Chapter One
Introduction

All evolution in thought and conduct must at first appear as heresy and misconduct.
George Bernard Shaw (1856-1950)[1]

Someday in years ahead a variation of Charles Darwin will look back on Windows and wonder how it evolved from Windows the API into Windows Objects, an object-oriented operating system. OLE 2.0 is the genesis of this transformation that changes how you program–and eventually how you use–Windows. In the beginning it will probably seem utterly strange and difficult, no matter what your background. But don't feel too threatened. I won't ask you to throw away any knowledge you've accumulated. Instead, we'll ease into the features of OLE 2.0 and see how those features, combined with everything you already know, can help you achieve new heights.

Today Windows is exposed to applications in a very large–and ever growing–list of randomly named APIs. Every API is created equal, so to speak, and is available from virtually any piece of code regardless of how useful such accessibility provides . Over the years, many different sets of new APIs have emerged, each in their own way describing some new capability of the system, each in their own way providing yet another different set of functions by which an application implements various features.

Such an environment is ripe for object-oriented languages like C++ and class libraries like Microsoft's Foundation Classes (MFC) or Borland's Object Windows Library (OWL) to flourish. These products provide some order to the turbulent chaos of the API waters. Instead of dealing with something like a window through a handle and numerous APIs spread thin around the reference manuals, these products shelter a window handle in a C++ object class and provide functions to manipulate the window through the object instead of the handle.

Windows Objects is the same sort of treatment to expose system features as objects instead of APIs, providing the much-needed order to the ballooning list of technologies: your use of system features means using system objects instead of handle-based APIs. Furthermore, Windows Objects describes how applications expose their pieces to both the system and other applications. Realize that the technique of *exposing* an object does not restrict how you *implement* the object. That is, while C++ is the most natural language in which to express the concepts you can write objects just as effectively in C or even assembly. That means that eventually even device drivers could be exposed as Windows Objects, but by no means must they be implemented in a less efficient or streamlined language like C++.

Windows Objects are built on a foundation that also allows an object's code to live anywhere–within a particular application, in a DLL loaded into your application task, in another application, or even on another machine (in the future when OLE is network enabled). The object model that OLE 2.0 introduces lays the evolutionary groundwork for distributed object computing in the years ahead.

OLE 2.0 exposes a number of key systems features, such as the clipboard and the file system, through specific objects. These objects are implemented on top of the existing Windows APIs such as SetClipboardData and OpenFile. Using these objects today will, of course, cause a slight decrease in overall performance since there is another layer of function calls to accomplish the same task as before. For the programmer, however, the overall surface of APIs is markedly reduced, as many of those APIs are moved into member functions on a particular object that you only see when you are manipulating that object. The only APIs that remain are a few to initially obtain one of the system objects.

While you will suffer from a small performance penalty today, the object implementations of the system features will be gradually absorbed into system, becoming the native expression of those features. While the APIs will still be available, they could be implemented on top of the objects transferring the performance penalty to those applications using the APIs.

Eventually, the APIs will be provided as some sort of compatibility layer that exists only for the ability to run old applications. All new systems features will be provided exclusively though objects. Only those applications that have made the transition to using these systems objects will be able to benefit from the newest and more powerful features.

This chapter will introduce each specific feature of OLE 2.0 describing briefly how your application might take advantage of it (that is, profit from it) today. Through these features you begin to transform your application that will more readily take advantage of Windows' future evolution (that is, profit tomorrow).

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

There is a stick with the fish (or proverbial carrot), which is that you have to read this book. However, the final chapter of this book will take the idea of component objects to the extreme, evolving a few control DLLs used in this book's sample applications from supporting many random APIs into supporting clean object-oriented interfaces. This is the model for years to come. ***Preview Note: No promises on this chapter since it might be impractical or too involved for the book's schedule.***

Charles Darwin will then have plenty to say about the origin of a new species of incredibly sophisticated and powerful applications for Windows.

Windows Objects: The Component Object Model

By measures of new functionality I typically call OLE 2.0 a third of an operating system. It presents a number of key operating system features as system objects such as memory allocation, file management, and data transfer. By similar measures, and by staring down its intimidating documentation, OLE 2.0 can be very overwhelming. The first step in adopting these new and powerful technologies is to realize that one just does not learn and exploit a new operating system overnight–there are a few fundamental concepts to learn first. In addition, many higher-level features of any system generally builds on the lower-level features, and OLE 2.0 is no different. In fact, OLE 2.0 makes tremendous use of the idea as shown in Figure 1-1.

Figure 1-1: Each feature in OLE 2.0 builds on lower-level features.

First is the Component Object Model which is part specification (hence the Model) and part implementation contained in COMPOBJ.DLL. The specification comes in defining a binary standard for object implementation that is independent of the programming language you choose to use. Those objects adhering to this standard earn the right to be called Windows Objects. This binary standard enables two applications to communicate through object-oriented interfaces without requiring them to know anything about each others implementation. For example, you might choose to implement a Windows Object in C++ that supports an interface through which a user (code) of that object can learn names of functions that can be invoked on that object. The user of this object might be a programming environment like Visual Basic or might be another application written in C.

