Chapter Seven
Clipboard Transfers with Data Objects

What do you do with a data object?
What do you do with a data object?
What do you do with a data object?
Put it on the clipboard!

Ancient Sailor Hymn

As we've seen in Chapter 6, data objects encompass the functionality in all the existing data transfer protocols. Using raw data objects by themselves to exchange data between applications is rich enough to replace DDE entirely. This chapter now looks at how you use data objects to place data on the clipboard and to retrieve data from the clipboard.

Conceptually the operations are simple: to place data on the clipboard you pass an IDataObject pointer to OLE2.DLL, which asks that data object about its supported formats and calls IDataObject::GetData when some consumer actually wants that data. To retrieve data from the clipboard you call an OLE2.DLL function that returns you an IDataObject pointer through which you can ::GetData and so forth. In other words, the consumer of the data is the user of an IDataObject pointer that it retrieves by calling a specific function.

Simple? On the surface yet, but it seems like a tremendous overkill to have to create an entire data object and an EnumFORMATETC object just to do something simple like copy a small piece of text to the clipboard. Since this is a major consideration for just about every developer I know, the first part of this chapter will address just how much of a data object you need for clipboard transfers as well as provide the implementation of a DLL component object that greatly simplifies creation of a data object for such transfers.

I'll apply this data transfer object DLL for implementations in both Schmoo and Patron. Schmoo is the simplest one through which we can demonstrate use of the three OLE 2.0 clipboard APIs which replace use of the Windows clipboard APIs. Like we paralleled file I/O to IStream member functions in Chapter 5, here we can parallel traditional clipboard APIs to use of the OLE 2.0 APIs and a data transfer object.

For Component Schmoo we'll take a slightly different approach: instead of creating a separate data transfer object, it creates a copy Polyline object that holds a snapshot of the same data as the current polyline. We can then just throw the Polyline's IDataObject on the clipboard. In essence this is doing exactly what we do with the data transfer object, just on a more application-specific basis.

Finally we'll implement clipboard operations for Patron, enabling it to copy and paste bitmaps and metafiles using the Freeloader code implemented in Chapter 6. All the work we do here will set Patron up for drag-drop features in Chapter 8 and for minor changes to become a compound document container in Chapter 9. The additions to Patron in this chapter will be significant as we implement the idea of tenants in each page.

Let me make it absolutely clear that at this time you are not in any way required to change your current clipboard implementation just to do clipboard operations. OLE 2.0 offers an alternate way to deal with the clipboard using data objects, but you can continue to use APIs for existing data formats. Where data object transfers become important is when we deal with drag-drop and compound documents. For drag-drop that we'll see in the next chapter, you can reuse a lot of the code you create for clipboard operations in this chapter. With the clipboard code in place, drag-drop implementation can be almost trivial. Data objects will be required for compound document implementations in Chapters 9 and 10, so if you're planning to pursue these goals I do suggest that you work these clipboard changes into your code now. This will save you a considerable amount of work later and will reduce the amount that you have to absorb at once. Finally, if you are only going to be a source of data (that is, consume no external data, like a data collection front-end) and you are not interested in compound documents and have no need for drag-drop features, then really any data transfer using data objects at all could be an overkill. Don't feel compelled to change your code 'just because.' Find a reason first.

With that little bit of preaching, let's move directly to OLE 2.0 functions, interfaces, and code examples for clipboard operations.

The OLE 2.0 Clipboard Protocol

As described in Chapter 6, a data transfer protocol in the OLE 2.0 sense is some mechanism to communicate a data object pointer, that is an IDataObject pointer, from the source of the data to a consumer of the data. The clipboard is one such communication mechanism and to support it, OLE 2.0 provides three clipboard related APIs in OLE2.DLL:

OleSetClipboard	Places a data object pointer on the clipboard. The given data object is tied to all the

	data that you want to copy. This function will AddRef the IDataObject pointer.
OleGetClipboard	Retrieves a data object pointer that represents the data available on the clipboard. This function will AddRef the pointer before returning, so the consuming application must call ::Release when finished with the pointer.
OleFlushClipboard	Clears a data object from the clipboard, calling ::Release on the data object to reverse the ::AddRef from OleSetClipboard.

The overall mechanism of a clipboard data transfer in OLE 2.0 is shown in Figure 7-1. The source application creates a data object, as well as the enumerator for FORMATETCs, and calls OleSetClipboard. In turn, OLE, that is OLE2.DLL, calls OpenClipboard, IDataObject::AddRef and IDataObject::EnumFormatEtc. For each enumerator format, it the calls the Windows API SetClipboardData(formatetc.cfFormat, NULL) to mark that format as available on the clipboard without actually rendering the data, then calls CloseClipboard. When any other piece of code generates a call to GetClipboardData on any of these formats, the Windows clipboard handler generates a WM_RENDERFORMAT message. But to what window?

Figure 7-1: The OLE 2.0 clipboard mechanisms involving the APIs OleGetClipboard and OleSetClipboard as well as an IDataObject implementation.

The window that receives this message is actually owned by OLE2.DLL and is kept hidden at all times. This window is created when you call OleInitialize, which is why you must use OleInitialize when using OLE 2.0 clipboard support rather than just the simpler CoInitialize.[1] Regardless of how many times OleInitialize is called for a given task, only one clipboard-handling window of this type is created. The window also remains hidden at all times.

When this window receives WM_RENDERFORMAT, only then does it generate calls to IDataObject::QueryGetData (just to make sure about the format) and IDataObject::GetData. In other words, OLE 2.0 provides delayed clipboard rendering with data objects. This does, however, have an impact on data sources, as delayed rendering means that you must snapshot your current data into the data object, regardless of whether you are implementing a Cut or Copy operation. I'll treat this subject in a little more detail in a bit.

The consumer of data does not, of course, have to deal with the complications of delayed rendering. A consumer makes a single call to OleGetClipboard to obtain a pointer to an IDataObject interface. The pointer returned, however, is not the exact data object that a source placed there because data may have been placed on the clipboard by an application that does not know OLE 2.0. So OleGetClipboard returns an OLE2.DLL owned data object pointer that represents the data on the clipboard regardless of how that data got there. In any case, once the consumer calls OleGetClipboard, it can call IDataObject::EnumFormatEtc,[2] ::QueryGetData, and ::GetData as it would call EnumClipboardFormats, IsClipboardFormatAvailable, and GetClipboardData. Only ::GetData calls will generate WM_RENDERFORMAT messages and a ::GetData call to the real source data object. As far as a data consumer is concerned, changing from clipboard handling using the Windows APIs to using the OleGetClipboard and the IDataObject interface takes minimal effort as we shall see.

I should mention here that the clipboard data object implemented in OLE2.DLL is quite limited as it only implements IDataObject::GetData, ::GetDataHere (for only limited formats), ::QueryGetData, and ::EnumFormatEtc. All other functions either return OLE_E_NOTSUPPORTED (::SetData and advise functions) or in the case of ::GetCanonicalFormatEtc, NOERROR. In addition, ::QueryInterface will only acknowledge IUnknown and IDataObject, so forget any hopes that the data object here will know how to draw or save itself. That sort of functionality must be implemented using the techniques demonstrated in the Freeloader example of Chapter 6.

But All I Want To Do is Copy Some Simple Data!

I imagine at this point that you might be screaming or cursing because clipboard data transfers under OLE 2.0 seem to have become much more complex than you would like. To even copy the simplest piece of data to the clipboard, say a short but passionate string like "*Why* is Microsoft doing this to me?", involves an implementation of a data object, an implementation of a FORMATETC enumerator, and the complexity of taking a snapshot of the data involved in the operation to handle delayed rendering. Ouch! Whatever happened to OpenClipboard, SetClipboardData, and

---

1If you have no reason to call OleInitialize, then continue to use CoInitialize: OleInitialize resides in OLE2.DLL and would require loading that DLL just for this one call. On the other hand, CoInitialize requires only loading COMPOBJ.DLL which is smaller and more optimized than OLE2.DLL.

2I noticed in my work on Patron for this chapter that EnumFormatEtc will fail on the IDataObject interface on a static object, that is, and object of CLSID_StaticMetafile or CLSID_StaticDib as used in Freeloader in Chapter 6.

CloseClipboard?

