 © Microsoft Corporation 1992-1993. All Rights Reserved.

3. Architectural Overview

OLE 2 introduces incremental changes from OLE 1, leveraging the elements of the OLE 1 design for extensibility and enabling applications to provide and use significantly richer capabilities.

Almost all OLE 2 compliant applications will provide some server functionality, both to objects they contain, and to other external applications or system components. In OLE 1, an application could be only a client and could not provide services to other components. In OLE 2, client and server are roles that applications interactively play with other applications. When describing a compound document situation, it is often more appropriate and clearer to use the terms “container” and “object” in place of “client” and “server” respectively.

The OLE 2 design provides some fundamental interfaces and services, and allows for an extensible set of interfaces for application integration. This document contains specifications of interfaces to support in-place editing and richer compound document integration. The OLE 2 developer’s kit includes the “glue” code necessary to support communication between applications and object handlers using OLE-defined interfaces.

The OLE 2 design explicitly supports interconnection of OLE 1 and OLE 2 applications, so that when such a connection is made, OLE 1 functionality is available to the user. OLE 2 applications will have to be aware that they are connected to a OLE 1 application. This is done through the interface negotiation mechanism, which is described later. Simply put, OLE 1 objects look and behave much like OLE 2 objects for which the in-place active state is not allowed.

3.1. OLE 2 Component Object Model

In OLE 2, objects provide services to their clients through *interfaces.*[17] An object may provide one or more interfaces to its clients, and compound document objects and containers *will* provide more than one. Getting a second interface on an object given an initial one is done using a QueryInterface() operation. Ownership of interface pointers is usually managed through a reference counting paradigm. Many objects allow themselves to be *aggregated,* to be used in an encapsulated way to as a building block for a larger object.

Taken as a whole, this paradigm[18] for how components communicate with each other is known as the “Component Object Model.” Though it indeed forms the lower-level infrastructure on which the rest of OLE 2 is constructed, the Component Object Model has nothing *per se* to do with compound documents. Rather, the model is a general one which forms a sound engineering basis by which software components written and revised by different development organizations over various periods of time can interact with each other and still have a reasonable expectation of interoperating correctly. It is expected that in the future the Component Object Model will have broad application far beyond compound documents.

The rest of this section expands on these concepts.

3.1.1. Interfaces

An *interface* is a grouping of a set of semantically related functions into a named unit. The idea is that if an object implements any one function in this group of functions, then it must implement them all. The set of functions as a whole and their associated semantics defines a contract between the object and its clients. If an object declares that it supports a certain interface, then clients of that object are entitled to assume that it conforms to all of the semantic contract implied by that interface, not just the syntactic signature of the methods defined therein.

The encapsulation of functionality into objects accessed through interfaces makes OLE an open, extensible system. It is open in the sense that anyone can provide an implementation of a defined interface, and anyone can develop and application that uses such interfaces, such as a compound document application. It is

[17] This is as in OLE 1, where interfaces were called “protocols.” This is purely a change in terminology; the semantics of the term remain unchanged. Note, also, that in OLE2 and more generally in the Component Object Model the term “interface” is used in a technically precise sense.

[18] along with transparent remoting. See the next chapter.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

extensible in the sense that new or extended interfaces can be defined without changing existing applications, and those applications that understand the new interfaces can exploit them, while continuing to interoperate with older applications through the older interfaces. Conversely, an implementor of a component can consciously know when he faces the possibility of breaking existing clients in a revision to his product: so long as he continues to support the old interfaces correctly, they will not break; when he ceases to support those interfaces, they will break. In short, communication between components through interfaces makes very clear what services a client is in fact relying on from the service provider. Only if this is very clear can either party expect to revise their implementation without fear of breaking existing, shipped, code with which their revised implementation will interact.

3.1.1.1. Run-time representation of interfaces

Interfaces are a *binary* standard for interaction between components, much as various calling conventions (cdecl, pascal, etc.) have historically been binary standards for interaction between procedures. The run-time manifestation of an interface instance is a data structure that provides access to the functions defined for that interface. This model was introduced in OLE 1, and is continued in OLE 2. The data structure has the following general form:

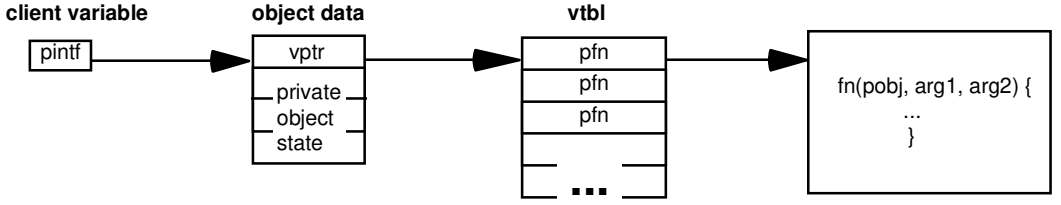

Figure 54. Run-time representation of an interface instance.

Interface instances are always referenced by their clients using pointers. Interface pointers to an object are obtained when the object is instantiated, or by querying the object (through an interface) to obtain a pointer to another interface that the object optionally implements. The interface pointer held by a client is a pointer to a data structure containing the internal state of the object, the first element of which is a pointer to an array of function pointer known as a *vtbl*. Functions pointed to by a vtbl are termed *member functions* (sometimes also called *methods*).

As in OLE 1, the other state information in the object data is not defined for use by the client of the object; it is strictly for the private use of the object itself. Clients cannot see or manipulate the state data directly. The state data is allocated and initialized by the implementor of the object, and is thereafter maintained by the implementations of member functions provided by the object implementor, and finally freed up when the object is released (which is described below).

The convention for calling member functions is to dereference the vtbl to get the address of the function to invoke, and pass in the pointer to the data structure (pintf) as an extra parameter so that the object can access its state information. This is exactly the mechanism used for member function dispatch by some C++ compilers.[19] In this OLE 2 specification, we take advantage of this, and adopt C++ as a *notation* for describing the functionality contained within an interface (programmers are *not* required to actually use C++; all OLE 2 interfaces can both be called from and implemented in C++, C, or any other language with the ability to use tables of function pointers). For example, the interface known as "IUnknown" is documented in this specification as:

```
interface IUnknown {
    virtual    HRESULT    QueryInterface(IID& iid, void** ppvObj) =0;
    virtual    ULONG      AddRef() =0;
    virtual    ULONG      Release() =0;
    };
```

Interfaces, by their nature, have *no* implementation; the definition of a given interface merely specifies the set of function signatures that are found in a particular shape of vtbl (along with, of course, the necessary semantics behind those functions). As a consequence, all member functions are listed in this specification as

[19] Specifically, both the Microsoft and the Borland compilers support this representation, as do very many other C++ compilers.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

"pure virtual"[20] by using "= 0" at the end. The use of "interface" in the first line of code here is purely a documentation technique used to reinforce the lack of any implementation in this definition. "interface" is in fact declared in C++ as:

```
#define  interface  struct
```

(For ease of reading, in this specification parameter types have generally been omitted in member function definitions. These types are, however, documented along with the member function, and of course also appear in the actual header file.)

A particular object will provide *implementations* for the virtual functions in the one or more interfaces that it supports. Objects which have the same set of interfaces and the same implementations for each are often said (loosely) to be instances of the same *class*; instance of a class implements these one or more interfaces in a certain way. However, all access to the instances of the class by clients will only be through interfaces; clients know nothing about an object other than it supports certain interfaces. As a result, classes play a much less significant role in OLE 2 than they do in other object oriented systems.

