Chapter 4
Component Windows Objects
(The Component Object Model)

Arthur: Camelot!
Galahad: Camelot...
Launcelot: Camelot...
Gawain: It's only a model...
Arthur: Sh!

From "Monty Python and the Holy Grail"[1]

Just about everyone who has ever tried to present all the material in OLE 2.0 in a comprehensible way (myself included) has tried to bring out the Component Object Model as some "feature" of OLE 2.0. Because "The Component Object Model" itself basically confuses people, this chapter is about using and implementing very general Windows Objects that involve the base APIs and interfaces specified in the Component Object Model.

The Component Object Model is part specification and part implementation. The specification is mostly about interfaces, reference counting, and QueryInterface, that is, the basic standardization of a Windows Object. The implementation is a number of fundamental APIs that provide for object creation and code management as well as code that handles marshaling of interface function calls across process boundaries. I refer to this implementation, contained in COMPOBJ.DLL, as the "component object library."

This implementation provides one answer to the ultimate question posed in Chapter 3 in the form of a "Component Object." A component object is a Windows Object identified with a CLSID that exists in any DLL or EXE on your file system.[2] To obtain a pointer to a component object the object's user passes that CLSID to one of two component object library APIs. The library in turn locates and loads the code implementing that object, instantiates the object, and asks the object for an interface pointer to return to the object's user. Note that a compound document object are just special cases of the more general component object, so the discussion here is relevant if you are interested in implementing a compound document server.

Before diving into using and implementing Component Objects, we must first discuss a few requirements of all applications (EXEs) that either use or implement objects. Applications (which define a task) must initialize the component object library before using any other OLE 2.0 APIs (from any OLE 2.0 DLL) and part of this initialization has to do with memory management within the application's task. Since both operations are crucial and used in all remaining sample applications in this book initialization and memory management will be the first two topics of this chapter.

The user of a Component Object, which we can call a Component User, then only uses the library to obtain that first pointer to an object identified with a class ID. The overall impact on such a component user is minimal as this chapter will demonstrate–the user need not be concerned about where the code for the object is actually located or how the object is implemented. The greater impact is on the implementation of a Component Object that allows the library to locate, load, and instantiate it based on a class ID. To accommodate such a capability you must implement a standard structure around the object, one structure for EXEs, another structure for DLLs. In addition, you must store information in the registration database under your object's class ID that identifies the name of your object and where it lives. The component object library APIs use this information to find you and get you connected to your object user.

This chapter will demonstrate a simple object implemented in both a DLL and EXE as well as a user of those objects. We will also implement Schmoo's Polyline object as a Component Object in a DLL that we'll take forward in following chapters as we add more OLE 2.0 features.

This chapter closes with a discussion about object reusability through a mechanism called aggregation. One object, called the aggregate, internally creates instances of other objects, possibly exposing the interfaces of those objects as an interface on the aggregate. Aggregation accomplishes code reuse like C++ inheritance but without the problems of inheritance. While the topic is appropriate to discuss here, we won't actually put it to use until later chapters. You may, however, find it a useful mechanism around which to architect code reuse in your own application.

So to explain what the Component Object Model is, we really want to analyze the model's impact on

1Copyright 1974, Python (Monty) Pictures, Ltd., 20 Fitzroy Square, London, W.1.
2Again, in the future when OLE 2.0 becomes network enabled these may live and execute on a different machine.

applications in general, on the user of a component object, and on the implementation of a component object. The specifications of the Component Object Model provide the foundation of how Windows will evolve from an API-based system to an object-oriented system, a very romantic walk through the lush gardens of Camelot. But that could be another whole topic in itself, so we'll stick to implementation details; it is, after all, only a model.

Where the Wild Things Are

There are component users and component objects, both of which can reside in any piece of code, EXE or DLL alike. The system features provided in OLE 2.0 are themselves both objects and users, all of which live in DLLs. The implementation portion of the component object library is considered part of the OLE 2.0 system features.

Whether an object user lives in an EXE or DLL is of little importance–EXEs have a little more work as described in the next section since they define a task. In any case, disregarding where an object user lives we can illustrate the relationship between object and user as shown in Figure 4-1. Note that the word "server" in this figure applies to the module that services an object, either a DLL or EXE.

Figure 4-1: Component Users (in DLLs or EXEs) see other Component Objects either in DLLs or other EXEs. The component object libraries live between the user and an EXE object to provide marshaling. There is no middleman between the user an a DLL object.

Because an EXE object requires marshaling, performance is typically slower than using a DLL object. However, a 16-bit DLL object cannot currently be loaded into a 32-bit user space, nor can a 32-bit DLL be loaded into a 16-bit user space. Herein lies the advantage to EXE objects in that the marshaling layer takes care of the 16-32 bit thunking.

The component object library is the agent responsible for getting at the first object in any series of communication between object and user. It worries making sure the right piece of object code is in memory whenever something else wants to use that object. However, once the initial object has been created and is handed to the user, the object and user may create other objects themselves and pass them to their partner–the component object library is only there for marshaling (if even necessary) and is otherwise out of the picture. The objects we implement in this chapter require a certain structure and registration such that any user can instantiate that object through the component object library APIs. Since these APIs are used underneath many of the compound document APIs (prefixed with Ole, such as OleCreate) all compound document objects are also objects that fit this model–they just have a more defined behavior. However, the component object library is highly useful for creating component software, and Windows itself is headed in this direction, not to be a compound document system, but to be a component object system.

Compound Document Terminology

A number of the terms generally used in discussions of compound documents will be used in this chapter to discuss much more generic concepts. Therefore the list below provides the crucial compound document terms and how they apply to the information in this chapter:

Container	A container is a user of compound document objects which are special cases of component objects. A container exposes site objects to the linked or embedded objects they contain, but those site objects are not separately addressable components and are only passed to the contained object at run-time.
Server (sometimes just 'Object')	A server is an implementor of an object, either a DLL or EXE. A server may implement a component object or a compound document object and expose both through identical structures usable by the component object model. Servers may use any other component object to implement themselves.
In-Proc Server or DLL Server	A server of objects specifically implemented in a DLL.
Server Application or EXE Server	A server of objects specifically implemented in an EXE.
Object Handler	A lightweight DLL server containing a partial implementation of an object in an EXE. Handlers are not

expected to implement full objects (especially not editing capabilities) and are intended for redistribution. Structurally they are identical to DLL servers.

The New Application for Windows Objects

Any and all applications that plan to **use or implement** Windows Objects must insure that the component object library is properly initialized before attempting to use other OLE 2.0 APIs. In addition, applications that intend to use objects implemented in other applications must also make special considerations for LRPC's use of PostMessage. Note that any object or object user in a DLL need not be concerned with any of these requirements since they are only necessary from the application that defines a task:

1. Call the Windows API SetMessageQueue(96) to set your application's message queue size to 96. This is the recommended size for LRPC handling.
2. Verify the library build version by calling CoBuildVersion or OleBuildVersion.
3. Call CoInitialize or OleInitialize on startup.
4. Call CoUninitialze or OleUninitialize when shutting down to free DLL objects.

NOTE: CoBuildVersion, CoInitialize, and CoUninitialize have counterparts with Ole prefixes: OleBuildVersion, OleInitialize, and OleUnintialize. The Co* functions control your access to component object model functions. If you use any data transfer, drag-drop, or compound document related APIs functions, you must use the Ole* functions listed above instead of their Co* counterparts. The Ole* versions simply perform a few specific operations and call the Co* versions. OLE 2.0 applications, containers included, *always* use the Ole* versions.

Absolutely all of the sample applications in this book that compile EXEs include these four steps. Most of the samples in this chapter as well as Chapters 5 and 6 use the Co* variants. All samples in Chapters 7 and beyond will use the Ole* functions as those samples will depend on other Ole* functions as well. The first sample in the next section, "Memory Management and Allocator Objects," will demonstrate each of these steps. In the meantime, let's look at each step in detail and why they are necessary.

Enlarge the Message Queue

OLE 2.0's Lightweight Remote Procedure Call implementation works on top of the Windows API PostMessage. In a nutshell, when the user of an object in another application calls one of the object's member functions, it generates an 'LRPC call' which in actuality is a PostMessage from the one application process space into the other. In order to handle all the possible PostMessage traffic, Microsoft recommends that all OLE 2.0 applications that have even the slightest chance of engaging in LRPC calls call SetMessageQueue with 96 on startup. This should, in fact, be your very first call inside WinMain to insure that no messages yet exist in your queue since SetMessageQueue will destroy anything there already:

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
  , LPSTR pszCmdLine, int nCmdShow)
  {
  [variables, but NO code]

  //Recommended for all OLE 2.0 applications.
  SetMessageQueue(96);

  [Initialization code, message loop, etc]

  }
```

Failure to enlarge your message queue sufficiently could cause the component object library to fail or reject some LRPC calls. Internally the library attempts to queue up pending LRPC calls if the target application is slow to respond, but it works better for those calls to be stored in your message queue instead.

Verify The Library Build Version

Before using any other Component Object API (Co*) an application should call **CoBuildVersion(void)** to get major and minor build numbers in a returned DWORD. If you are planning to go on to use OLE 2.0's data transfer or compound document features you must call **OleBuildVersion(void)** instead which returns a similar DWORD. The high-order word of the return value is the major version number and the low-order word is the minor version number.

The application can run one major version and any minor version. The version of your libraries to check against are defined as the symbols **rmm** (major) and **rup** (minor) in OLE2VER.H. Note that these version

numbers will not be 2 and 0 for OLE 2.0 because they are build numbers and not product release numbers. The application must check for equality on the major version only and fail loading if the expected and actual values differ. In addition, OLE2VER.H defines a symbol "rmj" which looks like what should be the major build number, but in fact this is not used. The slightly confusing names of rmm and rup for major and minor are unfortunate, but we have to deal with them nonetheless:

```
#include <compobj.h>    //For Ole* functions, use OLE2.H
#include <ole2ver.h>

...

DWORD    dwVer;

dwVer=CoBuildVersion();    //Or OleBuildVersion

if (rmm==HIWORD(dwVer))
   {
   //Major versions match.

   if (rup <= LOWORD(dwVer))
      {
      //Library is newer or as old as me, use normally.
      }
   else
      {
      /*
       * I was written for newer libraries:  disable features that depend
       * on APIs or bug fixes in newer libraries, or simply fail altogether.
       */
      }
   }
else
   //Major version mismatch, fail loading application.
```

Minor version numbers are useful to applications that want to know if the libraries they've loaded contain a particular function or have a specific bug fix. Let's say the minor version 12 of OLE 2.0 added a function that improves our performance over minor version 11. If I load the minor version 11 libraries, I cannot attempt to call that functions. If, however, I find that I am running against minor version 12, I can take advantage of what's available.

Call CoInitialize or OleInitialize

On startup an application must call **CoInitialize** or **OleInitialize** before calling any other function in their respective libraries. OleInitialize must be used for any feature above compound files, that is, any data transfer (including drag-drop) and any compound document functionality. component object library and compound file APIs can be used after only CoInitialize.

```
if (FAILED(CoInitialize(NULL)))    //Or OleInitialize
   //Fail loading the application

m_fInitialized=TRUE;
```

Both functions identically take a pointer to an allocator object that supports the IMalloc interface. Through this object all other parts of this application and DLLs that live in this application's task can allocate task memory (as opposed to shared memory). If NULL is passed as shown above then OLE uses a default allocator. Any code in this application or in a DLL loaded into this task may call the **CoGetMalloc** API to retrieve an IMalloc pointer to this same allocator. We'll see this in more detail in the next section below called "Memory Management and Allocator Objects."

Any code within the same task can call CoInitialize multiple times; in such circumstances the IMalloc passed to the first CoInitialize wins. This allows any code (usually that in a DLL) to insure that it can use the component object model library whether or not the main application called it. When CoInitialize is called more than once in the same task, it will return an HRESULT with S_FALSE–a code that does not mean failure but means 'nothing happened.' As we have seen, the FAILED() macro will return FALSE for S_FALSE just as it will for S_OK, so the code fragment above is valid for all uses of CoInitialize.

An application must remember whether or not CoInitialize or OleInitialize worked so it knows whether or not to call CoUninitialize or OleUninitialize when it shuts down. In other words, every Uninitialize call must be matched one-to-one with and Initialize call.

Call CoUninitialize or OleUninitialize

When an application is done with the libraries it must call CoUninitialize if it had previously called CoInitialize, or OleUninitialize if it has previously called OleInitialize. Neither function takes any parameters. You should remember whether or not the Initialize call succeeded and only call Uninitialize if so, that is, balance the calls as you would GlobalAlloc and GlobalFree.

Internally, OleUnintialize cleans up the specifics from OleInitialize and calls CoUninitialize. This latter function will call another **CoFreeAllLibraries** which forcibly unloads all object DLLs that were loaded on behalf of the application, regardless of reference or lock counts. You may have use for this function yourself if your application's debugging version has the ability to suddenly terminate and unload (say on an assert failure) which might not normally call CoUninitialize.

Memory Management and Allocator Objects

Up to now, the only system-supported memory management functions have been the various Local* and Global* Windows APIs (LocalAlloc, LocalFree, GlobalAlloc, GlobalFree, etc.). OLE 2.0 introduces a new object-oriented technique to deal with memory management through use of an **allocator objects.** Within any given task, that is, a process space in which is running a single EXE, there is a task allocator object and a shared allocator object. The application may implement the task allocator, or it may use the default task allocator implemented in the component object library. The shared allocator is not replacable-the implementation in the component object library is always used to insure that memory is truly sharable.

So how do you specify the task allocator? The answer is the only parameter to the **CoInitialize** and **OleInitialize** functions. Both take a pointer to an allocator object that defines how memory is managed within the task where a NULL means "use the default task allocator" and a non-NULL pointer points to application implementation of a task allocator. Since only applications call the Initialize functions, the application defines task allocations.

An allocator object implements the IMalloc interface, defined in COMPOBJ.H (IUnknown members have been removed for brevity; you will see them explicitly in the include file), so CoGetMalloc always provides an LPMALLOC pointer (far pointer to IMalloc) as an out-parameter. The IMalloc interface describes most of the same functions that Windows provides for local and global memory, such as LocalAlloc, LocalFree, and LocalCompact. For specific details for each member function, see the OLE 2.0 Programmer's Reference, but it's fairly obvious to guess at how to use each function in this interface on their signatures alone:

```
DECLARE_INTERFACE_(IMalloc, IUnknown)
  {
  STDMETHOD_(void FAR*, Alloc) (THIS_ ULONG cb) PURE;
  STDMETHOD_(void FAR*, Realloc) (THIS_ void FAR* pv, ULONG cb) PURE;
  STDMETHOD_(void, Free) (THIS_ void FAR* pv) PURE;
  STDMETHOD_(ULONG, GetSize) (THIS_ void FAR* pv) PURE;
  STDMETHOD_(int, DidAlloc) (THIS_ void FAR* pv) PURE;
  STDMETHOD_(void, HeapMinimize) (THIS) PURE;
  };

typedef     IMalloc FAR* LPMALLOC;
```

At any time any piece of code in the application or any DLL loaded into this task (including the OLE 2.0 libraries) can and will call **CoGetMalloc** to obtain an IMalloc pointer on the task allocator object, that is, a task allocator is a Windows Object and you use the API CoGetMalloc to obtain the first interface pointer. In this case, your application may be the object implementor and OLE 2.0 may be the object user. It works both ways!

All the OLE 2.0 libraries always use the task allocator for all non-shared memory needs. Some OLE 2.0 functions, like StringFromCLSID (in COMPOBJ.DLL), return a pointer to the caller that was allocated from this task memory. The caller must free that pointer when it's no longer necessary using the allocator object from CoGetMalloc.

The only parameter to CoGetMalloc is either MEMCTX_TASK or MEMCTX_SHARED to identify which allocator object you want. Again, the task allocator is always determined by your application's call to Co/OleInitialize whereas OLE 2.0 always provides the shared allocator.

The default task allocator is based on multiple-local heap management (or far local heaps). This allocator allows you to allocate more than 64K–since there are multiple heaps–but each allocation is as efficient as a local allocation since you only use one selector per heap instead of one per allocation (as GlobalAlloc). The only limitation is that any single allocation must be smaller than 64K. The shared allocator provides memory that different processes can independently access, such as global memory provides under Windows 3.1, but uses the same efficiency of multiple local heaps.

The Malloc program (CHAP04\MALLOC) shown in Listing 4-1 exercises the functions in both the standard task allocator (it does not implement its own allocator) and the shared allocator. It is an application that is most interesting in a debugger but does indicate success or failure of its operations in message boxes (yes, a most advanced user interface!).

MALLOC.CPP

```
/*
 * MALLOC.CPP
 *
 * Demostration of IMalloc object use.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include <windows.h>
#include <ole2.h>
#include <initguid.h>
#include <ole2ver.h>
#include "malloc.h"


int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   MSG         msg;
   LPAPPVARS   pAV;

   //Recommended for all OLE 2.0 applications.
   SetMessageQueue(96);

   //Create and initialize the application.
   pAV=new CAppVars(hInst, hInstPrev, nCmdShow);

   if (NULL==pAV)
      return -1;

   if (pAV->FInit())
      {
      while (GetMessage(&msg, NULL, 0,0 ))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      }

   delete pAV;
   return msg.wParam;
   }
```

```
LRESULT FAR PASCAL __export MallocWndProc(HWND hWnd, UINT iMsg
   , WPARAM wParam, LPARAM lParam)
   {
   LPAPPVARS      pAV;
   LPVOID         pv;
   ULONG          cb;
   UINT           i;
   BOOL           fResult=TRUE;
   HRESULT        hr;

   //This will be valid for all messages except WM_NCCREATE
   pAV=(LPAPPVARS)GetWindowLong(hWnd, MALLOCWL_STRUCTURE);

   switch (iMsg)
      {
      case WM_NCCREATE:
         //CreateWindow passed pAV to us.
         pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);

         SetWindowLong(hWnd, MALLOCWL_STRUCTURE, (LONG)pAV);
         return (DefWindowProc(hWnd, iMsg, wParam, lParam));

      case WM_DESTROY:
         PostQuitMessage(0);
```

Listing 4-1: The MALLOC program that exercises task and shared allocator objects.

```
         break;

      case WM_COMMAND:
         switch (LOWORD(wParam))
            {
            case IDM_IMALLOCCOGETMALLOCTASK:
               pAV->FreeAllocations(TRUE);

               hr=CoGetMalloc(MEMCTX_TASK, &pAV->m_pIMalloc);
               fResult=SUCCEEDED(hr);

               MessageBox(hWnd, ((fResult) ? "CoGetMalloc(task) succeeded."
                  : "CoGetMalloc(task) failed."), "Malloc", MB_OK);

               break;


            case IDM_IMALLOCCOGETMALLOCSHARED:
               pAV->FreeAllocations(TRUE);
```

```
            hr=CoGetMalloc(MEMCTX_SHARED, &pAV->m_pIMalloc);
            fResult=SUCCEEDED(hr);

            MessageBox(hWnd, ((fResult) ? "CoGetMalloc(shared) succeeded."
               : "CoGetMalloc(shared) failed."), "Malloc", MB_OK);
            break;


         case IDM_IMALLOCRELEASE:
            pAV->FreeAllocations(TRUE);
            break;


         case IDM_IMALLOCALLOC:
            if (NULL==pAV->m_pIMalloc)
               break;

            pAV->FreeAllocations(FALSE);

            for (i=0; i < CALLOCS; i++)
               {
               LPBYTE    pb;
               ULONG     iByte;

               cb=pAV->m_rgcb[i];
               pAV->m_rgpv[i]=pAV->m_pIMalloc->Alloc(cb);

               //Fill the memory with letters.
               pb=(LPBYTE)pAV->m_rgpv[i];

               if (NULL!=pb)
                  {
                  for (iByte=0; iByte < cb; iByte++)
                     *pb++=('a'+i);
                  }

               fResult &= (NULL!=pAV->m_rgpv[i]);
               }

            MessageBox(hWnd, ((fResult) ? "IMalloc::Alloc succeeded."
               : "IMalloc::Alloc failed."), "Malloc", MB_OK);
            break;


         case IDM_IMALLOCFREE:
            pAV->FreeAllocations(FALSE);
```

```
        MessageBox(hWnd, "IMalloc::Free finished.", "Malloc", MB_OK);
        break;


    case IDM_IMALLOCREALLOC:
        if (NULL==pAV->m_pIMalloc)
            break;

        for (i=0; i < CALLOCS; i++)
            {
            LPBYTE      pb;
            ULONG       iByte;

            pAV->m_rgcb[i]+=128;

            //Old memory is not freed is Realloc fails here.
            pv=pAV->m_pIMalloc->Realloc(pAV->m_rgpv[i]
                , pAV->m_rgcb[i]);

            if (NULL!=pv)
                {
                pAV->m_rgpv[i]=pv;

                //Fill the new memory with something we can see.
                pb=(LPBYTE)pAV->m_rgpv[i];
                cb=pAV->m_rgcb[i];

                if (NULL!=pb)
                    {
                    for (iByte=cb-128; iByte < cb; iByte++)
                        *pb++=('a'+i);
                    }
                }
            else
                fResult=FALSE;
            }

        MessageBox(hWnd, ((fResult) ? "IMalloc::Realloc succeeded."
            : "IMalloc::Realloc failed."), "Malloc", MB_OK);

        break;


    case IDM_IMALLOCGETSIZE:
        if (NULL==pAV->m_pIMalloc)
```

```
            break;

        for (i=0; i < CALLOCS; i++)
            {
            cb=pAV->m_pIMalloc->GetSize(pAV->m_rgpv[i]);

            //We test that the size is *at least* what we wanted.
            fResult &= (pAV->m_rgcb[i] <= cb);
            }

        MessageBox(hWnd, ((fResult) ? "IMalloc::GetSize matched."
            : "IMalloc::GetSize mismatch."), "Malloc", MB_OK);

        break;


    case IDM_IMALLOCDIDALLOC:
        if (NULL==pAV->m_pIMalloc)
            break;

        /*
         * DidAlloc may return -1 if it does not know whether
         * or not it actually allocated something.  In that
         * case we just blindly & in a -1 with no affect.
         */
        for (i=0; i < CALLOCS; i++)
            fResult &= pAV->m_pIMalloc->DidAlloc(pAV->m_rgpv[i]);

        MessageBox(hWnd, ((fResult) ? "IMalloc::DidAlloc is TRUE."
            : "IMalloc::DidAlloc is FALSE."), "Malloc", MB_OK);

        break;


    case IDM_IMALLOCHEAPMINIMIZE:
        if (NULL!=pAV->m_pIMalloc)
            pAV->m_pIMalloc->HeapMinimize();

        MessageBox(hWnd, "IMalloc::HeapMinimize finished."
            , "Malloc", MB_OK);

        break;


    case IDM_IMALLOCEXIT:
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
```

```
                break;
            }
        break;

    default:
        return (DefWindowProc(hWnd, iMsg, wParam, lParam));
    }

return 0L;
}


CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev, UINT nCmdShow)
    {
    UINT        i;
    ULONG       cb;

    m_hInst       =hInst;
    m_hInstPrev   =hInstPrev;
    m_nCmdShow    =nCmdShow;

    m_hWnd        =NULL;
    m_pIMalloc    =NULL;
    m_fInitialized=FALSE;

    //100 is arbitrary--IMalloc's can handle in allocs from 0 to 65535 bytes
    cb=100;

    for (i=0; i < CALLOCS; i++)
        {
        m_rgcb[i]=cb;
        m_rgpv[i]=NULL;
        cb*=2;
        }

    return;
    }


CAppVars::~CAppVars(void)
    {
    FreeAllocations(TRUE);

    if (m_fInitialized)
        CoUninitialize();
```

```
    return;
    }


BOOL CAppVars::FInit(void)
    {
    WNDCLASS    wc;
    DWORD       dwVer;

    //Make sure COMPOBJ.DLL is the right version
    dwVer=CoBuildVersion();

    if (rmm!=HIWORD(dwVer))
        return FALSE;

    //Call CoInitialize so we can call other Co* functions
    if (FAILED(CoInitialize(NULL)))
        return FALSE;

    m_fInitialized=TRUE;

    if (!m_hInstPrev)
        {
        wc.style         = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc   = MallocWndProc;
        wc.cbClsExtra    = 0;
        wc.cbWndExtra    = CBWNDEXTRA;
        wc.hInstance     = m_hInst;
        wc.hIcon         = LoadIcon(m_hInst, "Icon");
        wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName  = "MALLOC";

        if (!RegisterClass(&wc))
            return FALSE;
        }


    m_hWnd=CreateWindow("MALLOC", "IMalloc Object Demo"
        , WS_OVERLAPPEDWINDOW, 35, 35, 350, 250, NULL, NULL, m_hInst, this);

    if (NULL==m_hWnd)
        return FALSE;

    ShowWindow(m_hWnd, m_nCmdShow);
```

```
   UpdateWindow(m_hWnd);

   return TRUE;
   }



void CAppVars::FreeAllocations(BOOL fRelease)
   {
   UINT    i;

   if (NULL==m_pIMalloc)
      return;

   for (i=0; i < CALLOCS; i++)
      {
      if (NULL!=m_rgpv[i])
         m_pIMalloc->Free(m_rgpv[i]);

      m_rgpv[i]=NULL;
      }

   if (fRelease)
      {
      m_pIMalloc->Release();
      m_pIMalloc=NULL;
      }

   return;
   }
```

MALLOC.H

```
/*
 * MALLOC.H
 *
 * Definitions for a demonstration of IMalloc.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */



#ifndef _MALLOC_H_
```

```
#define _MALLOC_H_


//Menu Resource ID and Commands
#define IDR_MENU                    1


#define IDM_IMALLOCCOGETMALLOCTASK      100
#define IDM_IMALLOCCOGETMALLOCSHARED    101
#define IDM_IMALLOCRELEASE              102
#define IDM_IMALLOCALLOC                103
#define IDM_IMALLOCFREE                 104
#define IDM_IMALLOCREALLOC              105
#define IDM_IMALLOCGETSIZE              106
#define IDM_IMALLOCDIDALLOC             107
#define IDM_IMALLOCHEAPMINIMIZE         108
#define IDM_IMALLOCEXIT                 109


//MALLOC.CPP
LRESULT FAR PASCAL __export MallocWndProc(HWND, UINT, WPARAM,
LPARAM);


#define CALLOCS 10

/*
 * Application-defined classes and types.
 */

class __far CAppVars
   {
   friend LRESULT FAR PASCAL __export MallocWndProc(HWND, UINT, WPARAM,
LPARAM);

   protected:
      HINSTANCE       m_hInst;          //WinMain parameters
      HINSTANCE       m_hInstPrev;
      UINT            m_nCmdShow;

      HWND            m_hWnd;           //Main window handle
      LPMALLOC        m_pIMalloc;       //IMalloc interface we have.
      BOOL            m_fInitialized;   //Did CoInitialize work?

      ULONG           m_rgcb[CALLOCS];  //Array of sizes to allocate
      LPVOID          m_rgpv[CALLOCS];  //Array of allocated pointers
```

```
  public:
    CAppVars(HINSTANCE, HINSTANCE, UINT);
    ~CAppVars(void);
    BOOL FInit(void);

    void FreeAllocations(BOOL);
  };


typedef CAppVars FAR * LPAPPVARS;


#define CBWNDEXTRA              sizeof(LONG)
#define MALLOCWL_STRUCTURE      0


#endif //_MALLOC_H_
```

Using the Heapwalker application in the Windows SDK we can see where OLE 2.0 allocates each type of memory, task and shared, using its own allocator objects. As shown in Figure 4-2, task memory is allocated from multiple heaps belonging to the Malloc application task. When this memory is freed, the heaps are not necessarily freed themselves, but the space inside those heaps are freed, as shown in Figure 4-3.

Figure 4-2: Local Walk on a the heap shows allocated blocks. The blocks are filled with letters to show their location in a hex dump.

Figure 4-3: Local Walk after freeing the memory shows free space in the heaps. Freed blocks are filled with 0xCC.

Shared memory is allocated on behalf of COMPOBJ.DLL as shown in Figure 4-4, again using the same heap management technique as the standard task allocator. Freeing memory generates the same results as shown in Figure 4-3 for the task allocator.

Figure 4-4: Local Walk on a the heap shows allocated blocks exactly like the task allocator, but owned by COMPOBJ.DLL instead of the application.

Component Objects from Class IDs: A Component User

Let's suppose I'm an application and I know there exists a Windows Object called "Koala" which is, in fact, what we'll implement in the next section "Implementing a Component Object and Server." I can identify the Koala object with the CLSID of "00021102-0000-0000-C000-000000000046" (Whoa! That's a long name there! Remember that CLSIDs are 128 bits: this string is the hexadecimal representation of those bits). Let's say I also know that the Koala object supports the interface called IPersist which is a very simple interface only capable of returning the class ID of its object. Given this knowledge, how do I create a Koala object

with this class ID and obtain an pointer to its IPersist interface?

This question should ring a harmonic with the ultimate question posed in Chapter 3, so let's look at the answer. For the benefit of those readers writing component document container applications I want to mention that the APIs and interface functions we use to instantiate a component object are used within more complex APIs that we'll see in Chapter 9 to instantiate a compound document object. Again, compound document objects are more refined and specialized component objects; what we discuss here is simply the logical equivalent of the C++ *new* operator. If you are in a hurry to implement a compound document container, you can skip to the next chapter after finishing this section.

The OBJUSER program in Listing 4-2 implements a component user of the Koala component objects that we'll implement in the next section. Koalas implement the IPersist interface only, but by virtue of implementing one interface they also implement IUnknown: IPersist includes all IUnknown member functions plus one other called GetClassID that returns, what else, the CLSID of the object. IPersist is only used in OLE 2.0 as a base class for the IPersistStorage, IPersistStream, and IPersistFile interfaces that we'll see in Chapter 5. My reasons for choosing it here over a more interesting custom interface is that IPersist, being a standard interface, has built-in marshaling support so we can implement it in a DLL or application object identically with little extra effort.

OBJUSER.CPP

```
/*
 * OBJUSER.CPP
 *
 * A user of the Koala objects.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#define INITGUIDS
#include <windows.h>
#include <ole2.h>
#include <ole2ver.h>
#include "objuser.h"


int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   MSG        msg;
   LPAPPVARS   pAV;

   /*
    * This is necessary to support LRPC when using EXE objects.  THe
    * default message queue of 8 is a tad too small for some typical
    * operations.
    */
   SetMessageQueue(96);

   //Create and initialize the application.
   pAV=new CAppVars(hInst, hInstPrev, nCmdShow);

   if (NULL==pAV)
      return -1;

   if (pAV->FInit())
      {
      while (GetMessage(&msg, NULL, 0,0 ))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
```

```
    }

  delete pAV;
  return msg.wParam;
  }


LRESULT FAR PASCAL __export ObjectUserWndProc(HWND hWnd, UINT iMsg
  , WPARAM wParam, LPARAM lParam)
  {
  HRESULT        hr;
  LPAPPVARS      pAV;
  CLSID          clsID;
  LPCLASSFACTORY  pIClassFactory;
  DWORD          dwClsCtx;

  //This will be valid for all messages except WM_NCCREATE
  pAV=(LPAPPVARS)GetWindowLong(hWnd, OBJUSERWL_STRUCTURE);

  switch (iMsg)
    {
    case WM_NCCREATE:
      //CreateWindow passed pAV to us.
      pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);
```

Listing 4-2: The OBJUSER program that uses Koala Objects.