The implementation in the Component Object Model provides a small number of fundamental APIs that allow you to instantiate a Windows Object of a specific class providing a pointer through which you call that object's functions. This is a significant improvement over using mechanisms like SendMessage and PostMessage because you are not restricted to two parameters and you do not need a window in order to make the call. This is because the object called through your pointer may live not only in a DLL but also in another executable somewhere on your file system, opening the possibility for distributed object architectures under Windows. The Component Object Model APIs are also designed to handle the eventual capability for an object's code to also execute on a different machine in a different address, that is, full distributed objects. OLE 2.0 does not in itself provide this capability, but the OLE 2.0 APIs contain the necessary mechanisms.

A Windows Object does not always have to be structured such that it can be instantiated using the COMPOBJ.DLL APIs. Those APIs are just one way through which you can obtain the **first** pointer to an object. There are, of course, other APIs and routes in OLE 2.0 through which you can obtain that first pointer as well–many of the chapters in this book describe how you generally obtain and use a pointer to specific kinds of objects. When implementing an object, how you choose to allow others to get at your object impacts your overall code structure. For those you want addressable via COMPOBJ.DLL APIs you must 'house' your object inside either a DLL or EXE with specific code, that is, specific functions you call and export from your module. The object itself, however, can be independent of the housing as we will explore in Chapter 4.

The other key piece of implementation in COMPOBJ.DLL handles a process called marshaling, or passing function calls and parameters across process boundaries. Since an object's code may execute in another process space, and eventually execute on another machine, COMPOBJ.DLL handles translation of calling conventions and 16- to 32-bit parameter translation, when the object and that object's user are running in different process spaces. For example, an object may be executing in a 32-bit process space and so treats types like UINT as a 32-bit value. The user of that object may be running in a 16-bit process space and calls a function in the object passing a 16-bit UINT. COMPOBJ.DLL sits in the middle to marshal that UINT from a 16-bit world into a 32-bit world. Other types, like pointers, memory handles, etc., are handled in a similar manner: COMPOBJ makes sure that each side, object and user, sees the other in terms of their own process space. In the future when the object can execute on another machine, COMPOBJ.DLL will also account for considerations like byte-ordering.

This problem of marshaling is not new: OLE 1.0 had to move parameters and memory across process boundaries as well for which it used Dynamic Data Exchange (DDE). A major problem of OLE 1.0 resulting from the asynchronous DDE protocol was that a function call made on an object was inherently asynchronous, forcing the caller to sit and wait in a message loop until that function was complete, with all the associated problems of time-outs, error recovery, and blocking other requests on the same object. OLE 2.0's marshaling mechanism, Lightweight[1] Remote Procedure Calls (LRPC) is inherently synchronous–calls made on objects don't return until completed–simplifying the programming model. Some calls, however, do remain asynchronous due to the general uses of those calls.

Objects and Interfaces

A technique to describe objects such as the component object model does require some change in typical understanding of what an "object" really is. "Object" is probably the most overused and ambiguous term in the computer industry. They are mentioned everywhere but generally mean wildly different things; use of the term in this book is no exception. Chapter 3 will describe a Windows Object in detail, showing exactly how to implement one in both C and C++. Later chapters illustrate a number of routes by which you can obtain a pointer to a specific type of Windows Object. I will warn C++ programmers now that a Windows Object is a little different than a C++ object, although you can effectively use C++ objects to implement Windows Objects.

The new term that requires a little explanation here (in order to understand the rest of this chapter) is interface, another hackneyed and ambiguous term. The notion of interface that applies throughout this book is defined as "a set of semantically related functions implemented in an object." Interface by itself means definition (or prototype or signatures) for those functions: the OLE 2.0 include files contain these definitions. An instantiation of an interface is simply an array of pointers to functions; any code that has access to that array, that is, a pointer to the top of the array, can call the functions in that interface as shown in Figure 1-2. The interface definition allows that code to call functions by name and provides type checking on parameters instead of calling functions by an index into the array. Since it's generally inconvenient to draw function tables in expanded form for every interface, this book and other OLE 2.0 documentation shows each function table as a circle (or a jack) connected to the object, also as shown in Figure 1-2.

Figure 1-2: An instantiation of an interface is simply an array of function pointers with a more convenient representation of an interface function table.

A Windows object implements one or more interfaces, that is, provides pointers to instantiated function tables for each supported interface. Simple objects, like a data object we'll implement in Chapter 6, support only one specific interface describing data operations like GetData and SetData. More complex objects, like compound document objects, support at least three interfaces, perhaps more depending on the features that object implements. Overall, an object is completely described by the collection of interfaces it supports, as each separate interface provides the essential manipulation API to a user of that object.