3.1.1.2. Object pictures

In the sequel we will have several occasions to draw pictures of an object along with the interfaces it supports that it has chosen to expose to its clients. In such pictures we will use a pin-cushion-like notation in order to precisely show which objects expose which interfaces. Figure 55 shows an example of this. Here we have an object that supports three interfaces, A, B, and C.

3.1.1.3. Interface identifiers

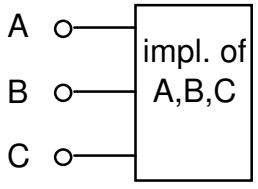

Figure 55. Diagram of an object supporting three interfaces

Each interface has a globally unique identifier by which it is known at run-time. This identifier enables a caller to query an object for a desired interface. Interface identifiers are large: 16 bytes in size. Identifiers are allocated using a special tool (see the next chapter) by the person who defines the semantics of a new interface. The tool guarantees that the allocated identifier will in fact be unique over space and time;[21] in all programs, on all machines, for all time, the run-time identifier for a given interface will in fact have the same 16 byte value.

Programmers who define interfaces convey the interface identifier to implementors or clients of that interface along with the other information about the interface (in the form of header files, accompanying semantic documentation; whatever). To make application source code independent of the representation of particular interface identifiers, it is standard practice that in header files a constant is defined for each interface identifier. The name of the constant can be algorithmically derived from the name of the interface. For example, the interface

```
IUnknown
```

has the associated interface identifier

```
IID_IUnknown
```

For brevity in this specification, this definition will not be repeated with each interface, though of course it is present in the OLE implementation.

3.1.1.4. Interface inclusion and polymorphism

As we have noted, the expression of an interface definition in C++ is a class definition which solely contains public virtual member functions. A question that naturally arises is how C++ inheritance fits into the picture. OLE 2 does make limited use of single inheritance of interfaces as a way to include one interface into another: the extended interface includes all the member functions of the inherited or included one, and the extended interface pointer is polymorphic with the included one.[22] The most common example of this

---

[20] "pure virtual" is C++ terminology meaning roughly "without implementation specified here"

[21] The tool uses the same technology for allocating interface identifiers as OSF DCE uses for essentially the same purpose. The allocation algorithm uses a combination of a machine id and the local clock. Clearly, the guarantee of uniqueness is not totally absolute (what is in this world), but for any practical purposes it can assumed to be.

[22] That is, a pointer pExtended of type IExtended* may safely assigned to a pointer variable of type IIncluded*.

are inclusions of interface IUnknown: most OLE 2 interfaces include IUnknown in order to include the interface negotiation and memory management functionality found there.

However, in OLE 2 we use interface inheritance in a very controlled way, limiting its use to those situations where the client of the functions in the interface extension in fact semantically requires the functions in the base interface as part of its operation; the extension would be useless without also having the base. We are quite careful, by contrast, not to group together using interface extension functions that do not have such an intrinsic semantic connection, for doing so inevitably constrains the use the extension in later otherwise useful situations. Instead, two or more independent interfaces would be created, and clients would travel from one to another using QueryInterface().

This philosophy tends to lead to smaller and more numerous interfaces than otherwise might be the case, but its soundness has proven itself in the ability to reuse well-factored interfaces in otherwise unexpected situations. In fact, earlier drafts of this specification used interface inclusion more heavily than this one. We have found it beneficial to remove the inheritance in cases where we can reuse the extended part of the interface independently.

3.1.1.5. QueryInterface

QueryInterface() is a method in IUnknown interface. It is the mechanism by which given one interface pointer on an object, a second interface is requested on the same object. A input-parameter to QueryInterface() is the interface identifier (IID) of the interface being requested. If the object indeed supports this interface, it returns that interface on itself through an accompanying output-parameter typed as a generic void**; if not, NULL is returned.[23]

(This stylistic use of a pair of parameters, an input IID and an output void**, is a common one in OLE. See for example, CoGetClassObject(), IOleItemContainer::GetObject(), and so on. Clearly this bypasses the compiler's type system; it is up to the object implementation to correctly return through the void** an instance of the requested interface. However, it is very important to realize that aside from this very controlled usage, this is *only* place in the architecture where the compile-time type checking is necessarily sacrificed;[24] at all other times, clients are invoking methods on objects through interfaces whose method signatures (and semantics) are known.)

In effect, what QueryInterface() accomplishes is a switch between contracts on the object. A given interface embodies the interaction required by a certain contract. Interfaces are *groups* of methods because contracts in practice invariably require more than one supporting method.[25] QueryInterface() separates the request "do you support a given contract" from the high-performance use of that contract once negotiation has been successful. Conversely, QueryInterface() provides a robust and reliable way for a component to indicate that in fact it does *not* purport to support a given contract.

It is commonplace and routine that components support many interfaces. More or fewer are used depending on the purpose for which the component was written. Simple, OLE 2 embedding servers, for example, support the interfaces known as IOleObject, IDataObject, IPersistStorage, and IUnknown. If they support in-place activation, then they will support yet further interfaces. By contrast, the OLE 2 extensible link source names known as "monikers" are in fact components which typically support but a single interface, namely IMoniker.

3.1.2. Reference counting and releasing

OLE 2 uses a simple reference counting scheme to manage the freeing of interface pointers on an object. Clients of an interface pointer pintf should call pintf->Release() when then have finished with it, in much the same way that one might call GlobalFree() to indicate that one is done with a global memory block. Release() decrements the reference count; when it goes to zero, the interface pointer is freed. When there are no longer any extant interface pointers on an object, the object itself is freed.

There are cases where one piece of code passes an interface pointer to another, as well as keeping one for later use. Since the code that receives the pointer will Release() it when finished, the code that passes it on

---

[23] See the next chapter for precise details.
[24] Of course, anyone can themselves define an interface that has methods whose method semantics include lack of type safety.
[25] Contrast to traditional inheritance-based language systems, such as C++, Smalltalk, IBM SOM, etc.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

should call AddRef() on the pointer to indicate that there will be two copies of the pointer. This enables the copies of the interface pointer to have independent lifetimes without further communication between them. Detailed rules are presented in he next chapter.

AddRef() and Release() are defined in the interface IUnknown, since there are cases where generic code has to be able to manage objects without knowing much about their supported interfaces; for example, when getting an object from somewhere like the clipboard, it must be possible to free it if it does not support any interfaces known to the client. By definition, such objects support IUnknown, so that is where the reference counting is done.

3.1.3. Aggregation

This subsection discusses aggregation in detail. Though an important part of the overall OLE 2 object model, it is not directly pertinent to understanding how the rest of the OLE 2 architecture hangs together. First time readers of this specification may wish to skip ahead and come back to this when they wish to know more about object implementations.

An effective method of implementing an object is to reuse other objects which support some or all of the required interfaces of the new object. Because of the semantic cohesiveness and independence of well-designed interfaces, it is often the case that the new object would like to take advantage of the entire existing implementation of certain interfaces, while at the same time changing or inhibiting the exposure of other interfaces or exposing new ones. OLE 2 objects support this reuse of existing implementations through a composition technique known as *aggregation.*

An aggregate is an object like any other object in that it implements one or more interfaces. The aggregate has a single identity, and follows the reference counting rules. Internal to the aggregate, the implementation of certain interfaces is actually provided by one or more contained objects. Clients of the object are, of course, unaware of this internal structure. They cannot tell and do not care that the object is an aggregate; aggregation is purely an implementation technique.