```
      SetWindowLong(hWnd, OBJUSERWL_STRUCTURE, (LONG)pAV);

      return (DefWindowProc(hWnd, iMsg, wParam, lParam));

    case WM_DESTROY:
      PostQuitMessage(0);
      break;

    case WM_COMMAND:
      switch (LOWORD(wParam))
        {
        case IDM_OBJECTUSEDLL:
          pAV->m_fEXE=FALSE;
          CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEDLL, MF_CHECKED);
          CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEEXE,
MF_UNCHECKED);
          break;


        case IDM_OBJECTUSEEXE:
```

```
            pAV->m_fEXE=TRUE;
            CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEDLL,
MF_UNCHECKED);
            CheckMenuItem(GetMenu(hWnd), IDM_OBJECTUSEEXE, MF_CHECKED);
            break;


        case IDM_OBJECTCREATECOGCO:
            if (NULL!=pAV->m_pIPersist)
                {
                pAV->m_pIPersist->Release();
                CoFreeUnusedLibraries();
                }

            dwClsCtx=(pAV->m_fEXE) ? CLSCTX_LOCAL_SERVER :
                CLSCTX_INPROC_SERVER;

            hr=CoGetClassObject(CLSID_Koala, dwClsCtx, NULL
                , IID_IClassFactory, (LPVOID FAR *)&pIClassFactory);

            if (SUCCEEDED(hr))
                {
                //Create the Koala asking for IID_IPersist
                pIClassFactory->CreateInstance(NULL
                    , IID_IPersist, (LPVOID FAR *)&pAV->m_pIPersist);

                //We're done with the class factory, so release it.
                pIClassFactory->Release();
                }

            break;


        case IDM_OBJECTCREATECOCI:
            if (NULL!=pAV->m_pIPersist)
                {
                pAV->m_pIPersist->Release();
                CoFreeUnusedLibraries();
                }

            //Simpler creation:  use CoCreateInstance
            dwClsCtx=(pAV->m_fEXE) ? CLSCTX_LOCAL_SERVER :
                CLSCTX_INPROC_SERVER;

            CoCreateInstance(CLSID_Koala, NULL, dwClsCtx
                , IID_IPersist, (LPVOID FAR *)&pAV->m_pIPersist);
```

```
        break;


    case IDM_OBJECTRELEASE:
        if (NULL==pAV->m_pIPersist)
            break;

        pAV->m_pIPersist->Release();
        pAV->m_pIPersist=NULL;

        CoFreeUnusedLibraries();
        break;


    case IDM_OBJECTGETCLASSID:
        if (NULL==pAV->m_pIPersist)
            break;

        hr=pAV->m_pIPersist->GetClassID(&clsID);

        if (SUCCEEDED(hr))
            {
            LPSTR       psz;
            LPMALLOC    pIMalloc;

            //String from CLSID uses task Malloc
            StringFromCLSID(clsID, &psz);
            MessageBox(hWnd, psz, "Object Class ID", MB_OK);

            CoGetMalloc(MEMCTX_TASK, &pIMalloc);
            pIMalloc->Free(psz);
            pIMalloc->Release();
            }
        else
            {
            MessageBox(hWnd, "IPersist::GetClassID call failed"
                , "Koala Demo", MB_OK);
            }

        break;

    case IDM_OBJECTEXIT:
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        break;
    }
```

```
            break;

        default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
        }

    return 0L;
    }



CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev, UINT nCmdShow)
    {
    m_hInst       =hInst;
    m_hInstPrev   =hInstPrev;
    m_nCmdShow    =nCmdShow;

    m_hWnd        =NULL;
    m_fEXE        =FALSE;

    m_pIPersist   =NULL;
    m_fInitialized=FALSE;
    return;
    }



CAppVars::~CAppVars(void)
    {
    if (NULL!=m_pIPersist)
        m_pIPersist->Release();

    if (IsWindow(m_hWnd))
        DestroyWindow(m_hWnd);

    if (m_fInitialized)
        CoUninitialize();

    return;
    }



BOOL CAppVars::FInit(void)
    {
```

```
WNDCLASS    wc;
DWORD       dwVer;

dwVer=CoBuildVersion();

if (rmm!=HIWORD(dwVer))
   return FALSE;

if (FAILED(CoInitialize(NULL)))
   return FALSE;

m_fInitialized=TRUE;

if (!m_hInstPrev)
   {
   wc.style         = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc    = ObjectUserWndProc;
   wc.cbClsExtra     = 0;
   wc.cbWndExtra     = CBWNDEXTRA;
   wc.hInstance      = m_hInst;
   wc.hIcon          = LoadIcon(m_hInst, "Icon");
   wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
   wc.lpszClassName  = "OBJUSER";

   if (!RegisterClass(&wc))
      return FALSE;
   }

m_hWnd=CreateWindow("OBJUSER", "Koala Component Object Demo"
   , WS_OVERLAPPEDWINDOW,35, 35, 350, 250, NULL, NULL, m_hInst, this);

if (NULL==m_hWnd)
   return FALSE;

ShowWindow(m_hWnd, m_nCmdShow);
UpdateWindow(m_hWnd);

CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTUSEDLL, MF_CHECKED);
CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTUSEEXE, MF_UNCHECKED);

return TRUE;
}
```

OBJUSER.H

```
/*
 * OBJUSER.H
 *
 * Definitions for an user of the Koala objects.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#ifndef _OBJUSER_H_
#define _OBJUSER_H_


#include <bookguid.h>


//Menu Resource ID and Commands
#define IDR_MENU                1


#define IDM_OBJECTUSEDLL        100
#define IDM_OBJECTUSEEXE        101
#define IDM_OBJECTCREATECOGCO   102
#define IDM_OBJECTCREATECOCI    103
#define IDM_OBJECTRELEASE       104
#define IDM_OBJECTGETCLASSID    105
#define IDM_OBJECTEXIT          106


//OBJUSER.CPP
LRESULT FAR PASCAL __export ObjectUserWndProc(HWND, UINT, WPARAM,
LPARAM);


/*
 * Application-defined classes and types.
 */

class __far CAppVars
    {
    friend LRESULT FAR PASCAL __export ObjectUserWndProc(HWND, UINT,
WPARAM, LPARAM);

    protected:
```

```
    HINSTANCE      m_hInst;          //WinMain parameters
    HINSTANCE      m_hInstPrev;
    UINT           m_nCmdShow;

    HWND           m_hWnd;           //Main window handle
    BOOL           m_fEXE;           //For tracking menu selection.

    LPPERSIST      m_pIPersist;      //IPersist interface we have.
    BOOL           m_fInitialized;   //Did CoInitialize work?

  public:
    CAppVars(HINSTANCE, HINSTANCE, UINT);
    ~CAppVars(void);
    BOOL FInit(void);
  };


typedef CAppVars FAR * LPAPPVARS;


#define CBWNDEXTRA              sizeof(LONG)
#define OBJUSERWL_STRUCTURE     0


#endif //_OBJUSER_H_
```

OBJUSER's only interesting output is a message box that shows the CLSID retrieved from the object when you make a call to IPersist::GetClassID. Otherwise you should step through this program in a debugger to really understand what is happening.[1] In any case, the first two items on the Koala Object menu control whether you use the object implemented in an application or a DLL. Either way, the rest of the functions remain the same. You can choose one of two ways to instantiate an object, you can ::Release the object, and call IPersist::GetClassID which displays the CLSID as a string in a message box.

Note that in order to run OBJUSER you must have compiled versions of both object implementations registered in the registration database. You can create the necessary entries by merging the file CHAP04\CHAP04.REG with REGEDIT.EXE. CHAP04.REG does not contain paths to the modules containing the object implementations. You must update the paths in REGEDIT such that the OBJUSER can file the DLL and EXE.

For OBJUSER and any other component user, there are three steps to instantiate and manage a component object (Note that OBJUSER also performs the four steps outlined in "The New Application for Windows Objects" since it's an EXE and defines a task):

1. #include <initguid> in one source file of the compilation to create a code segment containing CLSIDs and IIDs.
2. Create an object based on a CLSID using one of two routes:
    a. If you only need one object call **CoCreateInstance** with the CLSID and the IID of the interface you desire on the object.
    b. If you need more than one object, call **CoGetClassObject** to obtain an class factory (an IClassFactory pointer) for the CLSID and call IClassFactory::CreateInstance as much as you want with the IID of the interface

1***Preview Note: I might be inclined to add more message boxes to indicate success or failure of the API calls to make it a little more friendly.***

you desire on the object. Call IClassFactory::Release when finished.

3. Use the object through the interface pointer, ::Release it through that pointer when finished, and call **CoFreeUnusedLibraries**.

The first step only affects your build environment and compilation but does not really matter to programming. The second step is the real meat of our discussion as it shows exactly how to instantiate a component object. The third step deals with how you manage and free the object through your interface pointer.

#include <initguid.h> and Precompiled Headers

Anything that ever references any GUID, be it a CLSID or IID must #include the file *initguid.h* once, and only once, in the entire compilation of your module. This includes all component users and all objects (component objects or not) and means that you should *#include <initguid.h>* in one, and only one, file of your application. Including the file insures that all standard GUIDs are defined and that all defined GUIDs, including your own, end up in a discardable code segment instead of your data segment which is preferred since defined GUIDs are always constant. INITGUID.H also allows you to use the DEFINE_GUID or DEFINE_OLEGUID macros for defining your own IIDs and CLSIDs as shown in the BOOKGUID.H file in the INC directory.

If you have typically use a central include file for all files in your project, wrap an *#ifdef* statement around the *#include*. The samples have such a statement in the shared BOOKGUID.H file (in the INC directory):

```
#ifdef INITGUIDS
#include <initguid.h>
#endif
```

Only one file in each sample project #defines INITGUIDS. Note that there is a similar symbol, INITGUID used in COMPOBJ.H for similar purposes. However, you cannot use this symbol itself because COMPOBJ.H later will not pull in another necessary include file (COGUID.H), that is, you will not compile.

Including INITGUID.H only once is a trick when using precompiled headers, as will be popular when we start including all the lengthy OLE 2.0 header files. Create the precompiled header on a file that does not include INITGUID.H–the samples using precompilation all use a file precomp.cpp that contains only one #include statement. The precompiled header from this step can then be used to compile with all files **except** the one in which you **want** to include INITGUID.H. You should compile that single file without using the precompiled header pulling in the extra file.

Instantiate a Component Object

The component object library provides two fundamental object creation functions: CoCreateInstance, and CoGetClassObject combined with IClassFactory::CreateInstance. Which API you use depends on how many objects you need at a given time.

NOTE: OLE 2.0 Container applications do not directly use the CoCreateInstance or CoGetClassObject functions to create compound document objects. Instead, they use functions such as **OleCreate** that internally use CoCreateInstance as discussed in Chapter 9. As a container implementor, you should still understand how compound document objects are created through these mechanisms.

To create a single object given a CLSID, use CoCreateInstance which internally uses CoGetClassObject as described a little later. The code below to demonstrate this call is adapted from OBJUSER, modifying the symbols and their locations for ease of explanation:

```
HRESULT    hr;
DWORD      dwClsCtx;
LPPERSIST  pIPersist;
LPUNKNOWN  punkOuter=NULL;

//This controls where the object lives based on a meny selection.
dwClsCtx=(fEXE) ? CLSCTX_LOCAL_SERVER : CLSCTX_INPROC_SERVER;

CoCreateInstance(CLSID_Koala, punkOuter, dwClsCtx
   , IID_IPersist, (LPVOID FAR *)&pIPersist);
```

First note the naming of pointers to interfaces as shown with LPPERSIST and LPUNKNOWN. OLE 2.0 follows a convention where a far pointer to an interface, that is of type IInterface FAR *, is typed as LPINTERFACE where the interface name sans 'I' is appended in all-caps to an LP. So LPUNKNOWN is an IUnknown FAR * and LPPERSIST is IPersist FAR *.

CoCreateInstance takes five parameters, the names of which vary with the object class and interfaces you are using in your own implementation. Those shown here are similar to those in the OBJUSER program:

CLSID_Koala REFIID: A reference (a real C++ reference) to the class ID of the object you wish to create. In this example we are creating a "Koala" object implemented in "Implementing a Component Object and Server" section below. Note that in C, since there is no concept of a reference, you must precede this value with the & operation, that is, &CLSID_Koala.

punkOuter LPUNKNOWN: A pointer to the controlling unknown if the object is being created as part of an aggregate. See the section titled "Object Reusability" for more information.

dwClsCtx DWORD: Flags indicating the context in which the object is allowed to run which can be any combination of CLSCTX_LOCAL_SERVER (object in .EXE), CLSCTX_INPROC_SERVER (object in DLL) or CLSCTX_INPROC_HANDLER (compound document object handler DLL). OBJUSER chooses to run either a DLL or EXE based object depending on a menu choice.

IID_Persist REFIID: A reference to the interface ID you wish to obtain for this object. If the object does not support this interface, CoCreateInstance will fail. Note that as with the class ID, C programs must again prepend the & operator to an IID.

&pIPersist LPVOID FAR *: A pointer to the location in which CoCreateInstance is to store the interface pointer on return. If CoCreateInstance fails, the contents of this variable will be NULL on return.

Note that when more than one CLSCTX_* flag is specified, the libraries will attempt to load them in the order CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, then CLSCTX_LOCAL_SERVER, that is, always look for a DLL first (for better performance) only trying another EXE as a last resort.

Internally CoCreateInstance executes a three-step process to create the new object which can be written in pseudo-code as:

```
BEGIN
   Obtain a class factory (IClassFactory) for the desired class.
   Call IClassFactory::CreateInstance to create the object.
   Call IClassFactory::Release to free the class factory.
END
```

The first step, obtaining a class factory (also called a class object) is the exact function of the other relevant API, CoGetClassObject. A class factory is an object that implements the IClassFactory interface as defined in COMPOBJ.H:

```
DECLARE_INTERFACE_(IClassFactory, IUnknown)
  {
  [IUnknown methods included]

  //IClassFactory methods
  STDMETHOD(CreateInstance) (THIS_ IUnknown FAR* pUnkOuter, REFIID riid,
    LPVOID FAR* ppvObject) PURE;
  STDMETHOD(LockServer) (THIS_ BOOL fLock) PURE;
  };

typedef     IClassFactory FAR* LPCLASSFACTORY;
```

For cases where you only want to create one object of this class, CoCreateInstance suffices. However, if you want to create more than one object at a time, call CoGetClassObject to retrieve a class factory for the class, call IClassFactory::CreateInstance as many times as necessary, and call IClassFactory::Release when you are through. The code below shows an equivalent implementation to the previous code above but using CoGetClassObject instead:

```
HRESULT         hr;
DWORD           dwClsCtx;
LPPERSIST       pIPersist;
LPUNKNOWN       punkOuter=NULL;
LPCLASSFACTORY  pIClassFactory;

dwClsCtx=(fEXE) ? CLSCTX_LOCAL_SERVER :
  CLSCTX_INPROC_SERVER;

hr=CoGetClassObject(CLSID_Koala, dwClsCtx, NULL
  , IID_IClassFactory, (LPVOID FAR *)&pIClassFactory);

if (SUCCEEDED(hr))
  {
  //Create the Koala asking for IID_IPersist
  pIClassFactory->CreateInstance(punkOuter
    , IID_IPersist, (LPVOID FAR *)&pIPersist);

  //We're done with the class factory, so release it.
  pIClassFactory->Release();
  }
```

This code is almost the exact implementation of CoCreateInstance inside the component object library: the parameters you pass to CoCreateInstance are simply passed to CoGetClassObject and IClassFactory::CreateInstance. The extra parameters to CoGetClassObject are a NULL (a reserved LPVOID that should always be NULL), the interface ID you desire on the class object (always IID_IClassFactory in OLE 2.0 but could have more options in the future), and a location in which to store the pointer to the class object.

Remember that since CoGetClassObject is a function that creates a new interface pointer (reference counting rule #1 in "Reference Counting" in Chapter 3), you are responsible to ::Release that pointer when you are finished which is shown in the code above.

NOTE: If you want to hold the class factory object for a longer period of time, you must call IClassFactory::LockServer(TRUE). A reference count on a class factory does not guarantee that the server will stay in memory and that you could use the class factory at a later time. For the reasons, read the section titled "Provide an Unloading Mechanism" underneath "Implementing a Component Object and Server" below. In short, if a class factory reference could be used to keep a server in memory, then it can only shut down when that reference goes to zero. In the case of an EXE server, that reference can only go to zero if the server is being unloaded. Catch-22. Therefore you must use ::LockServer(TRUE) when you hold on to a class factory and ::LockServer(FALSE) **after** you release. Besides, if the server is locked, retrieving another class factory from it is cheap.

Manage the Object and Call CoFreeUnusedLibraries

What you do with an object once you have obtained an interface pointer is entirely dependent on the object itself and really what most of the chapters in the book are about. You must, in any case, be absolutely sure to call ::Release through that interface pointer once you are done with the object. Otherwise you doom the object to live immortal in memory for all eternity until the universe collapses (power off or the jolly three-finger reset).

Releasing the object is not the only consideration, however. When you initially instantiate an object

implemented in a DLL, COMPOBJ.DLL loads that DLL into memory using a function CoLoadLibrary. When the DLL is no longer needed, COMPOBJ.DLL calls CoFreeLibrary. Both functions map to LoadLibrary and FreeLibrary under Windows but are named differently for portability to other platforms such as the Apple MacIntosh.

However, the component object library does not know when an object in a DLL is destroyed because once it has facilitated loading and instantiating that object, communication between the object and its user is direct, completely bypassing everything in OLE 2.0. Therefore a DLL may remain loaded in memory even when no objects exist for that code to service. Over time, many DLLs may be loaded and chew up valuable memory. The component object library needs a cue to free those DLLs that are no longer needed. This is very much like discardable global memory where memory allocated and freed will stay in memory until discarded, even if no one is using that memory.

For this reason an all object users should periodically call CoFreeUnusedLibraries, primarily after you just ::Release an object for good. In this function, COMPOBJ.DLL can ask each DLL loaded in your task if it can be unloaded. If the DLL answers yes, then CoFreeLibrary is called on it to free the memory the DLL occupies. CoFreeUnusedLibraries does not affect other EXEs that are servicing objects, because those EXEs unload themselves when they are no longer servicing any objects. CoFreeUnusedLibraries is something like calling GlobalCompact(-1) which will purge memory of all unreferenced discardable memory segments.

NOTE: The OLE 2.0 implementation of CoFreeUnusedLibraries does nothing. However, your user code should still call the function after destroying an object so when the function is implemented you will work correctly.

A user can control whether or not the server code stays in memory even when that server has no outstanding objects through a second function in IClassFactory called ::LockServer. For various reasons, simply calling IClassFactory::AddRef does not guarantee that the code implementing that class factory actually stays in memory. Instead, ::LockServer(TRUE) increments a lock count in the server and ::LockServer(FALSE) decrements that count. When the server has a positive lock count, the DLL will not allow itself to be unloaded and an EXE will prevent itself from closing automatically.

Implementing a Component Object and Server[1]

Let's now implement the simple Koala component object with the IPersist interface as shown in the source code in Listing 4-3. The reason Koala implements the IPersist interface is because that interface has standard OLE 2.0-provided marshaling support meaning we can place this object in a DLL or EXE as we'll actually demonstrate. In real-world use, IPersist is never implemented by itself as it always serves as a base class for a few other interfaces.

KOALA.CPP

```
/*
 * KOALA.CPP
 * Koala Object version 1.00
 *
 * Implementation of the CKoala and CImpIPersist objects that works
 * in either an EXE or DLL.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 *
 */
```

Listing 4-3: Implementation of the Koala object structured to live in either DLL or EXE.

1Note again that "Server" here means a generic object server an not a compound document server. The latter has much stronger notions of 'windows' and 'documents' and provide storage, data transfer, and editing capabilities on their objects. These types of features will not be discussed in this chapter but instead introduced gradually over the rest of the chapters in this book.

```
#include "koala.h"


CKoala::CKoala(LPUNKNOWN punkOuter, LPFNDESTROYED pfnDestroy)
   {
   m_cRef=0;
   m_punkOuter=punkOuter;
   m_pfnDestroy=pfnDestroy;

   //NULL any contained interfaces initially.
   m_pIPersist=NULL;

   return;
   }


CKoala::~CKoala(void)
   {
   //Free contained interfaces.
   if (NULL!=m_pIPersist)
      delete m_pIPersist;     //Interface does not free itself.

   return;
   }


BOOL CKoala::FInit(void)
   {
   LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;

   if (NULL!=m_punkOuter)
      pIUnknown=m_punkOuter;

   //Allocate contained interfaces.
   m_pIPersist=new CImpIPersist(this, pIUnknown);

   return (NULL!=m_pIPersist);
   }



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

   /*
```

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

   /*
    * For IPersist we return our contained interface.  For EXEs we
    * have to return our interface for IPersistStorage as well since
    * OLE 2.0 doesn't support IPersist implementations by themselves
    * (assumed only to be a base class).  If a user asked for an
    * IPersistStorage and used it, they would crash--but this is
    * a demo, not a real object.
    */
   if (IsEqualIID(riid, IID_IPersist) || IsEqualIID(riid, IID_IPersistStorage))
      *ppv=(LPVOID)m_pIPersist;

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

   return ResultFromScode(E_NOINTERFACE);
   }


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


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

   cRefT=--m_cRef;

   if (0==m_cRef)
      {
      /*
       * Tell the housing that an object is going away so it can
       * shut down if appropriate.
```

```
        */
       if (NULL!=m_pfnDestroy)
          (*m_pfnDestroy)();

       delete this;
       }

   return cRefT;
   }



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


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



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


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

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

```
STDMETHODIMP CImpIPersist::GetClassID(LPCLSID pClsID)
   {
   *pClsID=CLSID_Koala;
   return NOERROR;
   }
```

KOALA.H

```
/*
 * KOALA.H
 *
 * Classes that implement the Koala object independent of whether
 * we live in a DLL or EXE.
 *
 * Copyright (c)1993 Microsoft Corporation, All Right Reserved
 */


#ifndef _KOALA_H_
#define _KOALA_H_


#include <windows.h>
#include <ole2.h>       //ole2.h has IPersist, compobj.h doesn't

//This defines all GUIDs used in this book so we can track them easily.
#include <bookguid.h>


//Type for an object-destroyed callback
typedef void (FAR PASCAL *LPFNDESTROYED)(void);


/*
 * The Koala object is implemented in its own class with its own
 * IUnknown to support aggregation.  It contains one CImpIPersist
 * object that we use to implement the externally exposed interfaces.
 */

class __far CKoala : public IUnknown
   {
   //Make any contained interfaces your friends so they can get at you
```

```
   friend class CImpIPersist;

   protected:
      ULONG          m_cRef;        //Object reference count.
      LPUNKNOWN      m_punkOuter;   //Controlling Unknown for aggregation

      LPFNDESTROYED  m_pfnDestroy;  //Function to call on closure.
      LPPERSIST      m_pIPersist;   //Contained interface implemetation

   public:
      CKoala(LPUNKNOWN, LPFNDESTROYED);
      ~CKoala(void);

      BOOL FInit(void);

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

typedef CKoala FAR * LPCKoala;


/*
 * Interface implementations for the CKoala object.
 */

class __far CImpIPersist : public IPersist
   {
   private:
      ULONG          m_cRef;        //Interface reference count.
      LPVOID         m_pObj;        //Back pointer to the object.
      LPUNKNOWN      m_punkOuter;   //Controlling unknown for delegation

   public:
      CImpIPersist(LPVOID, LPUNKNOWN);
      ~CImpIPersist(void);

      //IUnknown members that delegate to m_punkOuter.
      STDMETHODIMP        QueryInterface(REFIID, LPVOID FAR *);
      STDMETHODIMP_(ULONG) AddRef(void);
      STDMETHODIMP_(ULONG) Release(void);

      //IPersist members
      STDMETHODIMP        GetClassID(LPCLSID);
```

```
    };

typedef CImpIPersist FAR * LPIMPIPERSIST;

#endif //_KOALA_H_
```

The Koala object is implemented to support aggregation and to be identically usable in either an EXE or DLL server, but not without some impact. In addition, remember that we can call CoGetMalloc at any time to obtain access to shared or task memory, although the implementation of Koala shown here do not have occasion to use this feature.

To support aggregation, as we'll see in "Object Reusability", the object must be aware of a **controlling unknown**, if there is one, that is cognizant of all interfaces supported by the aggregate object. The object itself, implemented using the CKoala C++ class,[1] implements IUnknown but contains the **interface implementation** of IPersist in another C++ object called CImpIPersist. In aggregation, the interface implementations must always delegate all their IUnknown calls to the object that controls that interface's lifetime. When we're not aggregated and punkOuter passed to CKoala::CKoala is NULL, then CKoala passes its own IUnknown implementation to CImpIPersist instead. When the Koala object is aggregated, CKoala will receive a non-NULL punkOuter which it passes to CImpIPersist. In either case, the IPersist interface implementation will always delegate its IUnknown calls to a full object, only performing trivial reference counting on the interface for debugging purposes. If this seems confusing, defer aggregation until later.

The only interesting function of CImpIPersist is ::GetClassID, which just returns the CLSID defined the Koala object. However, the implementation of CKoala, the entire object, has a few more interesting features. First of all, note that even though we hold on to the punkOuter pointer, we do not ::AddRef it because we know that if punkOuter is non-NULL, the lifetime of that pointer completely contains our object's lifetime, so an extra reference count is unnecessary.

Second, we supply a two-phase instantiation process for use by the class factory we provide below. The CKoala constructor only saves variables whereas ::FInit performs any operations that are prone to failure so the caller can determine if a failure did occur. Since instantiating the interface implementation CImpIPersist might fail, we defer that action until ::FInit is called.

Next, the ::QueryInterface implementation in CKoala, which knows all the interfaces implemented in this object, makes a special case for the IPersistStorage interface. When a user like OBJUSER asks the DLL implementation for an interface identified with IID_IPersist, that IID comes directly into ::QueryInterface. However, when OBJUSER asks for IID_IPersist and the object lives in an EXE, that request goes through the marshaling layer in the component object model. The OLE 2.0 implementation of this marshaling does not single out IPersist and will always ask the object for IPersistStorage even if the user only asked for IPersist. So we also check for IPersistStorage here. Of course, **this must be avoided in real applications**, since the user might have actually asked for IPersistStorage but only received an IPersist. But like I pointed out above, IPersist is never useful implemented by itself–it's only used here for demonstration.

The final feature of the Koala object allows it to notify its server, either a DLL or an EXE, when the object is destroyed in ::Release. This is a special technique I created to isolate the object from any specifics about its DLL or EXE server–it's not part of the OLE 2.0 specifications so use whatever method is convenient for you. In any case, when the server receives this notification it decrements the count of active objects in the server. If the object lives in a DLL, that DLL might be able to then mark itself as unloadable; it the object lives in an EXE and it was the last object, the EXE might shut down on this notification. The class factory we'll write that instantiates these Koala objects will provide a pointer of type LPFNDESTROYED (see KOALA.H) that the object calls on its final ::Release. The server may take whatever action is deems necessary on this notification.

With the object isolated from any concern about where it lives, we can now concentrate on seeing how you expose that object from a DLL or EXE which share the same four steps to manage an object although their exact implementations of each step differ:

1. Register the CLSIDs for every class implemented in the server in the registration database.
2. Implement the class factory for each object class supported by the server. A single DLL or EXE

1Remember here that a C++ class is a *convenient* way to create interface function tables. CKoala is never exposed to anything outside its DLL or EXE, period. It only exposes it's IUnknown and IPersist interface function tables.

server can handle any number of classes.
3. Expose the class factory to the component object library.
4. Provide a shutdown or unloading mechanism when there are no more objects or lock counts on the server.

The DLL housing of Koala, DKOALA.DLL, is shown in Listing 4-4 with the EXE housing, EKOALA.EXE, shown in Listing 4-5. Note that the Koala object implementation itself shown in Listing 4-3 is identical in both the CHAP04\DKOALA and CHAP04\EKOALA directories in the sample code. When you run the OBJUSER program using each of these servers, you will notice a difference in the response time of calling the object's ::GetClassID. When using a DLL server, the response is quick because the call goes directly to the object implementation. When using an EXE server the response is slower since the ::GetClassID call must be worked through the marshaling process.

DKOALA.CPP

```
/*
 * DKOALA.CPP
 *
 * Example object implemented in a DLL.  This object supports
 * IUnknown and IPersist interfaces, meaning it doesn't know anything more
 * than how to return its class ID, but it demonstrates how any object
 * is presented inside an DLL.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


//Must do this once in the entire build or we can't define our own GUIDs
#define INITGUIDS

#include "dkoala.h"


//Count number of objects and number of locks.
ULONG       g_cObj=0;
ULONG       g_cLock=0;


HANDLE FAR PASCAL LibMain(HINSTANCE hInst, WORD wDataSeg
   , WORD cbHeapSize, LPSTR lpCmdLine)
   {
   if (0!=cbHeapSize)
      UnlockData(0);

   return hInst;
   }
```

```
void FAR PASCAL WEP(int bSystemExit)
   {
   return;
   }
```

Listing 4-4: The DKOALA.DLL implementation to house the Koala object.