Whenever the user of some object first obtains a pointer to that object, it only has a pointer to **one interface**; the user never obtains a pointer to the entire object. This pointer only allows the user to call the functions in that one interface's function table as illustrated in Figure 1-3. Through this pointer the user has no access to any data members of the object nor does it have any direct access to other interface. That is, data must be manipulated through the interface functions and the interface must have a function through which the caller can obtain a pointer to a different interface.

Figure 1-3: Object users with a pointer to interface A can only access member functions of interface A.

While OLE 2.0 does not define standard interface functions to access data members of the object, it does define a standard function through which the user of one interface on that object can obtain a pointer to another interface on that object. This function is called QueryInterface as shown in Figure 1-4. We'll

1Means "no network"

examine this function in detail in Chapter 3. When the user queries for another interface it will either receive back a NULL, meaning the object does not support the functionality described by the interface, or a valid pointer through which the user many then manipulate the object through that new interface. Since QueryInterface is so fundamental, it is part of an interface called IUnknown (The 'I' stands for Interface) that describes the group of fundamental functions that all Windows Objects support. All other interfaces in OLE 2.0 are derived from IUnknown and so all interfaces contain the QueryInterface function and by implementing one interface on a Windows object you automatically implement IUnknown since the first few functions in each function table will be those of IUnknown as shown in Figure 1-5.

Figure 1-4: An object user asks calls the QueryInterface member of any interface to retrieve pointers to other interfaces on the same object.

Figure 1-5: The first few members of any interface are always IUnknown members.

Through QueryInterface the user of an object to discover the capabilities of that object at run-time by asking for pointers to specific interfaces. By returning a pointer to that interface the object is contractually obliged to support the behavior implied by that interface. This enables every object to implement as many interfaces as it wants such that when it meets a user that knows how to use many of those interfaces the two can communicate on a high level. When the object meets an user with less knowledge, the two can still communicate through their common set of interfaces. The major point is that they can still communicate.

While OLE 2.0 defines a large number of standard interfaces you are absolutely free to define and publish your own custom interfaces without requiring any changes whatsoever to the OLE 2.0 DLLs or any other part of the Windows operating system. The only complication is that you must also provide a DLL for marshaling support since OLE 2.0's marshaling only knows its own interfaces. But that is a small price to pay for the ability to essentially publish your own new APIs without having to wait for any system revision from Microsoft.

Structured Storage and Compound Files

The OLE 2.0 specification defines a number of storage-related interfaces collectively called Structured Storage. These interfaces, which according to the meaning of 'interface' carry no implementation, describe a way to create a "file system within a file" and provides some extremely powerful features for applications. Instead of having one large contiguous sequence of bytes on the disk manipulated through a single file handle with a single seek pointer, Structured Storage describes how to treat a single file-system entity as a structured collection of two types of objects, storages and streams, that act like directories and files, respectively.

A stream object is the conceptual equivalent of a single disk file as we understand them today. They are the basic file system component in which data lives and each stream in itself has access rights and a single seek pointer. Streams are named using a text string (up to 31 characters in OLE 2.0) and can contain any internal structure you desire.

A storage object is the conceptual equivalent of a directory. Each storage, like a directory, may contain any number of storages (subdirectories) and any number of streams (files) shown in Figure 1-6. In turn, each sub-storage may itself contain any number of storages and streams ad infinitum until your disk is full.

A storage object does not contain any user defined data–just like an file-system directory cannot–as it only maintains information related to the storage structure, that is, what other streams and sub-storages live below it. Each storage has access rights as streams do, a feature lacking in MS-DOS directories. Given a storage you can ask it to enumerate, copy, move, rename, delete, or change times on elements within it, providing more than just the equivalents of MS-DOS commands.

As Structured Storage is only specification, OLE 2.0 provides a complete implementation called Compound Files, you can use as a new systems feature to replace traditional file-handle based API like _lread and _lwrite. Do not confuse the word 'compound' as used here to mean that Compound Files are only useful to

Figure 1-6: Conceptual structure of storage and stream objects in a compound file.

compound document implementations: Compound Files is a completely independent technology in the OLE 2.0 package and, in fact, lives independently in STORAGE.DLL. A similar and 100% compatible implementation of Compound Files will also become the native file system in future versions of Windows, and so such as basic technology cannot be restricted to high-level integration features like compound documents.

Since Compound Files isolates your application from the exact placement of bytes within a file, just like MS-DOS isolates applications from the exact sectors on the hard disk that your file occupies. MS-DOS presents disparate sectors as a contiguous byte array when you access that file through a file handle. In the same manner Compound Files presents information in a stream as one contiguous entity although the exact information in that stream may be discontinuous in the actual file itself. So even though the file itself will change, the exact data structures you write into each stream requires no changes. In fact, you can easily create a compound file that contains a single stream with the exact structure of your existing file format

Microsoft recognized that changing your on-disk file format may not be an option and so use of Compound Files is optional. The only type of application required to use some aspect of this storage model is a compound document container which must provide a storage object to any contained compound document object. However, you can create a storage object in memory and later write the contents of that memory into your own file format as you see fit as detailed in Chapter 5. Storage objects created on different storage devices, like memory and disk files, are indistinguishable from one another.