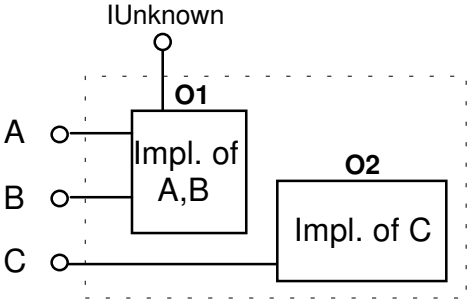

Figure 56. Example aggregate containing two internal objects

An aggregation providing the same interfaces as shown in Figure 55 is shown in Figure 56. For this to work in general, there must be some communication between the implementation of C in object O2 and the other object, O1. Minimally, a Release() on a pointer to the C interface when there are no extant pointers to the A or B interfaces should release the aggregate as a whole. Furthermore, QueryInterface() on C must be able to return A or B.

In the aggregation model, this internal communication is achieved through coordination with a distinguished instance of IUnknown interface known as the *controlling unknown* of the aggregate. The controlling unknown is responsible for which interfaces on which internal objects are exposed outside the aggregate to its clients. Its reference count represents the liveness of the aggregate as a whole. Consequently, the controlling unknown is always implemented as part of the new code written when the aggregate is put together from building block pieces. Objects which allow themselves to be aggregated must be written to cooperate with the controlling unknown by forwarding their QueryInterface(), AddRef, and Release() calls to the controlling unknown. An object learns whether it is being aggregated or not only at instance-creation time; see IClassFactory::CreateInstance(). Not all objects need be aggregateable; whether a given object is aggregateable is an implementation decision on the object programmer's part.

The services required of an element of an aggregate by its controlling unknown in principle need not be discussed here, since the designer of the aggregate (the implementor of the controlling unknown) knows completely the semantics of the objects he is aggregating, and can code the controlling unknown to deal with them appropriately. However, by far the most common case is that the element implements one or more interfaces needed by the controlling unknown, and so must expose IUnknown interface in order that the controlling unknown may use QueryInterface() to obtain these other interfaces.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

Figure 56 redrawn to show its internal structure is found in Figure 57. By convention, we draw the controlling unknown of any object on top of the object. The pointer from object O1 to the implementation of C holds O2 alive as long as the aggregate is alive and needs the implementation of C provided by O2. The pointer from O2 back to the controlling unknown is used by the implementation of IUnknown that is a part of the C interface (assuming, as is usually the case, that C interface is an extension of IUnknown) to delegate back to the aggregate. This pointer was supplied at the time that O2 was instantiated as a parameter to its creation function (see IClassFactory::CreateInstance() later in this specification). A non-NULL value indicates that the object is being instantiated in the context of an aggregate.

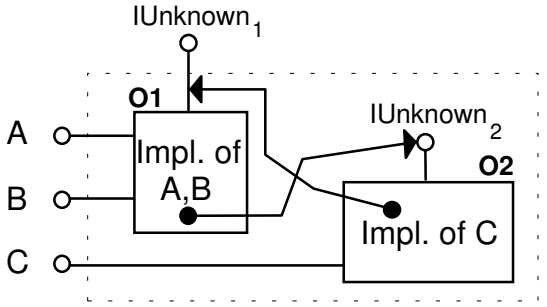

Figure 57. Internal structure of an aggregate

The responsibilities of when QueryInterface() delegates to the controlling unknown and when it does not are a little subtle, and worth clarifying. In Figure 57, there are five occurrences of IUnknown interface (assuming that A, B, and C are each extensions to it), and thus five implementations of IUnknown::QueryInterface(). The QueryInterface() implementations which are parts of interfaces A, B, and C each merely forward the call on to the $IUnknown_1$ interface on O1, which is the controlling unknown for the aggregate. The implementations in A and B have internal knowledge of this controlling unknown, since all three are written together. The implementation in C indirects through the pointer passed at instantiation time. The exception to this delegate-to-the-controlling-unknown rule is the $IUnknown_2$ interface on O2, the object which is being aggregated. Its QueryInterface() implementation always works locally on the object O2. This instance of IUnknown is not obtainable from outside the object; it is called solely by the implementation of the controlling unknown in order to, for example, obtain interfaces on O2 to expose outside the aggregate. The controlling unknown obtained this IUnknown interface pointer at instantiation time (since there is no means to do so after the fact because of the forwarding).

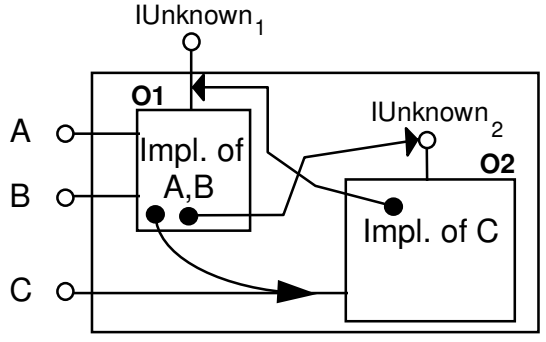

Figure 58. Internal structure of an aggregate

Careful readers will notice that Figure 57 depicts an internal cycle of object references. As we use reference counts to manage object lifetimes, naively such a situation would cause the aggregate as a whole to never be freed. This problem is avoided by declaring that aggregatees must in fact *not* AddRef() the controlling unknown that they are given.[26] Here, for example, O2 would *not* AddRef() the pointer it has to $IUnknown_1$.

Richer interaction between the pUnkOuter controlling unknown an the aggregatee complicates matters yet a little bit more. Consider Figure 58. The difference from Figure 57 is that the controlling unknown here has chosen to store internally a pointer to interface C on the aggregatee, where C is one of the interfaces that is exposed external to the aggregate. As usual, the controlling unknown obtained this pointer using $IUnknown_2$::QueryInterface(). However, O2 did not know at that time of that request that C was being requested not for use *outside* the aggregate, but for *internal* use; therefore, as usual, it must be assumed by O2 that the returned pointer is for *external* use. Thus, a non-zero reference on this pointer will keep O2 alive, which in turn will keep the aggregate as a whole alive. As O1's reference to C will only be released when the aggregate is released, we find ourselves in a situation where the aggregate is never reclaimed.

This problem is avoided by restricting the flexibility that aggregateable objects have in maintaining reference counts: an aggregated object must maintain the validity of *all* of its interface pointers so long as outstanding references exist to *any* of them (non-aggregated objects by contrast are free to invalidate an interface pointer when the last reference to *that interface* disappears). When storing internally what the aggregatee believes to be an external interface reference, controlling unknowns *should release the pointer*

[26] This is just an application of the "nested lifetime" rule for eliding redundant reference counting discussed in the next chapter.

immediately after obtaining it. Here, the reference from O1 to C does not have an outstanding reference count. In practice, this restriction has little impact on most programmers, as it tends to be more straightforward internally to maintain an *object* reference count rather than *interface* counts anyway, especially in C++, where the common implementation technique of inheriting from the interfaces one wishes to implement will fold together IUnknown implementations, making it impossible to distinguish interface-specific releases. We expect that the restriction can be removed in the future when new interfaces are defined by which a controlling unknown can communicate to its aggregatees which pointers it is maintaining internally; however, such a mechanism is not defined at this time.

The following is the most straightforward technique for implementing an aggregateable object.

