chapter Nine
Compound Documents & Embedding Containers

There are things and there are places to put things.
Tony Williams, Microsoft OLE Architect

In our kitchen, my wife and I have probably somewhere on the order of 150 "Resealable Plastic Storage Units" (as Tupperware would have you call them) in all shapes and sizes, from tiny ones that barely hold a quarter-cup to tremendously volumous cathedrals that should hold salad to feed a cast of thousands. (OK, so I'm exaggerating slightly...) Some are square, some are round, some are annoying contortions that prohibit creative stacking in our freezer. Some are clear, some are bold "Seventies" colors (like Avocado), and some are recycled yogurt containers that still say "Sell by June 1, 1962" (OK, so I'm exaggerating again.) Some we obtained as gifts from mom, some we bought ourselves, and at least one I know of I found at a campground in the Cascade Mountains. (No, I'm not exaggerating.)

Besides the ubiquitous plastic, there are boxes, bins, pots, jugs, jars, cans, bags, baskets, bottles, and small paper packets containing powders with ingredient lists long enough to choke a Toastmaster. What lives inside these various storage units are just as diverse. Some contain dry goods like assortments of beans, lentils, split peas, whole-wheat stone-ground flour, rye flour, brown rice, couscous, millet, Wheat Chex, and spinach grown according to the California Organic Food Act of 1990. Others hold in frozen comfort at least five different kinds of soups while others live happily in the refrigerator with their treasures of last week's radishes, tomorrow's lunch, and usually some sort of edible yet unidentifiable leftover.

Now the problem with kitchen storage is that the stuff you want to put in any given container may not necessarily fit the container. Carrots, for example, do not lend well to storage in an egg carton. What we would really like is that any container, no matter how bizarre, has at least a basic set of attributes that allow it to contain any kind of stuff regardless of how otherwise bizarre that container might be. In addition, we would really like to have foods that all share a basic set of attributes that allow them to be fit into any of these standardized containers, regardless of how fantastic the food.

I doubt this will ever happen with food, but the same problem exists in computing with trying to integrate data from different applications into one centralized place that we call a compound document. It has always been relatively easy to create some kind of functional interface in two different applications that allow them to communicate and exchange their data. When the two applications are made for each other with intimate knowledge about the other, they fit together as well as eggs in an egg carton. But like an egg carton really doesn't work well to store anything but eggs, such a specialized interface in a computer program is not very useful to other applications besides the one for which it was written. What we would like is the ability to program the container, the application that collects and stores information, such that it generically can use any information from any other application without having to know any intimate details about that other application. We would also like the ability for an application that provides or serves data to have that data generically usable by any other application again without having to know anything about those others. What we want are standards for implementing both the container application and the server application, as they're called, such that they can communicate without knowing any of each others details. If a container application could treat all servers in the world identically, as a server could identically treats all containers, then data is generically sharable between everything. To do this requires a standard.

The standard is called Object Linking and Embedding[1] which is otherwise referred to as Compound Documents (with the capital C and D). In the compound document model (whenever I mention "compound documents" sans capitals I mean them in the OLE 2.0 sense) there are embedded and linked objects that are the units of exchange between whatever agent appears as the container and whatever agent appears as the server of those objects. All three pieces, object, container, and server, implement their parts of the standard through specific interfaces, and I mean OLE 2.0 Interfaces with the big "I." Anything that implements the container interfaces, for example, can act in the capacity of a compound document container regardless of what else it implements. The same is true for the server and object interfaces. This means that we can, in fact, fit lasagna noodles into a bottle of red wine vinegar without any trouble whatsoever. We can't do it in the kitchen, but hey, this is just software...anything is possible.

But the standard is so rich that it's going to take us from here through Chapter 17 to explain it all, from embedding to linking to in-place activation, so we'll start in this chapter with a full step-by-step treatment about how you write an application that can contain embedded objects, using Patron as the example. We start with containers because outside of in-place activation the container is the most active agent, that is, it provides the most user interface and really drives the servers and objects. Once we have developed a container, we can look at embedded object servers in Chapters 10 and 11, where Chapter 10 describes servers in EXEs whereas Chapter 11 covers those in DLLs, including what we call object handlers.

The other third of the OLE acronym, linking, is deferred until Chapter 12, which covers linking containers, and Chapter 13, covering linked object servers. Some containers and servers will never be concerned with linking, so I've separated it out in this manner to keep our discussion about embedding as uncluttered as possible.

Once we get past linking and embedding, there are a few more features for both containers and servers that are interesting which will be the topics in Chapter 14. That takes us up to in-place activation in Chapters 15 through 17. But before any of this, we need to look at how all the pieces of compound documents fit together: containers, object handlers, and servers in DLLs and EXEs.

Compound Document Mechanisms

The OLE 2.0 standard for compound documents involves not only the container application but also includes at least one DLL and possibly another application. In order to make sense out of what we'll be doing in this and subsequent chapters, we must explore how these pieces work together.

A compound document object can be in any one of three states: passive, loaded, and running. A *passive* object exists entirely on disk, that is, does not occupy any memory. In such a state the object cannot be displayed in any way and there are, of course, no pointers anywhere in memory to that object since there is no memory to point to. In order to get a pointer to an interface on that object, it must be *loaded* which means it does occupy some memory, there is at least one pointer to it, and it can be displayed, printed, or activated. When activated, the object is put into the *running* state where it may interact with an end-user for editing or other manipulation. A running object may also be activated in-place, but that is strictly a user interface issue.

In any given state there are specific modules in memory; by modules I mean the container application, the OLE 2.0 libraries, an object handler, and the object's server. The following sections describe exactly what's loaded into memory at any given time.

The Passive State

When an end user first launches an application that is an OLE 2.0 container, we have in memory the

1The fact that the product name of OLE 2.0 is so hard-wired to relate to compound documents is unfortunate, which is why we have the more generic Windows Objects to collectively describe all the technologies.

container application and some of the OLE 2.0 libraries, typically COMPOBJ.DLL and OLE2.DLL, as shown in Figure 9-1. Let's assume that at this point there is also a document on disk in a compound file where a storage object within that compound file contains all the information about an object, including the object's native data and possibly a cached presentation (a metafile or bitmap) for that object. We'll conveniently forget about how the object got there in the first place, but we'll come back to that later.

At this point the object is sleeping comatose in the passive state and the only loaded modules are the container application and the OLE 2.0 libraries. The container has only called a few functions like OleBuildVersion and OleInitialize in OLE2.DLL which is, of course, why that module and COMPOBJ.DLL are loaded. So the only things going on at this point are a few function calls between the container and the OLE 2.0 libraries. STORAGE.DLL is not yet loaded since the container has yet to access the compound file.

Figure 9-1: When a container application first starts, generally all that is in memory are some OLE 2.0 DLLs and the container application itself.

The Loaded State

Now the end user opens the compound file using the container's File Open command. The container calls a function like StgOpenStorage, which loads STORAGE.DLL, and begins reading data from the file as shown in Figure 9-2. At some point the container will encounter the object within the file at which time it will call the OleLoad function in OLE2.DLL, passing the IStorage pointer to the object in the compound file as shown in Figure 9-3. The container also passes the IID of the first interface pointer it wants on the loaded object, usually IUnknown or IOleObject (an interface we'll be using in this chapter and implementing in Chapter 10).

Figure 9-2: When a container reads from a compound file, STORAGE.DLL must be loaded into memory to access that file. STORAGE.DLL does use services provided in COMPOBJ.DLL as well.

Figure 9-3: When a container encounters an object in its compound file, it calls OleLoad to bring that object into the loaded state.

Now things get interesting because OleLoad wakes the object from the passive state into the loaded state, and a number of things happen. First, OleLoad calls ReadClassStg on the still disk-based object to determine the CLSID (which will have been put there by a previous call to OleSave). With that CLSID it calls our old friend CoCreateInstance with that CLSID and the parameters CLSCTX_INPROC_HANDLER | CLSCTX_INPROC_SERVER. As we learned in Chapter 4, CoCreateInstance (through CoGetClassObject) will go out to the registration database and attempt to find the entry for the CLSID. There will either be an entry for InProcServer, which is checked first, one for InProcHandler, or none at all in which case OleLoad (not CoCreateInstance) uses a default handler.

Case 1: InProcServer

The first possibility inside OleLoad is that the registration database has an entry under the CLSID\InProcServer key. In this case the InProcServer is completely responsible for providing the complete

implementation of the object as we'll see in Chapter 11. The InProcServer is structured with the DllGetClassObject export and contains and implementation of IClassFactory as shown in Figure 9-4. OleLoad, by virtue of calling CoCreateInstance, will load this DLL into memory in the container's process space and ask its IClassFactory to create a new object. OleLoad specifically asks CoCreateInstance to return a pointer to the object's IOleObject interface, regardless of the container's requested interface, so if the container asked for something else, OleLoad will QueryInterface that first pointer for the one the container wanted, and returns that pointer to the container as also shown in Figure 9-4.

Figure 9-4: When an InProcServer exists for the object's CLSID, the OleLoad function loads that DLL and returns a pointer to the container application that points into that DLL.

The interface pointer we now have in the container references an object that exists inside the InProcServer, not inside OLE2.DLL. Any calls through that pointer will enter the InProcServer first. The compound document object provided by the server must implement at least the IOleObject, IDataObject, IPersistStorage, and IViewObject interfaces as was shown in Figure 9-4. These interfaces, as we will see in Chapter 10, are what make any object appear to any container as a basic embedded compound document object.

Case 2: InProcHandler

If there is not an entry for InProcServer, or there is an error in loading the DLL listed in the registration database, then CoCreateInstance will attempt to load the DLL listed under the CLSID\InProcServer key (if it fails, then we go to case 3 below). An object handler looks exactly the same as an InProcServer, complete with the DllGetClassObject export and an implementation of a class factory. As far as the loaded state is concerned, there is no difference between an object handler and an InProcServer as you can see by comparing Figures 9-4 and 9-5 (below). The handler, just like the InProcServer, implements objects with at least IOleObject, IDataObject, IPersistStorage, and IViewObject, so from the container's point of view there is absolutely no difference.

Figure 9-5: To a container, an InProcHandler looks exactly like an InProcServer.

Case 3: The Default Handler

If there are any errors loading an InProcServer or an InProcHandler then OleLoad calls OleCreateDefaultHandler to load the default handler, which is, in fact, OLE2.DLL. Since OLE2.DLL is already in memory, albeit acting in a different capacity as just a library of functions, loading the default handler always works. What you will also notice in browsing the registration database using REGEDIT.EXE -v is that many servers directly register OLE2.DLL as their InProcHandler. This extra registration, while unnecessary, is recommended for the benefit of any end-users who trust themselves enough to run REGEDIT.

The default handler itself looks no different than any other handler or any other InProcServer as far as the container is concerned, so for an illustration just replace the word "InProcHandler" in Figure 9-5 with "Default Handler (OLE2.DLL)" and you have the whole picture.

This is not, however, the only time when OLE2.DLL may be loaded in its capacity as the default handler. As we learned in Chapter 6 with the Freeloader application, the default handler has a lot of

nice functionality within it to draw and serialize graphics and to maintain cached presentations for an object, among others. In fact, OLE2.DLL is chock full of codethat is useful to implementors of servers and handlers. So much in fact, that commonly InProcServers and InProcHandlers will themselves call OleCreateDefaultHandler to create an instance of, that is aggregate on top of, the default handler. To this instance they delegate any interface functions as well as complete interfaces for which the handler or server have no need to implement as shown in Figure 9-6. In this case handlers are much like bank tellers–as far as customers (or containers) are concerned, the tellers do everything for you themselves. But behind the scenes they are delegating much of the processing and handling of your account to other people that you never see. You know those other people are there, but you don't have any way of knowing or articulating what tellers do and what everyone else does.

Figure 9-6: Handlers aggregate on the default handler to expose interfaces that need no customization.

As we'll see in detail in Chapter 11, InProcHandlers that you implement will delegate much of their functionality to the default handler whereas InProcServers will only delegate only a little. The distinctions between handlers and servers, in fact, lie mostly in how much they delegate.

16- and 32-bit Interoperability

OLE 2.0 on Windows NT as well as on Win32s/Windows 3.1 presents some interesting problems for InProcServers and InProcHandlers. 16-bit OLE 2.0 running for a 16-bit container cannot load 32-bit DLLs into that 16-bit process space. Likewise 32-bit OLE 2.0 for a 32-bit container cannot load 16-bit DLLs. At the time of this writing there are no solutions to this other than avoiding DLLs altogether. An eventual solution would be to write both 16- and 32-bit handlers or InProcServers and store their respective paths under different keys in the registration database. Note that when LRPC is involved the marshaling mechanism takes care of the differences in process spaces, so there are no problems.

Loading the Object: All Cases

Before OleLoad actually returns the container's requested interface pointer, it will itself QueryInterface the new object for IPersistStorage and call IPersistStorage::Load passing to the object the IStorage pointer provided by the container. As far as the object is concerned, it loads whatever is necessary to operate by calling member functions in IStorage and IStream which access the compound file through STORAGE.DLL as shown in Figure 9-7. The object then holds onto a copy of the IStorage pointer (after calling IStorage::AddRef of course!) if it wants to have incremental access to the storage for its lifetime as described in Chapter 5 in "Other OLE 2.0 Technologies and Structured Storage."

Figure 9-7: Handlers and InProcServers access the object's data on disk through the IStorage pointer provided by the container.

At this point the object is officially "loaded" and OleLoad returns an interface pointer to the

container application. OleLoad is only one of the many ways in which a container obtains the first pointer to a compound document object as we'll explore in this chapter. In any case, the container may now ask the object to perform tasks, such as drawing itself.

Drawing the Object

The vast majority of compound document containers will be interested in showing a graphic for the object after loading. It would look pretty silly to load a word-processor document to see the text but to be faced with large black holes where there should be charts or other types of object pictures. So a container application will eventually QueryInterface for IViewObject and call IViewObject::Draw (remember that the OleDraw function does this internally). This call, of course, first enters the IViewObject implementation inside the loaded InProcServer or InProcHandler but there are a number of things that might happen here.

First, the server or handler may draw the object directly on the hDC given in IViewObject::Draw using the data loaded from the object's storage. In this case, we're done with one quick function call to a DLL. Both InProcServers and InProcHandlers can choose to implement IViewObject:Draw for whatever aspects they want and depend on the default handler to draw the other aspects.