Well, in all reality it has never been *quite* that simple because somewhere along here you have to allocate some global memory that contains the data you want to copy. For instance, before calling SetClipboardData for the example string above you would probably have some function to make a global memory copy of that string:

```
HGLOBAL CopyStringToHGlobal(LPSTR psz)
   {
   HGLOBAL    hMem;
   LPSTR      pszDst;

   hMem=GlobalAlloc(GHND, (DWORD)(lstrlen(psz)+1));

   if (NULL!=hMem)
      {
      pszDst=GlobalLock(hMem);
      lstrcpy(pszDst, psz);
      GlobalUnlock(hMem);
      }

   return hMem;
   }
```

With that the code to copy some text to the clipboard would be as follows:

```
HGLOBAL    hMem;

hMem=CopyStringToHGlobal("Why is Microsoft doing this to me?");

if (NULL!=hMem)
   {
   if (OpenClipboard(hWndMain))
      {
      SetClipboardData(CF_TEXT, hMem);
      CloseClipboard();
      }
   else
      GlobalFree(hMem);  //We must clean up.
   }

...
```

Under OLE 2.0, we would like to write code that looks like that below, showing a strong resemblance to the Windows API code shown above:

```
LPDATAOBJECT    pIDataObject;
HRESULT         hr;
FORMATETC       fe;
STGMEDIUM       stm;
```

```
stm.tymed=TYMED_HGLOBAL;
stm.hGlobal=CopyStringToHGlobal("Why is Microsoft doing this to me?");

if (NULL!=stm.hGlobal)
   {
   hr=FunctionToCreateADataTransferObject(&pIDataObject)

   if (SUCCEEDED(hr))
      {
      SETDefFormatEtc(fe, CF_TEXT, TYMED_HGLOBAL);
      pIDataObject->SetData(&fe, &stm);
      OleSetClipboard(pIDataObject);
      pIDataObject->Release();   //Our ref count on this pointer
      }
   else
      GlobalFree(stm.hGlobal);
   }

...
```

This code shows how we would *like* to translate our existing Windows code into OLE 2.0 code. SetClipboardData with a clipboard format and a memory handle turns into an IDataObject::SetData with a FORMATETC and a STGMEDIUM followed by OleSetClipboard. CloseClipboard turns into an IDataObject::Release call. But what's the FunctionToCreateSomeDataTransferObject? That's not in the OLE 2.0 Programmer's Reference?

That's right, even though in my opinion it's a key feature left out of the OLE 2.0 implementation that would greatly simplify clipboard transfers using data objects. A function like this would create some data object in which we could stuff data renderings (that is, a FORMATETC and a STGMEDIUM) to copy to the clipboard using ::SetData. It would assume ownership of each rendering and maintain it's own reference count, making sure to call ReleaseStgMedium for each rendering when it's reference was reduced to zero.

But alas, OLE 2.0 does not provide such a useful tool, so I implemented one for this book, in the form of a component object DLL

A Data Transfer Component Object

An object that would simplify clipboard transfers is essentially a data cache with an IDataObject interface slapped on top of it. I had considered using the IOleCache interface on a default handler object (from OleCreateDefaultHandler) but that would require calls to IOleCache::Cache and IOleCache::SetData, then IOleCache::QueryInterface for IDataObject. What I really wanted was a data object that I would grab using something like CoCreateInstance. Just this object, housed in DATATRAN.DLL, is shown in Listing 7-1 and found in the sample code under CHAP07\DATATRAN.

DATATRAN.CPP

```
/*
 * DATATRAN.CPP
 *
 * Transfer data object implemented in a DLL.  This data object will
 * cache specific formats and renderings such that its IDataObject
 * interface could be plopped on the clipboard or used in drag-drop.
 *
```

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

#define INITGUIDS
#include "datatran.h"

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

//Make this global for the data object to create listboxes.
HINSTANCE   g_hInst=NULL;


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

    g_hInst=hInst;
    return hInst;
    }

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



/*
 * DllGetClassObject
 *
 * Purpose:
 *  Provides an IClassFactory for a given CLSID that this DLL is
 *  registered to support.  This DLL is placed under the CLSID
 *  in the registration database as the InProcServer.
 */

HRESULT __export FAR PASCAL DllGetClassObject(REFCLSID rclsid, REFIID riid
    , LPVOID FAR *ppv)
    {
    if (!IsEqualIID(riid, IID_IUnknown) && !IsEqualIID(riid, IID_IClassFactory))
        return ResultFromScode(E_NOINTERFACE);
```

```
   *ppv=NULL;

   if (IsEqualCLSID(rclsid, CLSID_DataTransferObject))
      *ppv=(LPVOID)new CDataTransferClassFactory();

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

   ((LPUNKNOWN)*ppv)->AddRef();
   return NOERROR;
   }



/*
 * DllCanUnloadNow
 *
 * Purpose:
 *  Answers if the DLL can be freed, that is, if there are no
 *  references to anything this DLL provides.
 */

STDAPI DllCanUnloadNow(void)
   {
   SCODE   sc;

   sc=(0L==g_cObj && 0==g_cLock) ? S_OK : S_FALSE;
   return ResultFromScode(sc);
   }




/*
 * ObjectDestroyed
 *
 * Purpose:
 *  Function for the DataObject 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;
```

```
    }



/*
 * CDataTransferClassFactory::CDataTransferClassFactory
 * CDataTransferClassFactory::~CDataTransferClassFactory
 *
 * Constructor Parameters:
 *  None
 */

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


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




/*
 * CDataTransferClassFactory::QueryInterface
 * CDataTransferClassFactory::AddRef
 * CDataTransferClassFactory::Release
 */

STDMETHODIMP CDataTransferClassFactory::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) CDataTransferClassFactory::AddRef(void)
    {
    return ++m_cRef;
    }


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

    cRefT=--m_cRef;

    if (0L==m_cRef)
        delete this;

    return cRefT;
    }




/*
 * CDataTransferClassFactory::CreateInstance
 * CDataTransferClassFactory::LockServer
 *
 * IClassFactory members.
 */

STDMETHODIMP CDataTransferClassFactory::CreateInstance(LPUNKNOWN punkOuter
    , REFIID riid, LPVOID FAR *ppvObj)
    {
    LPCDataObject      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 the data size to work with
   pObj=new CDataObject(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 CDataTransferClassFactory::LockServer(BOOL fLock)
   {
   if (fLock)
      g_cLock++;
   else
      g_cLock--;

   return NOERROR;
   }



/*
 * DATAOBJ.CPP
 * Data Transfer Object for Chapter 7
 *
 * Implementation of the CDataObject for the Data Transfer Component Object.
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "dataobj.h"

extern HINSTANCE g_hInst;
```

```
/*
 * CDataObject::CDataObject
 * CDataObject::~CDataObject
 *
 * Parameters (Constructor):
 *  punkOuter       LPUNKNOWN of a controlling unknown, if it exists.
 *  pfnDestroy      LPFNDESTROYED to call when an object is destroyed.
 */

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

   m_hList=NULL;

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

   return;
   }


CDataObject::~CDataObject(void)
   {
   if (NULL!=m_pIDataObject)
      delete m_pIDataObject;

   Purge();

   if (NULL!=m_hList)
      DestroyWindow(m_hList);

   return;
   }



/*
 * CDataObject::FInit
 */

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

```
    LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;

    if (NULL!=m_punkOuter)
        pIUnknown=m_punkOuter;

    //Allocate contained interfaces.
    m_pIDataObject=new CImpIDataObject(this, pIUnknown);

    if (NULL==m_pIDataObject)
        return FALSE;

    m_hList=CreateWindow("listbox", "renderings", WS_POPUP | LBS_OWNERDRAWFIXED
        , 0, 0, 100, 100, HWND_DESKTOP, NULL, g_hInst, NULL);

    if (NULL==m_hList)
        return FALSE;

    return TRUE;
    }



/*
 * CDataObject::Purge
 *
 * Purpose:
 *  Cleans out all entries in our listbox.
 */

void CDataObject::Purge(void)
    {
    UINT        i, cItems;
    LPRENDERING pRen;
    DWORD       cb;

    if (NULL==m_hList)
        return;

    cItems=(UINT)SendMessage(m_hList, LB_GETCOUNT, 0, 0L);

    for (i=0; i < cItems; i++)
        {
        cb=SendMessage(m_hList, LB_GETTEXT, i, (LPARAM)(LPVOID)&pRen);

        if (sizeof(LPRENDERING)==cb)
            {
```

```
            /*
             * Release the data completely being sure to reinstate
             * the original pUnkForRelease.
             */
            pRen->stm.pUnkForRelease=pRen->pUnkOrg;
            ReleaseStgMedium(&pRen->stm);
            delete pRen;
            }
        }

    SendMessage(m_hList, LB_RESETCONTENT, 0, 0L);
    return;
    }




/*
 * CDataObject::QueryInterface
 * CDataObject::AddRef
 * CDataObject::Release
 *
 * Purpose:
 *  IUnknown members for CDataObject object.
 */

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

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

    if (IsEqualIID(riid, IID_IDataObject))
        *ppv=(LPVOID)m_pIDataObject;

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

    return ResultFromScode(E_NOINTERFACE);
```

```
    }


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


STDMETHODIMP_(ULONG) CDataObject::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;
    }
```

IDATAOBJ.CPP

```
/*
 * IDATAOBJ.CPP
 * Data Tranfer Object for Chapter 7
 *
 * Implementation of the IDataObject interface for CDataObject.
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "dataobj.h"

/*
 * CImpIDataObject::CImpIDataObject
```

```
 * CImpIDataObject::~CImpIDataObject
 *
 * Parameters (Constructor):
 *  pObj            LPVOID of the object we're in.
 *  punkOuter       LPUNKNOWN to which we delegate.
 */

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

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


/*
 * CImpIDataObject::QueryInterface
 * CImpIDataObject::AddRef
 * CImpIDataObject::Release
 *
 * Purpose:
 *  IUnknown members for CImpIDataObject object.
 */

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


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

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