As for today, Compound Files provide a number of key features that you can use to make a more powerful application, perhaps adding additional features yourself that would otherwise be too difficult of time-consuming, such as transactioning and incremental saves. Chapter 5 will discuss all the features of Compound Files and demonstrate both simple and complex uses of this technology. All of the features can greatly affect or improve an application's design and treatment of storage.

Structured Storage as well as Compound Files were motivated by a number of factors, but one of the most important was to standardize the layout of pieces of information within a file. Such standardization enables any piece of code, be it the system shell or an application, to examine the structure of the entire compound file. The exact data formats of each individual stream is still private to whatever wrote that data, but anyone can look into a compound file and enumerate the storages and streams it contains. The OLE 2.0 toolkit even contains a tool called DFVIEW.EXE that displays the structure of any compound file and allows you to dump the hex data of any stream.

Further standardization of the contents of a few very specific streams (but by no means all streams) enables the system shell, and other applications, to allow end-users to search for occurrences of data within files that conform to attributes like creation date, author, keywords, and so on. Microsoft is determined to work with other ISVs to define standard names and structures for streams that contain information useful in such queries. The long-range goal is to have all information on the file system structured such that end users can browse the contents of many streams using the system shell. This capability is far more powerful yet easier to use than requiring the end user to first find a file, then find the application that can load that file, then use the application to open an browse files to eventually find the data. Structured storage enables shell-level document searching, an important manifestation of *Information At Your Fingertips*™.

Uniform Data Transfer and Notification

Built of top of both the component object model and compound files is a subset of OLE 2.0 called Uniform Data Transfer which provides the functionality to represent all data transfers–clipboard, drag-drop, DDE, and OLE–through a single piece of code called a data object. Such data objects are not limited to transferring data through global memory either as they may use other mediums like compound files. In general then a data source can choose the *best* method for data transportation, that is, the most efficient form of transport. End users benefit from better performance. Add that to direct streamlined capabilities like drag-drop and you have a more usable environment overall..

Up to now, all data transfer between an application and anything external (clipboard, DDE, OLE 1.0) has used global memory. The specific data format contained in that global memory was described using a clipboard format such as CF_TEXT or CF_BITMAP. Windows (not to mention programmers) has suffered immensely from inherent limitations of global memory transfers as well as from having radically different protocols and unrelated APIs for exchanging data via clipboard, DDE, and OLE 1.0.

OLE 2.0 makes two major improvements. First, it allows you to describe data using not only a clipboard format, but also a specification about how much detail the data contains, what device (printer primarily) it was rendered for, and what sort of medium is used to transfer the data. This new method of describing and exchanging data that we'll examine in Chapter 6 is much more powerful than anything previously available. Instead of just saying "I have a DIB" I can say "I have a thumbnail sketch of a DIB rendered for a 300dpi PostScript printer and it lives in a storage object." As a source of data I can choose the best possible medium in which I transfer data and make it the preferred format, providing other mediums as backups (such as global memory, the lowest common denominator). So if I happen to generate 30MB 24-bit DIBs, I can keep those in disk files or storage objects even through a data exchange. I don't have to load that entire DIB into memory just for such a transfer.

Data transfer in OLE 2.0, therefore, can use a compound file, disk file, global memory, or whatever medium is most preferable for data. Since data transfer works on top of compound files, you can see how this OLE 2.0 feature builds on an earlier feature, much in the way that today's clipboard takes advantage of Windows' Kernel memory allocation primitives.

Secondly, OLE 2.0 separates the means of setting up a data exchange, that is, the protocol, from the actual operation of exchanging data. The problem today is that the four transfer protocols (clipboard, DDE, OLE 1.0, drag-drop) use widely different functions as well as widely different data structures. Under OLE 2.0, applications use new APIs to transfer a **pointer** to a data object from the data source to the consumer of that data. These APIs form the protocol as discussed in Chapters 6-8. Once this pointer has been exchanged, the protocol disappears and all exchange of data happens through the data object. In other words, protocol worries about exchanging a data object; the data object standardizes how to exchange data rendered in some medium independent from the protocol. Since the data object does not know anything about protocols, you can write one piece of code to perform an operation like Paste regardless of how you obtained the data object, hence the Uniform in Uniform Data Transfer.

Notification

Consumers of data from an external source are generally interested when that data changes. OLE 2.0 handles notifications of this kind through an object called an advise sink, that is, a body that absorbs notifications from a source. The advise sink not only handles notifications for data changes but is also generally used to detect changes in another OLE object status, such as when it's saved, closed, or renamed. We'll first see advise sinks in Chapter 6 but see them again in Chapter 9 and beyond.