This brings us to the second case where the default handler has to come up with some way to draw the object. Unfortunately the default handler's clairvoyance skills are weak and thus cannot use the object's native data sitting out in storage to render some image. So the default handler likes to keep a object photo album with pictures of each aspect requested by the container; when the container asks to see the object in a particular aspect the default handler can show it one of the photographs in the album. OLE 2.0's photo album is a thing called the Cache, a piece of code in OLE2.DLL that implements the IDataObject, IViewObject, IPersistStorage, and IOleCache interfaces to maintain presentations in memory and on disk (in IStorages) and to draw and exchange them. Technically all handlers must implement the IOleCache interface, but rarely will you find it necessary to implement your own. Handlers generally delegate all IOleCache functions to the default handler, as well as some IDataObject and IPersistStorage members, which delegates directly to the cache implementation. Through aggregation, the default handler aggregates on the cache and provides the cache's IOleCache pointer as it's own. Likewise handlers generally expose the default handler's IOleCache interface as their own, which is directly exposing the cache's interface to the outside world as shown in Figure 9-8.

Figure 9-8: Any IOleCache interface exposed on a handler or the default handler is usually the IOleCache interface implemented on the cache itself.

The cache manages one or more presentations of the object within the object's IStorage on disk. Again, IOleCache is the mechanism through which you specify what should be in the cache. The cache's IPersistStorage interface is how we get presentations to and from disk, and IDataObject interface is how we ask the cache to return a memory copy of a presentation. These presentations first got here when the object was initially created, which again, is something we don't want to get involved with yet. So imagine, for example, that there is only one presentation in the cache: a metafile containing DVASPECT_CONTENT and a NULL target device (meaning it was rendered for the screen).

When the container asks to draw DVASPECT_CONTENT on the screen, it calls IViewObject::Draw as it always would. One way or another this request works its way into the default handler's implementation of IViewObject::Draw, and not knowing anything about the object, the default

handler has to go see if there's a suitable presentation in the cache by calling the cache's IDataObject::GetData. In this example the cache will locate the content metafile with a NULL target device, which is suitable, so it loads that presentation from storage and returns it to the default handler which in turn draws that metafile to the hDC passed to IViewObject::Draw.

This now brings us to the third case where the container asks IViewObject::Draw to render some really funky FORMATETC such that the default handler asks in the cache who responds "nope, ain't got none of those." The handler can now choose to outright fail, which is terribly rude. Instead, it elects to put the object into the running state which is the last resort to getting a presentation.

The Running State

This third state of an object means that there is "one-stop shopping" available for anything you want to do with the object, and if the capability is not there, it's not available anywhere. So Both InProcServers and what we know as LocalServers, or servers implemented in EXEs, are considered the Fred Meyer or Wall-Mart[1] of Objects. When an InProcServer exists, there is no difference between the loaded and running states. When there is not an InProcServer available, then the object is loaded when a handler is present and running only when the LocalServer has been launched. But when is it launched?

When there is no InProcServer available, the handler attempts to do everything possible to avoid having to launch the LocalServer in order to service a request. However, there will eventually be some request outside the capabilities of the handler as the example in the previous section where the container wanted the object to draw some funky FORMATETC. The handler, having tried so hard to avoid this moment, capitulates into calling CoCreateInstance with CLSCTX_LOCALSERVER, which as we saw in Chapter 4 launches the EXE listed under the CLSID\LocalServer key in the registration database. This server must call CoRegisterClassObject on startup to provide OLE 2.0 with its IClassFactory pointer which is then asked to create a new object which must have the IOleObject, IDataObject, and IPersistStorage interfaces to qualify for compound documents.[2] Once CoCreateInstance returns to the default handler with the appropriate interface pointer to that new object, the default handler makes a number of calls to put this new object into the same state as the one in the handler. Yes, we do have two instances of an object of the same CLSID, and they much now be synchronized. So the handler calls the new object's IPersistStorage::Load as shown in Figure 9-9, again passing the same IStorage pointer, along with a number of others that are not important for this discussion. Once the object has had a chance to load its data, the handler calls IDataObject::GetData passing the FORMATETC it could not render itself. If the object fails this GetData, then drawing just doesn't happen, otherwise the handler gets the metafile and draws it to the container's hDC.

Figure 9-9: When the object is put into the running state by the handler, OLE 2.0 launches the LocalServer EXE which provides a complete implementation of the object.

The other most common case where an object must be put into the running state is when the container calls IOleObject::DoVerb which up to this pointer we've called *activating* the object. DoVerb asks the object to execute some action such as playing a wave file or showing the object's data in a window in which the end user can make modifications. Since the definition of a verb is the sole responsibility of the object and its server, there is absolutely no chance that the default handler will know what to do with it. Some custom handlers themselves may know how to execute certain verbs, but for the most part handlers do not implement this function. So eventually the request again works

1HACK: Can anyone think of stores that advertise something like "if you didn't find it here, you can't get it at all?"
2Note again that the LocalServer cannot implement IViewObject because the hDC parameters in IViewObject::Draw cannot be marshalled.

into the default handler which launches the LocalServer to provide the request. And again, if an InProcServer is loaded, it is responsible for implementing this function itself because an InProcServer usually exists exclusive of a LocalServer.

Eventually, either programmatically or through user action, the LocalServer shuts down which takes the object from the running state back into the loaded state. If the handler again needs the services of the LocalServer, it again must relaunch the EXE and start the whole process over again. Overall, the transitions between all three states, Passive, Loaded, and Running, are illustrated in Figure 9-10. Note that it's the function OleLoad that moves from passive to loaded with a Release on the interface returned from OleLoad that moves from loaded to passive. An object is ultimately moved from loaded to running hidden using the OleRun API which is called from within the default handler's implementation of IOleObject::DoVerb (as well as a few other interface members). DoVerb with OLEIVERB_SHOW moves the object from running hidden to running visible. DoVerb with OLEIVERB_HIDE moved from running visible to running hidden. IOleObject::Close moves any object from running (regardless of visibility) to loaded once again.

Figure 9-10: Various function calls and events that cause state transitions.

OLE 1.0 and OLE 2.0 Interoperability

OLE 2.0 provides an OLE 1.0 compatibility layer that sits between a container and a server application (EXE) such that one of the applications can be written to OLE 1.0 and the other to OLE 2.0 without either knowing the difference. Essentially OLE 2.0 is translating the OLE 1.0 interfaces on one side into OLE 2.0 interfaces on the other. Therefore OLE 2.0 containers have full access to all OLE 1.0 servers and all OLE 2.0 servers are usable from OLE 1.0 containers. This allows you to upgrade your application to OLE 2.0 knowing that you will not alienate the OLE 1.0 market.

But truthfully, it's not that perfect because of these handler things. Since an OLE 2.0 container talks directly to a handler, it cannot use an OLE 1.0 handler. (This is not too unfortunate because there are very few OLE 1.0 handlers. Windows 3.1 Paintbrush is one of the few.) Likewise an OLE 2.0 handler or InProcServer cannot be used by an OLE 1.0 container. The differences between 1.0 and 2.0 designs simply do not allow it, so if you are thinking about handlers or InProcServers at this point, you may need, for at least a little while, to maintain both OLE 1.0 and OLE 2.0 versions of your DLLs.

Mommy, Daddy, Where do New Objects Come From?

Um, well, ah, you see, there's, ah, a stork. Yeah. The Object Stork.

Sure, such an explanation might work on a two-year-old[1] but I don't think it works on programmers, so there must be a better explanation, and hopefully one that would pass indecency laws. Container applications create new objects using the function OleCreate which is typically called after the container invokes the Insert Object dialog and the user has selected a name and pressed OK. Before calling OleCreate you cannot say the object is in the passive state because there is not yet an object (maybe it's

1Parents correct me. I don't have children yet so I don't quite know when they start asking.

in the waiting-for-reincarnation state). OleCreate creates a new object that is immediately in the loaded state, that is, a handler or InProcServer has been loaded for it. Instead of OLE 2.0 calling IPersistStorage::Load as it does in OleLoad, it calls IPersistStorage::InitNew. After that, everything is pretty much the same, and the handler will launch the LocalServer for the same reasons as before. As we'll see, containers will usually call IOleObject::DoVerb shortly after calling OleCreate to immediately bring the object up for editing.

Containers may also create new objects from a data object (on the clipboard or from a drag-drop operation) when either CF_EMBEDSOURCE or CF_EMBEDDEDOBJECT formats are available. Both these formats are an IStorage containing the object's native data as it would appear on disk but with no presentation caches, so the existence of these formats essentially means that an object exists in the passive state inside the data object. When a container does Edit/Paste or accepts a drop as well implement in this chapter, it can call OleCreateFromData which does everything OleLoad does except that the data passed to IPersistStorage::Load comes from the data object instead of from the containers document file. But again, OleCreateFromData transitions the object from the passive state into the loaded state.

The Structure of a Container Application

Now that we understand how compound documents generally work and what each state of an object implies, we can start our exploration of containers capable of managing embedded objects. The previous section explained how objects look to a container: they always have the IOleObject, IDataObject, IPersistStorage, IViewObject, and IOleCache interfaces, regardless of how many servers, handlers, or caches are in use. That's the object's part of the compound document standard, so now it's time to see the container's part.

Compound documents affect a container application on all levels which I define as Application, Document, Page, and Site,[1] where the site is a place on a page, a page is part of a document (in some cases the page is the document, or there is no distinction between a document and the pages in the document, so the structures merge[2]), and the document is managed by the application which may have more than one document open at any given time as shown in Figure 9-11. Patron is a good example of this structure where Patron (the application) loads documents consisting of pages on which reside tenants (the sites). Each site is a place for one object, or one graphic in Patron's case. As we will see, each site is responsible for implementing the IOleClientSite and IAdviseSink interfaces in order to use an embedded object.

Figure 9-11: The general structure of container applications.

Let me illustrate this with what I call The Allegory of the Cookie Jar[3] which will serve as a useful example in later chapters as well. An object is like a batch of cookies. Different cookies have their own form and taste just like different objects have different classes and behavior. Cookies are great by themselves right out of the oven, but eventually we need to store the batch somewhere or else they'll quickly become stale or will be devoured by an rabid pack of mongrel specter hounds. Where we decide to store these cookies depends on how we later want to access them. For this chapter, we have Embedded Cookies and they must always be stored inside a cookie jar (in Chapter 12 we'll see Linked

1In MFC 2.0 these are called App, Document, View, Item (in view).
2A word processor generally doesn't have a separate notion of pages since it will be constantly repaginating. Spreadsheets as well only define pages when printing. However, databases can view a database as the document with specific forms or tables as the pages, with specific fields in those forms or tables as the sites.
3With, of course, apologies to Plato.

Cookies where we put a treasure map in the jar that tells you where they really are). In the same way, an embedded object must always be stored at some container site. Our cookie jar is thus something we can call a site.

On the outside of this cookie jar is generally some sort of representation of the cookies inside. For our purposes, let's imagine a high-tech cookie jar that has a video camera on the inside that transmits a picture of the current cookies to a small screen on the outside, such that at any time we can see exactly what's in the jar.[1] This is equivalent to always having a presentation for an object that reflects the object's current native data. A broken video camera or a loss of power is like having a handler that cannot find a suitable presentation or a server that can render one. So we have a cookie jar that always shows what's in it, like we have a site that always displays a picture of the object it maintains.

That's all well and done, but the cookie jar itself needs a place to sit too, like a shelf or a countertop, just like a site needs a page in which to live. But even the shelf needs a place, like in the kitchen, just like a page needs a document. And what good is a kitchen without a house around it? What good is a document without an application that knows how to open it?

What I call a document, a page, and a site will, of course, vary from application to application. In a database, for example, the database is the document, the page is a table or a record, and the site is a specific field in that record or table. Whatever the application, you can generally find structures that fit this model. This brings us to where we can see what we need to do at each level to become a compound document container.

Step-By-Step Embedding Containers

The remainder of this chapter will follow modifications I made to Patron for it to become an container for embedded objects. There are changes at many levels, and much of the necessary work involves user interface. If we return to our cookie jar for a moment, the cookies only need to define user interface insofar as much as how they look, feel, and taste; very cookie-oriented sort of stuff. The cookie jar has to define how it opens and how it looks from the outside. In the same manner the shelf on which the cookie jar sits has its own interface of color and dimension, just like the kitchen and the house have to define their own characteristics. So much like most of the user interface in a house is shown in rooms, shelves, and storage devices, most of the user interface in embedding compound documents falls to the container.

Nevertheless, we can reduce implementation down to a clean sequence of steps shown below. Each is designed such that you can at least compile after coding the step and in most cases you also have something you can run and test. I strongly recommend that you test as much as you can in the early steps as the later ones build on these foundations:

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. Define what is to be a site in your application and manage unique IStorage instances with each site as sites are created and destroyed.

3. Make your sites Windows Objects and implement the IOleClientSite and IAdviseSink interfaces on the sites as well as adding variables to manage an embedded object in this site.

4. Implement a function to shade the site when the object it contains transitions to and from the running state.

5. Invoke the Insert Object dialog, call OleCreate to create a new embedded object, and call a

1Note that cookie jars in OLE 2.0 do not have locking lids. That's left to future revisions that implement object security.

number of follow-up functions to initialize the object after creation.

6. Draw and print the object's presentations.

7. Activate the object on a double-click and add a menu that lists the object's verbs and activate the object when this menu is selected. Optionally implement simple right-mouse button popup menus on the object.

8. Call OleCreateFromData to create an embedded object from a data object (from the clipboard and drag-drop). Optionally invoke the Paste Special dialog to allow the user to select the exact format to paste from the clipboard.

9. Provide new data formats to copy an embedded object back to the clipboard or to source it in a drag-drop operation.

10. Delete objects from the document. Remember to CoFreeUnusedLibraries after deleting an object.

11. Save and load documents containing embedded objects.

12. (Optional) Handle iconic presentation aspects and control the cache as necessary.

After following these 12 steps in your own application, you'll end up with an embedding container that can communicate with both OLE 1.0 and OLE 2.0 servers.

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) (under Windows 3.1 only), OleBuildVersion, and OleInitialize on startup and OleUnintialize (and possibly OleFlushClipboard) on shutdown. As containers we use both storage objects but also embedded compound document objects, among others.

Remember 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.

Define Sites and Manage Site Storage

Before going any further you need to decide what exactly you want to act as a site in your container application. A site has a one-to-one correspondence with a embedded object, so for every object you wish to contain you'll need to instantiate your site structure. If you only plan to embed a single object, then of course you only need one site. Patron's sites are its tenants, implemented by its C++ class CTenant. Invariably you too will have some sort of data structure or C++ class to use in this regard, so now is the time to create one if you don't have anything like it yet.

Since Patron is essentially embedding bitmaps and metafiles already, much of what I need to make it a container is already present. One of the most important requirements of a site is that each site manages an IStorage instance that is for the exclusive use of this site and any embedded object it contains. Patron again, already has a storage for each tenant which it passes to a function like OleSave (or more accurately IPersistStorage::Save) such that the object can save its data appropriately.