- Maintain an internal member variable m_pUnkOuter. At creation time, initialize it with either the pUnkOuter supplied to CreateInstance() or with the object's *own* main IUnknown.

```
if (NULL == (m_pUnkOuter = pUnkOuter))
    m_pUnkOuter = (IUnknown *) this;        // here would be IUnknown2
```

- In all IUnknown implementations except the unextended main IUnknown itself (here, in the IUnknown in interface C but not in $IUnknown_2$), trivially delegate all IUnknown methods through m_pUnkOuter:

```
HRESULT MyClass::OtherInterface::QueryInterface (REFIID riid, LPVOID FAR* ppvObj) {
    return m_myClass->m_pUnkOuter->QueryInterface(riid, ppvObj);
    }

ULONG MyClass::OtherInterface::AddRef (void) {
    return m_myClass->m_pUnkOuter->AddRef();
    }

ULONG MyClass::OtherInterface::Release (void) {
    return m_myClass->m_pUnkOuter->Release();
    }
```

This approach applies only to objects who choose to maintain only an overall-object reference count; that is, those that do not maintain interface-specific reference counts which they use to free interface resource as the last reference to each interface disappears, but instead wait until the last reference to *any* interface is gone before freeing any of their internal state.[27] Furthermore, aggregation by its nature requires that an object implementation distinguish between different IUnknown implementations on itself, which is often not necessary otherwise. In C++, this prevents the object from using the straightforward implementation technique of simply inheriting all of the interfaces that it wishes to implement, as C++ will merge the various IUnknown implementations together. Here, we have instead chosen to implement the main IUnknown by means of a nested class; m_myClass is a member of this nested class that the constructor initializes to point to the object as a whole.[28]

Being able to be aggregated is an optional decision on the part of each object implementation. Here, the implementor of O2 consciously chose to support this ability. Providing this support has a small but non-zero cost to the implementor, mostly in making the correct delegation decision in both the aggregated and non-aggregated situations. However, the reward for this paying cost is large in that it enables the effective reuse of implementations as parts of later objects.

3.2. OLE 2 Compound Document Architecture

The OLE2 compound document architecture consists largely of the definition of a suite of Component Object Model interfaces that are implemented or used by the various parties to a compound document interaction. This section gives a brief introduction to some of the interfaces and the roles in which they are used.

---

[27] Not all aggregateable objects need be so implemented when they in fact are not aggregated, but those that choose other implementation approaches will have to do a little more work in order to support the rules described here.

[28] For the curious: pretty much all of OLE2 is internally implemented using this nested class approach.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

3.2.1. In-place Activation Architecture

OLE 2, like OLE 1 is implemented as a set of DLLs that provide services to applications, and manage loading and communication with other DLLs and server applications. The overall picture is as shown in Figure 59.

One of the major additions in OLE 2 is the support for in-place activation. This requires cooperation between applications in managing the user interface. An important architectural question is the relationship of the OLE interfaces to the Windows API. Applications use Windows APIs and abstractions (window handles, DCs, menus etc.) to implement editing and rendering. Applications will use OLE 2 interfaces to communicate and negotiate with each other.

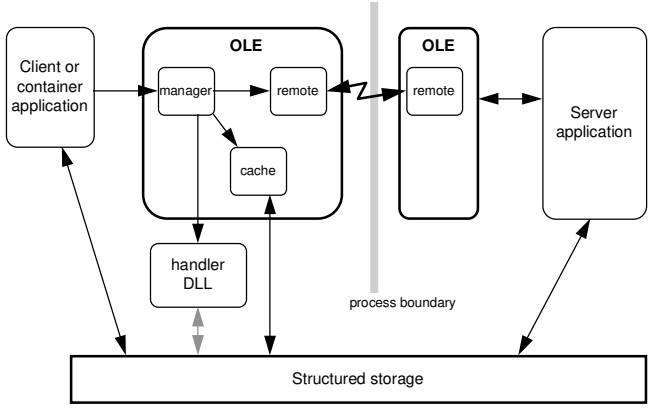

Figure 59. Overall structure of an OLE 2 embedded object.

One of the design goals of OLE 2 was to minimize the development cost required to convert a Windows (or OLE 1) application into an OLE 2 application. To this end, the design does not call for re-implementation of the editing and rendering code, rather it calls for supplementing that code with the OLE interfaces to negotiate the merging of the user interfaces.

In this context, another way of visualizing the structural differences between OLE 1 and OLE 2 is to look at the relationship between OLE applications and the Windows API. Figure 60 illustrates this relationship as it was in OLE 1. This relationship is largely undisturbed in OLE 2, which is shown in Figure 61.

Enabling the server to edit within the client's window space — while preserving the majority of the server's code and the programming model — requires that the window trees be merged, as shown in Figure 62. To support this, the object editor (or server) must be able to decouple its editing window from the frame and from any additional ornamentation. Frame-related controls like menus and toolbars must be associated with the container's frame. OLE 2 interfaces will be used to set up the frame controls and connection between windows. The Windows API and messages will be used to edit and to receive input and commands, etc.

Floating windows, pop-ups, etc., will be created as owned by the server's editing window. Adornments like rulers that must be attached to the object being edited (rather than the frame) must be children of the server's editing window. If necessary, the server can negotiate more space for these tools or temporarily overlay them on parts of the client.

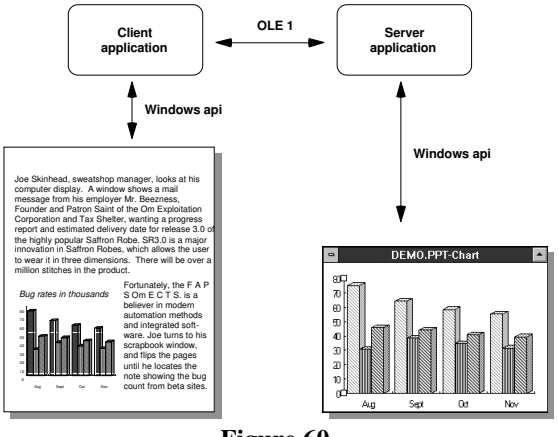

Figure 60.

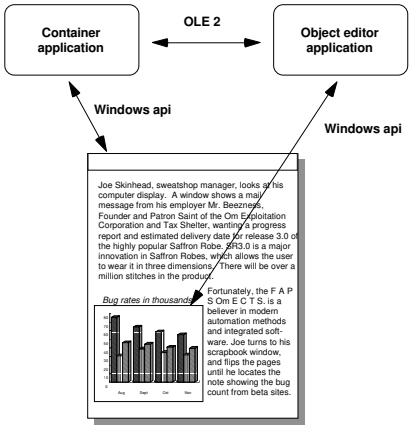

Figure 61.

Tools that are attached to the frame or document windows are created as children of the respective window. The object negotiates with the container about whether it is allowed to create such a window, and the container moves other UI elements to make room if necessary.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

3.2.2. OLE Interfaces, Extensibility and Layering

Interfaces have an associated data type for use by the programmer, such as IOleObject, and a name that is used as a system-wide identifier, such as IID_IOleObject. This identifier is analogous to the protocol names such as "StdFileEditing" found in OLE 1.

When a client instantiates an object (loads it from disk, creates one, or gets one from the clipboard or through a drag/drop, etc.), the client specifies the interface that it wants to use to talk to the object. In OLE 1 the interface was named "StdFileEditing." For OLE 2, there are a set of interface names that may be requested. If the object (i.e. its handler or server) does not support the requested interface, the instantiation request fails.