Data Objects and the Clipboard

Applications can first make use of data objects for Cut and Copy clipboard operations. As Chapter 7 shows, a data object is programmatically similar to common clipboard handling code. When your data object renders data you use the same functions that you used to generate a handle to pass to SetClipboardData. When your data object enumerates the formats it supports it does so in the same order as your clipboard code always has. In fact, a data object used for Copy and Cut operations can be implemented on top of whatever clipboard handling code you currently have, with some minor modifications, primarily to handle delayed rendering if you do not already.

Pasting data from the clipboard is matter of retrieving a data object that describes what data is currently on the clipboard. Instead of asking IsClipboardFormatAvailable, you ask such a data object if it can render a specific format for whatever device, content, and transfer media you wish, as specific as you want. If the data object can provide the data, you can, at any time, ask for a rendering through the object instead of GetClipboardData.

Data Objects and Drag-Drop

Converting an application to use data objects for clipboard transfers is not much of a benefit in itself. However, once you have the data object implemented for the clipboard, you can use that same exact implementation for drag-drop. OLE 2.0 does not deal with the simplistic drag-drop of files from File Manager: OLE 2.0 provides for *full* drag-drop of *any* data that you could transfer over the clipboard. Instead of being limited to files, or maybe just OLE objects, you can write your application to drag and drop any data that you can describe in a data object.

Think of drag-drop a streamlining your existing clipboard operations by eliminating menus, allowing direct manipulation, and providing dynamic feedback to the user about what data is being dragged and what might happen if the data is dropped. In this model the source of the drag provides the data object, determines what starts and stops the operation, and controls mouse cursor-related user interface. The target of a drag receives the data object, checks for usable formats, and determines what will happen with the data if it's dropped inside the target window.

Drag-drop is a tremendous user benefit, and if you implement a data object for the clipboard first then your drag-drop implementation is close to trivial as we'll see in Chapter 8: an implementation will not take more than a few days, depending on how fancy you want to get. For the simplest implementation of a drop source you can copy code straight from this book and have it working in under an hour. Targets are a little more complicated, but simple targets could be written in an afternoon. You won't find another feature this powerful and this easy to implement.

Data Objects and Compound Documents

Drag-drop is not the end; implementing linking and embedding involves augmenting the data object to handle OLE 2.0 formats to describe both linked and embedded objects. You will modify your data object code to enumerate and render a few new formats; most of the rendering can be delegated to functions already present in the OLE 2.0 SDK's sample code. We'll examine how data objects affect OLE in chapters 9, 10, and 11.

Once you have augmented the data object for OLE 2.0 formats you instantly enable transfers of OLE objects via clipboard and drag-drop since neither mechanism cares what the data object actually contains. In addition, by providing OLE 2.0 clipboard formats in a data object OLE automatically generates OLE 1.0 formats for backward compatibility with OLE 1.0 objects. You as a OLE 2.0 application get such backward compatibility for free by simple virtue of using a data object.

For those readers familiar with OLE 1.0 be aware that exchange of the 'native' data format is handled through a persistent storage interface separate from the data object. Where the data object interface handles exchange formats, the persistent storage interface provide the equivalent of a file into which an object stores its native structures.

Data Objects and DDE

OLE 2.0 itself doesn't attempt to address data transfers with DDE by use of data objects for reasons outlined in Chapter 6. It would be possible to design a protocol that you can use to isolate your application from DDE and treat it, again, with a data object just as you would treat any other data transfer. While outside the scope of this book, such a design it would allow us to come full circle with supporting the four data exchange mechanisms in Windows through data objects, keeping different protocols to retrieve a data object but treating that data object uniformly from that point onward.

Compound Documents: Object Embedding

The component object, compound file, and data transfer technologies comprise the bulk of OLE 2.0 that is not concerned with creating applications to support Compound Documents, or what is known as linking and embedding. Compound documents is now only a subset of the OLE 2.0 functionality[1] which builds on the lower-level technologies as illustrated previously in Figure 1-1. Compound Documents is first and foremost of standard for integration following the standards provided in the lower layers: the Component Object Model standardizes how an object and object user communicate; Compound Files standardize file structure; Uniform Data Transfer standardizes data exchange functions.

Compound Documents standardize how an object implementor (called a server) packages its objects such that any other arbitrary application (called a container) can integrate that object (data and a graphical presentation) into its own documents. Integration has always been possible when two applications know about each other, but Compound Documents enables integration without such knowledge. A container application is written to the compound document standard such that it can integrate any object that is also written to the standard. Neither container nor object require any specific information about each other because they communicate through standard interfaces that provide for editing, data transfer, and for storing the object's data into the container's document files. Custom interfaces are only necessary when the object and container do know about each other and want to communicate on a higher level, but the presence of these interfaces does not interfere with operation through standard interfaces because of the QueryInterface mechanism mentioned above in the "Objects and Interfaces" section. For most purposes, Compound Documents eliminates the need for custom interfaces altogether.