As a container you absolutely must provide every embedded object with some instance of IStorage. I say *some* because the object never knows nor cares where that storage actually exists. If you decided back in Chapter 5 that you could use Compound Files for your application's documents then you only need to create sub-storages within that document for each site. If, however, you need to preserve an older file format you can still create storage objects in memory and write their contents to you own disk

file.

Creating a memory IStorage for an object and to writing to a file involves five steps:

1. Call CreateILockBytesOnHGlobal with a NULL memory handle. OLE 2.0 will allocate memory for you and return an ILockBytes pointer.

2. Call StgCreateDocfileOnILockBytes to obtain an IStorage pointer.

3. Store whatever object data is necessary in the IStorage. Saving objects is covered in a section later in this chapter (we need something to save first) and generally means calling OleSave.

4. When you are ready to write to your file, call GetHGlobalFromILockBytes to obtain the handle to the memory containing the object data.

5. Call GlobalSize to determine the length of the data, GlobalLock the handle, write the data, the GlobalUnlock the handle. Remember to write the size of this data somewhere in your file so you know how much to allocate when you want to load the object again. When you are finished, you need only call IStorage::Release which will release the ILockBytes and release the memory in one swift stroke.

In the other direction you again have five steps:

1. Read the size of the data block from your file and allocate that much global memory.

2. Call CreateILockBytesOnHGlobal passing the memory allocated in step 1.

3. Call StgOpenStorageOnILockBytes to obtain an IStorage pointer that can access the object data.

4. Read the necessary data from the storage, usually by calling OleLoad.

5. When you no longer need the storage, remember to call IStorage::Release. Generally you will keep this storage around as long as the object is in the loaded state.

The OLE2UI library has a function called OleStdCreateIStorageOnHGlobal that wraps the CreateILockBytesOnHGlobal and StgCreateDocfileOnILockBytes or StgOpenStorageOnILockBytes functions. Calling it, of course, is equivalent to calling these functions yourself. Whatever works best.

Regardless of how you decide to treat storage, you want to make sure that each site has a storage available when the site is created and that you properly clean up that storage when the site is closed or destroyed. If you wanted to test committing the storage when the site is closed you can write some bogus stream of garbage into the storage and verify using a tool like the OLE 2.0 SDK's DFVIEW.EXE to make sure that the data is actually making it into the disk file.

Back in Chapter 5 I had briefly mentioned that any stream or storage with a name prefixed with ASCII 3 is for the exclusive use of a container application, and now I can put that into context. When you hand an embedded object a storage it (as well as OLE 2.0) has full control over everything within that storage with the exception of any element whose name starts with ASCII 3. Therefore if you wanted to write a stream of site-related information into the storage you pass to functions like OleSave and OleLoad, prepend "\0x03" or else who knows what could happen. This essentially marks those elements as "containers only" as illustrated in Figure 9-12.

Figure 9-12: All elements in a storage prefixed with \0x03 are for exclusive use of a container application, even if that storage is passed to an embedded object for its use as well.

You can again compile and verify your storage operation. In addition, if you know you are going to want to write specific site-related information into the site's storage (which Patron does not, by the way), now is a great time to implement the necessary structures in your code to handle those streams, even if they don't yet contain useful information. Just be sure to prefix their names with a "3"!

Implement Site Interfaces and Add Site Variables

"Sites do not live on storage alone," the maxim goes, "they need interfaces as well." In order to qualify a site as a place for an embedded object, the site must now become a Windows Object with two interfaces: IOleClientSite and IAdviseSink. Having these interfaces makes the site appear as a container to the outside world as shown before in Figure 9-11.

Of course, by virtue of implementing one of these interfaces you will have an implementation of IUnknown. To illustrate these two interfaces and what a site object really needs to hold an embedded object, let's look at Patron's CTenant implementation as revised for this chapter. First, I moved all tenant-related definitions into TENANT.H as shown in Listing 9-1. You'll see that I've added two interface implementation classes, CImpIOleClientSite and CImpIAdviseSink, defined a few new symbols, and added a number of new variables and functions to CTenant.

TENANT.H

```
[Unmodified sections omitted]

class __far CImpIOleClientSite : public IOleClientSite
  {
  protected:
    ULONG             m_cRef;     //Interface reference count.
    class CTenant FAR * m_pTen;     //Back pointer to the object.
    LPUNKNOWN         m_punkOuter; //Controlling unknown for delegation.

  public:
    CImpIOleClientSite(class CTenant FAR *, LPUNKNOWN);
    ~CImpIOleClientSite(void);

    STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    STDMETHODIMP SaveObject(void);
    STDMETHODIMP GetMoniker(DWORD, DWORD, LPMONIKER FAR *);
    STDMETHODIMP GetContainer(LPOLECONTAINER FAR *);
    STDMETHODIMP ShowObject(void);
    STDMETHODIMP OnShowWindow(BOOL);
    STDMETHODIMP RequestNewObjectLayout(void);
  };

typedef CImpIOleClientSite FAR * LPIMPIOLECLIENTSITE;


class __far CImpIAdviseSink : public IAdviseSink
  {
  protected:
    ULONG             m_cRef;     //Interface reference count.
    class CTenant FAR * m_pTen;     //Back pointer to the object.
    LPUNKNOWN         m_punkOuter; //Controlling unknown for delegation.

  public:
    CImpIAdviseSink(class CTenant FAR *, LPUNKNOWN);
    ~CImpIAdviseSink(void);

    STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    STDMETHODIMP_(void)  OnDataChange(LPFORMATETC, LPSTGMEDIUM);
```

```
    STDMETHODIMP_(void)  OnViewChange(DWORD, LONG);
    STDMETHODIMP_(void)  OnRename(LPMONIKER);
    STDMETHODIMP_(void)  OnSave(void);
    STDMETHODIMP_(void)  OnClose(void);
  };

typedef CImpIAdviseSink FAR * LPIMPIADVISESINK;

...

//Tenant types (not persistent, but determined at load time)
typedef enum
  {
  TENANTTYPE_NULL=0,
  TENANTTYPE_STATIC,
  TENANTTYPE_EMBEDDEDOBJECT,
  TENANTTYPE_EMBEDDEDFILE,
  TENANTTYPE_EMBEDDEDOBJECTFROMDATA
  } TENANTTYPE, FAR * LPTENANTTYPE;


//State flags
#define TENANTSTATE_DEFAULT     0x00000000
#define TENANTSTATE_SELECTED    0x00000001
#define TENANTSTATE_OPEN        0x00000002


class __far CTenant : public IUnknown
  {
  friend CImpIOleClientSite;
  friend CImpIAdviseSink;

  private:
    HWND          m_hWnd;          //Pages window, same as CPages.
    DWORD         m_dwID;          //Persistent DWORD identifier
    DWORD         m_cOpens;        //Count calls to FOpen.

    BOOL          m_fInitialized;  //Something here?
    LPUNKNOWN     m_pObj;          //The object that lives here.
    LPSTORAGE     m_pIStorage;     //Sub-storage for this tenant

    FORMATETC     m_fe;            //As used to create the object.
    DWORD         m_dwState;       //State flags
    RECTL         m_rcl;           //Space of this object.

    class CPages FAR *m_pPG;       //Pages window

    TENANTTYPE    m_tType;         //Type identifier
    ULONG         m_cRef;          //We're an object now!
    LPOLEOBJECT   m_pIOleObject;   //IOleObject on m_pObj
    LPVIEWOBJECT  m_pIViewObject;  //IViewObject on m_pObj

    LPOLECLIENTSITE m_pIOleClientSite;  //Our interfaces
    LPADVISESINK  m_pIAdviseSink;

  protected:
    BOOL  FObjectInitialize(LPUNKNOWN, LPFORMATETC, DWORD);
    HRESULT CreateStatic(LPDATAOBJECT, LPFORMATETC, LPUNKNOWN FAR *);

  public:
    CTenant(DWORD, HWND, CPages FAR *);
    ~CTenant(void);

    //Gotta have an IUnknown for delegation.
    STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    DWORD   GetID(void);
    UINT    GetStorageName(LPSTR);
    UINT    UCreate(TENANTTYPE, LPVOID, LPFORMATETC, LPPOINTL
            , LPSIZEL, LPSTORAGE, LPPATRONOBJECT, DWORD);
```

```
    BOOL    FLoad(LPSTORAGE, LPFORMATETC, LPRECTL);
    BOOL    FOpen(LPSTORAGE);
    void    Close(BOOL);
    BOOL    Update(void);
    void    Destroy(LPSTORAGE);

    void    Select(BOOL);
    void    ShowAsOpen(BOOL);
    void    ShowYourself(void);
    void    AddVerbMenu(HMENU, UINT);
    void    CopyEmbeddedObject(LPDATAOBJECT, LPFORMATETC, LPPOINTL);
    void    NotifyOfRename(LPSTR);

    BOOL    Activate(DWORD);
    void    Draw(HDC, DVTARGETDEVICE FAR *, HDC, int, int, BOOL, BOOL);
    void    Repaint(void);
    void    Invalidate(void);

    void    ObjectGet(LPUNKNOWN FAR *);
    void    FormatEtcGet(LPFORMATETC, BOOL);
    void    SizeGet(LPSIZEL, BOOL);
    void    SizeSet(LPSIZEL, BOOL);
    void    RectGet(LPRECTL, BOOL);
    void    RectSet(LPRECTL, BOOL);
  };

typedef CTenant FAR * LPTENANT;

...
```

Listing 9-1: Site-related definitions and classes for the Container version of Patron.

Tenants have a number of new variables that I find necessary to maintain an embedded object in this site:

m_tType	(TENANTTYPE) Describes what lives at this site, either a static object or an embedded object. This will expand in later chapters to include other types, such as linked.
m_cRef	(ULONG) Reference count for the tenant which implemented an IUnknown interface and deletes itself when this reference count is zero.
m_pIOleObject	(LPOLEOBJECT) A pointer to the object that lives at this site. This is an interface we use, not implement.
m_pIViewObject	(LPVIEWOBJECT) A secondary pointer object obtained from QueryInterface on m_pIOleObject. This is another used interface, not implemented.
m_pIOleClientSite	(LPOLECLIENTSITE) A pointer to the tenant's implementation of IOleClientSite.
m_pIAdviseSink	(LPADVISESINK) A pointer to the tenant's implementation of IAdviseSink.

Patron initializes all these variables to zero or NULL in CTenant::CTenant:

```
CTenant::CTenant(DWORD dwID, HWND hWnd, LPCPages pPG)
    {
    [Other initialization]

    m_cRef=0;
    m_pIOleObject=NULL;
    m_pIViewObject=NULL;

    m_pIOleClientSite=NULL;
    m_pIAdviseSink=NULL;
    return;
    }
```

In the destructor, CTenant::~CTenant, we must release the interfaces we're using and delete the interface implementations we might have created (we'll come back to that IViewObject::SetAdvise call in a later

section):

```
CTenant::~CTenant(void)
  {
  if (NULL!=m_pIViewObject)
    {
    m_pIViewObject->SetAdvise(m_fe.dwAspect, 0, NULL);
    m_pIViewObject->Release();
    }

  if (NULL!=m_pIOleObject)
    m_pIOleObject->Release();

  if (NULL!=m_pIAdviseSink)
    delete m_pIAdviseSink;

  if (NULL!=m_pIOleClientSite)
    delete m_pIOleClientSite;

  [Other cleanup]

  return;
  }
```

The interfaces we implement on a tenant are initialized in CTenant::FOpen after creating or opening the IStorage for this tenant:

```
m_pIOleClientSite=new CImpIOleClientSite(this, (LPUNKNOWN)this);
m_pIAdviseSink=new CImpIAdviseSink(this, (LPUNKNOWN)this);

if (NULL==m_pIOleClientSite || NULL==m_pIAdviseSink)
  return FALSE;
```

Both of these interface implementation delegate IUnknown functions to the tenant, which itself now implements IUnknown (notice that CTenant inherits from IUnknown as well). CTenant's QueryInterface knows IUnknown, IOleClientSite, and IAdviseSink:

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

  if (IsEqualIID(riid, IID_IUnknown))
    *ppv=(LPVOID)this;

  if (IsEqualIID(riid, IID_IOleClientSite))
    *ppv=(LPVOID)m_pIOleClientSite;

  if (IsEqualIID(riid, IID_IAdviseSink))
    *ppv=(LPVOID)m_pIAdviseSink;

  if (NULL!=*ppv)
    {
    ((LPUNKNOWN)*ppv)->AddRef();
    return NOERROR;
    }

  return ResultFromScode(E_NOINTERFACE);
  }
```

In addition, AddRef and Release now control the tenant's lifetime where Release will call delete this when the reference count is zero. Because of this, any other piece of code that uses a tenant, primarily that in PAGE.CPP which has pointers like *pTenant*, now call pTenant->Release instead of delete pTenant as you can see in a function like CPage::Close. You can find the matching AddRef in CPage::FTenantAdd which happens

There are also a number of new functions in CTenant: FObjectInitialize, ShowAsOpen, ShowYourself, AddVerbMenu, CopyEmbeddedObject, an NotifyOfRename, which are used to implement some of the additional container requirements we'll see in later sections. Some of these are

called directly from the implementations of IAdviseSink and IOleClientSite.

Implement IAdviseSink

Container sites need an implementation of IAdviseSink solely for receiving IAdviseSink::OnViewChange notifications. All other member functions need only be stubbed as shown in Patron's implementation of Listing 9-2.

IADVSINK.CPP

```
/*
 * IADVSINK.CPP
 * Implementation of the IAdviseSink interface for Patron's tenants.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "patron.h"

/*
 * CImpIAdviseSink::CImpIAdviseSink
 * CImpIAdviseSink::~CImpIAdviseSink
 *
 * Parameters (Constructor):
 *  pTenant        LPTENANT of the tenant we're in.
 *  punkOuter      LPUNKNOWN to which we delegate.
 */

CImpIAdviseSink::CImpIAdviseSink(LPTENANT pTenant, LPUNKNOWN punkOuter)
   {
   m_cRef=0;
   m_pTen=pTenant;
   m_punkOuter=punkOuter;
   return;
   }

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


/*
 * CImpIAdviseSink::QueryInterface
 * CImpIAdviseSink::AddRef
 * CImpIAdviseSink::Release
 */

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

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

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


/*
 * IAdviseSink::OnDataChange
 *
 * Unused since we don't IDataObject::Advise.
```

```
 */

STDMETHODIMP_(void) CImpIAdviseSink::OnDataChange(LPFORMATETC pFEIn
   , LPSTGMEDIUM pSTM)
   {
   return;
   }