```
   }



/*
 * CImpIDataObject::GetData
 * CImpIDataObject::QueryGetData
 * CImpIDataObject::SetData
 * CImpIDataObject::EnumFormatEtc
 *
 * Substantial member functions.
 */

STDMETHODIMP CImpIDataObject::GetData(LPFORMATETC pFE, LPSTGMEDIUM pSTM)
   {
   UINT       i, cItems;
   LPRENDERING pRen;
   DWORD      cb;
   HWND       hList;

   if (NULL==m_pObj->m_hList || NULL==pFE || NULL==pSTM)
      return ResultFromScode(DATA_E_FORMATETC);

   hList=m_pObj->m_hList;
   cItems=(UINT)SendMessage(hList, LB_GETCOUNT, 0, 0L);

   for (i=0; i < cItems; i++)
      {
      cb=SendMessage(hList, LB_GETTEXT, i, (LPARAM)(LPVOID)&pRen);

      if (sizeof(LPRENDERING)==cb)
         {
         /*
          * Check if the requested FORMATETC is the same as one
          * that we already have.  If so, then copy that STGMEDIUM
          * to pSTM and AddRef ourselves for pUnkForRelease.
          */
         if (pFE->cfFormat==pRen->fe.cfFormat
            && (pFE->tymed & pRen->fe.tymed)
            && pFE->dwAspect==pRen->fe.dwAspect)
            {
            *pSTM=pRen->stm;
            AddRef();
            return NOERROR;
            }
         }
```

```
    }

  return ResultFromScode(DATA_E_FORMATETC);
  }



STDMETHODIMP CImpIDataObject::QueryGetData(LPFORMATETC pFE)
  {
  UINT       i, cItems;
  LPRENDERING pRen;
  DWORD      cb;
  HWND       hList;

  if (NULL==m_pObj->m_hList || NULL==pFE)
     return ResultFromScode(S_FALSE);

  hList=m_pObj->m_hList;
  cItems=(UINT)SendMessage(hList, LB_GETCOUNT, 0, 0L);

  for (i=0; i < cItems; i++)
     {
     cb=SendMessage(hList, LB_GETTEXT, i, (LPARAM)(LPVOID)&pRen);

     if (sizeof(LPRENDERING)==cb)
        {
        /*
         * Check if the requested FORMATETC is the same as one
         * that we already have.
         */
        if (pFE->cfFormat==pRen->fe.cfFormat
           && (pFE->tymed & pRen->fe.tymed)
           && pFE->dwAspect==pRen->fe.dwAspect)
           {
           return NOERROR;
           }
        }
     }

  return ResultFromScode(S_FALSE);
  }


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

```
   LPRENDERING     prn;

   //We have to remain responsible for the data.
   if (!fRelease)
      return ResultFromScode(E_FAIL);

   //If we're handed NULLs, that means clean out the list.
   if (NULL==pFE || NULL==pSTM)
      {
      m_pObj->Purge();
      return NOERROR;
      }

   /*
    * Here we take the rendering we're given and attach it to the
    * end of the list.  We save the original pSTM->pUnkForRelease and
    * replace it with our own such that each 'copy' of this data
    * of actually just a reference count.
    */

   prn=new RENDERING;

   if (NULL==prn)
      return ResultFromScode(E_OUTOFMEMORY);

   prn->fe=*pFE;
   prn->stm=*pSTM;
   prn->pUnkOrg=pSTM->pUnkForRelease;
   prn->stm.pUnkForRelease=(LPUNKNOWN)this;

   SendMessage(m_pObj->m_hList, LB_ADDSTRING, 0, (LONG)(LPVOID)prn);
   return NOERROR;
   }



STDMETHODIMP CImpIDataObject::EnumFormatEtc(DWORD dwDir
   , LPENUMFORMATETC FAR *ppEnum)
   {
   LPCEnumFormatEtc    pEnum;

   *ppEnum=NULL;

   /*
    * From an external point of view there are no SET formats,
    * because we want to allow the user of this component object
```

```
    * to be able to stuff ANY format in via Set.  Only external
    * users will call EnumFormatEtc and they can only Get.
    */

   switch (dwDir)
      {
      case DATADIR_GET:
          pEnum=new CEnumFormatEtc(m_punkOuter);
          break;

      case DATADIR_SET:
      default:
         pEnum=NULL;
         break;
      }

   if (NULL==pEnum)
      return ResultFromScode(E_FAIL);
   else
      {
      //Let the enumerator copy our format list.
      if (!pEnum->FInit(m_pObj->m_hList))
         {
         delete pEnum;
         return ResultFromScode(E_FAIL);
         }

      pEnum->AddRef();
      }

   *ppEnum=pEnum;
   return NOERROR;
   }




/*
 * CImpIDataObject::GetDataHere
 * CImpIDataObject::GetCanonicalFormatEtc
 * CImpIDataObject::DAdvise
 * CImpIDataObject::DUnadvise
 * CImpIDataObject::EnumDAdvise
 *
 * Trivial member functions.
 */
```

```
STDMETHODIMP CImpIDataObject::GetDataHere(LPFORMATETC pFE, LPSTGMEDIUM pSTM)
    {
    return ResultFromScode(E_NOTIMPL);
    }


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

//No advise support for this sort of data transfer.
STDMETHODIMP CImpIDataObject::DAdvise(LPFORMATETC pFE, DWORD dwFlags
    , LPADVISESINK pIAdviseSink, LPDWORD pdwConn)
    {
    return ResultFromScode(E_FAIL);
    }

STDMETHODIMP CImpIDataObject::DUnadvise(DWORD dwConn)
    {
    return ResultFromScode(E_FAIL);
    }

STDMETHODIMP CImpIDataObject::EnumDAdvise(LPENUMSTATDATA FAR *ppEnum)
    {
    return ResultFromScode(E_FAIL);
    }
```