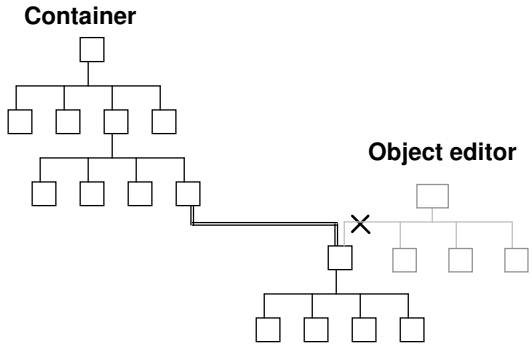

Figure 62.

OLE 2 exploits interface extensibility to provide a layered architecture where not all interfaces need be supported in order to be compliant. Some optional support will be handled by providing default behavior that conforms to the interface and that provides degenerate handling. One example of this will be for OLE Automation, support for which is connected with the IDispatch interface. Not all OLE objects need support this interface; an external client that could make use of it can inquire if it is present and use it if so.

An brief introduction to some of the important interfaces that compound document application developers commonly see are as follows. The discussion of interfaces should be read in the context of the system structure—how the parts fit together.

Name	Description
IUnknown	Provides negotiation of other interfaces and reference counting.
IClassFactory	Provides the method used to create instances in memory. This is approximately the equivalent of OLESERVER in OLE 1, and is only used by OLE 2 during the object creation process. IClassFactory will not typically be used directly by containers.
IOleObject	The primary interface used by containers to interact with their embeddings. IOleObject together with IDataObject and IViewObject provides approximately the same functionality as OLEOBJECT in OLE 1. IOleObject adds the methods appropriate to compound document support that are not included in IDataObject or IViewObject: methods for managing verbs, update control, layout and shape information.
IDataObject	Contains methods for retrieving, storing and enumerating data, and requesting notification of changes. IDataObject interface provides a data transfer capability very much in the spirit of DDE, but generalized and more robust. It is used uniformly in several places in OLE where data exchange needs to occur: the clipboard, drag-drop, etc. The uniform use of a single mechanism in each of these situations leverages data transfer code in applications.
IViewObject	Contains functionality for displaying/printing objects, locking out updates (e.g. for band printing), and requesting change notification.
IOleCache	The interface used by a container application to control the kind of data that is cached its embedded objects.
IOleContainer	IOleContainer is an interface that is generic to many kinds of container. It provides name parsing and enumeration of objects; that is, an outside-in view of the composite set of objects managed by a container. OLE 2 applications will normally use IOleItemContainer (see below) to expose their contents.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

IOleItemContainer	This is an extension to IOleContainer that adds the method used to bind to an object in the container. This interface subsumes the functionality associated with the OLECLIENTDOC and OLESERVERDOC in OLE 1.
IOleClientSite	The objects view of its context: where it is anchored in the document and where it gets its storage, UI, and other resources. IOleClientSite provides approximately the same functionality as the OLECLIENT interface in OLE 1.
IPersistStorage IPersistStream IPersistFile	Interfaces used by clients to ask objects to load / save themselves using the respective storage media.
IStorage, IStream	Storage media interfaces used by applications to store and retrieve data from persistent store, for structured (objects) and flat (data) storage respectively.
IOleInPlaceObject	An interface containing methods for controlling an object that can activate in place.
IOleInPlaceSite	Interface that the object uses to obtain information from the client site, and negotiate over UI.
IDropTarget	Interface used to communicate with the (potential) target of a drag/drop action.
IDropSource	Interface used to communicate with the source of a drag/drop action.

3.2.3. Containers and Objects

As in OLE 1, containers are clients for services (rendering etc.) provided by objects. Containers also provide services to their objects and negotiate with them over in-place editing. When a container instantiates an OLE object, it provides a pointer to a Client Site interface, through which the object calls on these services.

Containers also provide services to other components through OLE container interfaces. These services relate to the objects in the container: enumeration, name resolution, etc. In addition, containers may offer services related to their own data by permitting binding to parts of their data through OLE object interfaces. While a container is active, it may be called to provide services, subject to the same sorts of design rules as the OLE 1 server interfaces.

Containers and objects may also offer command interface services, to permit external programming tools and other applications (perhaps with application programming languages) to invoke commands.

As an illustration, let us examine some of interfaces that would be (optionally) implemented and used by various kinds of OLE2 compound document application.

3.2.3.1. Container-Only Applications

First, consider a simple container application such as Windows Write.

Interfaces Implemented	Interfaces Used
IClassFactory	IStorage
IOleClientSite	IPersistStorage
IOleItemContainer	IOleObject
IPersistFile	IDataObject and/or IViewObject
	IOleCache
Interfaces Optionally Implemented	**Interfaces Optionally Used**
IOleInPlaceSite	IOleInPlaceObject

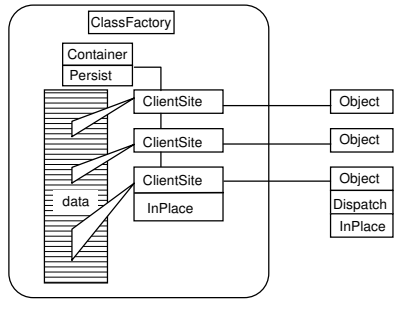

Figure 63. Structure of a container-only application

The application that manages the container document will normally associate contained objects with a position within the document. There will often be container-supplied properties associated with the objects, a name for linking, etc. The Client Site interface is where the application should provide this information to

 © Microsoft Corporation 1992-1993. All Rights Reserved.

its objects. The application may contain embedded objects which in turn have links to their parts; IPersistFile and IOleItemContainer support this functionality.

3.2.3.2. Single-Object Server Application

A simple single-object server application like MS WordArt would see the following interfaces:

Interfaces Implemented	Interfaces Used
IClassFactory	IStorage
IPersistStorage	IOleClientSite
IOleObject	
IDataObject	
Interfaces Optionally Implemented	**Interfaces Optionally Used**
IDispatch	IOleInPlaceSite
IOleInPlaceObject	

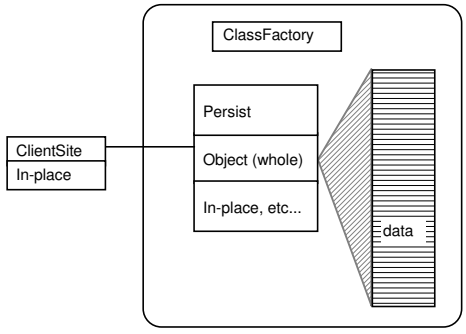

Figure 64. Structure of a single-object server application.

3.2.3.3. Multiple-Object Server Application

A server application that supports links to ranges of its data (such as bookmarks in a word processor, or cell ranges in a spreadsheet) would be similar to the above server case, with the addition of IOleItemContainer interface to support binding to named items within the object.

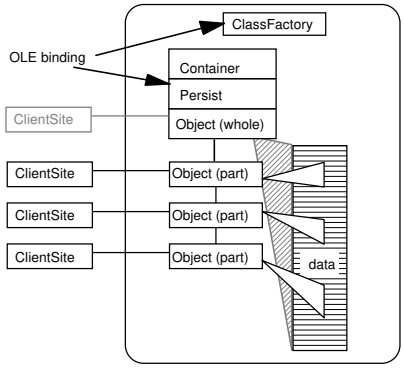

Figure 65. Structure of a multiple-object server application.