/*
 * IAdviseSink::OnViewChange
 *
 * Purpose:
 *  Notifes the advise sink that presentation data changed in the data
 *  object to which we're connected providing the right time to update
 *  displays using such presentations.
 *
 * Parameters:
 *  dwAspect       DWORD indicating which aspect has changed.
 *  lindex         LONG indicating the piece that changed.
 */

STDMETHODIMP_(void) CImpIAdviseSink::OnViewChange(DWORD dwAspect, LONG lindex)
   {
   //This only requires a repaint, which our tenant makes simple for us.
   m_pTen->Repaint();
   m_pTen->m_pPG->m_fDirty=TRUE;
   return;
   }


STDMETHODIMP_(void) CImpIAdviseSink::OnRename(LPMONIKER pmk)
   {
   return;
   }

STDMETHODIMP_(void) CImpIAdviseSink::OnSave(void)
   {
   return;
   }

STDMETHODIMP_(void) CImpIAdviseSink::OnClose(void)
   {
   return;
   }
```

Listing 9-2: Patron's implementation of IAdviseSink on container sites.

Your implementation of OnViewChange only needs to repaint your site and to set your document's dirty flag, however you do that. This function will be called whenever something changes in a *running* object, that is, whenever the end user generally modifies the object in some way, by adding more graphics or typing more text. If the object is being serviced by an InProcServer then you'll call IViewObject::Draw when repainting which will cause the server to redraw the object's presentation in your window. If the object is being serviced by a LocalServer and an InProcHandler, and the handler does not implement IViewObject::Draw specifically for this object, then the handler will ask the LocalServer to render a metafile (alternately a bitmap) through the server's IDataObject::GetData, and use that to repaint the object in the container site. Through this mechanism any change made in the server's editing window will be reflected in the container's window as well, even for OLE 1.0 servers although they don't necessarily send a change notification on every modification.

You may wonder why we don't use OnDataChange to do this instead of having this extra OnViewChange. Since containers are interested in displaying the object's presentation, it really wants to know when that presentation changes. It may be true that a data change in the object does not

necessarily precipitate a view change. The converse is true also, where a view change may not necessarily mean a data change. OnDataChange is only useful when the container knows a little more about the object and its data formats and is stepping a little outside the basic notion of embedded objects where the container knows nothing about the object. For example, a chart server may act like a normal object server when used with any container, but when used with the same vendor's spreadsheet the end user may have more ways to link together data in the spreadsheet with the presentation in the chart. In that sense the two modules are talking on a higher level than specified in compound documents alone.

All of the other member functions, OnClose, OnRename, and OnSave, are important for the default handler and its management of linked objects. Remember that the handler always sits between the container and a LocalServer and so all notifications from that server pass through the handler on their way to the container, so the handler may itself take action on such notifications. Note that linked objects cannot be serviced by an InProcServer, so there is always a handler in the linked case. Again, that's a topic for a later chapter.

Implement IOleClientSite

IOleClientSite is an interface implemented in the container through which an object informs the container of specific events or asks the container to perform specific operations. Its member functions are shown in Table 9-1.

Table 9-1: The IOleClientSite Interface

IOleClientSite Member	Description
SaveObject	Request that the container calls OleSave to insure that the object's data is saved to its IStorage. This is commonly called just before a server shuts down.
GetMoniker	Requests a container-defined moniker that references the object in this site.
GetContainer	Requests an IOleContainer interface pointer through which the object can see what else is in the container's document. This interface is implemented as part of a document, not as part of a site.
ShowObject	Asks the container to insure that the object in this site is visible within the container's window.
OnShowWindow	Informs the container that the object in the server is either becoming visible such that the user can edit it or is being hidden. On this function the container either draws or removes hatching from the site.
RequestNewObjectLayout	Asks the container to make more space for the object in the container's document.

Of these six member functions (besides IUnknown, of course) you will only need to implement three to support embeddings: SaveObject, ShowObject, and OnShowWindow. In fact, even SaveObject is the only functionally essential member as the other two are used strictly to implement user interface and do not interfere with the embedding aspects. As for the other three, we'll need to implement GetMoniker to support linking in Chapter 12, we'll need to implement GetContainer in Chapter 14 to

support some advanced yet optional container capabilities, and we'll never have to implement RequestNewObjectLayout until OLE itself is updated because OLE 2.0 does not support any use of this function. With that, let's looks at the specific requirements of each of the three necessary functions in Patron's implementation as shown in Listing 9-3. A template implementation of IOleClientSite can be found in INTERFAC\ ICLISITE.CPP for your use.

ICLISITE.CPP

```
/*
 * ICLISITE.CPP
 * Implementation of the IOleClientSite interface for Patron's tenants.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "patron.h"

/*
 * CImpIOleClientSite::CImpIOleClientSite
 * CImpIOleClientSite::~CImpIOleClientSite
 *
 * Parameters (Constructor):
 *  pTenant        LPTENANT of the tenant we're in.
 *  punkOuter      LPUNKNOWN to which we delegate.
 */

CImpIOleClientSite::CImpIOleClientSite(LPTENANT pTenant, LPUNKNOWN punkOuter)
   {
   m_cRef=0;
   m_pTen=pTenant;
   m_punkOuter=punkOuter;
   return;
   }

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


/*
 * CImpIOleClientSite::QueryInterface
 * CImpIOleClientSite::AddRef
 * CImpIOleClientSite::Release
 */

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

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

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


/*
 * CImpIOleClientSite::SaveObject
```

```
 *
 * Purpose:
 *  Requests that the container call OleSave for the object that lives
 *  here.  Typically this happens on server shutdown.
 */

STDMETHODIMP CImpIOleClientSite::SaveObject(void)
   {
   //Since we're already set up with the tenant to save, this is trivial.
   m_pTen->Update();
   return NOERROR;
   }


STDMETHODIMP CImpIOleClientSite::GetMoniker(DWORD dwAssign, DWORD dwWhich
   , LPMONIKER FAR *ppmk)
   {
   *ppmk=NULL;
   return ResultFromScode(E_NOTIMPL);
   }

STDMETHODIMP CImpIOleClientSite::GetContainer(LPOLECONTAINER FAR* ppContainer)
   {
   *ppContainer=NULL;
   return ResultFromScode(E_NOTIMPL);
   }


/*
 * CImpIOleClientSite::ShowObject
 *
 * Purpose:
 *  Tells the container to bring the object fully into view as much
 *  as possible, that is, scroll the document.
 */

STDMETHODIMP CImpIOleClientSite::ShowObject(void)
   {
   m_pTen->ShowYourself();
   return NOERROR;
   }


/*
 * CImpIOleClientSite::OnShowWindow
 *
 * Purpose:
 *  Informs the container if the object is showing itself or
 *  hiding itself.  This is done only in the opening mode and allows
 *  the container to know when to shade or unshade the object.
 *
 * Parameters:
 *  fShow         BOOL indiciating that the object is being shown
 *                (TRUE) or hidden (FALSE).
 */

STDMETHODIMP CImpIOleClientSite::OnShowWindow(BOOL fShow)
   {
   //All we have to do is tell the tenant of the open state change.
   m_pTen->ShowAsOpen(fShow);
   return NOERROR;
   }


STDMETHODIMP CImpIOleClientSite::RequestNewObjectLayout(void)
   {
   return ResultFromScode(E_NOTIMPL);
   }
```

Listing 9-3: Patron's implementation of IOleClientSite

As you can see from the listing, this implementation of IOleClientSite delegates just about everything to some of the functions added in CTenant. We'll look at OnShowWindow in the next section but can see what has to happen in SaveObject and ShowObject.

SaveObject simply tells the container to save the object in this site. In short, that means the following steps:

1. QueryInterface the object for IPersistStorage.
2. Call OleSave passing the IPersistStorage and the IStorage in which to save the object.
3. Call IPersistStorage::SaveCompleted.
4. Call IPersistStorage::Release

These are exactly the four steps taken in Patron's function CTenant::Update that we've been using since Chapter 7 and which requires no modification here:

```
BOOL CTenant::Update(void)
   {
   LPPERSISTSTORAGE    pIPS;

   if (NULL!=m_pIStorage)
      {
      m_pObj->QueryInterface(IID_IPersistStorage, (LPVOID FAR *)&pIPS);
      OleSave(pIPS, m_pIStorage, TRUE);
      pIPS->SaveCompleted(NULL);
      pIPS->Release();

      m_pIStorage->Commit(STGC_ONLYIFCURRENT);
      }

   return FALSE;
   }
```

ShowObject tells the container to bring the object, that is the site, into view if at all possible. An object calls this function when it's opened for editing such that the object in the container's site is visible alongside the object in the server's editing window. This is very nice for end-users to see both their editing session and the context around that object in the container.[1] Of course, since it only has to do with user interface, it's not necessary for the actual function of embedding, so if it's too much trouble, feel free to ignore it altogether.

Implementing ShowObject means one thing: make the object visible. How you actually do that will be specific to the nature of your application, document, page, and site, of course. In Patron, ShowObject calls CTenant::ShowYourself:

```
void CTenant::ShowYourself(void)
   {
   RECTL      rcl;
   RECT       rc;
   POINT      pt1, pt2;

   //Scrolling deals in device units, so get our rectangle in those.
   RectGet(&rcl, TRUE);

   //Get the window rectangle offset for the current scroll position.
   GetClientRect(m_hWnd, &rc);
   OffsetRect(&rc, m_pPG->m_xPos, m_pPG->m_yPos);

   //Check if the object is already visible. (macro in bookguid.h)
   SETPOINT(pt1, (int)rcl.left,  (int)rcl.top);
   SETPOINT(pt2, (int)rcl.right, (int)rcl.bottom);

   if (PtInRect(&rc, pt1) && PtInRect(&rc, pt2))
      return;
```

1Of course, full in-place activation is a much stronger way to express this same feature, but that takes a lot more work. This is a small courtesy in comparion, and one that is much easier to provide.

```
//Check if the upper left is within the upper left quadrant
if (((int)rcl.left > rc.left && (int)rcl.left < ((rc.right+rc.left)/2))
  && ((int)rcl.top > rc.top && (int)rcl.top < ((rc.bottom+rc.top)/2)))
   return;

//These are macros in INC\WIN1632.H
SendScrollPosition(m_hWnd, WM_HSCROLL, rcl.left-8);
SendScrollPosition(m_hWnd, WM_VSCROLL, rcl.top-8);
return;
}
```

A good rule of thumb here is to avoid scrolling if at all possible, so ShowYourself first checks if the site's rectangle (which is the same as the object's in the container) is already visible, that is, if both upper-left and lower-right corners are already visible in the page window. If so, then nothing needs to happen and we can return. If this first check fails, then either the site is not visible at all, or the site is too big to be entirely shown in the window. So if the upper-left corner of the site is in the upper-left quadrant of the window, it must be true that the site is not completely visible, but visible enough that we would still want to avoid scrolling. If this second check fails, then ShowYourself capitulates and scrolls the window such that the upper-left corner of the site is visible just below the upper-left corner of the window. The SendScrollPosition macros (defined in INC\WIN1632.H) send WM_*SCROLL messages with SB_THUMBPOSITION, messages which are processed in PagesWndProc of PAGEWIN.CPP.

After coding both IAdviseSink and IOleClientSite interfaces and the rest of your site objects, you can compile and insure that the interfaces are created and destroyed as necessary. Since we can't yet create an object to call these interface functions, you can write some test code that would call each interface function so you can test that IAdviseSink::OnViewChange repaints and sets a dirty flag, that IOleClientSite::SaveObject calls OleSave, etc., and that IOleClientSite::ShowObject actually scrolls the site into view. After one more step to implement IOleClientSite::OnShowWindow, we're ready to put an object in this site and have these functions called for real.

Implement Site Shading

The user interface specifications for OLE 2.0 application ask that when an object is "open," generally in its running state and visible in a server window, that the container site is shaded with a hatch pattern as shown in Figure 9-13. Conveniently we have IOleClientSite::OnShowWindow to tell us exactly when to shade and unshade the site: the functions only parameter, a BOOL *fShow*, says the object is now open (TRUE) or not open (FALSE).

§

Figure 9-13: A typical appearance of an site with a loaded object and shaded with an open object.

Patron's implementation of OnShowWindow calls CTenant::ShowAsOpen passing *fShow* as the only parameter:

```
void CTenant::ShowAsOpen(BOOL fOpen)
  {
  BOOL      fWasOpen;
  DWORD     dwState;
  RECT      rc;
  HDC       hDC;

  fWasOpen=(BOOL)(TENANTSTATE_OPEN & m_dwState);

  dwState=m_dwState & ~TENANTSTATE_OPEN;
  m_dwState=dwState | ((fOpen) ? TENANTSTATE_OPEN : 0);

  //If this was not open, then just hatch, otherwise repaint.
  if (!fWasOpen && fOpen)
```

```
    {
    RECTFROMRECTL(rc, m_rcl);
    RectConvertMappings(&rc, NULL, TRUE);
    OffsetRect(&rc, -(int)m_pPG->m_xPos, -(int)m_pPG->m_yPos);

    hDC=GetDC(m_hWnd);
    OleUIDrawShading(&rc, hDC, OLEUI_SHADE_FULLRECT, 0);
    ReleaseDC(m_hWnd, hDC);
    }

  if (fWasOpen && !fOpen)
    Repaint();

  return;
  }
```

This function first checks if any state change has actually occurred, that is, if the object was already open and we're being asked to show as open again, then there's nothing to do, just as if we are not open and told the same fact again. If the state differ, then there are two possibilities.

First, if the change is from loaded to open, then we need to draw the hatch pattern across the site rectangle. The OLE2UI library provides a convenient function for this called OleUIDrawShading which takes the rectangle to shade, the hDC on which to draw, some flags from OLEUI_SHADE_* values, and a width (the zero in the code above). There are three possible (and exclusive) flags you can pass that affect how the function shades in or around the rectangle:

- OLEUI_SHADE_FULLRECT: Shades the entire rectangle. The width parameter is ignored.
- OLEUI_SHADE_BORDERIN: Shades the width inside the border of the rectangle.
- OLEUI_SHADE_BORDEROUT: Shades the width outside the border of the rectangle.

The OLEUI_SHADE_BORDER* flags are used for in-place activation and are not necessary for use here. Also note that OLE2UI.H defined another called OLEUI_SHADE_USEINVERSE which is a no-op: OleUIDrawShading doesn't do anything with it. If it did, it might draw a hatch pattern using an XOR ROP code. Instead, OleUIDrawShading always draws the hatch pattern in black using PatBlt with a hatch pattern brush and the ROP 0x00A000C9 which performs the logical AND between the black pattern and whatever is on hDC, that is, draws a black hatch regardless of what's underneath.

Since this hatching is destructive, the other case we need to handle, going from an open state into a loaded state, requires that we repaint the entire object to remove the hatching. Fortunately for Patron, CTenant::Repaint does just that. You will generally need to do the same, or implement your own version of OleUIShadeBorder that uses the inverse (remember that you have the code for this function in the OLE 2.0 SDK).

Invoke the Insert Object Dialog

We've finally arrived at the point where we can create a new object to occupy a site, as all the necessary container pieces are in place. In other words, we have some sort of object we can call a site with IOleClientSite and IAdviseSink interfaces and we have a storage object we can give to the object in this site. What's left is to somehow obtain the CLSID of an embeddable object and call the OleCreate function to obtain our first pointer to that new object. However you want to determine the CLSID is up to you depending on your application and its use. For most cases, however, the standard Microsoft-recommended way is to invoke the Insert Object dialog. This dialog allows the use to create a new

embedded object based on a CLSID when the user has selected "Create New" as shown in Figure 9-14, or an embedded object that contains a copy of a file[1] when using has selected "Create From File" as shown in Figure 9-15. In Chapter 12 we'll also be able to create a link to a file using this same dialog. The OLE2UI library provides an implementation of this dialog through the OleUIInsertObject function that we'll make use of here.

§

Figure 9-14: The Insert Object dialog with Create New selected.

§

Figure 9-15: The Insert Object dialog with Create From File selected.

The list of names in the listbox shown in Figure 9-14 are enumerated from the registration database for any entry that appears as below:

```
\
    <ProgID>= <User-readable name of this type of object>
        CLSID = {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}
        Insertable
```

Any key that has the "Insertable" sub-key gets its user-readable name in the listbox. When the user selects that name and presses OK, the container application receives back the CLSID for this entry in the registration database. The container may instruct the dialog to verify that an InProcServer or LocalServer exists for that CLSID which takes considerably more time to invoke the dialog.

Dealing with the Insert Object dialog will require small modifications at just about every level of your container application:

1. Add a menu item in the Edit menu with the string "Insert Object..." or, if you have an Insert menu already, add an item with just the string "Object..." This item should be enabled whenever you have an open document.

2. When the user selects the Insert Object command, invoke the dialog to retrieve the CLSID (for Create New) or the filename (for Create From File).

3. Call OleCreate (for CreateNew) or OleCreateFromFile (for the other, of course), and store the returned pointer with your site object.

4. After creating the object, initialize the object by calling a number of miscellaneous functions as well as interface members on the object.

5. If you do not support iconic aspects (see the last section in this chapter) then you must provide a modified template for this dialog such that the "Display As Icon" checkbox is out of view, or you must hook it to insure that the checkbox is always disabled.

Adding the menu item should not be much trouble, basically add a #define and a MENUITEM in your .RC file (remember the ellipses on "Insert Object..."):

```
//In Patron's RESOURCE.H
#define IDM_EDITINSERTOBJECT      (IDM_CUSTOMEDITMIN+2)

//In PATRON.RC
MENUITEM "&Insert Object...",      IDM_EDITINSERTOBJECT
```

That was the simple step. The others are more involved.

1In OLE 1.0 this capability was provided by the Packager server. Packager's functions are now incorporated into OLE 2.0 itself.

Call OleUIInsertObject

When Patron's frame window procedure detects a WM_COMMAND message with IDM_EDITINSERTOBJECT is calls the active document's FInsertObject function which I added new to this chapter's version of Patron:

```
BOOL CPatronDoc::FInsertObject(HWND hWndFrame)
  {
  OLEUIINSERTOBJECT   io;
  DWORD             dwData=0;
  char            szFile[CCHPATHMAX];
  UINT              uTemp;
  BOOL              fRet=FALSE;

  if (NULL==m_pPG)
    return FALSE;

  _fmemset(&io, 0, sizeof(io));

  io.cbStruct=sizeof(io);
  io.hWndOwner=hWndFrame;

  szFile[0]=0;
  io.lpszFile=szFile;
  io.cchFile=sizeof(szFile);

  io.dwFlags=IOF_SELECTCREATENEW | IOF_DISABLELINK;

  uTemp=OleUIInsertObject(&io);

  if (OLEUI_OK==uTemp)
    {
    TENANTTYPE     tType;
    LPVOID         pv;
    FORMATETC      fe;

    SETDefFormatEtc(fe, 0, TYMED_NULL);

    if (io.dwFlags & IOF_SELECTCREATENEW)
      {
      tType=TENANTTYPE_EMBEDDEDOBJECT;
      pv=(LPVOID)&io.clsid;
      }
    else
      {
      tType=TENANTTYPE_EMBEDDEDFILE;
      pv=(LPVOID)szFile;
      }

    if ((io.dwFlags & IOF_CHECKDISPLAYASICON) && NULL!=io.hMetaPict)
      {
      fe.dwAspect=DVASPECT_ICON;
      dwData=(DWORD)(UINT)io.hMetaPict;
      }

    fRet=m_pPG->TenantCreate(tType, pv, &fe, NULL, dwData);

    //Free this regardless of what we do with it.
    if (NULL!=io.hMetaPict)
      OleUIMetafilePictIconFree(io.hMetaPict);

    if (fRet)
      {
      //Disable Printer Setup once we've created a tenant.
      m_fPrintSetup=FALSE;
      FDirtySet(TRUE);
      }
    }

  return fRet;
  }
```

To call OleUIInsertObject we must first initialize an OLEUIINSERTOBJECT structure. This structure contains the usual fields for the size of the structure (*cbStruct*) and the parent window (*hWndOwner*) as well as others for hooking and template customization that are common to all the dialogs in the library. For our purposes, we need to fill in a few other Insert-Object-specific fields: *lpszFile* points to a buffer to receive a filename in the Create From File case, *cchFile* is the length of *lpszFile*, and *dwFlags* describes various options for this dialog. In our case we use the IOF_CREATENEW flag to initially select the "Create New" radiobutton (the standard) and IOF_DISABLELINK which removes a Link checkbox that would otherwise appear when the "Create From File" radiobutton is selected (we'll remove this flag later, of course). We can then call OleUIInsertObject and let the dialog do what dialogs do.

The return value from OleUIInsertObject indicates if the user pressed OK or Cancel. The Cancel case is uninteresting so let's look at the return of OLEUI_OK. When this happens *dwFlags* in the OLEUIINSERTOBJECT structure will first of all contain which radiobutton was selected on closing the dialog: IOF_CREATENEW or IOF_CREATEFROMFILE. In Patron we use this to determine the type of object to create and what data is necessary for a tenant to create the object. For Create New, the type is a straight embedded object and we want to pass a pointer to the selected CLSID which is in the *clsid* field in the structure; for Create From File, the type is an embedded object from a file and we want to pass a pointer to the filename from the dialog (which was written to *lpszFile*). The code that looks at IOF_DISPLAYASICON has to do with iconic aspects of the object which we'll discuss at the end of this chapter, so ignore it for now.

Once Patron determines what type of object to create, it gives the order to create the object by calling CPages::TenantCreate which passes through to CPage::TenantCreate, which creates a new tenant and tells the tenant to create the object as we'll see in a moment. I wanted to first point out that there may be a memory handle in the *hMetaPict* field of the structure on return from OleUIInsertObject, and it is your responsibility to clean it up by calling OleUIMetafilePictIconFree as shown above. Be sure you do this to avoid memory leaks, regardless of what else happens.

Call OleCreate or OleCreateFromFile

Patron's object creation involved both creating a tenant in CPage::TenantCreate and creating the object in CTenant::UCreate. The CPage code is responsible for placing the tenant on the page and making sure it's repainted and selected properly. This code has not changed appreciably from what we first wrote in Chapter 7. Our main focus is on the changes to CTenant::UCreate:

```
UINT CTenant::UCreate(TENANTTYPE tType, LPVOID pvType, LPFORMATETC pFE
  , LPPOINTL pptl, LPSIZEL pszl, LPSTORAGE pIStorage
  , LPPATRONOBJECT ppo, DWORD dwData)
  {
  HRESULT         hr;
  LPUNKNOWN       pObj;
  UINT            uRet=UCREATE_GRAPHICONLY;

  [Unmodified code to validate params and obtain placement data from ppo]

  //Now create an object based specifically for the type.
  switch (tType)
    {
    case TENANTTYPE_NULL:
      break;

    case TENANTTYPE_STATIC:
      hr=CreateStatic((LPDATAOBJECT)pvType, pFE, &pObj);
      break;

    case TENANTTYPE_EMBEDDEDOBJECT:
      hr=OleCreate(*((LPCLSID)pvType), IID_IUnknown, OLERENDER_DRAW
```

```
        , NULL, NULL, m_pIStorage, (LPLPVOID)&pObj);
      break;

   case TENANTTYPE_EMBEDDEDFILE:
      hr=OleCreateFromFile(CLSID_NULL, (LPSTR)pvType, IID_IUnknown
        , OLERENDER_DRAW, NULL, NULL, m_pIStorage
        , (LPLPVOID)&pObj);
      break;

   case TENANTTYPE_EMBEDDEDOBJECTFROMDATA:
      hr=OleCreateFromData((LPDATAOBJECT)pvType, IID_IUnknown
        , OLERENDER_DRAW, NULL, NULL, m_pIStorage
        , (LPLPVOID)&pObj);
      break;

   default:
      break;
   }

//If creation didn't work, get rid for the element FOpen created.
if (FAILED(hr))
   {
   Destroy(pIStorage);
   return UCREATE_FAILED;
   }

FObjectInitialize(pObj, pFE, dwData);

//We depend here on m_pIOleObject having been initialized.
if ((0==pszl->cx && 0==pszl->cy))
   {
   SIZEL   szl;

   //Try to get the real size of the object, default to 2"*2"
   SETSIZEL((*pszl), 2*LOMETRIC_PER_INCH, 2*LOMETRIC_PER_INCH);

   if (SUCCEEDED(m_pIOleObject->GetExtent(pFE->dwAspect, &szl)))
      {
      //Convert HIMETRIC to our LOMETRIC mapping, if meaningful
      if (0!=szl.cx && 0!=szl.cy)
         SETSIZEL((*pszl), szl.cx/10, szl.cy/10);
      }
   }

return uRet;
}
```

There are three new ways a tenant can now create an object: OleCreate, OleCreateFromFile, and OleCreateFromData, the latter of which is treated in a later section. The first, OleCreate, creates a new embedded object from a CLSID. We pass it the CLSID from the Insert Object dialog, the interface we want (IUnknown), a render option (OLERENDER_DRAW), a pointer to a FORMATETC (NULL in this case because we're using OLERENDER_DRAW), a pointer to an IOleClientSite interface (NULL because we'll give it to the object later), the storage for this object (m_pIStorage), and the address in which to store the interface pointer we requested. OleCreateFromFile creates a new embedded "packager" object from the contents of a file and takes all the same parameters except the CLSID is CLSID_NULL and we have to also pass the filename to use. In either case, what we get back is the first interface pointer to this new object to whatever interface we asked for, in this case IUnknown.

If OleCreate* worked, then Patron continues by initializing the object and obtaining the initial size. Since the initialization sequence involves a number of functions, and since its the same as when we load an object from a file later on, I've centralized the code in CTenant::FObjectInitialize. I strongly recommend you do the same.

Initialize the Object

After creating (or loading) an object, perform the following initialization steps:

1. Save the type of the object: static or embedded. We will use this later to determine if we can activate the object at all.

2. Call IViewObject::SetAdvise to establish a connection between the object and our IAdviseSink::OnViewChange for the aspect of the object we have. Most often the aspect is DVASPECT_CONTENT, but later in this chapter we'll enable our container to deal with DVASPECT_ICON as well.

3. Pass out IOleClientSite pointer to the object via IOleObject::SetClientSite. This is the only way in which the object knows its site. We could also pass the IOleClientSite pointer to an OleCreate* function, but we also need to do this when loading an object with OleLoad, and OleLoad does not take such a parameter. So to keep it all central (as well as explicit) we do it here.

4. Call IOleObject::Advise with a pointer to our IAdviseSink so that the object can inform the handler of events such as OnClose and OnSave. This is necessary for proper operation of the handler but really doesn't do much for our application.

5. Call the weirdo function OleSetContainedObject in OLE2.DLL that basically makes OLE 2.0 work correctly with your container. Failure to call this will generally leave a few extra reference counts on your site object through the IOleClientSite interface. Don't worry. Be happy. Just do it.

6. Provide the object with strings for its user interface requirements by sending your application and document names to IOleObject::SetHostNames. Patron does this through CTenant::NotifyOfRename which passes the "Patron 2.0" as the application name and the short 8.3 filename of the document (or "Untitled"). The object will generally use the document name in window titles and on menus when it opens editing windows as we'll do in Chapter 10.

You can see these in Patron's CTenant::FObjectInitialize:

```
BOOL CTenant::FObjectInitialize(LPUNKNOWN pObj, LPFORMATETC pFE, DWORD dwData)
   {
   HRESULT       hr;
   LPPERSIST     pIPersist=NULL;
   DWORD         dw;
   LPCDocument   pDoc;
   char          szFile[CCHPATHMAX];

   if (NULL==pObj || NULL==pFE)
      return FALSE;

   m_pObj=pObj;
   m_fe=*pFE;
   m_dwState=TENANTSTATE_DEFAULT;

   /*
    * Determine the type:  Static or Embedded
    * If Static, this will have CLSID_FreeMetafile or CLSID_FreeDib.
    * Otherwise it's embedded.  Later we'll add a case for links.
    */
   m_tType=TENANTTYPE_EMBEDDEDOBJECT;

   if (SUCCEEDED(pObj->QueryInterface(IID_IPersist,(LPLPVOID)&pIPersist)))
      {
      CLSID   clsid;

      pIPersist->GetClassID(&clsid);

      if (IsEqualCLSID(clsid, CLSID_FreeMetafile)
         || IsEqualCLSID(clsid, CLSID_FreeDib))
```

```
      m_tType=TENANTTYPE_STATIC;

   pIPersist->Release();
   }


m_pIViewObject=NULL;
hr=pObj->QueryInterface(IID_IViewObject, (LPLPVOID)&m_pIViewObject);

if (FAILED(hr))
   return FALSE;

m_pIViewObject->SetAdvise(pFE->dwAspect, 0, m_pIAdviseSink);

//We need an IOleObject most of the time, so get one here.
m_pIOleObject=NULL;
pObj->QueryInterface(IID_IOleObject, (LPLPVOID)&m_pIOleObject);

//Follow up object creation with advises and so forth.
if (FAILED(hr))
   return FALSE;