IENUMFE.CPP

```
/*
 * IENUMFE.CPP
 * Data Tranfer Object for Chapter 7
 *
 * IEnumFORMATETC implementation specifically for the Data Tranfer
 * objects.  This enumerator copies the state of the data list in
 * the data object and uses that to maintain what FORMATETCs it knows.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "dataobj.h"
```

```
extern HINSTANCE g_hInst;


/*
 * CEnumFormatEtc::CEnumFormatEtc
 * CEnumFormatEtc::~CEnumFormatEtc
 *
 * Parameters (Constructor):
 *  punkRef         LPUNKNOWN to use for reference counting.
 */

CEnumFormatEtc::CEnumFormatEtc(LPUNKNOWN punkRef)
    {
    m_cRef=0;
    m_punkRef=punkRef;
    m_iCur=0;
    m_cItems=0;
    return;
    }


CEnumFormatEtc::~CEnumFormatEtc(void)
    {
    if (NULL!=m_prgfe)
        {
        LPMALLOC    pIMalloc;

        if (SUCCEEDED(CoGetMalloc(MEMCTX_TASK, &pIMalloc)))
            {
            pIMalloc->Free(m_prgfe);
            pIMalloc->Release();
            }
        }

    return;
    }



/*
 * CEnumFormatEtc::FInit
 */

BOOL CEnumFormatEtc::FInit(HWND hList)
    {
```

```
   UINT        i, cItems;
   LPMALLOC    pIMalloc;
   LPRENDERING pRen;
   LPFORMATETC pFE;
   DWORD       cb;

   if (NULL==hList)
      return FALSE;

   cItems=(UINT)SendMessage(hList, LB_GETCOUNT, 0, 0L);

   if (FAILED(CoGetMalloc(MEMCTX_TASK, &pIMalloc)))
      return FALSE;

   m_prgfe=(LPFORMATETC)pIMalloc->Alloc(cItems*sizeof(FORMATETC));
   pIMalloc->Release();

   if (NULL!=m_prgfe)
      {
      pFE=m_prgfe;

      for (i=0; i < cItems; i++)
         {
         cb=SendMessage(hList, LB_GETTEXT, i, (LPARAM)(LPVOID)&pRen);

         if (sizeof(LPRENDERING)==cb)
            {
            //Copy just the FORMATETC
            *pFE++=pRen->fe;
            m_cItems++;
            }
         }
      }

   return TRUE;
   }



/*
 * CEnumFormatEtc::QueryInterface
 * CEnumFormatEtc::AddRef
 * CEnumFormatEtc::Release
 *
 * Purpose:
 *  IUnknown members for CEnumFormatEtc object.  For QueryInterface
```

```
*  we only return out own interfaces and not those of the data object.
*  However, since enumerating formats only makes sense when the data
*  object is around, we insure that it stays as long as we stay by
*  calling an outer IUnknown for ::AddRef and ::Release.  But since we
*  are not controlled by the lifetime of the outer object, we still keep
*  our own reference count in order to free ourselves.
*/

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

   /*
    * Enumerators do not live on the same level as the data object, so
    * we only need to support out IUnknown and IEnumFORMATETC interfaces
    * here with no concern for aggregation.
    */
   if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IEnumFORMATETC))
      *ppv=(LPVOID)this;

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

   return ResultFromScode(E_NOINTERFACE);
   }


STDMETHODIMP_(ULONG) CEnumFormatEtc::AddRef(void)
   {
   ++m_cRef;
   m_punkRef->AddRef();
   return m_cRef;
   }

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

   cRefT=--m_cRef;

   m_punkRef->Release();
```

```
   if (0==m_cRef)
      delete this;

   return cRefT;
   }


/*
 * CEnumFormatEtc::Next
 * CEnumFormatEtc::Skip
 * CEnumFormatEtc::Reset
 * CEnumFormatEtc::Clone
 *
 * Standard enumerator members for IEnumFORMATETC
 */

STDMETHODIMP CEnumFormatEtc::Next(ULONG cFE, LPFORMATETC pFE
   , ULONG FAR * pulFE)
   {
   ULONG          cReturn=0L;

   if (NULL==m_prgfe)
      return ResultFromScode(S_FALSE);

   if (NULL!=pulFE)
      *pulFE=0L;

   if (NULL==pFE || m_iCur >= m_cItems)
      return ResultFromScode(S_FALSE);

   while (m_iCur < m_cItems && cFE > 0)
      {
      *pFE++=m_prgfe[m_iCur++];
      cReturn++;
      cFE--;
      }

   if (NULL!=pulFE)
      *pulFE=(cReturn-cFE);

   return NOERROR;
   }


STDMETHODIMP CEnumFormatEtc::Skip(ULONG cSkip)
   {
```

```
   if ((m_iCur+cSkip) >= m_cItems)
      return ResultFromScode(S_FALSE);

   m_iCur+=cSkip;
   return NOERROR;
   }


STDMETHODIMP CEnumFormatEtc::Reset(void)
   {
   m_iCur=0;
   return NOERROR;
   }


STDMETHODIMP CEnumFormatEtc::Clone(LPENUMFORMATETC FAR *ppEnum)
   {
   LPCEnumFormatEtc    pNew;
   LPMALLOC            pIMalloc;
   LPFORMATETC         prgfe;
   BOOL                fRet=TRUE;
   ULONG               cb;

   *ppEnum=NULL;

   //Copy the memory for the list.
   if (FAILED(CoGetMalloc(MEMCTX_TASK, &pIMalloc)))
      return ResultFromScode(E_OUTOFMEMORY);

   cb=m_cItems*sizeof(FORMATETC);
   prgfe=(LPFORMATETC)pIMalloc->Alloc(cb);

   if (NULL!=prgfe)
      {
      //Copy the formats
      _fmemcpy(prgfe, m_prgfe, cb);

      //Create the clone
      pNew=new CEnumFormatEtc(m_punkRef);

      if (NULL!=pNew)
         {
         pNew->m_iCur=m_iCur;
         pNew->m_prgfe=prgfe;
         pNew->AddRef();
         fRet=TRUE;
```

```
        }
    }

pIMalloc->Release();

*ppEnum=pNew;
return fRet ? NOERROR : ResultFromScode(E_OUTOFMEMORY);
}
```

Listing 7-1: Code that implements the DATATRAN component object, including the IDataObject and IEnumFORMATETC interfaces.

DATATRAN is a component object DLL like those described in Chapter 4. It implements a class factory object along with the DllGetClassObject and DllCanUnloadNow exports (DATATRAN.H and DATATRAN.CPP). The object is implemented using a C++ class called CDataObject (DATAOBJ.H, .CPP) which holds a pointer to the IDataObject implementation from CImpIDataObject (IDATAOBJ.CPP). We also use the same CEnumFormatEtc that we implemented in Chapter 6 (IENUMFE.CPP).

To this object we need to assign a CLSID, defined in BOOKGUID.H as CLSID_DataTransferObject. The file CHAP07\CHAP07.REG will create the appropriate registration database entries for this DLL which acts as an "InprocServer" for the object class.

So now if we're writing code that wants to use this object, we can call CoCreateInstance as the "SomeFunctionToCreateADataTransferObject" using CLSID_DataTransferObject and IID_IDataObject. CoCreateInstance will load DATATRAN.DLL, call DllGetClassObject, and call our IClassFactory::CreateInstance (CDataTransferClassFactory::CreateInstance) as we would expect, where the class factory instantiates an object of class CDataObject and calls its FInit function.

CDataObject::FInit performs two functions: create an interface implementation of IDataObject and create a listbox (via CreateWindow) in which this object will track which data renderings it currently holds. This listbox is created as LBS_OWNERDRAWFIXED since we'll only be using it to store pointers to a structure called RENDERING. This structure as shown in DATAOBJ.H contains the FORMATETC of the rendering, the STGMEDIUM referencing the data, and an LPUNKNOWN for reasons that will become clear in a moment.

These RENDERING structures are allocated and stored in the listbox in the implementation of IDataObject::SetData. Whatever FORMATETC and STGMEDIUM structures are passed into SetData we simply copy into this RENDERING structure with one exception. We replace the *pUnkForRelease* field of the STGMEDIUM with a pointer to ourselves since we now own the data. However, we must still preserve the original *pUnkForRelease* such that we can restore the STGMEDIUM to its original state before we try to release it. Once the RENDERING is completely filled, we then add it to the list of formats in the listbox with a simple LB_ADDSTRING message. Note that we always place the data given to SetData at the end of the list, so when using this data object we want to make sure that we call SetData in the same order as we would call SetClipboardData.

Implementations of functions like QueryGetData and EnumFormatEtc are fairly trivial since we already have the formats in a list, and implementation of the advise functions, GetDataHere, and GetCanonicalFormatEtc can be left empty. QueryGetData needs only to walk through the listbox item by item looking for a match. EnumFormatEtc still needs to create an enumerator object, but in initializing that object we pass the listbox such that the enumerator can walk the list and create its array of FORMATETC structures.

That leaves ::GetData and ::Release as the only remaining points of interest in this component object. ::GetData walks through the list looking for a match on the requested FORMATETC. If it finds the requested rendering, it doesn't make a copy of the actual data but copies the rendering's STGMEDIUM into the out-parameter for the caller. What's strange is the odd AddRef call in here. Remember that the STGMEDIUM we provide to the caller here contains our own IUnknown in pUnkForRelease. That means that whenever the data is no longer necessary, OLE will call ReleaseStgMedium and we'll receive a call to ::Release. As far as the consumer is concerned, they've freed the data and its memory, but in actuality we're reference counting each rendering. Only when we reset or free the entire data object will we actually free the data.

This shows a very good use for pUnkForRelease besides being able to control how the data is freed. In this case we're overriding the *ownership* of the data, taking over from the real source. Just before we want to actually free the data (as far as we're concerned) with ReleaseStgMedium, we restore original ownership. This is done in the function CDataObject::Purge (an internal function in this class) that cleans up all the RENDERING allocations in the listbox. Purge is called from two places: first is when SetData is called with NULL pointers and second when the data object itself is

destroyed.

If You Already Have a Data Object...Component Schmoo

You may run into a situation where some object you already have has an IDataObject interface on it. For example, the Polyline object used from Component Schmoo already implements IDataObject. Therefore it's a little unnecessary to create or use a whole new object (and class) just to handle data transfers. It makes more sense to just use that existing implementation.

Remember, however, that you must make some sort of snapshot of the data you wish to attach to the data object. For Component Schmoo, it means that we cannot just take the IDataObject pointer for its current Polyline and throw that up on the clipboard. If we did, then you have a potential for extreme end-user confusion. End-users will expect that the data they would later paste is exactly the same as what they copied. If Component Schmoo places the visible Polyline's IDataObject on the clipboard, then that data object is really 'live' instead of a snapshot since any change made to the Polyline would be reflected in a later paste operation. What the user saw at the time of the copy would not be what's seen at the time of paste. Not good.