```
HRESULT __export FAR PASCAL DllGetClassObject(REFCLSID rclsid, REFIID riid
   , LPVOID FAR *ppv)
   {
   if (!IsEqualCLSID(rclsid, CLSID_Koala))
      return ResultFromScode(E_FAIL);

   //Check that we can provide the interface
   if (!IsEqualIID(riid, IID_IUnknown) && !IsEqualIID(riid, IID_IClassFactory))
      return ResultFromScode(E_NOINTERFACE);

   //Return our IClassFactory for Koala objects
   *ppv=(LPVOID)new CKoalaClassFactory();

   if (NULL==*ppv)
      return ResultFromScode(E_OUTOFMEMORY);

   //Don't forget to AddRef the object through any interface we return
   ((LPUNKNOWN)*ppv)->AddRef();

   return NOERROR;
   }



STDAPI DllCanUnloadNow(void)
   {
   SCODE   sc;

   //Our answer is whether there are any object or locks
   sc=(0L==g_cObj && 0==g_cLock) ? S_OK : S_FALSE;
   return ResultFromScode(sc);
   }



/*
 * ObjectDestroyed
 *
```

```
 * Purpose:
 *  Function for the Koala object to call when it gets destroyed.
 *  Since we're in a DLL we only track the number of objects here
 *  letting DllCanUnloadNow take care of the rest.
 */

void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;
    return;
    }



CKoalaClassFactory::CKoalaClassFactory(void)
    {
    m_cRef=0L;
    return;
    }


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



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

    //Any interface on this object is the object pointer.
    if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IClassFactory))
        *ppv=(LPVOID)this;

    /*
     * If we actually assign an interface to ppv we need to AddRef it
     * since we're returning a new pointer.
     */
    if (NULL!=*ppv)
        {
        ((LPUNKNOWN)*ppv)->AddRef();
        return NOERROR;
        }
```

```
    return ResultFromScode(E_NOINTERFACE);
    }


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


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

    cRefT=--m_cRef;

    if (0L==m_cRef)
        delete this;

    return cRefT;
    }



STDMETHODIMP CKoalaClassFactory::CreateInstance(LPUNKNOWN punkOuter
    , REFIID riid, LPVOID FAR *ppvObj)
    {
    LPCKoala          pObj;
    HRESULT             hr;

    *ppvObj=NULL;
    hr=ResultFromScode(E_OUTOFMEMORY);

    //Verify that if there is a controlling unknown it's asking for IUnknown
    if (NULL!=punkOuter && !IsEqualIID(riid, IID_IUnknown))
        return ResultFromScode(E_NOINTERFACE);

    //Create the object telling it a function to notify us when it's gone.
    pObj=new CKoala(punkOuter, ObjectDestroyed);

    if (NULL==pObj)
        return hr;

    if (pObj->FInit())
        hr=pObj->QueryInterface(riid, ppvObj);
```

```
   //Kill the object if initial creation or FInit failed.
   if (FAILED(hr))
      delete pObj;
   else
      g_cObj++;

   return hr;
   }



STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
   {
   if (fLock)
      g_cLock++;
   else
      g_cLock--;

   return NOERROR;
   }
```

DKOALA.H

```
/*
 * DKOALA.H
 *
 * Definitions, classes, and prototypes for a DLL that
 * provides Koala objects to any other object user.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#ifndef _DKOALA_H_
#define _DKOALA_H_


//Get the object definitions that also includes windows.h, et. al.
#include "koala.h"


void FAR PASCAL ObjectDestroyed(void);
```

```
//DKOALA.CPP
//This class factory object creates Koala objects.

class __far CKoalaClassFactory : public IClassFactory
   {
   protected:
      ULONG          m_cRef;        //Reference count on class object

   public:
      CKoalaClassFactory(void);
      ~CKoalaClassFactory(void);

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

      //IClassFactory members
      STDMETHODIMP        CreateInstance(LPUNKNOWN, REFIID, LPVOID FAR *);
      STDMETHODIMP        LockServer(BOOL);
   };

typedef CKoalaClassFactory FAR * LPCKoalaClassFactory;


#endif //_DKOALA_H_
```

EKOALA.CPP

```
/*
 * EKOALA.CPP
 *
 * Object implemented in an application.  This object supports
 * IUnknown and IPersist interfaces, meaning it doesn't know anything more
 * than how to return its class ID, but it demonstrates how any object
 * is presented inside an EXE.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


//Must do this once in the entire build or we can't define our own GUIDs
#define INITGUIDS
```

Listing 4-5: The EKOALA.EXE implementation to house the Koala object.

```
#include <ole2ver.h>
#include "ekoala.h"


//Count number of objects and number of locks.
ULONG       g_cObj=0;
ULONG       g_cLock=0;


//Make window handle global so other code can cause a shutdown
HWND        g_hWnd=NULL;



int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
    , LPSTR pszCmdLine, int nCmdShow)
    {
    MSG         msg;
    LPAPPVARS   pAV;

    //This is suggested to handle many LRPC calls we might make
    SetMessageQueue(96);

    //Create and initialize the application.
    pAV=new CAppVars(hInst, hInstPrev, pszCmdLine, nCmdShow);

    if (NULL==pAV)
        return -1;

    if (pAV->FInit())
        {
        while (GetMessage(&msg, NULL, 0,0 ))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    delete pAV;
    return msg.wParam;
    }



LRESULT FAR PASCAL __export KoalaWndProc(HWND hWnd, UINT iMsg
```

```
    , WPARAM wParam, LPARAM lParam)
    {
    LPAPPVARS   pAV;

    //This will be valid for all messages except WM_NCCREATE
    pAV=(LPAPPVARS)GetWindowLong(hWnd, KOALAWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            //CreateWindow passed pAV to us.
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);

            SetWindowLong(hWnd, KOALAWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));


        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
        }

    return 0L;
    }



/*
 * ObjectDestroyed
 *
 * Purpose:
 *  Function for the Koala object to call when it gets destroyed.
 *  We destroy the main window if the proper conditions are met for
 *  shutdown.
 */

void FAR PASCAL ObjectDestroyed(void)
    {
    g_cObj--;

    //No more objects and no locks, shut the app down.
    if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
        PostMessage(g_hWnd, WM_CLOSE, 0, 0L);
```

```
    return;
    }



CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev, LPSTR pszCmdLine
    , UINT nCmdShow)
    {
    //Initialize WinMain parameter holders.
    m_hInst     =hInst;
    m_hInstPrev =hInstPrev;
    m_pszCmdLine=pszCmdLine;
    m_nCmdShow  =nCmdShow;

    m_hWnd=NULL;
    m_dwRegCO=0;
    m_pIClassFactory=NULL;
    m_fInitialized=FALSE;
    return;
    }


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

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

    if (m_fInitialized)
        CoUninitialize();

    return;
    }


BOOL CAppVars::FInit(void)
    {
    WNDCLASS    wc;
    HRESULT     hr;
    DWORD       dwVer;
```

```
//Check command line for -Embedding
if (lstrcmpi(m_pszCmdLine, "-Embedding"))
   return FALSE;

dwVer=CoBuildVersion();

if (rmm!=HIWORD(dwVer))
   return FALSE;

if (FAILED(CoInitialize(NULL)))
   return FALSE;

m_fInitialized=TRUE;

if (!m_hInstPrev)
   {
   wc.style         = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc    = KoalaWndProc;
   wc.cbClsExtra     = 0;
   wc.cbWndExtra     = CBWNDEXTRA;
   wc.hInstance      = m_hInst;
   wc.hIcon          = NULL;
   wc.hCursor        = NULL;
   wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName   = NULL;
   wc.lpszClassName  = "Koala";

   if (!RegisterClass(&wc))
      return FALSE;
   }

m_hWnd=CreateWindow("Koala", "Koala", WS_OVERLAPPEDWINDOW
   ,35, 35, 350, 250, NULL, NULL, m_hInst, this);

if (NULL==m_hWnd)
   return FALSE;

g_hWnd=m_hWnd;

/*
 * Create our class factory and register it for this application
 * using CoRegisterClassObject.  We are able to service more than
 * one object at a time so we use REGCLS_MULTIPLEUSE.
 */
m_pIClassFactory=new CKoalaClassFactory();
```

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

    //Since we hold on to this, we should AddRef it.
    m_pIClassFactory->AddRef();

    hr=CoRegisterClassObject(CLSID_Koala, (LPUNKNOWN)m_pIClassFactory
        , CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &m_dwRegCO);

    if (FAILED(hr))
        return FALSE;

    return TRUE;
    }


CKoalaClassFactory::CKoalaClassFactory(void)
    {
    m_cRef=0L;
    return;
    }

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

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

    //Any interface on this object is the object pointer.
    if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IClassFactory))
        *ppv=(LPVOID)this;

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

    return ResultFromScode(E_NOINTERFACE);
    }


STDMETHODIMP_(ULONG) CKoalaClassFactory::AddRef(void)
```

```
    {
    return ++m_cRef;
    }


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

    cRefT=--m_cRef;

    if (0L==m_cRef)
        delete this;

    return cRefT;
    }



STDMETHODIMP CKoalaClassFactory::CreateInstance(LPUNKNOWN punkOuter
    , REFIID riid, LPVOID FAR *ppvObj)
    {
    LPCKoala           pObj;
    HRESULT             hr;

    *ppvObj=NULL;
    hr=ResultFromScode(E_OUTOFMEMORY);

    //Verify that if there is a controlling unknown it's asking for IUnknown
    if (NULL!=punkOuter && !IsEqualIID(riid, IID_IUnknown))
        return ResultFromScode(E_NOINTERFACE);

    //Create the object telling it a function to notify us when it's gone.
    pObj=new CKoala(punkOuter, ObjectDestroyed);

    if (NULL==pObj)
        return hr;

    if (pObj->FInit())
        hr=pObj->QueryInterface(riid, ppvObj);

    g_cObj++;

    /*
     * Kill the object if initial creation or FInit failed.  If
     * the object failed then we handle the g_cObj increment above
```

```
    * in ObjectDestroyed.
    */
   if (FAILED(hr))
      {
      delete pObj;
      ObjectDestroyed();  //Handle shutdown cases.
      }

   return hr;
   }


STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
   {
   if (fLock)
      g_cLock++;
   else
      {
      g_cLock--;

      //No more objects and no locks, shut the app down.
      if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
         PostMessage(g_hWnd, WM_CLOSE, 0, 0L);
      }

   return NOERROR;
   }
```

EKOALA.H

```
/*
 * EKOALA.H
 *
 * Definitions, classes, and prototypes for an application that
 * provides Koala objects to any other object user.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#ifndef _EKOALA_H_
#define _EKOALA_H_
```

```
//Get the object definitions that also includes windows.h, et. al.
#include "koala.h"


//EKOALA.CPP
LRESULT FAR PASCAL __export KoalaWndProc(HWND, UINT, WPARAM,
LPARAM);

class __far CAppVars
   {
   friend LRESULT FAR PASCAL __export KoalaWndProc(HWND, UINT, WPARAM,
LPARAM);

   protected:
      HINSTANCE       m_hInst;           //WinMain parameters
      HINSTANCE       m_hInstPrev;
      LPSTR           m_pszCmdLine;
      UINT            m_nCmdShow;

      HWND            m_hWnd;            //Main window handle

      BOOL            m_fInitialized;    //Did CoInitialize work?
      LPCLASSFACTORY  m_pIClassFactory;  //The class factory we own.
      DWORD           m_dwRegCO;         //Key from CoRegisterClassObject.

   public:
      CAppVars(HINSTANCE, HINSTANCE, LPSTR, UINT);
      ~CAppVars(void);
      BOOL FInit(void);
   };


typedef CAppVars FAR * LPAPPVARS;

#define CBWNDEXTRA              sizeof(LONG)
#define KOALAWL_STRUCTURE     0


void FAR PASCAL ObjectDestroyed(void);


//This class factory object creates Koala objects.

class __far CKoalaClassFactory : public IClassFactory
   {
   protected:
```

```
    ULONG       m_cRef;     //Reference count on class object

  public:
    CKoalaClassFactory(void);
    ~CKoalaClassFactory(void);

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

    //IClassFactory members
    STDMETHODIMP        CreateInstance(LPUNKNOWN, REFIID, LPVOID FAR *);
    STDMETHODIMP        LockServer(BOOL);
  };

typedef CKoalaClassFactory FAR * LPCKoalaClassFactory;

#endif //_EKOALA_H_
```

Register CLSIDs

Every object class must have a unique CLSID associated with it in the registration database. The registration entries for a simple object as we're implementing here are few; as you create objects with more features that support linking and embedding there will be much more information to add as described in Chapter 10 as well as in Appendix A of the OLE 2.0 Programmer's Reference. For the purposes of the sample code, the necessary entries are contained in the file CHAP04.REG. Creating a .reg file is the preferred method of registering objects and applications since it can be done at install time instead of programmatically at run time, which is tedious, to say the least. The OLE 2.0 Reference describes the details about installation in its Appendix A.