/*
 * We could pass m_pIOleClientSite in an OleCreate* call, but
 * since FPostCreate could be called after OleLoad, we still
 * need to do this here, so it's always done here...
 */
m_pIOleObject->SetClientSite(m_pIOleClientSite);
m_pIOleObject->Advise(m_pIAdviseSink, &dw);

OleSetContainedObject((LPUNKNOWN)m_pIOleObject, TRUE);

/*
 * For IOleObject::SetHostNames we need the application name
 * and the document name (which is passed in the object parameter).
 * The design of Patron doesn't give us nice structured access to
 * the name of the document we're in, so I grab the parent of
 * the Pages window (the document) and send it DOCM_PDOCUMENT
 * which returns us the pointer.  Roundabout, but it works.
 */

pDoc=(LPCDocument)SendMessage(GetParent(m_hWnd), DOCM_PDOCUMENT, 0
   , 0L);

if (NULL!=pDoc)
   pDoc->FilenameGet(szFile, CCHPATHMAX);
else
   szFile[0]=0;

NotifyOfRename(szFile);

if ((DVASPECT_ICON & pFE->dwAspect) && NULL!=dwData)
   {
   DWORD      dw=DVASPECT_CONTENT;
   BOOL       fUpdate;

   OleStdSwitchDisplayAspect(m_pIOleObject, &dw, DVASPECT_ICON
      , (HGLOBAL)(UINT)dwData, TRUE, FALSE, NULL, &fUpdate);
   }

return TRUE;
}
```

Some of this is included in an OLE2UI function called OleStdSetupAdvises, which may or may not be useful to your application. As for Patron's code above, the stuff at the end that deals with DVASPECT_ICON is again, a subject for the end of this chapter. In addition to the initialization steps above, this function also initializes CTenant's m_pIOleObject and m_pIViewObject fields using QueryInterface. I hold on to these interface pointers for the lifetime of the object simply because we

need them in a variety of places and we can avoid excess QueryInterface calls. IPersistStorage is not included here because we only use it once more in CTenant::Update.

Finally, when you create a new embedded object there are two more steps to follow that I did not include in the list above because you only execute them after using the Insert Object dialog and not after loading an object or creating one with data from the clipboard (after which Patron calls CTenant::FObjectInitialize):

1. Invoke IOleObject::DoVerb with OLEIVERB_SHOW (see below under "Activate Objects").

2. Save the object with OleSave and IPersistStorage::SaveCompleted.

Both of these are called from CPage::TenantCreate for a new object from OleCreate or OleCreateFromFile. Calling IOleObject::DoVerb (which happens in CTenant::Activate) with OLEIVERB_SHOW instructs the object to show itself in a window ready for editing. Saving the object immediately afterwards saves the initial editing state of the object once it's open. This is most important for working well with OLE 1.0 servers that may not update themselves before closing.

Here I would suggest you compile your code and work out those errors and verify that objects are, in fact created and that all the initialization flows smoothly. However, things won't look like much until we can display the object.

Creating an Object Based on a Selection: IOleObject::InitFromData

> For certain compound document objects (particularly ones you wrote yourself or those written to work with your container) you may elect, after calling OleCreate, to initialize the object based on a selection of other data in your container. For example, a spreadsheet application as a container may have a range of cells selected and the user would like to create a new chart object. The spreadsheet can package the current selection up in an IDataObject and pass that to the newly created object's IOleObject::InitFromData, along with an *fCreation* flag set to TRUE.
>
> I strongly encourage containers such as word-processors and spreadsheets, that have their own data, to use this function and to document the formats you send in order to enable third-party add-on packages that are designed specificially to value-add your application.

Draw and Print Objects

Showing some sort of presentation for the object is, of course, one of the most important aspects about a container's user interface. This really boils down to calling OleDraw, or IViewObject::Draw, whenever you repaint the site exactly as Patron has been doing in CTenant::Draw. One thing to remember is that if the object is still open when you repaint you need to make sure you draw the shading as described above. This is why Patron's tenants have a state flag to mark them as open so repaints will produce the proper results.

Printing is no different although you generally want to pass a non-NULL target device to the drawing functions to give the objects a chance to render themselves specifically for that device. Most custom handlers, for example, exist only to optimize an object's output for specific printers. It's important that you let them do their job.

It's a good idea here to quickly test that objects in your sites repaint appropriately when you receive notifications like IAdviseSink::OnViewChange. When they do, then you can see your sites being

updated as changes are made in the object's server.

Resizing Objects, IOleObject::SetExtent, and IOleObject::GetExtent

In the course of managing objects your container may have course to resize the site in which the object lives as Patron does. In such a situation you may want to call IOleObject::SetExtent to let is know the exact size of its display. The object can then optimize its output for that size. On the other hand, the container may not want to be ultimately responsible for the object's size in which case it can call IOleObject::GetExtent to ask the object how large it would like to be. Patron calls GetExtent after creating a new object to set the initial size of the site, but thereafter will always tell the object new extents when the site is resized.

Activate Objects and Add the Object Verb Menu

Creating new objects and displaying or printing them doesn't do anything more for us than a static bitmap or metafile. One of the most important features of compound documents under OLE is that you can activate the object, that is, ask it to execute one of its verbs and thus perform some action such as playing a sound or opening its data for editing. This ability for activation is the only thing that separates an embedded or linked object from a static one. But some way or another you have to allow the end-user to select the actions available for that object which varies object to object. For this reason the OLE 2.0 user interface defines two methods to allow end-users to invoke verbs.

The first method is to execute what is known as the primary or default verb when the object is double-clicked with the left mouse button. The exact meaning of this primary verb is defined by the object, not by the container, so the container blindly tells the object to execute without any knowledge of what will happen. The second method, of which the container is equally ignorant, is to provide a menu item on the container's edit menu that lists all the available verbs on the currently selected object. When the end-user selects one of these menu items, the container again blindly tells the object to execute a verb. However, the container can ask the object to perform known actions by using pre-defined verbs, although these options are generally not shown directly to the end user.

All verbs are a simple integer index. Verbs with values less than zero are pre-defined verbs defined in the OLE 2.0 specifications:

OLEIVERB_SHOW	(-1) Instructs the object to show itself.
OLEIVERB_OPEN	(-2) Instructs the object to open itself in its own window for editing. This mostly applies to in-place activation. For a non-in-place capable object, this is identical to OLEIVERB_SHOW.
OLEIVERB_HIDE	(-3) Instructs the object to hide its editing window.
OLEIVERB_UIACTIVATE	(-4) For in-place activation
OLEIVERB_INPLACEACTIVATE	(-5) For in-place activation.
OLEIVERB_DISCARDUNDOSTATE	(-6) For in-place activation.

As mentioned in this list, a number of these deal only with in-place activation which we'll see in later chapters, so ignore them for now.

In addition to the negative values, OLE 2.0 also defines zero as the primary verb:

OLEIVERB_PRIMARY	(0) Instructs the object to execute its default action.

All other positive verb indexes are object-specific, so all verbs zero or above are complete

enigmas to the container and must be invoked only at the request of an end-user as mentioned above. Executing this verb is a matter of calling IOleObject::DoVerb with the index of the verb but also with a few other parameters. Whenever Patron needs to execute a verb on an object, it asks the tenant to do so through CTenant::Activate:

```
BOOL CTenant::Activate(DWORD iVerb)
  {
  RECT      rc, rcH;
  HCURSOR   hCur;

  //Can't activate statics.
  if (TENANTTYPE_STATIC==m_tType)
    {
    MessageBeep(0);
    return FALSE;
    }

  RECTFROMRECTL(rc, m_rcl);
  RectConvertMappings(&rc, NULL, TRUE);
  XformRectInPixelsToHimetric(NULL, &rc, &rcH);

  hCur=SetCursor(LoadCursor(NULL, MAKEINTRESOURCE(IDC_WAIT)));
  ShowCursor(TRUE);

  m_pIOleObject->DoVerb(iVerb, NULL, m_pIOleClientSite, 0
    , m_hWnd, &rcH);

  SetCursor(hCur);
  ShowCursor(FALSE);

  return FALSE;
  }
```

Since you cannot activate static objects, Patron simply beeps on any attempt at activation. Otherwise it needs to prepare itself to call IOleObject::DoVerb which takes a number of parameters:

iVerb	(LONG) Index of the verb to execute.
lpMsg	(LPMSG) Pointer to the MSG structure describing the event that invoked the verb (such as a mouse click). Inclusion of this is optional for non-in-place containers so you can generally pass NULL for now.[1]
pActiveSite	(LPOLECLIENTSITE) Pointer to the IOleClientSite that invoked this verb. In some specialized cases it may not be the one in which the object is contained.
lindex	(LONG) Index of the piece of the object on which this verb was invoked. Always 0 in OLE 2.0 meaning "the whole object."
hWndParent	(HWND) Handle of whatever window immediately contains the site.
lprcPosRect	(LPCRECT) Rectangle describing the boundaries of the object in *hWndParent*.

The *hWndParent* and *lprcPosRect* parameters specifically support multimedia-type objects that want to play in-place, but not edit. Video clips are a good example. Such objects need to have a window for which they can call GetDC and a rectangle that defines exactly where on that hDC they can draw without disturbing anything but themselves. It was somewhat unreasonable to ask in-place media players to do a full in-place activation just to play in this manner.

1The message is important for in-place objects like pushbuttons that would want to know if they should show themselves in a pressed or unpressed state. Presumably for such an object a container would call DoVerb on both button up and button down messages which would allow the object to look and act like any other button.

With that we can now look at the specific cases where Patron calls CTenant::Activate. Once you have added those cases which are important to your application, you can compile and test activation by creating new objects with the Insert Object dialog, making initial edits, closing the server window, then reactivating the object to edit the data again. Note that if a server window is already open, activating the object will generally switch the focus to that window as most servers call SetFocus on themselves inside IOleObject::DoVerb.

Mouse Double-Clicks

In Patron, a WM_LBUTTONDBLCLK on a page comes into PagesWndProc (PAGEWIN.CPP) and gets dispatched to CPage::OnLeftDoubleClick. Note that before this happens, Patron will have processed WM_LBUTTONDOWN and selected the tenant under the mouse if there was one, so that tenant is now the current. CPage::OnLeftDoubleClick (PAGEMOUS.CPP) is simple where it checks if there was a tenant under the current mouse position (determined elsewhere in CPage::OnNCHitTest), and if there is one, calls CTenant::Activate on it with OLEIVERB_PRIMARY:

```
BOOL CPage::OnLeftDoubleClick(UINT uKeys, UINT x, UINT y)
  {
  if (HTNOWHERE!=m_uHTCode)
    return m_pTenantCur->Activate(OLEIVERB_PRIMARY);

  return FALSE;
  }
```

However you want to communicate the double-click even to your site is, of course, your choice. I will point out that Patron's design here may later need to pass a MSG structure to CTenant::Activate, so you may want to incorporate that into your design right away. I have chosen not to until it becomes necessary.

Object Verb Menu

Double-clicking only lets the user execute one verb on the object even when the object has multiple verbs. For example, a sound object will generally have a primary verb that plays the sound and a secondary verb to edit the sound itself. Other objects may have even more verbs. To present these all to the end user, OLE 2.0 specifies that these verbs appear on the container's Edit menu in one of two ways:

1. If the object has only one verb, then create a menu item with the string "<*verb*> <*object name*>."
2. If the object has multiple verbs, then create a cascading menu item with the string "<*object name*>" where its popup contains a separate item for each verb.

For example, an "Equation" object may have a single verb called "Edit" and so the container's menu would appear as:

§

Something like a Sound Recorder "Sound" object has two verbs "Play" and "Edit," and so the container's menu would appear as:

§

The OLE2UI library provides a function called OleUIAddVerbMenu to create this menu item, cascaded or not, for you with little pain (and believe me, it's not fun to implement this yourself). This function takes a number of parameters:

lpObj	(LPOLEOBJECT) Points to the IOleObject interface of the currently selected object.
lpszShortType	(LPSTR) Pointer to the object's short descriptive name as defined by the

object itself. The container may pass NULL in which case this function uses a name for the object from the registration database which is generally what you want.

hMenu (HMENU) Handle of the container's Edit menu or the menu to modify. This should not be your top-level menu.

uPos (UINT) Position of the item to modify in the Edit menu. This item is generally removed altogether and recreated in this function.

idVerbMin (UINT) Starting WM_COMMAND identifier to use for the verbs.

bAddConvert (BOOL) Flag indicating whether to add a "Convert..." item to this menu. See Chapter 14.

idConvert (UINT) WM_COMMAND identifier for the Convert menu item. See Chapter 14.

lphMenu (HMENU FAR *) Pointer to an HMENU to receive the handle of the popup menu added if there are multiple verbs for this object.

The most important parameter here is *idVerbMin* because when the user selects a verb from this menu it will generate a WM_COMMAND message to your top-level menu with an identifier of *<verb index> + idVerbMin*, so by subtracting *idVerbMin* you can get to the real verb indexes of 0, 1, 2, etc. to pass to IOleObject::DoVerb. Be sure as well that you define no other menu identifiers with a value greater than this starting index.

So there's two parts to this story: creating the menu and processing the items. Patron creates the menu based on the currently selected tenant by overloading the CPatronDoc::FQueryObjectSelected function. This is called whenever the frame window detects a WM_INITPOPUPMENU message on the Edit menu. On the document level this function does nothing but pass it on down to the page level, through CPages::FQueryObjectSelected and finally to CPage::FQueryObjectSelected:

```
BOOL CPage::FQueryObjectSelected(HMENU hMenu)
  {
  HMENU     hMenuTemp;

  if (NULL!=m_pTenantCur)
    {
    m_pTenantCur->AddVerbMenu(hMenu, MENUPOS_OBJECT);
    return TRUE;
    }

  OleUIAddVerbMenu(NULL, NULL, hMenu, MENUPOS_OBJECT
    , IDM_VERBMIN, FALSE, 0, &hMenuTemp);

  return FALSE;
  }
```

If there is no currently selected tenant, then we call OleUIAddVerbMenu with NULL pointers which creates a grayed and disabled menu item with just the word "Object" in it as described by the OLE 2.0 user interface guidelines. If there is a selected tenant, then this function asks it to call OleUIAddVerbMenu itself, since the tenant knows the object pointer:

```
void CTenant::AddVerbMenu(HMENU hMenu, UINT iPos)
  {
  HMENU     hMenuTemp;
  LPOLEOBJECT pObj=m_pIOleObject;

  //If we're static, say we have no object.
  if (TENANTTYPE_STATIC==m_tType)
    pObj=NULL;