What Component Schmoo does instead is to instantiate a new hidden Polyline object as the Data Transfer object. CoSchmoo creates this new object with the same dimensions as the visible one and copies the current Polyline's data into it. A perfect snapshot. Then it can toss the new hidden Polyline's IDataObject to the clipboard as shown in the code below taken from CHAP07\COSCHMOO\DOCUMENT.CPP:

```
BOOL CSchmooDoc::FClip(HWND hWndFrame, BOOL fCut)
   {
   //CHAPTER7MOD
   LPPOLYLINE          pPL;
   LPDATAOBJECT        pDataSrc, pDataDst;
   FORMATETC           fe;
   STGMEDIUM           stm;
   BOOL             fRet=TRUE;
   HRESULT             hr;
   RECT             rc;

   //Create a transfer Polyline Object
   hr=CoCreateInstance(CLSID_Polyline6, NULL, CLSCTX_INPROC_SERVER
      , IID_IPolyline6, (LPVOID FAR *)&pPL);

   if (FAILED(hr))
      return FALSE;

   //Make the new transfer Polyline same size as the current one.
   m_pPL->RectGet(&rc);

   if (FAILED(pPL->Init(m_hWnd, &rc, WS_CHILD, ID_POLYLINE)))
      {
      pPL->Release();
      return FALSE;
      }

   //Copy the data.
   m_pPL->QueryInterface(IID_IDataObject, (LPVOID FAR *)&pDataSrc);

   SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL);
   fRet=SUCCEEDED(pDataSrc->GetData(&fe, &stm));
```

```
pDataSrc->Release();

if (!fRet)
   {
   pPL->Release();
   return FALSE;
   }

pPL->QueryInterface(IID_IDataObject, (LPVOID FAR *)&pDataDst);
pPL->Release();

pDataDst->SetData(&fe, &stm, TRUE);

fRet=SUCCEEDED(OleSetClipboard((LPDATAOBJECT)pDataDst));
pDataDst->Release();

if (!fRet)
   return FALSE;

if (fCut)
   {
   m_pPL->New();
   FDirtySet(TRUE);
   }

return TRUE;
}
```

You will notice that the code above handles both Cut and Copy operations with identical vigor. The only difference between Cut and Copy is that a Cut operation removes the affected data from whatever document it came from. For Component Schmoo, this means clearing out the visible Polyline by calling it's New member, but of course it will mean different things for other applications. The point that should be clear from this, however, is that you still need to create a snapshot of the data regardless of whether you are doing a cut or a copy.

If You Already Have Extensive Clipboard Handling Code

Certainly there are those application, one of which might be yours, that has a great deal of highly optimized clipboard code already in place, especially if you already have a scheme to handle delayed rendering and snapshot copies of your data. If that's the case, then I encourage you to implement a data object on top of your existing code. That is, the implementation of functions like IDataObject::GetData call other functions within your code, perhaps the same one you call when handling the WM_RENDERFORMAT message. This is essentially what Component Schmoo does in the examples above with its RenderFormat function.

This technique then allows you to preserve all your existing code or perhaps only having to restructure it to make it more generally accessible.

Simple Data Source and Consumer: Schmoo

Now that we have the necessary means to simplify clipboard operations, we can use the Schmoo application to demonstrate how to convert existing clipboard code into OLE 2.0 clipboard code. This involves a few steps for startup/shutdown and for the operations of Copy/Cut, Paste, and enabling the Edit/Paste menu item. Each of the sections below list these steps along with Schmoo's original (where appropriate) and now modified implementations of these operations.

Startup/Shutdown

The requirements for startup and shutdown insure that OLE will create its clipboard handling window and that this window

and all OLE2.DLL owned data renderings are properly freed, shown implemented in Schmoo's SCHMOO.CPP in Listing 7-2:

1. At startup, call OleInitialize instead of CoInitialize. This insures that OLE will create its clipboard handling window.

2. During shutdown first call OleFlushClipboard to generate a Release to the data transfer object to match the AddRef from OleSetClipboard. The data object should be destroyed at this time. Also call OleUninitialize instead of CoUninitialize to insure cleanup of OLE's clipboard window.

SCHMOO.CPP

```
BOOL CSchmooFrame::FInit(LPFRAMEINIT pFI)
   {
   DWORD       dwVer;

   //We need OLE versions of Initialize for Clipboard
   dwVer=OleBuildVersion();

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

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

   m_fInitialized=TRUE;
   return CFrame::FInit(pFI);
   }


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

   for (i=0; i<5; i++)
      DeleteObject(m_hBmpLines[i]);

   OleFlushClipboard();

   if (m_fInitialized)
      OleUninitialize();

   return;
   }
```

Listing 7-2: Schmoo's use of OleInitialize, OleUninitialize, and OleFlushClipboard.

Copy/Cut

A copy or a cut, as you probably know, is the process of gaining access to the clipboard, copying the data renderings, and releasing the clipboard. The OLE 2.0 way of executing this is a little different because there is no analog per se of opening or closing the clipboard. Instead, there's the single OleSetClipboard call. The steps for Copy and Cut in the list below are implemented in Schmoo as shown in Listing 7-3.

1. Create a data transfer object like DATATRAN and attach the appropriate data to that object. This is similar to calling OpenClipboard (although a window handle is not required) and calling SetClipboardData for each format you wish to copy. Note also that EmptyClipboard is unnecessary here since OLE automatically calls this function from within OleSetClipboard.

2. Pass the data transfer object to OleSetClipboard which will AddRef the data object and call its EnumFormatEtc to know what formats to place on the actual windows clipboard.

3. Release the data transfer object since you are done with your IDataObject pointer. The object will not be destroyed since OleSetClipboard. This is like calling CloseClipboard.

4. (Cut only) Remove the affected data from the current document.

Schmoo performs these steps in it's CSchmooDoc::FClip function as shown in Listing 7-3 alongside the previous code that filled this function. At least half of the code, and the entire structure of the function, remains the same. What changes are deal primarily with creating the data object and placing data in it. Schmoo conveniently has already a function that retrieves a specific data format, CSchmooDoc::RenderFormat, which works to our favor here.

DOCUMENT.CPP

```
BOOL CSchmooDoc::FClip(HWND hWndFrame, BOOL fCut)
   {
   BOOL           fRet=TRUE;
   HGLOBAL        hMem;
   UINT           i;
   static UINT    rgcf[3]={0, CF_METAFILEPICT, CF_BITMAP};
   const UINT     cFormats=3;

   static DWORD   rgtm[3]={TYMED_HGLOBAL, TYMED_MFPICT, TYMED_GDI};
   LPDATAOBJECT   pIDataObject;
   HRESULT        hr;
   STGMEDIUM      stm;
   FORMATETC      fe;

   hr=CoCreateInstance(CLSID_DataTransferObject, NULL, CLSCTX_INPROC_SERVER
      , IID_IDataObject, (LPVOID FAR *)&pIDataObject);

   if (FAILED(hr))
      return NULL;

   rgcf[0]=m_cf;

   for (i=0; i < cFormats; i++)
      {
      //Copy private data first.
```

```
    hMem=RenderFormat(rgcf[i]);

    if (NULL!=hMem)
        {
        stm.hGlobal=hMem;
        stm.tymed=rgtm[i];
        stm.pUnkForRelease=NULL;

        SETDefFormatEtc(fe, rgcf[i], rgtm[i]);

        pIDataObject->SetData(&fe, &stm, TRUE);
        }
    }

fRet=SUCCEEDED(OleSetClipboard(pIDataObject));
pIDataObject->Release();

//Delete our current data if copying succeeded.
if (fRet && fCut)
    {
    m_pPL->New();
    FDirtySet(TRUE);
    }

return fRet;
}
```

Listing 7-3: Schmoo's FClip function from DOCUMENT.CPP that places data on the clipboard.

Enabling Edit/Paste

Any programmer that has ever implemented any Paste functionality in an application has gone through the rite of processing WM_INITMENUPOPUP and deciding whether or not to enable the Paste menu item depending on the available formats on the clipboard. This immortalized user interface does not change with OLE 2.0; what does change is how you can implement it using data objects.

I want to stress the phrase "**can implement**" because OLE 2.0 does not exclude you from using any of the exiting APIs to implement this feature or Paste, for that matter. This is simply an alternate way of doing it. The advantage is that once you write a piece of code that determines whether or not you can paste from any given data object, you'll have a piece of code that can determine the "pastability" of data from the clipboard or from a drag-drop operation as we'll see in Chapter 8.

So as you can see in Listing 7-4 the OLE 2.0 method for checking formats is only a matter of asking the data object:

1. Call OleGetClipboard to retrieve an IDataObject pointer for the clipboard.

2. Call IDataObject::QueryGetData for each format you would pass to IsClipboardFormatAvailable. If querying on any of your pastable formats succeeds you should enable the Edit/Paste menu item. Only if none of those formats are available should you disable the item.

3. Call IDataObject::Release when you are done.

DOCUMENT.CPP

```
BOOL CSchmooDoc::FQueryPaste(void)
   {
   LPDATAOBJECT    pIDataObject;
   BOOL            fRet;

   if (FAILED(OleGetClipboard(&pIDataObject)))
      return FALSE;

   fRet=FQueryPasteFromData(pIDataObject);
   pIDataObject->Release();
   return fRet;
   }

BOOL CSchmooDoc::FQueryPasteFromData(LPDATAOBJECT pIDataObject)
   {
   FORMATETC       fe;

   SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL);
   return (NOERROR==pIDataObject->QueryGetData(&fe));
   }
```

Listing 7-4: Schmoo's FQueryPaste and FQueryPasteFromData functions from DOCUMENT.CPP that checks the formats available on the clipboard.

Schmoo's FQueryPaste function, that used to call IsClipboardFormatAvailable no retrieves the IDataObject pointer for the clipboard, passes it to a new function, FQueryPasteFromData, to determine pastability, then releases that pointer. The real core of the code lives in FQueryPasteFromData which is one call to IDataObject::QueryGetData.