Interfaces Implemented	Interfaces Used
IClassFactory	IStorage
IPersistStorage	IOleClientSite
IOleObject	
IDataObject	
IOleItemContainer	
Interfaces Optionally Implemented	**Interfaces Optionally Used**
IDispatch	IOleInPlaceSite
IOleInPlaceObject	

3.2.3.4. Container and Server Application

An application that is both a container of other OLE objects and also a server for its own data (presented as objects) will implement interfaces as shown in the following table. The IOleContainer interface will be used to bind both to objects that this object contains and to named items managed by this server. The application is responsible for determining whether a name refers to a contained object or a range.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

Interfaces Implemented	Interfaces Used
IClassFactory	IStorage
IPersistStorage	IPersistStorage
IOleObject	IOleClientSite
IDataObject, IViewObject	IOleObject
IOleItemContainer	IDataObject
IOleClientSite	
Interfaces Optionally Implemented	**Interfaces Optionally Used**
IDispatch	IDispatch
IOleInPlaceObject	IOleInPlaceSite

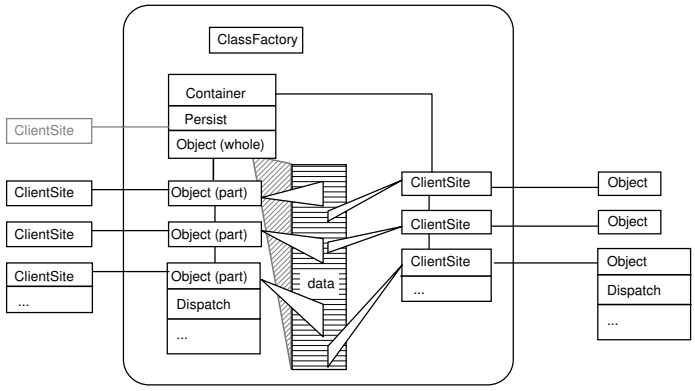

Figure 66. Structure of an application which is both a container and a server.

3.2.4. States of Embedded Objects

Figure 67 illustrates the main states of objects as they relate to the execution of servers when the objects are instantiated by containers, etc. In addition to the object states, there will be relevant states of server applications (e.g., dormant: invisible, not object, but server task is running so that getting into the Editing state is quick).

There is no requirement that all objects in a container be loaded at once; the container may choose to optimize the loading process. Similarly, containers may defer making the object running, in order to conserve memory, or may pre-activate server tasks for fast startup.

Passive: Object is on disk.

Loaded: Object structure is in client memory. The object may use the cached picture for rendering. The native object data will not normally be in memory.

Running: The server .EXE (or handler) for the object is running, with the object data available to it. The object can do full negotiation and rendering, etc. Clients who have links to this object may bind to it or receive notification of availability.

From the server's perspective, it is sometimes (though less often) useful to distinguish variations in the running state:

Ready: The server or handler has created UI resources, but does not have the focus. The server's tools and menu may be visible but inactive if the focus is in a different window, or invisible if the focus is elsewhere in the container's window hierarchy.

Active: The server or handler has the menus and is getting input.

Open: The server or handler has opened the object in a pop-up window, which may look like its normal frame window.

Executing: The server is asynchronously executing some command invoked by the client.

Note that there is no direct correspondence between the UI states described earlier and the states illustrated above. A UI Inactive object, for example, could be Passive, Loaded, Running, Ready, or Executing. The

 © Microsoft Corporation 1992-1993. All Rights Reserved.

states where the two classifications coincide are indicated by identical names: objects that are *Active* or *Open* in the UI sense, are also Active or Open in the sense defined here.

3.2.5. Instantiating, Binding, and Closing Objects

The term *object* refers both to structures that represent embedded or linked OLE objects (i.e., objects accessed through interfaces like IOleObject) and to instances of supporting interfaces provided by applications, like IOleItemContainer and IOleClientSite. In both cases, the services are accessed through a defined interface using an appropriately typed pointer. See the next chapter for a discussion of the representation of interfaces and pointers in C and C++.

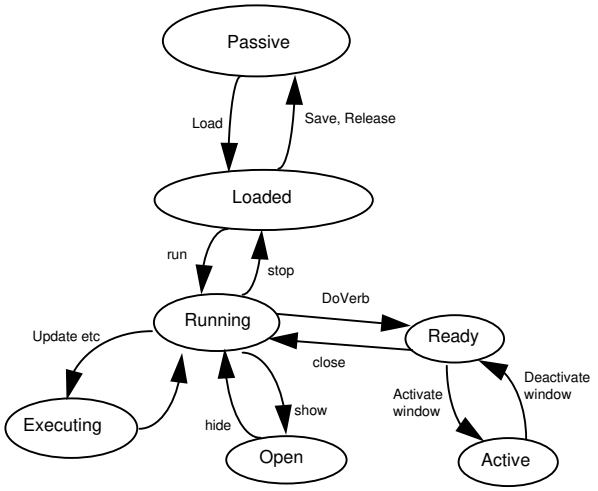

Figure 67. Sketch of object states transitions

Instantiation refers to the process of creating the in-memory structures that represent the object, so that operations can be invoked on it. Instantiation of an embedded or linked object occurs when an object is loaded from persistent storage (i.e., on transition from a Passive to Loaded state), or when an object is being created for the first time.

Internally, instantiation is a two-phase process: first, an object of the appropriate class is created, then a method on that object is called to perform initialization and to give access to the data for the object. The initialization method(s) is(are) defined in one of the interfaces that the object supports. The particular initialization method that is called depends on the context in which the object is being instantiated and the location of the initialization data.

From the application developer's point of view, instantiation happens in one of three ways:

- A container application calls an OLE-provided function to instantiate an object and initialize it either from persistent storage or as a new object. In this case, the application will call either OleLoad() or OleCreate(), specifying the desired interface. Internally, OLE will
    1. Determine the class of the object, either from storage or from the argument to OleCreate().
    2. Locate the code that manages this class of object and that can initialize it in this context; this will normally be the OLE library, but it can be a class-specific handler library; the library returns an instance of IClassFactory interface.
    3. Call the CreateInstance() method in the interface located in preceding step.
    4. Call the instance's Load() or InitNew() method, as appropriate, to initialize the object state; in the normal case (where OLE is the handler for the object), OLE will remember for later use where to find the native data, and will locate data needed to draw the object, etc.
    5. Query the object for the desired interface pointer and return it, if supported; otherwise, the release the object.
- An application acting as a server is called upon to instantiate an object on behalf of a third party. For example, IOleItemContainer::GetObject() might be invoked in the process of the third party trying to connect to a link source. In this case, the application is completely in control of the instance creation and returns the pointer to the created object. Subsequently, an initialization method will be called on that object.
- The application creates an instance of one of the OLE-defined supporting object structures (such as IOleContainer or IOleClientSite) and passes the pointer to one of the OLE functions or methods (e.g. IRunningObjectTable::Register() or IOleObject::SetClientSite()). In this case the application is completely in control of the instantiation.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

The generic term "binding" refers to the process of getting an object into the Running state so that server-supplied operations (such as editing) can be invoked. There are two cases: a) for an embedded object where the server code is located and run, and the object data is given to the server, and b) for a linked object, where first the object data is located and then the server is located and run as above. In the case of linked objects, the server has a little more flexibility on how the object data is opened. OLE objects are put into the running state either explicitly on request of the container (using OleRun()), or implicitly by OLE as a side-effect of invoking an operation on the object, such as IOleObject::DoVerb().

Internally, there are many components involved in binding. For a link, the object data may have to be located, which in turn may require binding to other containers. If the object is already running somewhere, it is necessary to establish a communication channel to it; otherwise, the server code must be located and run, the data made available to it, and a connection established. Remote procedure call proxies must be located and called for the interfaces requested by the client and/or used during instantiation. Various internal OLE components combine to achieve binding.