The required entries fall under the CLSID key where OLE 2.0 stores information about all classes under the CLSID spelled out in a string as you can see in the REGEDIT program and shown in Figure 4-5. OLE 2.0 also stores information about its standard interfaces and the code that handles parameter marshaling under the Interface key. The list below describes the necessary registration for DLL and EXE-based objects:

1. From HKEY_CLASSES_ROOT (the root key of the entire registration database), create the entry \CLSID\{*class ID*}=<*name*> where {*class ID*} is the value of your CLSID spelled out and <*name*> is a human-readable string for your object. The Koala object has the class ID string of {00021102-0000-0000-C000-000000000046}, which, except to a few odd individuals, is not something many consider readable.
2. Create an entry under the CLSID entry in step 1 to point to the object code:
    a. For DLL objects, register \InprocServer=<*path to DLL*>
    b. For EXE object, register \LocalServer=<*path to EXE*>
    c. For DLL object handlers, register \InprocHandler=<*path to DLL*>

3. (Optional) If you want to allow a user to look up your CLSID based on a text string, make an entry under HKEY_CLASSES_ROOT of <*ProgID*>=<*name*> where <*ProgID*> is a short name without spaces or punctuation, equivalent to an OLE 1.0 class name, and <*name*> is the human-readable name of your object identical to that in step 1. Under this key create another entry of \CLSID={*class ID*} where {*class ID*} is also the same as in step 1.

Entries of the type created in step 3 will be required for compound document objects that should appear in the Insert Object dialog inside a container application. But that is a subject for a later chapter. Without

those entries from steps 1 and 2, however, the CoGetClassObject API (which CoCreateInstance uses, remember) will not be able to locate your object implementation. Note also that the same DLL or EXE can serve multiple CLSIDs, and in such cases you must make a similar entry under each CLSID you support with the InprocServer and LocalServer keys can all contain the same path to the same server.

Implement the Class Factory

Telling the component object library where your object code lives is one thing–you still need to provide for creating objects once that code has been loaded. So the next step is to create a class object that implements the IClassFactory interface. Once we have this class factory implemented we can provide the code to expose it outside the server. This is somewhat like implementing a window procedure in order to call RegisterClass since you have to have a pointer to store in the WNDCLASS you pass to that Windows API. Both sample implementations, DLL and EXE, use a C++ class of CKoalaClassFactory for this purpose and the two are almost identical. The only differences between the DLL and EXE CKoalaClassFactory have to do with the unloading mechanisms we'll discuss below. For now, let's concentrate on those identical parts that instantiate a CKoala object. For the record, the IUnknown members of this class factory are pretty standard.

Figure 4-5: The populated CLSID section of the registration database showing the entries for Koala.

Each implementation of IClassFactory::CreateInstance is identical and contain three major points of interest. First, the first parameter to ::CreateInstance is punkOuter which is the controlling unknown for the object we've been asked to create, if our new object is becoming part of an aggregate. When we instantiate the object using *new CKoala*, we pass this punkOuter down to the object so it can delegate properly (again, see "Object Reusability" for more details). When an object is aggregated, the outer object **must** ask for an IUnknown interface on the new object. To enforce this rule, we check that IID_IUnknown is asked for when punkOuter is non-NULL.

Next, besides passing punkOuter to *new CKoala* we also pass a pointer to an independent function called ObjectDestroyed. When the final ::Release of the Koala object is about to free the object, it will call this function. This allows the object to isolate itself from the nature of its server housing (DLL or EXE) and allow that housing to act appropriately on the event. You can see again in the CKoala::Release function in Listing 4-3 how and when this function is called. We'll examine what ObjectDestroyed does in both servers in the section "Provide an Unloading Mechanism" below.

Finally, if the object is successfully instantiated, we still need to initialize it through an internal ::FInit implemented in the CKoala object. ::FInit is not a standard feature of OLE 2.0 objects and is used here to support a convenient two-phase creation model common in C++ coding. ::FInit performs all operations that might fail and is thus able to communicate success or failure back to the class factory during instantiation. If this second initialization step succeeds, then ::CreateInstance asks the object's ::QueryInterface to return the appropriate interface pointer, which has the convenient effect of calling that pointer's ::AddRef as required. Furthermore, ::CreateInstance increments a global object count that the server can use to determine unloading conditions. If the initialization fails, then ::CreateInstance deletes the object and returns an out-of-memory error to the caller.

IClassFactory also has a member called LockServer, which either increments or decrements a lock count on the DLL or EXE in which the class factory lives. ::LockServer provides a method through which a user can obtain keep a DLL or EXE in memory even if that server is not servicing any objects and has no outstanding reference counts on its class factory. This allows a user to optimize loading and reloading of servers, keeping the code in memory even when it's not immediately necessary. Such optimizations can greatly increase performance when a user deals with a very large server EXE or DLL.

The implementation of ::LockServer in the DLL and EXE versions differs slightly, again to handle the differences in unloading mechanisms. Their communality is to either increment or decrement a global lock counter which is used in different ways by the server's unloading mechanism.

Expose the Class Factory

The major difference between DLL and EXE servers is how they expose their class factories, primarily because a DLL does not define a task and an EXE may, in fact, be run stand-alone outside the context of object use. In other words, your class factory is an object and the component object library needs to obtain its IClassFactory pointer. It does so by either calling a function you export from an object DLL (that is, an API you implement), or by you calling a component object library API from your own code. In either case an API

is used to get the class factory pointer from your code to OLE 2.0's code.

DLL Server

The DLL exposure mechanism is the simplest, so let's start there. Every DLL server must export a function called DllGetClassObject with the following form:

```
HRESULT __export FAR PASCAL DllGetClassObject(REFCLSID rclsid, REFIID riid
   , LPVOID FAR * ppv);
```

The __export is a matter of convenience–if your compiler does not support __export you can still list the function in the EXPORTS section of your .DEF file. In addition, the macro STDAPI defined in COMPOBJ.H expands to HRESULT __export FAR PASCAL if you want to use it.

When a user calls CoCreateInstance or CoGetClassObject and passed CLSCTX_INPROC_SERVER, the component object library will look in the registration database for the location of an InprocServer for the given CLSID, call CoLoadLibrary to get that server into memory, then call GetProcAddress looking for DllGetClassObject. The component object library then calls DllGetClassObject with the CLSID and IID requested by the component user. Your export then creates the appropriate class factory for the CLSID and returns the appropriate interface pointer for IID, which is usually IClassFactory. By calling this function in your DLL, the component object library obtains a pointer to your class factory object; essentially DllGetClassObject is an API you implement for OLE 2.0. This is just like exporting the WEP function from a DLL such that Windows can locate and call it.

NOTE: Since DllGetClassObject is passed a CLSID, a single DLL (server or handler) can provide class factories for any number of classes, that is, a single module can be the server for any number of objects. OLE2.DLL is an example of such a server as it provides most of the internally used objects of OLE 2.0 from one DLL.

All implementations of DllGetClassObject should validate that it can support the requested CLSID as well as the requested interface for the class factory which can be either IUnknown or IClassFactory. If both checks succeed, it then instantiates the class factory object (in this case using CKoalaClassFactory). Remember that as a function that creates a new interface pointer to an object, DllGetClassObject must ::AddRef the new object before returning:

```
if (!IsEqualCLSID(rclsid, CLSID_Koala))
   return ResultFromScode(CO_E_CLASSNOTREG);

//Check that we can provide the interface
if (!IsEqualIID(riid, IID_IUnknown) && !IsEqualIID(riid, IID_IClassFactory))
   return ResultFromScode(E_NOINTERFACE);

//Return our IClassFactory for Koala objects
*ppv=(LPVOID)new CKoalaClassFactory();

//Don't forget to AddRef the object through any interface we return
((LPUNKNOWN)*ppv)->AddRef();
```

EXE Server

Exposing a class factory from an EXE is somewhat different because an EXE has a WinMain, a message loop, and a window to define its lifetime. When we implement compound document objects in Chapter 10, these parts of the application will be the same as the end user would see if they ran your application stand-alone. The real difference is that instead of having the component object library call an exported function like DllGetClassObject, you **pass** your class factory object (that is, an IClassFactory pointer) to the **CoRegisterClassObject** API, but only under the right circumstances.

The component object library informs an EXE that its being used to service objects through a command-line flag **-Embedding**. This flag is simply appended to the path entry for this local server in the registration database, so if you register your EXE with flags yourself, look for this at the end of the command line. Checking this flag is priority #1 in EKOALA's initialization. If this flag is not present, then the end user attempted to run the application stand-alone from the shell. Since this application doesn't live for any purpose other than to service objects, it fails to load if -Embedding is not present.

The next few steps in CAppVars::FInit are the same as those required of any OLE 2.0 application: CoBuildVersion and CoInitialize since we are using component object library APIs. After such initialization we create a window for this task that remains hidden; **in all cases where -Embedding is on the command line, the server window should remain hidden until explicitly asked to show itself.** For this

demonstration, the EKOALA program has no need to ever show its window since it has no user interface. For compound document servers, they may be asked to provide an updated rendering of an object without ever showing that object to the end user. Compound document objects implement the IOleObject interface whose ::DoVerb is told when to actually show the window, if it ever becomes necessary.

If we get past the initialization stage we must then create the class factory object and pass it to CoRegisterClassObject in the same way we are very accustomed to calling the Windows API RegisterClass. With RegisterClass you create a WNDCLASS structure, fill in the lpfnWndProc field with a pointer to your window's message procedure, and pass a pointer to that WNDCLASS to RegisterClass. Your window procedure is not actually called until you, or someone else, creates a window of your registered class. With CoRegisterClassObject you create a class factory object where that object has a function table for the IClassFactory interface that you must fill with pointers to your IClassFactory implementation. You then pass a pointer to the object to CoRegisterClassObject, but the interface functions like ::CreateInstance are not called until you or someone else creates a component object of your class.[1]

Creating the class factory object is just the matter of allocating the object's data structure and function table, which is conveniently handled in C++ with the *new* operator:

```
//Return our IClassFactory for Koala objects
m_pIClassFactory=new CKoalaClassFactory();

if (NULL==m_pIClassFactory)
   return FALSE;

//Since we hold on to this, we should AddRef it
m_pIClassFactory->AddRef();
```

The additional ::AddRef insures that the application controls the lifetime of the class factory because the CKoalaClassFactory constructor initializes its reference count to zero. Since the application makes the first AddRef, it will have to make the last ::Release that allows the class factory to destroy itself. Since we have this extra AddRef on the class factory we do not need to include it as part of the server's object count.

Once we have created the class factory we have to tell the component object library about it using **CoRegisterClassObject**. We have to be the one that calls this function because we have yet to yield from this task in our message loop, so the component object library does not have a chance to call us as happens in a DLL:

```
hr=CoRegisterClassObject(CLSID_Koala, (LPUNKNOWN)m_pIClassFactory
   , CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &m_dwRegCO);

if (FAILED(hr))
   return FALSE;   //Registration failed.

[Class factory successfully registered]
```

CoRegisterClassObject takes the CLSID of the factory we're providing, a pointer to the class factory, the context in which we're running (always CLSCTX_LOCAL_SERVER), a flag indicating how this class factory can be used, and a pointer to a DWORD in which CoRegisterClassObject returns a registration key we'll need later during shutdown.

NOTE: If your EXE is the server for multiple classes, you must call CoRegisterClassObject for each supported CLSID just like you would RegisterClass for each window class you support. The component object library will launch you when any user requests any CLSID you support, but cannot tell you through WinMain which CLSID that was. So you must register a class factory for each CLSID you placed in the registration database.

The fourth parameter to CoRegisterClassObject specifies how many objects can be created using this class factory: REGCLS_SINGLEUSE or REGCLS_MULTIPLEUSE. If you specify single use, then OLE will launch another instance of your application each time a user calls CoGetClassObject. If you specify multiple-use, then one instance of the application is used to service any number of objects. In the compound document scenario, this is equivalent to having a single-document (SDI) or multiple-document (MDI) applications since SDI applications can only edit one object at a time whereas MDI can edit more.

To prove this to yourself, run two instance of the OBJUSER program, specify "Use EXE Object" from both Koala Object menus, and use one of the Create commands from the menu. Since EKOALA registers

1In reality, calling CoRegisterClassObject immediately generates a number of calls to your IClassFactory::AddRef since the component object library is holding on to your IClassFactory pointer. So your object is called before ever creating an object unlike your window procedure.

itself as multiple-use, only one instance will be loaded to service both objects. Now change EKOALA to register as single-use and run two OBJUSERs again, using the same commands. This time two instances of EKOALA will be run, each only servicing one object.

Provide an Unloading Mechanism

Since the mechanisms we use to expose a class factory from both kinds of servers differ, the mechanisms for indicating when the server is no longer needed also differs. An unloading mechanism is not a consideration for normal Windows applications because they are almost always controlled by the user. OLE 2.0 allows DLLs and EXEs that serve objects to be controlled by another piece of component user code. Since there is no end-user involve to close applications, there must be a programmatic technique to accomplish the same ends.

The bottom line is that any server is no longer needed when there are no lock counts from IClassFactory::LockServer and there are no objects currently being serviced. However, since the EXE server has a window it must destroy its main window, cause a call to PostQuitMessage, exit the message loop, and quit the application. Since DLLs have no idea of "quitting" (that is, no message loop to exit) they instead mark themselves as "unloadable."

DLL Server

Again, let's start with the DLL since the mechanism is trivial. As we have seen, the DLL server increments and decrements a global[1] lock count in IClassFactory::LockServer and increments the object count in IClassFactory::CreateInstance. When any Koala object is destroyed we want to decrement the object count which is handled in the ObjectDestroyed function we provided to the Koala object:

```
void FAR PASCAL ObjectDestroyed(void)
  {
  g_cObj--;
  return;
  }
```

The DLL never tells anyone to unload it; instead, the component object model will periodically **ask** it "can you unload now?" by calling an export DllCanUnloadNow with the following form:

```
STDAPI DllCanUnloadNow(void)
  {
  SCODE  sc;

  //Our answer is whether there are any object or locks
  sc=(0L==g_cObj && 0==g_cLock) ? S_OK : S_FALSE;
  return ResultFromScode(sc);
  }
```

The implementation shown above will answer "yes" when both object and lock counts are zero and "no" otherwise. If this function answers yes, then the libraries will internally call CoFreeLibrary to reverse the call it made to CoLoadLibrary from within CoGetClassObject.

NOTE: The function that should call DllCanUnloadNow is CoFreeUnusedLibraries which, as we've seen, is called periodically by an object user. However, the OLE 2.0 implementation of CoFreeUnusedLibraries does nothing, so you will never see a call to DllCanUnloadNow. However, CoFreeUnusedLibraries will be implemented in the near future, so implement DllCanUnloadNow as if it were always called anyway.

Also note that there has been no mention of class factory reference counts in any other this because such reference counts are not used to keep the DLL in memory. Object users wishing to hold a class factory must also call your ::LockServer (as described in "Implementing an Object User" above). While a reference count could easily prevent a DLL from unloading, it's impossible to do so in an EXE which the discussion below illustrates.