Chapter 9 will explore container applications that provide 'site' objects that describe places in which a compound document object can live. These site objects implement at least two interfaces one of which describes containment functions and another which provides functions through which the container is notified of events in the object. Much of the implementation of a container is user-interface, providing dialogs such as Paste Special, Insert Object, and the new OLE 2.0 Change Type feature. Fortunately various groups at Microsoft have contributed to writing a source code library of these dialogs as well as other user interface helper functions that should save you tremendous amounts of time implementing a container.

Chapters 10 and 11 will explore compound document objects and how to implement them in either a DLL or EXE. These chapters will only deal with embedded objects which store their private data structures in a storage object provided by the container. This storage object, which is usually some piece of a larger Compound File, is for the object's exclusive use. The object may create any kind of structure within that storage object, that is, as many streams and sub-storages as desired. When asked to save itself, the object writes into this storage which is essentially writing directly into the container's file. This means that the object is the only agent that has to access that storage and has the ability to only access as much as necessary. This is a stark contrast to OLE 1.0 compound documents where the container always had to load the entire object's data from a file and pass it to the object via global memory. Under OLE 2.0, containers only need to pass a pointer to the object and the object worries about accessing the actual data. The end result is a promise of much better performance over OLE 1.0.

1As a historical note, OLE 1.0 was concerned *only* with Compound Documents and provided no other technologies.

Compound Documents: Object Linking & Monikers

Enabling container and object applications for linking is a matter of dealing with an addition OLE 2.0 data format (describing a link source) and adding a few more interfaces to the objects that each application implements (site and document objects in the container). The addition of linking capabilities requires very few changes to other compound document code you implement to handle embedding. Chapter 13 deals with the necessary changes for an object server to support linking.

Linking affects containers more than in OLE 1.0. Containers are more than just consumers of linked objects–they can become link sources in themselves. OLE 2.0 provides the mechanisms by which a container can provide link source information for objects embedded within their documents. Within the same container, therefore, you can create an embedded object to which another object is linked. Chapter 14 deals with this and other advanced container features.

Linked objects have been a significant difficulty from the beginnings of OLE. Since the data linked objects lives in a separate file on the file system, links are easily broken when the end user manually changes the location of that file. OLE 1.0 depended on absolute pathnames to linked files, so any change in that files location broke the link, even when the relative paths between the container and the linked file remained the same. In addition, OLE 1.0 could only describe a nested object through one reference. To solve most of the link breakage problems as well as to provide for arbitrarily deep object nestings, OLE 2.0 introduces a type of object called a **moniker**.

A simple moniker contains some reference to a linked object and contains code that knows how to "bind" to that linked object. Binding is a process of launching the application that handles the class of the linked object, asking the application to load a file in which the object lives, then asking the application to resolve the name of the object down to an object pointer. OLE 2.0 provides implementations of simple 'file' and 'item' monikers that contain pathnames and object names for these purposes.

A file moniker contains both absolute and relative pathnames which OLE uses in relocating a linked file. If it fails to find the file using the absolute pathname, it tries the relative pathname. Failing that it attempts to locate the file in the current directory. Relative pathnames address the cases where an entire directory tree was moved which breaks absolute links but not relative links. Searching the current directory handles cases where the user copied all the linked files into the same directory as the compound document (such as on a floppy) which breaks both absolute and relative pathnames. The only case that is not handled is when the user moves a linked file to a completely new location. That cannot be solved until the operating system is aware of every such change. When that happens, the system update the link paths without the application knowing or caring.

Complex object references are described using composite monikers that are sequences of any other simple or composite moniker. Most links will generally be expressed in a composite of one file moniker and one item moniker, even links to embedded objects (the item) in a container document (the file). Longer sequences of monikers express more complex notions such as nested objects where the composite contains many items monikers. An illustration of a composite moniker that contains an file and an item moniker is shown in Figure 1-7. OLE 2.0 provides several other moniker types as well as the ability to implement your own. We'll see monikers more in Chapter 12.

Figure 1-7: Conceptual file, item, and composite monikers

Compound Documents: In-Place Activation

Compound documents describes a way to embed or link an object into some container document. The container provides the appropriate functionality to activate the object which may invoke any number of actions on the object. When the action implies editing, both linked and embedded objects are opened in another window that provides the editing context, the same model as OLE 1.0 supported.

To stress a document-centric view of computing, OLE 2.0 provides the ability to *activate* an object in-place, inside the container application's window. Editing the object is a subset of the more generic term 'activating.' Instead of the object opening another window to execute some action it may choose to provide editing tools or other controls in the context of the container. The end user benefits from never having to leave the document context in which they are working–no distractions of other windows and other environments. Instead of seeing two copies of the data, one in the container and one in a separate editing window, the end user sees only the one in the container, in the full context of it's containing document.