  OleUIAddVerbMenu(pObj, NULL, hMenu, iPos, IDM_VERBMIN
    , FALSE, 0, &hMenuTemp);
```

```
    return;
    }
```

Here we again call OleUIAddVerMenu with NULL pointers for a static object since those can have no verbs. IDM_VERBMIN is the starting verb index which has no other defined menu identifiers above it.

We had to go through a number of layers (document, pages, page, tenant) to get create this menu, and it's no different for processing the WM_COMMAND messages it generates. This processing in Patron starts up in CPatronFrame::OnCommand which is called on WM_COMMAND messages:

```
LRESULT CPatronFrame::OnCommand(HWND hWnd, WPARAM wParam, LPARAM lParam)
  {
  [Other code omitted]

  pDoc=(LPCPatronDoc)m_pCL->ActiveDocument();

  if (NULL!=pDoc && (IDM_VERBMIN <= wID))
    {
    pDoc->ActivateObject(wID-IDM_VERBMIN);
    return 0L;
    }

  [Other command handling here.]
  }
```

The call to CPatronDoc::ActivateObject and the verb index passes unmolested to CPages::ActivateObject, CPage::ActivateObject, and finally CTenant::Activate. I didn't say Patron had the world's most elegant design!

One other side note: applications that have status lines may want to display status information for these verbs as the user selects them from the menu. Patron does this by watching for WM_MENUSELECT messages in CPatronFrame::FMessageHook with the appropriate identifiers. If it detects a selection of the popup verb menu itself (as opposed to an item with a verb) it displays no message (as described by OLE 2.0 UI). If it sees the selection of a verb item, it displays the string "Commands to manipulate the selected object." If you want, you can display a message more like "<*Verb*> the <*object name*> object" or something like it which describes the result more precisely.

The Right-Mouse Button Popup Menu

The OLE 2.0 User Interface also recommends that you display a small menu over an object in a container when the end-user singly clicks the right mouse button. This menu generally appears with various commands from your Edit menu, like Cut, Copy, and an object's verbs, along with optional commands like Delete and Convert. None of the strings on this menu should have keyboard mnemonics since it is a mouse-only feature anyway.

Patron creates a menu with Cut, Copy, Delete Object, and verb command on it when it detects a WM_RBUTTONDOWN in PagesWndProc (PAGEWIN.CPP) by calling CPage::OnRightDown:

```
BOOL CPage::OnRightDown(UINT uKeys, UINT x, UINT y)
  {
  HMENU      hMenu;
  HMENU      hMenuRes;
  HINSTANCE  hInst;
  HWND       hWndFrame, hWndT;
  POINT      pt;
  UINT       i, cItems;

  //Select the tenant under the mouse, if there is one.
  if (!FSelectTenantAtPoint(x, y))
    return FALSE;

  /*
   * Get the top-level window to which menu command will go.  This will
   * be whatever parent doesn't have a parent itself...
```

```
 */
hWndT=GetParent(m_hWnd);

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

/*
 * Build a popup menu for this object with Cut, Copy, Delete, and
 * object verbs.
 */
#ifdef WIN32
 hInst=(HINSTANCE)GetWindowLong(m_hWnd, GWL_HINSTANCE);
#else
 hInst=(HINSTANCE)GetWindowWord(m_hWnd, GWW_HINSTANCE);
#endif

hMenuRes=LoadMenu(hInst, MAKEINTRESOURCE(IDR_RIGHTPOPUPMENU));

if (NULL==hMenuRes)
   return FALSE;

//Resource-loaded menus don't work, so we'll copy the items.
hMenu=CreatePopupMenu();
cItems=GetMenuItemCount(hMenuRes);

for (i=0; i < cItems; i++)
   {
   char       szTemp[80];
   int        id, uFlags;

   GetMenuString(hMenuRes, i, szTemp, sizeof(szTemp), MF_BYPOSITION);
   id=GetMenuItemID(hMenuRes, i);

   uFlags=(0==id) ? MF_SEPARATOR : MF_STRING | MF_ENABLED;
   AppendMenu(hMenu, uFlags, id, szTemp);
   }

DestroyMenu(hMenuRes);

//Munge the Object menu item
m_pTenantCur->AddVerbMenu(hMenu, MENUPOS_OBJECTONPOPUP);

SETPOINT(pt, x, y);
ClientToScreen(m_hWnd, &pt);

TrackPopupMenu(hMenu, TPM_LEFTALIGN | TPM_RIGHTBUTTON
   , pt.x, pt.y, 0, hWndFrame, NULL);

DestroyMenu(hMenu);
return FALSE;
}
```

This function essentially sees if there is an object below the mouse, and if so, creates a menu using items from a resource-template menu (which as a top-level menu from LoadMenu cannot be used with TrackPopupMenu). It then adds the object's verbs using CTenant::AddVerbMenu and displays it with TrackPopupMenu with the result shown below.All command identifiers on this menu are identical to those on the top-level menu, so no other modifications are necessary to support this feature.

§

Create Objects from Clipboard and Drag-Drop Transfers

Creating a new object using the Insert Object dialog is only one way you might get an object into a container. It is also possible to paste an object using data from the clipboard or to accept an embedded object dropped on your container. Again, since both clipboard and drag-drop operations operate from a

single IDataObject, the discussion here applies identically to both.

First of all, there two new clipboard formats that a container must register using RegisterClipboardFormat:

CF_EMBEDDEDOBJECT	An IStorage containing the object's native data.
CF_OBJECTDESCRIPTOR	An OBJECTDESCRIPTOR structure in global memory that contains the object's size, aspect (iconic, content, etc.), class, and other information in which a potential consumer might be interested.[1]

The CF_* values are not integers as with Windows-defined formats. Instead they are strings defined in OLE2.H that you must register as Patron does in the CPatronDoc constructor:

```
m_cfEmbeddedObject  =RegisterClipboardFormat(CF_EMBEDDEDOBJECT);
m_cfObjectDescriptor=RegisterClipboardFormat(CF_OBJECTDESCRIPTOR);
```

Now, given any data object pointer you can determine if there's an embedded object available by calling OleQueryCreateFromData passing the IDataObject pointer. This function basically checks for the existence of CF_EMBEDDEDOBJECT or CF_EMBEDSOURCE (an identical format as CF_EMBEDDEDOBJECT but under a different name) as well as OLE 1.0 embedded object formats like "Native" and "OwnerLink." But what OleQueryCreateFromData does inside is of no concern to use; we can use the function to enable Pasting and dropping as we can see in CPatronDoc::FQueryPasteFromData:

```
BOOL CPatronDoc::FQueryPasteFromData(LPDATAOBJECT pIDataObject
  , LPFORMATETC pFE, LPTENANTTYPE ptType)
  {
  FORMATETC  fe;
  HRESULT    hr, hr2;
  [Other tests]

  hr2=OleQueryCreateFromData(pIDataObject);

  if (NOERROR==hr2)
    {
    if (NULL!=pFE)
      {
      /*
       * Default to content.  FPaste will use CF_OBJECTDESCRIPTOR
       * to figure the actual aspect.
       */
      SETDefFormatEtc(*pFE, m_cfEmbeddedObject, TYMED_ISTORAGE);
      }

    if (NULL!=(LPVOID)ptType)
      *ptType=TENANTTYPE_EMBEDDEDOBJECTFROMDATA;

    [Other tests]
    }

  ...
  }
```

If Patron sees an available embedded object, it remembers that fact by storing a FORMATETC containing the embedded object clipboard format. This information is used in pasting as well as dropping, which calls CPatronDoc::FPasteFromData. I don't want to show this lengthy function here so I will instead some of its features. First of all, this function will look for the

1CF_OBJECTDESCRIPTOR is somewhat the general format for the problem we had to solve in Chapter 8's Patron where we wanted a graphic on the clipboard to be accompanied by some data that indicates the size and placement of the object. Patron's private format contains both whereas CF_OBJECTDESCRIPTOR only contains extents (since placement data is generally meaningless between different applications). Patron will still look for its own format, then look for CF_OBJECTDESCRIPTOR as a backup.

CF_OBJECTDESCRIPTOR format which will describe the aspect of the object. This is necessary to paste objects that are displayed as icons as we'll see a little later. Also, as we implemented in Chapter 7, this function generates call to the current page to create a new tenant which tells the tenant to create a new object. If we're pasting an embedded object, it sets the tenant type to TENANTTYPE_EMBEDDEDOBJECTFROMDATA.

Eventually this type reaches its way into CTenant::UCreate which instead of calling OleCreate as we did for Insert Object it now calls OleCreateFromData which does exactly the same thing as OleCreate but takes an IDataObject pointer instead of a CLSID on which to base the creation. All other parameters to OleCreateFromData are exactly the same as OleCreate:

```
//In CTenant::UCreate:

case TENANTTYPE_EMBEDDEDOBJECTFROMDATA:
  hr=OleCreateFromData((LPDATAOBJECT)pvType, IID_IUnknown
    , OLERENDER_DRAW, NULL, NULL, m_pIStorage, (LPLPVOID)&pObj);
  break;
```

The interface pointer we receive from this function call is indistinguishable from a pointer returned in OleCreate, so from here we treat this new object with the exact same initialization steps. Note, however, that unlike Insert Object, pasting or dropping an object does not immediately activate it, that is, do not immediately call IOleObject::DoVerb as we did before.

If you are using the Paste Special dialog you will need to add a new entry such that it will now list an available object on the clipboard. This new entry should have the format of CF_EMBEDDEDOBJECT and the flag OLEUIPASTE_PASTE:

```
SETDefFormatEtc(rgPaste[1].fmtetc, m_cfEmbeddedObject, TYMED_ISTORAGE);
rgPaste[1].lpstrFormatName="%s Object";
rgPaste[1].lpstrResultText="%s Object";
rgPaste[1].dwFlags=OLEUIPASTE_PASTE;
```

Be careful to use **OLEUIPASTE_PASTE** and **not** OLEUIPASTE_PASTEONLY. If you use the latter (like we did for other static formats), you will not see this entry in the dialog at all. This can very much increase your job stress by a few orders of magnitude. Believe me. It took me six hours to figure this out.

Also, note that no matter where you put this entry in your array of pastable formats the Paste Special dialog will always list embedded objects first. If you don't agree with this behavior, you will need to create your own version of the dialog based on the sample code in the OLE 2.0 SDK (see the SDK's SAMP\OLE2UI\PASTESPL.C).

Pasting Into an Object: IOleObject::InitFromData

If an object is currently selected in the container when performing Edit/Paste, you may elect to paste into the object itself instead of into your container. If so, you can pass the IDataObject from the clipboard or a drag-drop operation to IOleObject::InitFromData with the *fCreation* parameter set to FALSE. The call will return S_FALSE if the object cannot accept the data to paste, in which case you can paste it as you normally would. If the object returns E_FAIL, then it tried but could not paste the data, so display an appropriate error message.

Copy and Source Embedded Objects

It's sure nice to have cookies in a cookie jar, but that's not the only place you ever want to store cookies. You want to be able to move some of them from one container into another. In other words, if end-

users can paste embedded object into our container then they will probably expect to copy or cut those same objects from our container and put them into another container (it would be nice to *copy* cookies, huh?). In order to do this we have to include CF_EMBEDDEDOBJECT (and CF_OBJECTDESCRIPTOR) whenever we create a data object for a clipboard operation or as a drag-drop source. It might seem like you could just QueryInterface the embedded object for its IDataObject and throw that out for a data transfer, but this is not a good idea because instead of copying the object you are passing a pointer to the real object meaning unpredictable things might happen to it outside control of the container. So you truly need to duplicate the whole object.

This affects Patron down in CPage::TransferObjectCreate which is called whenever we need a data object for some transfer operation. The only modifications here is to include a call to CTenant::CopyEmbeddedObject (after we've copied higher priority formats) which creates both of the new formats and stuffs them in our transfer object:

```
void CTenant::CopyEmbeddedObject(LPDATAOBJECT pIDataObject, LPFORMATETC pFE
   , LPPOINTL pptl)
   {
   LPPERSISTSTORAGE   pIPS;
   STGMEDIUM          stm;
   FORMATETC          fe;
   HRESULT            hr;
   UINT               cf;
   POINTL             ptl;

   //Can only copy embeddings.
   if (TENANTTYPE_EMBEDDEDOBJECT!=m_tType)
      return;

   if (NULL==pptl)
      {
      SETPOINTL(ptl, 0, 0);
      pptl=&ptl;
      }

   /*
    * Create CF_EMBEDDEDOBJECT.  This is simply an IStorage with a
    * copy of the embedded object in it.  The not-so-simple part is
    * getting an IStorage to stuff it in.  For this operation we'll
    * use a temporary compound file.
    */

   stm.tymed=TYMED_ISTORAGE;
   hr=StgCreateDocfile(NULL, STGM_TRANSACTED | STGM_READWRITE
      | STGM_CREATE | STGM_SHARE_EXCLUSIVE | STGM_DELETEONRELEASE
      , 0, &stm.pstg);

   if (FAILED(hr))
      return;

   m_pObj->QueryInterface(IID_IPersistStorage, (LPVOID FAR *)&pIPS);

   if (NOERROR==pIPS->IsDirty())
      {
      OleSave(pIPS, stm.pstg, FALSE);
      pIPS->SaveCompleted(NULL);
      }
   else
      m_pIStorage->CopyTo(0, NULL, NULL, stm.pstg);

   pIPS->Release();

   //stm.pstg now has a copy, so stuff it away.
   cf=RegisterClipboardFormat(CF_EMBEDDEDOBJECT);
   SETDefFormatEtc(fe, cf, TYMED_ISTORAGE);

   if (SUCCEEDED(pIDataObject->SetData(&fe, &stm, TRUE)))
      *pFE=fe;
   else
```

```
   stm.pstg->Release();

//Create CF_OBJECTDESCRIPTOR which OLE2UI handles.
stm.tymed=TYMED_HGLOBAL;