Congratulations! You're a mother! After implementing DllCanUnloadNow you now have a complete DLL object server into which you can put more and more complex objects and interfaces, continuing to use the same mechanisms. The framework for DLL-based objects developed here will be used for more complex DLL objects in this book. I certainly hope you will be able to use it for incredible objects of your own.

1I confess!! I used global variables! I normally try hard to avoid any use of global variables like many readers do. In this case having a few globals simplified both DLL and EXE implementation. You might also see me declare global instance handle when appropriate since instance handles are really application-wide and might be needed deep in a long chaing of function calls. In any case, global variables in this and other sample apps are prefixed with g_ for clear identification. Please forgive my transgressions!

EXE Server

Instead of being asked when you can be unloaded as in the DLL case, an EXE server must initiate shutdown itself when it detects the following conditions: there are no objects being serviced and there is a zero lock count. This detection complicates EXE servers especially when we deal with compound documents because we throw in another condition regarding the end user and what they might have done with the application (see "***Preview Note: I Don't Have the Section Yet***" in Chapter 10). But for now, we need to detect only when these two conditions are met and start shutdown by posting a WM_CLOSE to the main window. The two places where we must add a check is in IClassFactory::LockServer and in the ObjectDestroyed function that the Koala object will call when it's freed:

```
STDMETHODIMP CKoalaClassFactory::LockServer(BOOL fLock)
   {
   if (fLock)
      g_cLock++;
   else
      {
      g_cLock--;

      //No more objects and no locks, shut the app down.
      if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
         PostMessage(g_hWnd, WM_CLOSE, 0, 0L);
      }

   return NOERROR;
   }


void FAR PASCAL ObjectDestroyed(void)
   {
   g_cObj--;

   //No more objects and no locks, shut the app down.
   if (0==g_cObj && 0==g_cLock && IsWindow(g_hWnd))
      PostMessage(g_hWnd, WM_CLOSE, 0, 0L);

   return;
   }
```

To facilitate message posting the I sinned again in EKOALA by storing its window handle as a global variable. This slight bit of "cheating" as you may consider it works clean and easy without having to pass the window handle around. Such a global guarantees that any code in this application could start a shutdown with the same mechanism. We could also use PostAppMessage but that requires some changes to the application's message loop which wouldn't be any cleaner.

By posting WM_CLOSE I am starting shutdown of EKOALA exactly as if an end-user closed it from the system menu. In the process of shutting down, EKOALA destroys the main window (DefWindowProc's handling of WM_CLOSE), exits WinMain (by calling PostQuitMessage in WM_DESTROY), and ends up in CAppVars::~CAppVars. This destructor first calls **CoRevokeClassObject** that removes the class factory you passed to **CoRegisterClassObject** identified by the DWORD key that CoRegisterClassObject returned. If you registered multiple class factories for different CLSIDs you must revoke each one here. CoRevokeClassObject will ::Release any reference counts that the component object library was holding on the class factory. Furthermore, since we called ::AddRef ourself before CoRegisterClassObject (remember that, long ago?), we must now match it with a ::Release. This will reduce the class factory's reference count to zero and free that object. Finally, since we called CoInitialize we need to remember to call CoUninitialize.

CoRevokeClassObject is the reason why a class factory's reference count cannot be used by a component user to keep a server, DLL or EXE, in memory. If a positive reference count could keep the class factory in memory, then we could not shut the application down until the reference count was zero and the class factory was destroyed. But the reference count will never reach zero unless we call CoRevokeClassObject. But we only call CoRevokeClassObject when we shut down, after our window is gone, we've exited the message loop, and we're on the non-stop express to oblivion. So we can't revoke until we're shutting down and we can't shut down until we revoke. Aaaugh!. Fourth down and 100 yards to go...so we punt: a positive reference count on a class factory cannot be used to keep a server in memory. A component user must rely on ::LockServer to prevent shutdown, not the class factory reference count. Our saving grace is that this is the only special case of reference counting in all of OLE 2.0.

Object Handlers (Default Handler)

An object handler is a lightweight DLL server used to provide a partial implementation of an full EXE object, thereby reducing the need to launch the EXE to service that object. Since DLL objects generally load faster and need no parameter marshaling, use of handlers generally increases overall performance. Object handlers used in conjunction with a compound document provide for data transfer and object rendering (directly to screen or printer) but not for editing, making handlers ideal for licensing for redistribution with document, much like TrueType fonts in Windows 3.1 can be saved with a document given the proper license. Chapter 11 will examine Object Handlers for specific use with compound documents.

If an object user calls CoGetClassObject with CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER then the handler is loaded first and all calls to the object's interfaces are sent to the DLL. If an full DLL object exists as well, OLE will use that DLL first if the CLSCTX_INPROC_SERVER flag is specified.

When a handler discovers that it cannot provide the requested function for the caller, it can delegate to the full object implementation in the EXE. In order to delegate calls the handler has to first CoCreateInstance on the same CLSID as itself, specifying only CLSCTX_LOCAL_SERVER that forces OLE to launch the application. In any case, CoCreateInstance returns the pointer to the object of the same class in the EXE through which the handler can interact with the full implementation. When the handler's object is ::Released for the final time, it calls ::Release on the EXE object it created.

This is not, of course, without restrictions: the EXE object can only support standard interfaces with built-in marshaling support there is limited communication between the handler and the application. Furthermore, the handler must insure that data is synchronized between itself and the application. Most often, however, a handler exists to provide speedy rendering of specific data formats and delegates requests for more esoteric formats to the application.

OLE 2.0 provides a default handler OLE2.DLL that is always used in lieu of a specific handler for a class. We'll explore various aspects of the default handler over the next few chapters with a more complete list of its functionality in Chapter 11. An object handler for compound document objects can create a default handler object to which it delegates unimplemented calls, and the default handler takes care of launching an application when necessary. In the compound document scenario an object handler should never directly instantiate it's own server: let OLE2.DLL do that.

Finally, the only different between DLL object handlers and DLL object servers is their intended uses and expected performance. Technically and structurally, the handler is identical to a DLL server. All discussion in this chapter dealing with DLL servers applies equally to DLL handlers. The real significant differences lie in how the two are used and how they should be designed, which are topics for Chapters 10 and 11.

Schmoo's Polyline as a DLL Object

The Koala object that supports the IPersist interface pretty boring and, well, useless. To demonstrate an object much more useful and exciting the CHAP04\POLYLINE directory in the sample code contains an implementation of Schmoo's CPolyline class as a Windows Object in a DLL. The source code for POLYLINE.DLL is a little too long to show here, however. This implementation shows that a much more complicated object such as Polyline can fit into exactly the same housing (DLL or EXE) as a simple object like Koala.

The member functions of the original CPolyline from Chapter 2 are converted into a custom interface IPolyline4 defined in INC\IPOLY4.H (the '4' in the names are for Chapter 4 since we'll be making modifications to the interface through this book)and shown Listing 4-7. Note that since all interface functions should return HRESULT wherever possible, some return values in the original CPolyline are converted into out-parameters in the interface. IPOLY4.H also defines an interface IPolylineAdviseSink4 (which we'll also modify as time goes on) through which a Schmoo document receives notifications from the Polyline. This replaces the CPolylineAdviseSink class that Schmoo used before.

IPOLY4.H

```
/*
```

```
* IPOLY4.H
* Chapter 4 Polyline Object
*
* Definition of an IPolyline interface for a Polyline object used
* in the Schmoo implementation.  This interface is custom and is
* only supported from DLL-based objects.
*
* Copyright (c)1993 Microsoft Corporation, All Rights Reserved
*/


#ifndef _IPOLY4_H_
#define _IPOLY4_H_

//Versioning.
#define VERSIONMAJOR             2
#define VERSIONMINOR             0
#define VERSIONCURRENT             0x00020000

#define CPOLYLINEPOINTS            20

//Version 2.0 Polyline Structure
typedef struct __far tagPOLYLINEDATA
   {
   WORD       wVerMaj;              //Major version number.
   WORD       wVerMin;              //Minor version number.
   WORD       cPoints;             //Number of points.
   BOOL       fReserved;            //Previously fDrawEntire, obsoleted
   RECT       rc;                  //Rectangle of this figure (client)
   POINT      rgpt[CPOLYLINEPOINTS];  //Array of points on 0-32767 grid.

   //Version 2.0 additions
   COLORREF    rgbBackground;         //Background color
```

Listing 4-6: IPolyline4 and IPolylineAdviseSink4 custom interfaces.

```
   COLORREF    rgbLine;              //Line color
   int        iLineStyle;           //Line style
   } POLYLINEDATA, *PPOLYLINEDATA, FAR *LPPOLYLINEDATA;

#define CBPOLYLINEDATA   sizeof(POLYLINEDATA)


//We use the OLE 2.0 macro to define a new interface
#undef  INTERFACE
#define INTERFACE IPolylineAdviseSink4
```

```
/*
 * When someone initializes a polyline and is interested in receiving
 * notifications on events, then they provide one of these objects.
 */

DECLARE_INTERFACE_(IPolylineAdviseSink4, IUnknown)
   {
   //IUnknown members
   STDMETHOD(QueryInterface) (THIS_ REFIID, LPVOID FAR *) PURE;
   STDMETHOD_(ULONG,AddRef)  (THIS) PURE;
   STDMETHOD_(ULONG,Release) (THIS) PURE;

   //Advise members.
   STDMETHOD_(void,OnPointChange)     (THIS) PURE;
   STDMETHOD_(void,OnSizeChange)      (THIS) PURE;
   STDMETHOD_(void,OnDataChange)      (THIS) PURE;
   STDMETHOD_(void,OnColorChange)     (THIS) PURE;
   STDMETHOD_(void,OnLineStyleChange) (THIS) PURE;
   };

typedef IPolylineAdviseSink4 FAR * LPPOLYLINEADVISESINK;


//We use the OLE 2.0 macro to define a new interface
#undef  INTERFACE
#define INTERFACE IPolyline4

DECLARE_INTERFACE_(IPolyline4, IUnknown)
   {
   //IUnknown members
   STDMETHOD(QueryInterface) (THIS_ REFIID, LPVOID FAR *) PURE;
   STDMETHOD_(ULONG,AddRef)  (THIS) PURE;
   STDMETHOD_(ULONG,Release) (THIS) PURE;

   //IPolyline members

   //File-related members:
   STDMETHOD(ReadFromFile) (THIS_ LPSTR) PURE;
   STDMETHOD(WriteToFile)  (THIS_ LPSTR) PURE;

   //Data transfer members:
   STDMETHOD(DataSet)      (THIS_ LPPOLYLINEDATA, BOOL, BOOL) PURE;
   STDMETHOD(DataGet)      (THIS_ LPPOLYLINEDATA) PURE;
   STDMETHOD(DataSetMem)   (THIS_ HGLOBAL, BOOL, BOOL, BOOL) PURE;
   STDMETHOD(DataGetMem)   (THIS_ HGLOBAL FAR *) PURE;
   STDMETHOD(RenderBitmap) (THIS_ HBITMAP FAR *) PURE;
```

```
    STDMETHOD(RenderMetafile) (THIS_ HMETAFILE FAR *) PURE;
    STDMETHOD(RenderMetafilePict) (THIS_ HGLOBAL FAR *) PURE;

    //Manipulation members:
    STDMETHOD(Init)   (THIS_ HWND, LPRECT, DWORD, UINT) PURE;
    STDMETHOD(New)    (THIS) PURE;
    STDMETHOD(Undo)   (THIS) PURE;
    STDMETHOD(Window) (THIS_ HWND FAR *) PURE;

    STDMETHOD(SetAdvise) (THIS_ LPPOLYLINEADVISESINK) PURE;
    STDMETHOD(GetAdvise) (THIS_ LPPOLYLINEADVISESINK FAR *) PURE;

    STDMETHOD(RectGet) (THIS_ LPRECT) PURE;
    STDMETHOD(SizeGet) (THIS_ LPRECT) PURE;
    STDMETHOD(RectSet) (THIS_ LPRECT, BOOL) PURE;
    STDMETHOD(SizeSet) (THIS_ LPRECT, BOOL) PURE;

    STDMETHOD(ColorSet) (THIS_ UINT, COLORREF, COLORREF FAR *) PURE;
    STDMETHOD(ColorGet) (THIS_ UINT, COLORREF FAR *) PURE;

    STDMETHOD(LineStyleSet) (THIS_ UINT, UINT FAR *) PURE;
    STDMETHOD(LineStyleGet) (THIS_ UINT FAR *) PURE;
    };

typedef IPolyline4 FAR * LPPOLYLINE;

//Error values for data transfer functions, in SCODEs instead of #defines
#define POLYLINE_E_INVALIDPOINTER      MAKE_SCODE(SEVERITY_ERROR,
FACILITY_ITF, 1)
#define POLYLINE_E_READFAILURE         MAKE_SCODE(SEVERITY_ERROR,
FACILITY_ITF, 2)
#define POLYLINE_E_WRITEFAILURE        MAKE_SCODE(SEVERITY_ERROR,
FACILITY_ITF, 3)

//Color indices for color member functions
#define POLYLINECOLOR_BACKGROUND    0
#define POLYLINECOLOR_LINE          1

#endif //_IPOLY4_H_
```

The core implementation of the Polyline has not changed significantly. The DLLPOLY.CPP file is only a slight modification from the DKOALA.CPP file: a few name changes, a class registered in LibMain, and DLL instance handle passed to the CPolyline constructor. Oh yes, CPolyline still exists but is now more like the CKoala object in the previous examples. The member functions of the former CPolyline of Chapter 2 have been moved to the interface implementation CImpIPolyline.

As we move forward in this book we'll incrementally replace specific members of IPolyline4 with those of another interface. In the next chapter, for example, we'll remove the two file-related functions from

IPolyline5 replacing them with an IPersistStorage interface to the object. In Chapter 6 we'll replace the data transfer and graphics rendering functions in IPolyline6 with the IDataObject interface. In Chapter 10 we'll replace the advise and size manipulation functions with IOleObject, followed by replacement of ::Window with IOleInPlaceObject in Chapter 15, and finally eliminating all others using IDispatch in Chapter 19.

The version of Schmoo that uses the component Polyline object, Component Schmoo, is provided in CHAP04\COSCHMOO and only required a few of modifications. When you run Component Schmoo, however, you will notice absolutely no changes in the user interface or in any behavior. Component Schmoo just changed from being the user of a local C++ object, CPolyline, to a component user of the Polyline Windows object through the IPolyline4 interface.

You will notice that the component Polyline object does not concern itself with reading and writing old file formats: it only bothers with it's current format. This is because as we move forward into the next chapters this DLL will be more concerned with compound files that live in storage objects much more than with old traditional MS-DOS files. Since it's goal is to service embedded objects as an in-process DLL server, it has no concept of file versions. You can still use the self-contained Schmoo application to convert files.