At this point I strongly encourage you to implement a new function like FQueryPasteFromData, that is, one that takes some data object pointer as a parameter and returns whether or not there's any usable data there. Again, my reason for this encouragement is that with such a function in place, you can pass any data object to it irrespective of whether you got the pointer from OleGetClipboard, from a drag-drop operation, or from a QueryInterface on some other interface pointer. I addition, it gives you a single point in which to make changes to handle other formats that involve compound documents as we'll do in later chapters.

Paste

Paste is pretty much the same operation as checking pastability where instead of calling IDataObject::QueryGetData we call IDataObject::GetData with the steps listed below. These steps are implemented in Schmoo as shown in Listing 7-5:

1. Obtain a data object pointer by calling OleGetClipboard. The returned pointer will have been AddRef'd in OleGetClipboard

2. Call IDataObject::GetData to retrieve your preferred FORMATETC. Since you are the owner of this data, be sure to call ReleaseStgMedium when you are finished copying the data. Be sure to use ReleaseStgMedium since the data object may have filled pUnkForRelease in order to control the data. Note also that you do should not hold on to this data but instead copy it as necessary. This is no different from the requirement of data obtained from the clipboard today using GetClipboardData.

3. Call IDataObject::Release to match the AddRef in OleGetClipboard.

DOCUMENT.CPP

```
BOOL CSchmooDoc::FPaste(HWND hWndFrame)
   {
```

```
   LPDATAOBJECT    pIDataObject;
   BOOL            fRet;

   if (FAILED(OleGetClipboard(&pIDataObject)))
      return FALSE;

   fRet=FPasteFromData(pIDataObject);
   pIDataObject->Release();

   return fRet;
   }


BOOL CSchmooDoc::FPasteFromData(LPDATAOBJECT pIDataObject)
   {
   FORMATETC       fe;
   STGMEDIUM       stm;
   BOOL            fRet;

   SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL);
   fRet=SUCCEEDED(pIDataObject->GetData(&fe, &stm));

   if (fRet && NULL!=stm.hGlobal)
      {
      m_pPL->DataSetMem(stm.hGlobal, FALSE, FALSE, TRUE);
      ReleaseStgMedium(&stm);
      FDirtySet(TRUE);
      }

   return fRet;
   }
```

Listing 7-5: Schmoo's FPaste and FPasteFromData functions from DOCUMENT.CPP that paste data from the clipboard.

Like we did for FQueryPaste, FPaste here simply obtains the data object pointer from OleGetClipboard and passes it to FPasteFromData. Again, I encourage you to split off the code that actually performs the paste into a function that takes a data object, for we can reuse this function to implement a drop (essentially a paste) in a drag-drop operation.

Paste Special and a Functional Patron

It's finally time to bring together the storage enhancements we made to Patron in Chapter 5 and the freeloading technique developed in Chapter 6 to make Patron an app that finally does something that could be considered useful. Since the changes are extensive, I will not show the complete code listings here–instead I will pull in important fragments for the discussion. Besides adding a number of functions to the CPatronDoc, CPages, and CPage classes, I've added two more non-trivial source files, TENANT.CPP and PAGEMOUS.CPP. All of this primarily supports the pasting of metafiles and bitmaps into a page as 'tenants' which OLE 2.0 will kindly draw and serialize for us, resulting in Patron finally having something visible as shown in Figure 7-2. What's left for us to provide is storage for each tenant, insure that they are saved to files and that we can reload them again, and the ability to copy or cut one of these tenants back to the clipboard.

Patron also contains a few user-interface components such as mouse hit-testing to select a tenant, a menu item to delete a tenant, and code draw resizing handles on the selected object. Since we draw resizing handles we have to hit-test

those regions of the tenant and provide for giving visual feedback during a resize operation as shown in Figure 7-3. Since the goal of this book is to discuss OLE 2.0 and not to describe exactly how you implement an application of this nature, I will not go into much detail about these non-OLE features in Patron.

However, the other most noticeable feature of Patron is its addition of a Paste Special dialog box that allows you to paste either a bitmap or a metafile: by default Patron will prefer to paste a metafile over a bitmap, but with Paste Special the end-user can choose the format. The Paste Special dialog box, as it appears in Figure 7-4, is implemented in the OLE2UI library shipped with the OLE 2.0 Toolkit. This dialog not only supports traditional Paste Special commands but is also expandable to handle things like embedded and linked objects. We'll take advantage of such features in Chapters 9 and 12. Because this dialog is an integral part of the OLE 2.0 user interface specifications, we'll spend a little extra time with it up front, then finish off this short chapter with brief discussions about other Patron modifications.

§

Figure 7-2: Patron with a number of Tenants on a page.

§

Figure 7-3: Resizing a tenant in Patron.

§

Figure 7-4: The Paste Special dialog box in Patron.

The Paste Special Dialog Box and the OLE2UI Library

I begged and I pleaded and it finally happened...when I worked with OLE 1.0 a while back I figured that the hardest part about implementing an OLE 1.0 container (called a client then) was providing all the user interface. All those dialogs! When I began working with OLE 2.0, I tried hard to convince others that Microsoft should implement a common dialogs for OLE 2.0. While I initially met stiff resistance, that is "not enough resources," it eventually made sense to everyone that I was going to have to implement these dialogs anyway, the OLE 2.0 sample code would have to obtain the dialogs, and various Microsoft product groups including product support were going to have to implement them. So a number of us, myself included,[1] got together to divide the work down to one dialog a person. From that effort came OLE2UI, which will save you a tremendous amount of time in providing the proper OLE 2.0 interface.

Not only does OLE2UI give you dialogs, but also a number of other useful functions as well to make your OLE 2.0 programming life easier. I have purposely avoided their use until now as I prefer that you gain an understanding of what's really happening before using a function that hides such information from you. In any case, to use the OLE2UI library we must now #include <ole2ui.h> to gain all the necessary definitions, structures, and function prototypes necessary.

NOTE: Patron attempts to link to a build of OLE2UI named BOOKUI.DLL through BOOKUI.LIB. The OLE 2.0 Toolkit contains instructions on how to build a renamed version of OLE2UI which you will have to do before attempting to build Patron in this chapter or any following.

In addition, applications using the library are required to call **OleUIInitialize** passing your instance handle before using any other function in the library and **OleUIUnInitialize** when the library is no longer needed. These calls are made from the same code in PATRON.CPP where we also call OleInitialize and OleUnintialize. Be careful with OleUIUnInitialize as the 'I' in 'Initialize' is capitalized which is inconsistent with OleUninitialize where the 'i' is lower case. One of those glitches that slips through the cracks.

Anyway, our first real encounter with this great timesaving library is the Paste Special dialog which you invoke by filling an OLEUIPASTESPECIAL structure (shown below, taken from OLE2UI.H) and calling OleUIPasteSpecial. On return the dialog will indicate which format was chosen from the dialog box. Patron uses this format to determine whether to create an object using CLSID_StaticDib or CLSID_StaticMetafile. The real trick here is filling the OLEUIPASTESPECIAL structure:

```
//From OLE 2.0's OLE2UI.H
typedef struct tagOLEUIPASTESPECIAL
   {
   //These IN fields are standard across all OLEUI dialog functions.
   DWORD        cbStruct;      //Structure Size
   DWORD        dwFlags;       //IN-OUT:  Flags
```

1I did the Insert Object dialog. If you want to see the credits for everyone on the OLE 2.0 UI team, Shift double-click in the upper right of the About box in the OLE 2.0 toolkit's Outline programs.

```
    HWND          hWndOwner;      //Owning window
    LPCSTR        lpszCaption;    //Dialog caption bar contents
    LPFNOLEUIHOOK lpfnHook;       //Hook callback
    LPARAM        lCustData;      //Custom data to pass to hook
    HINSTANCE     hInstance;      //Instance for customized template name
    LPCSTR        lpszTemplate;   //Customized template name
    HRSRC         hResource;      //Customized template handle

    //Specifics for OLEUIPASTESPECIAL.

    //IN fields
    LPDATAOBJECT    lpSrcDataObj;       //Source IDataObject* (on the
                                        //clipboard) for data to paste

    LPOLEUIPASTEENTRY arrPasteEntries;  //Array of OLEUIPASTEENTRY structs
                                        //of the acceptable formats.
    int           cPasteEntries;      //Number of OLEUIPASTEENTRYs

    [These are not important for this chapter]
    UINT      FAR *arrLinkTypes;
    int           cLinkTypes;

    //OUT fields
    int           nSelectedIndex;     //User-selected arrPasteEntries
    BOOL          fLink;              //Paste or Paste Link selected?
    HGLOBAL       hMetaPict;          //Icon and icon title
    } OLEUIPASTESPECIAL, *POLEUIPASTESPECIAL, FAR
*LPOLEUIPASTESPECIAL;
```