3.2.5.1. Running an Embedded Object

The sequence of events for running an embedded object is roughly as follows, assuming that the object is Loaded.

```
find a Class Factory object for the class of the OLE object
if not found
    locate the server code through the registration database
    run the server code (will register the ClassFactory)
    find the running ClassFactory object
    (ClassFactory interface pointer will normally be to an RPC proxy)
send CreateInstance method call requesting some IPersist* interface
    (returns pointer to IPersist interface, usually an RPC proxy)
if creating from scratch
    send InitNew method passing a handle to an empty IStorage
else
    send Load method call to the object, passing the storage handle (see the chapter on persistent storage)
send QueryInterface method call for the client's requested interface; this may build another proxy
return the result of the above, and clean up.
```

All of the preceding sequence of events are normally only internal to the OLE implementation. The embedded object, as seen by the container application, will appear to have changed its state to Running. The preceding outline illustrates the places where server applications are involved: registering the server object, responding to the CreateInstance() call, and responding to the Load method to initialize the object. OLE takes care of finding registered servers for other clients, setting up RPC proxies, etc.

3.2.5.2. Binding a Linked Object

At first glance, binding a linked object would appear to be but a little more complicated than the outline for running an embedded object. However, three things complicate the story:

1. OLE 2 allows links to refer to objects (and possibly parts of objects) that are embedded in other containers. This of course implies that there is the potential for nested embedding. It follows that the link must include sufficient information to locate the object inside of the other containers. For this we use what is conceptually an extension of the path name scheme for locating files in a file system. It is, however, somewhat more complicated than merely concatenating component names for reasons that are described in a subsequent chapter. To indicate their special nature, these composite names are called *monikers*. In summary, the important properties of monikers are:
    - A moniker's representation is opaque to applications; monikers are to name components what OLE embeddings are to data formats.
    - Applications can compose (concatenate) and decompose a moniker without knowing its syntax. This is important, as it allows an application to give a composite moniker that describes, for example, a range within a spreadsheet. The application is able to do this without having to know the syntax name by which the spreadsheet itself is known.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

- During binding, names are *reduced* before the steps in binding take place; this allows for important functionality such as link tracking and user aliases to be implemented transparently.
- OLE defines a set of moniker (component) types that provide standard ways for applications to participate in naming support; most applications will not themselves have to implement their own monikers. The main such type is Item Moniker. Item names are string names assigned by the server or container application. The item names are used during binding in the IOleItemContainer::GetObject() call, analogous to OLE 1. An Item Moniker is an encapsulation of an item name that can be composed with other monikers; these monikers may encapsulate pathnames and other item names in the containment hierarchy, etc.

Typically, an object's moniker will be part of a filename (pathname) concatenated with a series of item name parts.

2. Running an object requires getting to the storage area for the data. When running an embedded object, OLE is given access to the storage while Loading the object. OLE then holds on to the handle so that it can pass it to the server on binding. OLE defines the particular interface by which the storage for embedded objects is accessed (details are in the chapter on "Persistent Storage for Objects"). However, when running a linked object, OLE does not mandate an interface; instead, it permits the server to use the regular file system interface or some other mechanism. This means that a different initialization method is called during binding.
3. Linking to objects permit multiple references to a given object; during binding it is possible that the object may already running, in which case OLE will bind to the running version rather than running another copy. Coupled with the first point above, we can see that OLE may have to bind to an already running object at any point in the containment hierarchy.

The following outlines illustrates what conceptually happens when binding a linked object connected to a link source in the most common scenarios. For more precise (and more accurate) details of the binding process, see the chapter later in this specification on linking and naming.

```
Extract the moniker from the link object
Reduce the moniker (reduces each component; many types reduce to themselves)
initial pcontainer is the file system
For each component from left to right
    if the object identified so far is running
        connect to it and verify that it has the required interface
    else
        if this is the last component
            required interface is what the caller wants
        else
            required interface is IOleItemContainer
        if this is a file moniker
            bind to file (see below) for required interface
        else (item moniker,we already have a container)
            call pcontainer->GetObject( itemname, required interface ID)
            result is either new pcontainer for next component, or final result

Bind to file (required interface)
Find a ClassFactory object for the class of the object in the file (must have been registered)
If not found
    locate the server code through the registration database
    run the server code (will register the ClassFactory)
    find the running ClassFactory object
ClassFactory interface pointer will normally be to an RPC proxy
send CreateInstance method call requesting IPersistFile interface, get back an object proxy
send Open method, passing in file name
return the result of the above, and clean up.
```

Cases similar to binding to files (for objects which are not actually in files) are omitted in the interests of exposition. Also omitted are details of how RPC proxies are instantiated and connected to the right servers. These happen during the RPC return from calls to servers, like CreateInstance(), Load(), GetObject(), and the infrastructure for finding running servers and objects.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

3.2.5.3. Closing Objects

Closing objects is normally initiated from the server's user interface (for example, **File/Close**). In this case, the server sends the appropriate notification to the object's Client Site (saved, closed, renamed, etc.). In cases where the object is being closed, the client will send a Release() call to the object. Objects (including such objects as IClassFactory implementations) must maintain a reference count; when a Release() call causes the reference count to go to zero, the object should be freed. A server application can safely terminate when it has no objects left (counting Class Factories).

When a client wants to close the object, it should call Release() (after optionally saving the object). Clients will normally release objects in a lazy manner when they are de-activating after in-place activation. This will allow the user to click outside the object but not lose the active editing session until some other action causes the client to Stop and/or Release the object.

3.2.6. Creation, Transfers and Clipboard Handling

A major part of the OLE 2 support required of an application deals with creating objects and moving them around among containers.

3.2.6.1. Creating New Objects

New objects are created in several ways. The recommended and possibly the most common way is to take a copy of an existing object and insert it into the desired place, and perhaps then modify it. Generally, the transfer commands described in the following section will be used.

The User Interface chapter of this document recommends an Insert Object command. This command can be used to create a new object either from scratch or initialized from a file that the user identifies in a browser dialog.

To create an object from scratch, an application calls OleCreate() to create the object; this will instantiate the object using the IPersistStorage::InitNew() method, passing in a handle to where the storage will reside. Once instantiated, the object will be put into the running state and the server application's IPersistStorage::InitNew() method will be called. The server should initialize the object to the default state for that kind of object. On return from the OleCreate() call, the object will be Running but not yet visible to the user. The container application may initialize the object from the container's current selection, and then call a method on the object to show the object to the user and activate for editing—either in place, or in another window. When creating objects, containers should inquire with IOleObject::InitFromData() as to whether the new object wishes to initialize itself from the selection. If so, then the container should provide the object with the appropriate data.

If the user wants to create an embedded object that is a copy of the data in a file, then the application should call OleCreateFromFile(), passing in the selected pathname. On return from this function, the object will be in the Running state as in the case of OleCreate() above, but again not visible to the user.

3.2.6.2. Transferring Data and Objects

A selection may be transferred from a source document to a destination document either by a drag/drop action or through the regular clipboard commands. The transfer may copy or move the data, or make a link. The selection may be a range of data in the source document, and may contain embedded objects and/or links. For some containers, the data portion may be empty (for example, a file folder that has no significant container-level data associated with the objects).