Both objects and containers need require a number of additional interfaces in order to support in-place activation, but these build on top and use much of the compound document implementations that you'll already have done by this time. We'll discuss in-place objects in Chapter 16 followed by in-place containers in Chapter 17. I strongly encourage you to think about implications of in-place activation while you are working on other features, but the in-place feature is not the right place to start since it builds so much on other functionality. Chapter 15 more fully describes some of the implications and considerations for in-place design that you should be aware of before diving into in-place implementation.

By implementing in-place activation interfaces you do not restrict your container or object to being useful to only other in-place applications. The presence of the in-place activation interfaces does not interfere in any way with the more basic compound document interfaces. So if you implement an in-place object, you are useful to an in-place container, which can activate you in-place, as well as a simple container which will always activate you in a separate window.

One of the major implications of in-place activation has to do with application design in general which is a special topic treated in Chapter 18. Here I'll demonstrate how you can build your own application out of in-place objects–your application can look exactly how it does today, but the process of breaking the application up into component objects makes those objects usable from other applications as well. You should not consider the information in this chapter important for implementing OLE 2.0 features but is a very interesting exercise of the possibilities that in-place activation enables.

Automation

Finally, there is one other key technology that is part of the OLE 2.0 system that is completely separate from the rest: OLE Automation. This technology allows an object, any object regardless of other features, to expose a set of commands and functions that some other piece of code can invoke upon it. Each command can take any number of parameters and automation provides the methods through which the objects describes the names and types of those parameters. This book will not explore this technology in excruciating detail but will demonstrate a few techniques you might find useful.

The intent of automation is to enable the creation of system macro programming tools. Such tools will ask automation-enabled objects for lists of their function names and lists of parameters (names and types) that those functions accept. At one point Microsoft was considering a single system macro programming tool but this would mean one language and one tool for all end-users. Through Automation, OLE 2.0 allows objects to describe their capabilities to any tool where the tool defines the programming environment. Ultimately this gives the end-user the choice of language, vendor, functionality, etc.

Automation objects generally describe user-level functions on the order of File Open or Format Character; automation tools display those functions to the user allowing the user to write macro scripts that span applications. From the programmer's perspective, you can either implement an automation object on top of your usual application message procedure, or you can implement your message procedure on top of an automation object. Looking forward to the evolution of Windows, the latter technique will likely become the norm.

The major motivation for this mechanism is to pave the way for programming tools that can affect any automation-enabled object, regardless of whether the system or an application implements that object. When the user chooses an object, such a tool asks the object for its list of function names and exposes them to the user as what operations are possible on that object. When the user selects a function they would like to use, the programming tool can further ask the object for the names and types of that function's parameters and provide the environment in which the end user indicates what to pass in each parameter.

With a system full of such objects from many applications, the end user can use any programming tool that understands Automation to write macros that *could* span applications. A more immediate benefit to your specific application is that such a tool can also write macros that operate only on your application, eliminating the need for you to create your own specific macro language. The end user is the free to choose their preferred programming tool, any of which use your automation interfaces in the same way. The user gains the benefit of having one tool that works with all automation applications; you benefit from exposing automation once and letting someone else provide the programming environment.

But it doesn't stop there. While automation exposes commands through which external agents invoke your functions, there is not reason whatsoever that you could not invoke those same commands on yourself. You might implement automation on top of your application's message procedure, or you might choose to implement your message procedure on top of automation. Looking ahead, centralizing such code in the automation interface enables the eventual elimination of message procedures, using instead a more general-purpose and powerful command processing object.

Using this Book

This book is intended for programmers familiar with programming in the Windows environment, either under Windows 3.1 or Windows NT. Only in special circumstances will I explain details about certain Windows API since our concentration is OLE 2.0. I do not expect that you know anything about OLE version 1.0, but certainly a familiarity with the concepts can be helpful. Be forewarned that OLE 2.0 is much more than OLE 1.0: do not allow yourself to believe that OLE 2.0 is just OLE 1.0 with the added feature of in-place activation.

I assume that you at least know C and if you are at least familiar with C++, all the better. Since I wrote the sample code in this book is with C++, I have provided a short explanation of C++ concepts and terminology in Chapter 2, written specifically for a C programmer. The code in this book is actually my first serious work in C++, so while I take advantage of some components of the language, I do not use it to such an extent that a C programmer will feel lost. After all, I have to understand it myself. Obviously you will be able to digest the OLE 2.0 sample code easier if you do not need to struggle with C++ at the same time. I have tried to keep the really strange C++ out of the samples.

With this book I do not intend to replace the OLE 2.0 Programmer's Reference, mostly for the reason that the Reference provides all the exact function prototypes, describes every last data structure, and provides information on some very specialized topics that I will not discuss in detail in this book. When appropriate, that is, when the topic is too esoteric for most readers, I will refer you to the Programmer's Reference. In addition, I do not necessarily assume that you have read any part of the Programmer's Reference–what I do assume is that somehow or another you have the motivation to incorporate OLE 2.0 technologies into your application.