To demonstrate the self-contained Schmoo as a server of Polyline objects, since the polyline is not an application in itself, we'll also develop Schmoo in parallel with the component Polyline object. Instead of restructuring Schmoo internals, we'll implement the interfaces on top of the existing CPolyline in order to expose CPolyline's functions externally. In Component Schmoo, we replace the custom interface with standard interfaces, allowing the object to be usable from more than one application.

Object Reusability

So what about inheritance?Windows Objects themselves and the classes they identify through CLSIDs have no notion of *implementation* inheritance whatsoever, that is, one Windows Object does no inherit the implementation of another Windows Object. But Windows Object are still **reusable** through two mechanisms called containment and aggregation. These mechanisms has several significant benefits over inheritance, which is why the Component Object Model has significant benefits over other object models that rely heavily on inheritance.

In the Component Object Model, inheritance is simply considered a tool useful for implementing classes in C++ as well as in defining interfaces. In your implementation of an object, you can either use multiple inheritance from all the interfaces you support, or you can contain implementations of each interface it supports where each interface implementation inherits a single interface. That's really what interface is for: to help the **implementor** of an object but not the **user** of an object. Inheritance greatly enhances programmer productivity, but does the user really care how the object was implemented? The answer is definitely NO since object users are *supposed* to be entirely **ignorant** of the object's implementation, especially when the implementation exists in other pieces of code that you did not implement or those for which you don't have the source, such as Windows itself.

The single most significant problem of inheritance is that a derived class invariably must know about the implementation of the base class. But this is crazy! The derived class really is, after all, *just another user of that base class* and should therefore not be knowledgeable about the base class implementation.

Consider subclassing a Windows edit control, something that many of us Windows programmers have tried to do with a great deal of pain. As you have probably experienced you have to know exactly what messages you can and cannot trap and override. You must also know when to call the default edit control's window procedure: do you call it before you do your own processing on the message? After? Do you call it at all? You absolutely must have knowledge about how the edit control is implemented if you have any chance of not breaking its behavior through your subclassing. This same problem manifests itself even with a simple function like DefWindowProc, which has led Microsoft to publishing at least part of the DefWindowProc source code in the Windows 3.0 and 3.1 Software Development Kits. All because you have two pieces of code, one that doesn't know anything about the other, trying to operate on the same data at the same time. A fragile relationship at best, just waiting for a fiery divorce.

This example brought up another serious problem with inheritance: two pieces of code working on the instance of an object. If I have a base class B and a derived class D that inherits from B, than an instantiation of class D is only one data structure in memory. If B contains virtual functions that D does not override, or if the implementation of D explicitly calls a member function in B (that is using B::<member function>), then we again have two pieces of code working on the same memory. But class B does not know the expected behavior of class D–so how on earth can D force the correct behavior of its objects? The answer is that D must know what B is going to do on that object such that it knows when to override a virtual function in B

and when exactly to explicitly call B's functions. This is exactly the same problem as trying to figure out when to call DefWindowProc for any given message: we've just replace the word 'message' with 'virtual function.' In any case, since B cannot know about the implementation of D, then D must know about the implementation of B which causes D to violate it's status as a user of B.

Systems built on inheritance have the key problem that they must ship all their source code in order to be usable. Take a look at application frameworks like Microsoft's Foundation Classes–source is shipped so you know how to inherit from any given class and can duplicate behavior as appropriate. Sure, inheritance works well to build large complex systems because it's a much better way to manage source code than a large stockpile of sample source files. It certainly works well when you control an have access to all the source code for all classes. It certainly helps me to develop the sample applications in this book. But it does not work for reusing object implemented in the operating system itself where source is either not available or you did not originally implement the object yourself.

The Component Object Model avoid all these problems but retains reusability through its mechanisms of containment and aggregation, which we are finally in a position to explain in detail. Both mechanisms achieve reusability literally by using, instead of inheriting, the implementation of another object. The object we're using remains entirely self-contained and operates on its own instance of data. Our own object, which is called the aggregate, works on its own instance of data and calls the other object as necessary to perform specific functions where we can pass it the data on which to operate.

Let's say I have an object called Animal that knows only of itself and exists as an atomic entity (like the Koala object). I can illustrate this object as a cube with circular jacks for each interface: IUnknown and IAnimal (with members like ::Eat, ::Sleep, and ::Procreate). Again by convention IUnknown is always shown on top with all other interfaces shown to the side:

A user of this object with a pointer to either interface can ::QueryInterface to get a pointer to the other. The implementation of IAnimal knows about the object's IUnknown and vice-versa. Now I want to create a more complicated Koala object that will expose interfaces IUnknown, IAnimal, and IMarsupial (maybe with members like ::CarryYoungInPouch and ::LiveInAustralia), with a more complicated picture:[1]

When I implement Koala, I know that Animal exists and I wish to reuse Animal's implementation. I can use Animal and its IAnimal in two ways, neither of which changes how the external world sees Koala:

1. Containment: Koala completely contains an Animal object and implements it's own of IAnimal to expose externally. This makes Koala a user of Animal and Animal need not care.
2. Aggregation: Koala exposes Animal's IAnimal interface directly as Koala's IAnimal. This requires that Animal knows that it's interface is exposed for something other than itself such that QueryInterface, AddRef, and Release behave as a user expects.

Case 1: Object Containment

Complete containment of Animal is necessary when I need to change some aspect about my implementation of IAnimal. Since all external calls to that interface will enter Koala first, Koala can override specific functions or simply pass that call to Animal's implementation. The internal structure of Koala will appear like:

In this case Animal always operates on its own data unless Koala explicitly passes other data to it (which is also true in aggregation). In other words, by default the two objects work on different data and only by conscious design of an interface would the two be able to communicate. This is different from inheritance where working on the same data is the default and it takes conscious effort to create separate data instances.

1Instinct tells you that IMarsupial should inherit from IAnimal because a marsupial is just another kind of animal. The Windows Object notion of interfaces means that through a pointer to IMarsupial you deal with the object as a marsupial but not generally as anything else. If you want to treat it like any other animal, call IMarsupial::QueryInterface(IID_IAnimal) for the appropriate interface. As a real example, consider compound document objects: they are all treated through IOleObject whether linked or embedded. A linked object is a further refinement of an embedded object, so you might expect that an interface like IOleLink for linked objects would inherit from IOleObject. But it doesn't: you QueryInterface through IOleObject for IOleLink. QueryInterface is the mechanism for getting at more functions on the same object.

To build this sort of structure, Koala calls CoCreateInstance on CLSID_Animal when Koala itself is created, asking for IID_IAnimal. Koala maintains this IAnimal pointer until the Koala object is destroyed at which time it calls IAnimal::Release to free the Animal object. Whenever Koala's IAnimal implementation wants to reuse Animal's IAnimal implementation, Koala just calls the appropriate Animal functions.

This technique is the simplest way to reuse another's implementation of an interface. However, you do not always care to override **any** functions in such an interface, wishing only to pass every call through to the object you're using. You could, of course, implement stubs for every IAnimal function that simply calls the contained object, but we would rather just expose that interface directly and eliminate any need for such stubs. That technique is aggregation.

Case 2: Object Aggregation

Aggregation on Animal is useful when Koala does not wish to change any aspect about how it appears through the IAnimal interface, that is, Koala as no need to just implement a bunch of Animal stubs that only delegate to a contained Animal object. Therefore Koala wants to expose the IAnimal interface of the Animal object directly, turning it into Koala's IAnimal. This yields an internal structure like:

Here's the problem. Since Animal's IAnimal is exposed directly, users of Koala will expect that IAnimal::QueryInterface(IID_IMarsupial) will return a pointer to Koala's IMarsupial, and that IAnimal::QueryInterface(IID_IUnknown) will return a pointer to Koala's IUnknown. But Animal was not written to know anything about IMarsupial, let alone know anything about the Koala object! How can it know the identify of the outer object?

The answer is that when Koala creates Animal, Koala passes its IUnknown pointer to the Animal class factory ::CreateInstance as the punkOuter parameter. In this fashion Koala identifies itself as the **controlling unknown** of the aggregate. Furthermore, Koala must always ask for Animal's IUnknown when creating Animal as part of an aggregate. Note that an object may not support aggregation in which case it fails ::CreateInstance when a non-NULL punkOuter is specified.

This punkOuter parameter in ::CreateInstance must be passed from the class factory to the object and made available to all interface implementations of IUnknown members. Let's look again at some of the code from the Koala object implementation again (Listing 4-3) renamed for Animal:

```
STDMETHODIMP CAnimalClassFactory(LPUNKNOWN punkOuter, ...)
  {
  LPUNKNOWN   pObj;
  ...
  pObj=new CAnimal(punkOuter, ...);    //Create the object

  ...
  pObj->FInit();                  //Initialize object
  ...
  }

CAnimal::CAnimal(LPUNKNOWN punkOuter, ...)
  {
  ...
  m_punkOuter=punkOuter;   //Save the controlling unknown
  ...
  }

BOOL CAnimal::FInit(void)
  {
  LPUNKNOWN      pIUnknown=(LPUNKNOWN)this;

  if (NULL!=m_punkOuter)
     pIUnknown=m_punkOuter;

  m_pIAnimal=new CImpIAnimal(this, pIUnknown);

  return (NULL!=m_pIAnimal);
  }
```

In this manner the Animal object knows about the controlling unknown punkOuter and save it within its own structure. When Animal goes to create its interface implementations, it plays a nasty trick on them. If there is no real controlling unknown, Animal passes its own IUnknown, the one knows all the interfaces implemented in Animal, as the controlling unknown to the interface. The interface, in turn, blindly delegates

all IUnknown member calls to whatever controlling unknown it has (besides doing interface reference counting for debugging purposes):

```
CImpIAnimal::CImpIAnimal(LPVOID pObj, LPUNKNOWN punkOuter)
   {
   ...
   m_punkOuter=punkOuter;
   ...
   }

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

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

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

So let's say there is no aggregation; the m_punkOuter to which CImpIAnimal delegates is the IUnknown implemented in CAnimal. This IUnknown implementation will return pointers for IUnknown and IPersist:

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

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

   if (IsEqualIID(riid, IID_IAnimal))
      *ppv=(LPVOID)m_pIAnimal;

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

If there is an aggregation, m_punkOuter points to the controlling unknown (the interface does not know the difference) and so calls to it initially bypass the IUnknown of Animal. This controlling unknown determines exactly which pointer is returned for which interface by executing the following steps:

1. If the requested interface is one implemented by the aggregating object and exposed externally, the controlling unknown returns a pointer to the aggregate's implementation.
2. If the requested interface is implemented within a contained object, delegate the call to that contained object's IUnknown.

The ::QueryInterface of an aggregate's controlling unknown, such as what Koala would implement, would therefore appear as:

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

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

   if (IsEqualIID(riid, IID_IMarsupial))
      *ppv=(LPVOID)m_pIMarsupial;

   if (NULL!=*ppv)
      {
```

```
    ((LPUNKNOWN)*ppv)->AddRef();
    return NOERROR;
    }
  else
    return m_pIUnknownAnimal->QueryInterface(riid, ppv);

  return ResultFromScode(E_NOINTERFACE);
  }
```

where m_pIUnknownAnimal is the IUnknown pointer we requested when creating the Animal object. The IUnknown implementation on an object like CAnimal above is always the last stop for a ::QueryInterface, ::AddRef, or ::Release call. It never worries about aggregation in itself because it has to be the controlling unknown for its own object. Only if the Animal object itself contained more primitive objects (like a TangibleThing) would it do any further delegation, but in no way will it pass any request to a higher unknown.

NOTE: Aggregation and delegating IUnknown calls applies equally to ::AddRef and ::Release although ::QueryInterface is the most interesting case. Since the user is expecting to affect the reference count of the entire aggregate object through whatever interface, all interfaces in an aggregable object must pass ::AddRef and ::Release calls up to the controlling unknown. The unknown of the contained object, however, will not see these reference counts since that contained object is nested within the lifetime of the aggregate.

Summary

A first requirement of all OLE 2.0 applications is that they use a message queue of size 96 and provide for initializing the OLE 2.0 libraries if, in fact, the application can run against the version of those libraries that currently exist on the machine. Checking versions is accomplished through the CoBuildVersion and OleBuildVersion functions whereas initialization occurs through CoInitialize and OleInitialize. On shutdown an application must also call CoUninitialize or OleUninitialize to reverse the corresponding Initialize call. These requirements are presented in this chapter because all later samples must comply with them.

Part of library initialization involves defining a task allocator object, one that implements the IMalloc interface, that is used for all task memory allocations. An OLE 2.0 application can either implement its own or use an OLE 2.0-provided allocator that works on the technique of multiple-local heaps. OLE 2.0 always implements a similar shared allocator that can provide memory sharable between applications. While applications can change the task allocator they cannot change the shared allocator. While the application is running, any other piece of code (such as the OLE 2.0 libraries themselves) may call CoGetMalloc to obtain a pointer to either the task or shared allocator objects.

The Ultimate Question presented in Chapter 3 asked how you obtain a pointer given knowledge and the identification of a specific Windows Object. This chapter deals with the specific case where you identify a Windows Object given a CLSID and use the function CoCreateInstance to instantiate an object of that class. Such an object is called a Component Object and the application using it is called a Component User. CoCreateInstance internally uses CoGetClassObject which obtains a class factory object (IClassFactory) for the CLSID and call IClassFactory::CreateInstance to perform the actual instantiation. What you do with the object once you have an interface pointer on it is your own business, although there are a few considerations when you release the object, such as calling CoFreeUnusedLibraries to purge unused DLLs from memory.

Implementing Component Objects that can be loaded and called by functions like CoCreateInstance and CoGetClassObject have different structural requirements depending on whether the object lives in a DLL or EXE. A DLL exports a function called DllGetClassObject that provides the API through which CoGetClassObject obtains a pointer to the DLL's class factory for a given CLSID. EXEs instead must pass a pointer to their class factory objects to the CoRegisterClassObject API for each supported CLSID. The two module types also differ in their shutdown conditions. Whereas the component object library asks a DLL if that DLL can be unloaded, and EXE must initiate its own shutdown when the proper conditions are met, that is, destroy its main window and exit its message loop. As examples, this chapter implements an object called Koala that supports the IPersist interface in both a DLL and EXE, then separates the Polyline object of the sample Schmoo application into a component object and shows a modification of Schmoo, called Component Schmoo, that uses the component Polyline object.

Object reusability in OLE 2.0 is achieved through mechanisms called Containment and Aggregation, not inheritance. The inheritance mechanism works well for source code management but generally requires that you have the source code available for any classes from which you inherit. Because of source availability and a host of other problems, OLE 2.0 works on mechanisms other than inheritance that provide the same

reusability of code, but avoids the problems with traditional techniques. There is, however, some impact on the implementation of an object that wants to allow itself to be reusable via containment and aggregation.