The transfer is negotiated using the IDataObject interface. When initiating a transfer, the source application creates an IDataObject and assigns the formats, etc. that describe the selection. For a drag transfer, the application calls the drag API function, DoDragDrop(). For a clipboard transfer (Copy, Cut) the application calls OleSetClipboard(), which puts the handle to the IDataObject on the clipboard. OleSetClipboard() also offers the regular data formats to the clipboard for non-OLE and OLE 1 applications.

When a drop happens, or a Paste command is invoked, the destination application retrieves the IDataObject and enumerates its contents to determine whether the transfer is valid and, what action to take (convert and merge the data, or embed it as an object, etc.). Applications will use the same code to provide and consume

 © Microsoft Corporation 1992-1993. All Rights Reserved.

these data components; only the place holder will differ between clipboard and drag transfers. The OLE clipboard-handling functions (OleCreateFromData(), OleQueryCreateFromData(), etc.) will handle this negotiation on behalf of the application. The application may itself examine the IDataObject to decide what Paste Special options to enable.

Non-OLE or OLE 1 applications of course do not directly offer IDataObject instances to the clipboard with OleSetClipboard(); they merely offer regular clipboard formats. In this situation, OleGetClipboard() will synthesize an appropriate IDataObject instance initialized with the clipboard contents. Thus, OLE 2 applications need not care whether or not data offered through the clipboard came from and OLE 2-aware source.

If the destination container wants to merge transferred data into its own data by requesting a data format that it understands, it can copy over any objects or links that the source contained as part of the selection. For this purpose, the destination application should choose the format it wants, request it from the IDataObject, then merge it in, noting in the process where objects are placed relative to the data. The destination application should then create IStorage instances for each object, and copy the objects from the source's storage.

3.2.7. Interoperability of OLE 1 with OLE 2

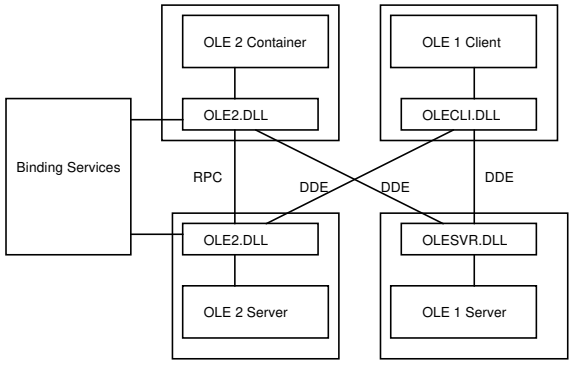

Figure 68. Communication between OLE 1 and OLE 2 applications.

As described in the preceding sections, it is explicitly possible to create and transfer objects and data between OLE 1- and OLE 2-compliant applications. Similarly, the objects themselves may be created by either kind of application. To an OLE 2 container, an OLE 1 object will appear to support IOleObject interface with no in-place activation capability. It will not be possible to create links to embedded objects contained inside OLE 1 containers, since they will not provide monikers in transfer operations. To an OLE 1 container, an OLE 2 object will provide the expected OLE 1 interface, and OLE will map the calls made by the container into calls on the object's IOleObject and other interfaces. The object will not be able to activate in place; it will instead have to Open itself if a executing verb requires provision of some user interface. Figure 68 illustrates the communication connections between OLE 1 and OLE 2 applications.

3.2.8. Handlers and Applications

OLE 2 allows object services to be provided both by lightweight handler (DLL) code or by an application running as a separate task. Handlers may be used for performance, better rendering etc. Separate server applications will probably be more common, since OLE 2 allows existing applications to become servers at lower development cost than rewriting them as (a set of) handlers.

In OLE 1, OLECLI.DLL is (roughly) equivalent to an OLE 2 proxy, and OLESVR.DLL is analogous to an OLE 2 stub; handlers (if they exist) are loaded and invoked when the object is instantiated, in preference to using the proxy. The handler may *itself* call OLECLI.DLL as a proxy. OLE 2 follows basically the same model: a handler has complete freedom to implement the services that it provides for its clients, and in doing so it can draw on the default implementations provided by OLE of those services.

Because of the need to support interoperability between OLE 1 and OLE 2,[29] and between 16-bit and 32-bit versions of applications, handlers should generally only be seen as an optimization; a class should generally not *require* that a handler be used by its clients. This does not, however, prejudice a developer who wants to write his code entirely as a handler DLL. That developer can make a simple server application that loads and calls the DLL to do most of its work. The application would have to provide the OLE registration calls,

[29] Remember that OLE 1 had both a DDE and a library-based implementation strategy. This complicates the compatibility issue.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

perhaps the frame window support, and the stack, message pump etc. which are the normal characteristics of a Windows application.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

4. The Component Object Model

With this chapter we begin the presentation in this specification of the actual functions and interfaces that form OLE 2 and its underlying foundations. Detailed function prototypes and descriptions are presented.

The body of functionality discussed in this chapter forms the core of what is known as the "Component Object Model." This object model and the infrastructure that supports it forms the heart of the central underpinnings on which OLE 2 is built; however the model has nothing directly to do with compound documents *per se;* rather, it has a much broader applicability to component software in general.

The Component Object Model defines:

- the concept of an *interface* as the *lingua-franca* by which clients of services communicate with entities which provide those services. These entities are referred to as "components" or "objects." Interfaces promote clarity of understanding as to the services from a given component that a client can actually rely on using long-term as the component is revised, updated, and otherwise replaced.
- an architecture (QueryInterface) by which objects can support *multiple interfaces*, which provides a robust means for an component to support optional or new interfaces without breaking existing clients, and conversely provides a ubiquitous means by which a updated client, aware of new functionality it desires to use, can safely inquire of a given component whether it *too* understands this new functionality. This inquiry is safe even if the component is an old, unaware one.
- a *reference counting* model of object management that enables objects to be reclaimed when they are no longer needed, and which permits an object to be simultaneously used by many non-communicating clients, yet still know when it is no longer needed.
- a mechanism by which pointers to interfaces can be passed through *remote procedure calls*, enabling clients in other processes to access and manipulate objects. This mechanism is transparent to both the client and the server side of the conversation: servers need take no special action to be remoted, and clients need take no special action in order to deal with remoted services.
- an architecture (CoGetClassObject() and related infrastructure) by which a desired object implementation can be identified and *dynamically loaded* into the running system, either as a .DLL or an .EXE server in a separate process. Clients need not be aware of whether an object is implemented locally or remotely.
- a level of indirection to object implementations though the System Registry (see CoGetClassObject()). Since static binding to the implementation (using DLL names, for example) is not used, this additional level of indirection frees clients from knowing the packaging into files (DLLs, EXEs) of the functionality they use from various services, allowing that packaging to be changed in the future without breaking existing clients.
- a standard mechanism (use of the memory allocator returned by CoGetMalloc()) by which ownership of data containing memory allocations can be passed between parties. To give but one motivating example, this allow strings to be returned from server to client without the client having to allocate a buffer in advance. Of course, this applies to any data value.

This chapter is organized as follows. We begin by examining the technical calling conventions for OLE interfaces used with both C and C++. Next, we discuss the stylistic convention used for returning function results and error status. Following this, IUnknown interface and enumerators are presented, then a discussion of memory management. A discussion of class binding and object instantiation is next, and we conclude with a look at part of the new remoting infrastructure put in place for OLE 2.

4.1. Calling conventions; Calling interfaces from C++, C, or other languages

As was mentioned in the previous chapter, OLE 2 interfaces are documented in this specification using C++ syntax as a notation, and programmers using C++ compilers will be able to call these functions directly and