/*
 * You want to make sure that if this object is iconic, that you
 * create the object descriptor with DVASPECT_ICON instead of
 * the more typical DVASPECT_CONTENT.  Also remember that
 * the pick point is in HIMETRIC.
 */
XformSizeInPixelsToHimetric(NULL, (LPSIZEL)pptl, (LPSIZEL)&ptl);
stm.hGlobal=OleStdGetObjectDescriptorDataFromOleObject(m_pIOleObject
   , NULL, m_fe.dwAspect, ptl);

cf=RegisterClipboardFormat(CF_OBJECTDESCRIPTOR);
SETDefFormatEtc(fe, cf, TYMED_HGLOBAL);

if (FAILED(pIDataObject->SetData(&fe, &stm, TRUE)))
   stm.pstg->Release();

return;
}
```

Creating CF_EMBEDDEDOBJECT really means creating some IStorage and saving the object into it. The storage used here is a temporary disk file with STGM_DELETEONRELEASE because the object is potentially so large that a memory storage is a bad idea. Once we have this new storage, get an IPersistStorage interface for the object and call IPersistStorage::IsDirty. If the object is not dirty, then a CopyTo from the object's storage into the new storage used in the transfer is sufficient because the data is current. If the object is dirty, however, then you must call OleSave followed by IPersistStorage::SaveCompleted to make sure that the current state of the object is what's copied. No surprises for the end-user!

The OLE2UI library has a convenient function to create the CF_OBJECTDESCRIPTOR format: OleStdGetObjectDescriptorFromOleObject. This function allocates and fills an OBJECTDESCRIPTOR structure with information it can obtain from an IOleObject pointer. We still have to provide the object's user-readable name (a NULL here in which case the function asks the object), the aspect (which in Patron's case might be content or icon), and a pick point (POINTL, in HIMETRIC) for drag-drop operations.

With these other two formats in the data object, another container can create embedded objects using this data. Patron itself uses this when it copies an object during drag-drop operations (with the Ctrl key pressed) which results in two independent embedded objects that just so happen to contain exactly the same data until either is modified.

Close and Delete Objects

We would quickly fill our hard disks with large documents if we could never remove objects from them. So we now must add some way to delete an object out of a document. This is similar, but not exactly the same, as closing an object when the document is being closed where the object still exists, that is, the object moves from the loaded state into the passive state. Deleting an object is taking it from running or loaded into outright non-existent.

Closing an embedded object means two things: make sure the object's storage is committed and call IOleObject::Close. The latter signals the object's server to completely shut down and purge itself from memory if there are no other containers using it. But as far as our container is concerned this object is now in the passive state such that we must call OleLoad to bring it back to loaded.

IOleObject::Close takes one parameter, called the save option, instructing the object how to proceed:

OLECLOSE_SAVEIFDIRTY	If the object is not dirty, then it can just shut down. Otherwise the object should call IOleClientSite::SaveObject first, then

	shut down. This is the most common flag.
OLECLOSE_NOSAVE	The object should just shut down, discarding all changes. This is a common flag when destroying an object.
OLECLOSE_PROMPTSAVE	If the object is not dirty, then it can shut down. Otherwise it will prompt the user with a Yes/No/Cancel message box asking if they want to save the object. If you use this flag you may get a return value of OLE_E_PROMPTSAVECANCELLED in which case you should not close the object. Use of this flag is rare.

If you are destroying an object (as Patron does when selecting Delete Object from the Edit menu) then you still call IOleObject::Close but generally with OLECLOSE_NOSAVE. This is in addition to destroying the storage element for this object that your site manages. Finally, after destroying an object you should call CoFreeUnusedLibraries just as a matter of habit.

When you test closing and destroying objects, try it with and without the object's server actually running. When it is running, make sure that the server is shutting down completely and is no longer in memory. If it stays in memory (and you are using a reliable server), then you may not be releasing a reference count somewhere.

Save and Load Document with Embedded Objects

At some point or another it would be nice to save all the objects we've been creating and editing such that we might reload them at a later time. We've pretty much covered all the pieces of getting an object saved in a document: provide an IStorage, call OleSave when asked through IOleClientSite or when closing the object, call IPersistStorage::SaveCompleted, and commit the IStorage when you are done as Patron does in CTenant::Update:

```
BOOL CTenant::Update(void)
  {
  LPPERSISTSTORAGE   pIPS;

  if (NULL!=m_pIStorage)
    {
    /*
     * We need to OleSave again because we might have changed the
     * size or position of this tenant.  We also need to save the
     * rectangle on the page, since that's not known to OLE.
     */
    m_pObj->QueryInterface(IID_IPersistStorage, (LPVOID FAR *)&pIPS);
    OleSave(pIPS, m_pIStorage, TRUE);
    pIPS->SaveCompleted(NULL);
    pIPS->Release();

    m_pIStorage->Commit(STGC_ONLYIFCURRENT);
    }

  return FALSE;
  }
```

On a larger scale, Patron's document saving starts in CPatronDoc::USave which first asks CPages to update which asks the currently open CPage to update which in turn asks each tenant to update using the code above. After all that the document commits itself, and being the owner of the root storage, thus writes the file to disk. Little of this storage code has changed from previous versions of Patron.

Patron, by the way is designed–and I won't argue if this is the best design–to only keep the current page open. That means that when you switch pages all open/running objects on the current are closed

back to the passive state and all objects on the new page are opened into the loaded state.

Loading a document in Patron starts with opening a root storage for the document and initializing the pages. Then it opens the current page through CPage::FOpen. This, in turn, recreates all the tenants on the page, but instead of calling CTenant::UCreate it calls CTenant::FLoad telling it the object's storage, what is in this storage, and the rectangle occupied by the tenant on the page:

```
BOOL CTenant::FLoad(LPSTORAGE pIStorage, LPFORMATETC pFE, LPRECTL prcl)
   {
   HRESULT       hr;
   LPUNKNOWN     pObj;

   if (NULL==pIStorage || NULL==pFE || NULL==prcl)
      return FALSE;

   //Fail if this is called for an already living tenant.
   if (m_fInitialized)
      return FALSE;

   m_fInitialized=TRUE;

   //Open the storage for this tenant.
   if (!FOpen(pIStorage))
      return FALSE;

   hr=OleLoad(m_pIStorage, IID_IUnknown, NULL, (LPVOID FAR *)&pObj);

   if (FAILED(hr))
      {
      Destroy(pIStorage);
      return FALSE;
      }

   FObjectInitialize(pObj, pFE, NULL);

   RectSet(prcl, FALSE);
   return TRUE;
   }
```

Most of this again is unchanged from previous versions as OleLoad brings passive objects into the loaded state and returns an interface pointer to us. Now, however, to support embedded objects, we call FObjectInitialize to perform the same initialization sequence that was required after OleCreate, then position the object on the page before repainting happen (by virtue of opening a new window which generates a WM_PAINT message). So we're doing the same steps of querying for IOleObject and IViewObject interfaces, setting up advises, and communicating our IOleClientSite and IAdviseSink interfaces to the object. The difference between this and creating a new object is that we do no immediately activate the object. It would look crazy for a user to open one innocent little document and be hit with a barrage of editing windows!

There is still one small modification to Patron's file I/O which can be found in CPatronDoc::Rename, an override of the CDocument::Rename function in CLASSLIB. This is called whenever the user does File Save As or otherwise changes the name of a document:

```
void CPatronDoc::Rename(LPSTR pszFile)
   {
   //We don't need to change the base class, just augment...
   CDocument::Rename(pszFile);
   m_pPG->NotifyTenantsOfRename(pszFile);
   return;
   }
```

Here we call through CPages::NotifyTenantsOfRename to CPage::NotifyTenantsOfRename which informs each tenant of the new document name:

```
void CPage::NotifyTenantsOfRename(LPSTR pszFile)
   {
   LPTENANT    pTenant;
```

```
UINT      i;

for (i=0; i < m_cTenants; i++)
   {
   if (FTenantGet(i, &pTenant, FALSE))
      pTenant->NotifyOfRename(pszFile);
   }

return;
}
```

Finally, down in CTenant::NotifyOfRename we need to tell any *loaded* objects, not just running objects, of the new host names by calling IOleObject::SetHostNames again.

```
void CTenant::NotifyOfRename(LPSTR pszFile)
   {
   char      szObj[40];
   char      szApp[40];

   if (NULL==m_pIOleObject)
      return;

   if (0==*pszFile)
      LoadString(m_pPG->m_hInst, IDS_UNTITLED, szObj, sizeof(szObj));
   else
      {
      GetFileTitle(pszFile, szObj, sizeof(szObj));

     #ifndef WIN32
      //Force filenames to uppercase in DOS versions.
      AnsiUpper(szObj);
     #endif
      }

   LoadString(m_pPG->m_hInst, IDS_CAPTION, szApp, sizeof(szApp));
   m_pIOleObject->SetHostNames(szApp, szObj);
   return;
   }
```

If you leave this small part out, then your current document name will not be reflected in open server windows which is not, of course, very user-friendly.

After making these additions to your code, you should verify using a tool like DFVIEW.EXE in the OLE 2.0 SDK that all your objects are indeed inside the compound file and that what you expect to be there is actually there. You should then reopen the file and attempt to activate each object. Then make some changes, save the file again, close it, reopen it, and make sure that those changes were actually saved. If all is well, then accept my congratulations, you have just built yourself a basic embedding container. You are now ready for more advanced container features which we'll see in Chapters 12 and 14. But there is just one more option you might want to add now.

Handle Iconic Presentations (Cache Control)

Finally we're at the last step which is optional but highly recommended: support of iconic aspects on objects. I will admit that when I first looked at this feature I was intimidated to the point that I wanted to defer it to a later chapter as some sort of avoidance strategy. After whacking myself in the head a few times with a 40oz Louisville Slugger I keep in my office, I got myself to it for this chapter. And it really didn't turn out to be all that hard.

The trick to iconic aspects is really a special case of the more general issue of controlling what's cached for an object. Back in Chapter 6 when we first used OleSave to serialize bitmaps and metafiles for us (see "Freeloading from OLE2.DLL") we learned that OLE 2.0 maintains a presentation cache for each object that we can control through the IOleCache interface. This interface is always provided from an object handler, and generally the one used is the one in OLE2.DLL. Through it a container can tell OLE 2.0 to not save any presentations at all or to save one different from the default

DVASPECT_CONTENT rendering.

This is exactly what we want to do with iconic presentations, or DVASPECT_ICON. So where do we get the iconic representations? They come from either the Paste Special or Insert Object dialogs when the user has selected "Display From Icon" checkbox, or from a paste or drop operation. The 'iconic representation' is not a handle to an icon; instead, it is a special metafile that contains both the icon and a label stored in such a way that the OLE2UI library can re-extract the icon and label from this metafile. The reason it's all packaged up in a metafile is so containers can throw such a representation into the cache and forget about it.

Icon handling affects a number of places in the code we've developed in this chapter, so let me just run down the list of small modifications that are necessary for Paste Special (as well as paste or drop) and Insert Object:

1. In Paste Special, add OLEUIPASTE_ENABLEICON for the CF_EMBEDDEDOBJECT entry. This enabled "Display As Icon" in the dialog when this format is selected.

2. If you previously hooked the Insert Object dialog or modified its template to disable or hide the "Display As Icon" checkbox, remove it now.

3. On return from Paste Special or Insert Objet the *hMetaPict* field of their respective structures may contain a non-NULL handle. It is your responsibility, regardless of anything else you do, to clean up this memory by calling OleUIMetafilePictIconFree.

4. If Display As Icon is checked on return from Paste Special or Insert Object, the *dwFlags* fields in their respective structures will contain PSF_CHECKDISPLAYASICON (Paste Special) or IOF_CHECKDISPLAYASICON (Insert Object). This is your signal to use DVASPECT_ICON to display the object rather than DVASPECT_CONTENT.

5. When pasting an object using Paste or a drop, examine the CF_OBJECTDESCRIPTOR data that should accompany the object. If there is no such data then the data cannot be an iconic object. Otherwise the *dwDrawAspect* field in the structure will contain the DVASPECT_* value to use. If this is set, you must ask the data object for its CF_METAFILEPICT in DVASPECT_ICON which will be the iconic representation. If you fail to use DVASPECT_ICON you'll get an error.

6. After creating an iconic object from paste, drop, Paste Special, or Insert Object, then you have to make sure that the correct aspect presentation is cached by calling OleStdSwitchDisplayAspects. This function calls IOleCache::Cache for DVASPECT_ICON and sends it the icon-metafile from the dialogs or from paste. It will also free any other currently cached formats and establish an view object advise link with DVASPECT_ICON.

7. When saving and reloading and object, be sure to remember that it's iconic if you later need to copy it to the clipboard or source it in a drag-drop operation–its aspect must then be part of the CF_OBJECTDESCRIPTOR format. However, after you have set the cache for DVASPECT_ICONIC, OleSave and OleLoad will ignorantly save and load whatever is in the cache.

8. Remember to pass DVASPECT_ICON to IViewObject::Draw or OleDraw. If you forget, your objects will draw blank and these functions will return OLE_E_BLANK.

9. Be forewarned that you will receive very few IAdviseSink::OnViewChange notifications for iconic objects because most changes to an object happen to DVASPECT_CONTENT.

Any of the steps above which have to differentiate between content and icon aspects apply equally well

if you wanted to manage thumbnail aspects as well. In all cases, once you tell the cache what to do, OleSave and OleLoad do the right thing. It's mostly a matter of your own structures indicating the correct aspect.

The one missing piece of this icon story is how does one either change the icon at a later time or switch back to viewing DVASPECT_CONTENT for the object? The answer lies in a more involved feature called Object Conversion which is a subject for Chapter 14. If, for now, you want to provide for just changing the icon and label, you can invoke the OleUIChangeIcon dialog box in the OLE2UI library. This function which takes an icon-metafile on input and returns the new one on output. You can then use IOleCache::SetData to change the currently cached metafile to this new image. There's not much else to say about it, because there are no user interface standards for using the Change Icon dialog.

Summary

Support for Compound Documents is a primary technology in OLE 2.0 that allows a container application to embed and link objects from any OLE 1.0 or OLE 2.0 server application. By implementing their part of the compound document standard, containers and servers need not have any knowledge about the other which leads to the ability to integrate data from arbitrary sources into a container document.

An object in a container document may have three states: passive, loaded, and running. A passive object exists entirely on disk and is not visible, printable, or available for any manipulation. When an end-user opens the document in which the object lives and the container application loads the object, it transitions to the loaded state where it may be seen and printed but not edited or otherwise manipulated in any way. Only when the object is activated does it transition to the running state where the user may perform any number of actions on that object such as playing or editing the data.

All three objects states are associated with particular modules being in memory. These modules may be the container application, various OLE 2.0 libraries, an object handler DLL, an in-process server DLL, or a local server EXE. Handlers, in-process servers, and local servers all have their place in the compound document picture with specific responsibilities in each.

This chapter focuses on the compound document container module and how it must be structured to place embedded object in its documents. This chapter gives detailed step-by-step instructions to adding the code necessary to support this feature which includes application initialization and shutdown, site creation and management, site interfaces and storage, creating new objects, initializing objects, drawing and printing objects, closing and deleting objects, dealing in objects with the clipboard and drag-drop operations, and saving and loading documents with objects. Also covered is the optional step of supporting iconic aspects for objects.

Support for linked objects is not covered in this chapter but instead left to Chapter 12 as are a few more advanced features which are left for Chapter 14.