You will notice that this book follows a different course than the OLE 2.0 Programmer's Reference and for good reason. The encyclopedic parts of the Reference (Parts 1 and 2) are oriented towards developers who want to create compound document applications that are in-place enabled whereas Part 3 is your treasury of function names and parameter descriptions. I encourage you to use Parts 1 and 2 for a second opinion or as an alternate explanation to that I present here, for I intend to show what you can do with all the separate features and technologies of OLE 2.0, with or without compound documents. For some time now Microsoft has been plugging OLE 2.0 as the proverbial 'thing' to do without ever qualifying how to do it beyond the scope of compound documents and in-place activation...so I've often been asked by developers what they can do with the technology when their application doesn't fit the compound document mold. In this book I hope to answer those questions and qualify just what exactly is possible with what is to be the at the core of the once and future Windows.

Road Map

Pat and Casey both decided to build cabins at the top of a mountain. Both cabins would have the latest siding, a Swedish finish floor, and a pressure-treated deck with a great view of the valley.

Pat was so excited about having this cabin that she quickly threw some wood and tools into a helicopter and went straight to the summit. "Time is Money," Pat philosophized, so she started hammering away. Soon she needed another tool and more materials and so rushed back down the mountain, grabbed what she needed, and hurried back up. This process repeated itself again and again and again–Pat never had the right things on hand to complete the job efficiently although on every flight back up the mountain she thought she did. But progress was impressive.

Meanwhile Casey had not started so quickly. She carefully planned an approach to the construction, organized all the materials she would need and arranged for them to be delivered just before they were necessary. There would be nothing she did not have on hand to complete each stage in the project.

When Casey eventually arrived at the summit she only had enough to build the foundation, but it went perfectly. Pat would often peer over and laugh, touting how much she had accomplished faster than Casey. "Time is Money!" she would shout. Casey would quietly think "If time is money, then why are you spending as much time *going up and down this mountain than you are building?"*

It seemed that Pat would complete the project far before Casey. Pat had an insatiable urge to keep building something, and so was finishing the floor and building parts of the deck with only half the walls and roof complete. Casey only had a foundation, the frame, and the roof completed on her cabin.

Unexpectedly powerful monsoons fell upon the two builders. Pat watched in horror as the incomplete walls and roof were torn away in the strong winds. She could only stand there helpless and get drenched, watching her beautiful Swedish finish floor warp and split under the intense pounding. Casey, keeping perfectly dry with a solid roof overhead, continued working through the rains, adding wallboards so they would withstand the wind and completing a magnificent interior, all the time staying warm and dry.

When the monsoons subsided and the sun returned, Pat cleaned up the wreckage and salvaged what she could. Months later, Pat eventually completed the cabin–it was finished, but it certainly wasn't what she had imagined. She was just glad that it was finally done.

Casey had only to spend a few more weeks on the deck and finished the cabin. While Pat was painfully recovering what she could, Casey was enjoying a wonderful spring in the mountains.

Catastrophes often occur in software development. A competitor releases a product sooner than expected or has more features than you knew about. If your work doesn't stand up to that competition it has to be scrapped or completely reworked.

The approach taken in this book will help you incorporate features of OLE 2.0 into your applications such that at a number of points you can stop completely but still have a lot to show for your efforts. For example, if you incorporate compound files early on, you will benefit from structured storage, incremental saves, and transactioning. Those may be significant features to show to your customers, enough that further work may not even be necessary.

If you go further to incorporate data objects you can gain significant benefit by changing large data transfers from global memory to a storage object. That in itself, again, may be utterly important for your customer base, and again, it would be a reasonable place to stop. From there you could add drag-drop, very visual and powerful, and stop again. Then move on to embedding, linking, and achieve the summit of in-place activation. Each step, again, takes advantage of all previous steps, and after each step you can choose to stop and ship.

Granted, not everyone has the luxury of time to address every OLE 2.0 feature. At a minimum, many applications, especially those porting from OLE 1.0, will desire to be either a container for compound document objects or a server of compound document objects. With either goal you can read a subset of the chapters in this book as shown in Figure 1-8. The chapters in the rectangles indicate the required reading for obtaining OLE 1.0 style linking and embedding with the other blocks indicating optional features.

In addition, at the beginning of Chapters 4 and beyond you will see a map of the specific sections you need to read in that chapter depending on your goals, identifying exactly why you must read those sections. The applications that many readers will want to implement use most of OLE 2.0's features, but exactly how much you need to use those features is what these maps will attempt to explain.

Of course, if you are using OLE 2.0 to implement features other than compound documents, you'll only need to read sections in a few earlier chapters, perhaps going straight from Chapter 4 to Chapter 19. There is also an exploration of component object implementations and the evolution of applications and DLLs from an API base to an object base in Chapter 20.

Also, I've made in-place activation an option on this list because if your time constraints do not allow it you can get by without it. I encourage you to make in-place activation and drag-drop your goals, but following the approach on the road map can keep you dry when the monsoons come.

Figure 1-8: Chapters to read depending on your goals.