Paste Special needs to present a listbox containing text descriptions of all the formats available for pasting, allowing the application to control which specific formats are at all possible. So first, in order for Paste Special to know what's on the clipboard, you must fill the *lpSrcDataObj* field of the OLEUIPASTESPECIAL dialog with whatever you get from OleGetClipboard:

```
OLEUIPASTESPECIAL   ps;

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

if (FAILED(OleGetClipboard(&ps.lpSrcDataObj)))
   return FALSE;
```

You must also fill in the *cbStruct* field of the OLEUIPASTESPECIAL structure (used to verify versions), the window that own and parents the dialog box in *hWndParent* :

```
ps.cbStruct=sizeof(ps);
ps.hWndOwner=hWndFrame;
```

In the *dwFlags* field you can specify any of the flags PSF_SHOWHELP, PSF_SELECTPASTE, PSF_SELECTPASTELINK, and PSF_CHECKDISPLAYASICON, the latter two of which we'll use later. To use this dialog for simple pasting, specify the PSF_SELECTPASTE flag (including PSF_SHOWHELP if desired):

```
ps.dwFlags=PSF_SELECTPASTE;
```

What's left is to describe the formats we can possibly paste and to provide text descriptions for those formats. This is accomplished by filling the *cPasteEntries* and *arrPasteEntries* with the number of allowable formats and a pointer to an array of OLEUIPASTEENTRY structures:

```
OLEUIPASTEENTRY     rgPaste[4];

ps.arrPasteEntries=rgPaste;
ps.cPasteEntries=4;
```

Each OLEUIPASTEENTRY structure contains four fields that describe the format and provide the necessary user interface strings for each format:

```
//From OLE 2.0's OLESTD.H
typedef struct tagOLEUIPASTEENTRY
   {
   FORMATETC  fmtetc;
   LPCSTR     lpstrFormatName;
   LPCSTR     lpstrResultText;
   DWORD      dwFlags;
   DWORD      dwScratchSpace;
   } OLEUIPASTEENTRY, *POLEUIPASTEENTRY, FAR *LPOLEUIPASTEENTRY;
```

where:

fmtetc	The FORMATETC for this entry
lpstrFormatName	A text description of the FORMATETC. For example, if fmtetc contains CF_DIB, an appropriate string is "Device-Independent Bitmap." Note that the specific storage medium, target device, and aspect of the FORMATETC should not be reflected in this description.
lpszResultText	A string that describes the result of the Paste Special operation. Typically this is the word "as" prepended to the format description. For example, a FORMATETC containing CF_DIB should be accompanied with a string of "as Device-Independent Bitmap."
dwFlags	Flags indicating what operations are allowed on this particular format. The only one of relevance here is OLEUIPASTE_PASTEONLY which indicates that the only operation allowed on this format within the dialog is a paste. The dialog also has Paste Link and Display as Icon options that are disabled with the PASTEONLY flag.
dwScratchSpace	Reserved

Patron has four formats that it can paste. The preferred format is a structure called PATRONOBJECT (see PAGES.H) which contains information about a tenant's location within a page such that a copy/paste between Patron documents places the object in the same place, if possible. This is followed in preference by CF_METAFILEPICT, CF_DIB, and CF_BITMAP. Therefore Patron fills in an array of four OLEUIPASTEENTRY structures:

```
SETDefFormatEtc(rgPaste[0].fmtetc, m_cf, TYMED_HGLOBAL);
rgPaste[0].lpstrFormatName="Patron Object";
```

```
rgPaste[0].lpstrResultText="as Patron Object";
rgPaste[0].dwFlags=OLEUIPASTE_PASTEONLY;

SETDefFormatEtc(rgPaste[1].fmtetc, CF_METAFILEPICT, TYMED_MFPICT);
rgPaste[1].lpstrFormatName="Metafile";
rgPaste[1].lpstrResultText="as Metafile";
rgPaste[1].dwFlags=OLEUIPASTE_PASTEONLY;

SETDefFormatEtc(rgPaste[2].fmtetc, CF_DIB, TYMED_HGLOBAL);
rgPaste[2].lpstrFormatName="Device-Independent Bitmap";
rgPaste[2].lpstrResultText="as Device-Independent Bitmap";
rgPaste[2].dwFlags=OLEUIPASTE_PASTEONLY;

SETDefFormatEtc(rgPaste[3].fmtetc, CF_BITMAP, TYMED_GDI);
rgPaste[3].lpstrFormatName="Bitmap";
rgPaste[3].lpstrResultText="as Bitmap";
rgPaste[3].dwFlags=OLEUIPASTE_PASTEONLY;
```

Finally, once we have this structure filled, we can call OleUIPasteSpecial:

```
uTemp=OleUIPasteSpecial(&ps);

if (OLEUI_OK==uTemp)
   {
   fRet=FPasteFromData(ps.lpSrcDataObj
      , &rgPaste[ps.nSelectedIndex].fmtetc
      , TENANTTYPE_STATIC, NULL, 0L);
   }

ps.lpSrcDataObj->Release();
return fRet;
```

If the function returns OLEUI_OK then the user pressed the OK button and we can actually perform a paste with the selected format which the dialog stores in the *nSelectedIndex* field of OLEUIPASTESPECIAL. Note again that Patron is using a "Paste from Data" function that takes a data object and effectively does a paste from the specified FORMATETC. Finally as well, we have to remember to release the data object we obtained from OleGetClipboard.

Tenant Creation, Paste

The difference between Paste Special and the typical Paste operation is that Paste does not allow specific selection of the pasted format. For Paste, Patron goes through a series of checks to find the best format on the clipboard by calling IDataObject::QueryGetData, and once it finds a format it uses the same FPasteFromData function that paste special uses. The function that checks for the available format is identical to what we want for enabling the Paste menu item. Patron's FQueryPasteFromData does just that as well as return the best FORMATETC it found on the clipboard:

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

```
    if (NULL!=(LPVOID)ptType)
       *ptType=TENANTTYPE_STATIC;

    //Any of our specific data here?
    SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL);
    hr=pIDataObject->QueryGetData(&fe);

    if (NOERROR!=hr)
       {
       //Try metafile, DIB, then bitmap, setting fe each time
       SETDefFormatEtc(fe, CF_METAFILEPICT, TYMED_MFPICT);
       hr=pIDataObject->QueryGetData(&fe);

       if (NOERROR!=hr)
          {
          SETDefFormatEtc(fe, CF_DIB, TYMED_HGLOBAL);
          hr=pIDataObject->QueryGetData(&fe);

          if (NOERROR!=hr)
             {
             SETDefFormatEtc(fe, CF_BITMAP, TYMED_GDI);
             hr=pIDataObject->QueryGetData(&fe);
             }
          }
       }

    if (NOERROR==hr && NULL!=pFE)
       *pFE=fe;

    return (NOERROR==hr);
    }
```

Patron's implementation of paste, in CPatronDoc::FPaste, calls FQueryPasteFromData to first find the best format, then calls CPatronDoc::FPasteFromData to do the real operation. Note again, like I have mentioned before, that it is extremely valuable to separate out a function to do your paste operation from a data object, for when we implement drag-drop in the next chapter this sort of function will be extremely useful:

```
BOOL CPatronDoc::FPaste(HWND hWndFrame)
   {
   LPDATAOBJECT    pIDataObject;
   BOOL            fRet=FALSE;
   FORMATETC       fe;
   TENANTTYPE      tType;

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

```
   if (FAILED(OleGetClipboard(&pIDataObject)))
      return FALSE;

   //Go get the type and format we *can* paste, then actually paste it.
   if (FQueryPasteFromData(pIDataObject, &fe, &tType))
      fRet=FPasteFromData(pIDataObject, &fe, tType, NULL, 0L);

   pIDataObject->Release();
   return fRet;
   }


BOOL CPatronDoc::FPasteFromData(LPDATAOBJECT pIDataObject
   , LPFORMATETC pFE, TENANTTYPE tType, LPPATRONOBJECT ppo, DWORD
dwData)
   {
   BOOL           fRet;
   HRESULT        hr;
   PATRONOBJECT    po;
   STGMEDIUM      stm;

   if (NULL==pFE)
      return FALSE;

   //If we're not given any placement data, see if we can retrieve it
   if (pFE->cfFormat==m_cf && NULL==ppo)
      {
      hr=pIDataObject->GetData(pFE, &stm);

      if (SUCCEEDED(hr))
         {
         ppo=(LPPATRONOBJECT)GlobalLock(stm.hGlobal);

         po=*ppo;
         ppo=&po;

         GlobalUnlock(stm.hGlobal);
         ReleaseStgMedium(&stm);
         }
      }

   fRet=m_pPG->TenantCreate(tType, (LPVOID)pIDataObject, pFE
      , ppo, dwData);

   if (fRet)
```

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

    FDirtySet(TRUE);
    }

return fRet;
}
```

The FPasteFromData function first checks to see if it was given explicit placement data for whatever it pastes, that is, a point for the upper-left corner of the object on the page (where (0,0) is the upper-left of the printable region) and both x- and y-extents which define the complete object's rectangle. I separated the point and the extents such that they could be manipulated independently–changing the size of an object doesn't necessarily change the upper-left corner, and moving an object doesn't necessarily change the extents.

In any case, during pastes from the clipboard the *ppo* parameter to this function will always be NULL, which means if the clipboard holds Patron Object data, then we paste the best available graphic format, placing that graphic at the coordinates in the PATRONOBJECT structure on the clipboard. You'll find that no code in this chapter's version of Patron passes anything but NULL as *ppo*: it's there in preparation for the drag-drop feature we'll add in the next chapter. In all honesty, I had implemented this differently on my first pass through this chapter's code and after finding it deficient for Chapter 8, I came back here and changed this code to suit. I hope it gives you a little insight if you are designing something similar.

Whew! That's been a mouthful, but now let's look at what actually creates this thing we call a Tenant. A Tenant is an object of my C++ class CTenant which has member functions like Open, Load, Update, Destroy, Select, Activate, Draw, SizeSet, just to name a few. This tenant object will also become a 'site object' in Chapter 9 as it will maintain information for a compound document object. But really we're most of the way there already, as a Tenant in this chapter holds on to some object with an IUnknown pointer, asks that object to draw itself through IViewObject, and uses the OLE2.DLL functions of OleSave and OleLoad to save the object and its presentations to a piece of storage.

The storage for each tenant is a uniquely named storage object created below the storage object for the page. You can see this happening in the function CTenant::FOpen which will either open the storage of a given name if it exists, and failing that will create a new storage of that name. To keep unique names, each page maintains a DWORD tenant ID counter which it saves to its storage along with a list of tenants, just like the CPages implementation saves lists of pages to its storage object one level higher.

When Patron needs to create a tenant, a request that usually starts from the document level, the creation request is passed from the document, to CPages, then to the current CPage that allocates a new tenant (new CTenant, in FTenantAdd) and asks that tenant to actually perform the necessary functions to obtain some IUnknown pointer which the tenant then stores. This happens in CTenant::UCreate which is the only piece of code that knows how to take something like a data object pointer and create an object out of whatever is there. Exactly what to do is determined by the TENANTTYPE (an enumeration) parameter passed to it. For this chapter, the type is always TENANTTYPE_STATIC, so UCreate generally uses the same technique as Chapter 6's Freeloader to create a static object. In later chapters we'll add more types of tenants here, such as compound document embedded objects for which we'll call OleCreate to create the object.

Saving and Loading Tenants

In Chapter 5 we added basic compound file functions to Patron where a document maintained a list of pages. We set up a mechanism such that each page's storage was committed when you switched away from it, and opened again when you switch to that page. When we saved the compound file, we write a "Page List" stream off the root storage that contained the number of pages in the document, the currently viewed page, the next ID to use for a page, and the list of pages, that is, the list of page IDs from which Patron can recreate the name of the page's storage.

For this chapter we just extend the same idea to each page, where a page maintains a list of tenants and writes a stream containing the number of tenants, the next tenant ID, and a list of tenant IDs that exist in the page. The name of the tenant's storage is generated from the tenant ID just as we do for the pages. Managing the storages, however, is a little different. When the page is open, all the tenants on that page are also considered "loaded," that is, Patron has a pointer to the object in that tenant. When you switch away from the page, each tenant's storage is committed before the page's

storage is committed. Still, nothing is written to the disk since we have yet to commit the root storage. But as far as each page is concerned, we don't have to try to keep pointers to any tenant that has been modified; instead, we save those objects to memory when closing the page and reload them, from memory, when the page is reopened.

In Chapter 6's Freeloader we explicitly used the object's IPersistStorage interface to affect the saving and loading of the object's and their presentations. Patron instead uses the two functions OleSave and OleLoad, some OLE2.DLL API 'wrappers,' which do exactly, and I mean exactly (I looked), the sequence of operations that we did in Freeloader. OleSave saves all the presentations in the cache and stores the object's class ID to its storage. OleLoad reinitializes the cache from the saved presentations and creates a pointer to the object, so when we reload a tenant and its object we do not use UCreate. That latter function is exclusively for first-time creation of the object residing in the tenant.

Patron's whole storage scheme really shows off the power of transactioned storage. By simple virtue of having the root storage transacted, we can write the rest of the application to think like its data is always on disk, that is, when we create a new object in a tenant we immediately save that object to its storage (see OleSave call in CTenant::CreateStatic). When any tenant is asked to update itself in its storage, it writes a small stream containing its FORMATETC and position information, then calls OleSave to write all the messy data, followed by a Commit. In all, the storage management on the tenant level is minimal, and the page only needs to insure that each tenant is given the chance to update itself before the page closes.

The most beautiful part of this storage mechanism is that we have now in place everything we need to handle storage for a compound document object. When we enable this feature in Patron in Chapter 9, you'll see that we need no modifications.

Copy and Cut

The final feature that I would like to discuss is that of copy and cut operations for the currently selected tenant. I will not be discussing exactly I implemented the sizing functionality because that's just a lot of Windows programming that's pretty clear from the source code itself. A discussion of it here would distract from discussing OLE 2.0. I will admit that the sizing code took about three times as long to write and debug than any other part of this application that is using OLE 2.0. I don't really want to sedate you with the details.

Patron uses the same technique developed in Schmoo at the beginning of this chapter. The data for the object is stuffed into one of our Data Transfer component objects and it is that object we put on the clipboard. The entire Copy/Cut operation starts in CPatronDoc::FClip which just calls CPages::TenantClip which calls the current page's CPage::TenantClip. Inside this last function the page calls it's internal ::TransferObjectCreate function. TransferObjectCreate places two data formats in the data transfer object. The first is a PATRONOBJECT structure that describes where the object lives on the page. If we later paste back into a Patron document we can use this data to try to put the tenant in the same place that it was in the source. We'll use this as well in Patron's drag-drop implementation next chapter. In addition to this placement data, we include the graphical presentation we're using for the object in the tenant. A rendering of this graphic is readily available through the object's IDataObject::GetData. Mostly harmless, I would say, and once TransferObjectCreate is finished CPage::TenantClip calls OleSetClipboard and possibly destroys the selected tenant if we're doing a Cut operation.

Note that I created the separate TransferObjectCreate function because we'll want to use it again next chapter when we need a data object as a drag-drop source. This is why there's the *pptl* parameter to TransferObjectCreate which describes the offset from the top left of the tenant's rectangle where it was picked up in the drag-drop operation.

What that really means is that we've come to the end of this chapter and are ready to dive into the drag-drop transfer protocol with data objects.

Summary

OLE 2.0 provides three APIs that allow applications to replace their current clipboard handling code with the use of data objects. OLE 2.0 only requires the use of it's clipboard handling in drag-drop and compound document scenarios, so applications that have little need to be involved with compound documents or will not implement drag-drop need not be too concerned with this new mechanism.

However, any potential source of data that will also wish to become a drag-drop data source as well as a clipboard data source, will need to implement a data object for use in that operation. If you are planning to go this route, it's beneficial for you to convert your current clipboard code into OLE 2.0 data object transfers, since most of the code you write is just as useful on a data object obtained from a drag-drop operation as from a clipboard Paste.

The first of the three APIs is OleSetClipboard, which places a data object pointer on the clipboard as well as the names of the actual data formats that data object knows about as determined from IDataObject::EnumFormatEtc. OLE 2.0 will not actually render the data, that is call IDataObject::GetData, until some data consumer asks for it. Therefore the source application must be written to handle what is known as delayed rendering which typically requires that the application snapshots the data copied or cut, and holds on to it until the clipboard is cleared.

The consumer of data calls OleGetClipboard to retrieve an IDataObject pointer representing the data on the clipboard. Through this data object pointer the consumer can retrieve data or ask about the availability of formats, that is, implement Paste functionality. I recommend here that you create a function to paste, or query pasting, from any arbitrary data object as such code centralizes the operation and enables you to use to regardless of what protocol you used to get the IDataObject pointer in the first place.

The other API is OleFlushClipboard which will clean any existing data object off the clipboard. This is generally called at the close of an application. In addition, apps that wish to use OLE 2.0's clipboard facilities must now be calling OleInitialize and OleUninitialize instead of CoInitialize and CoUninitialize. The Ole* versions create a hidden clipboard window that OLE 2.0 uses to generate calls to a source's IDataObject::GetData. OleUninitialize, of course, destroys this one-per-task window.

It may seem like an overkill to have to create a data object just for the purposes of copying one lousy little piece of text, and I agree. Therefore this chapter implements a component object to simplify OLE 2.0 clipboard operations making them almost parallel to how you probably use the clipboard today. This is, of course, one possible solution, so at least two other possibilities are presented here.

For this chapter the clipboard handling code in both Schmoo and Component Schmoo are replaced with the OLE 2.0 techniques. Patron is blessed with procreative ability, that is, can now paste real graphics from the clipboard, resize them, print them, and save and load them from compound files. This sets Patron up well for becoming a compound document container in Chapter 9.