Chapter Eight
Drag-Drop Operations with Data Objects

When I first started to work with Object Linking and Embedding[1] I watched a number of video tapes from OLE 1.0 seminars and demonstrations in a four-month crash effort to attain OLE enlightenment. At one point, when I was getting bored out of my skull, I decided to fast-forward through some of the demos. What I noticed during that fit of impatience was how often the presenter would select some data, pull down the Edit menu to choose Cut or Copy, switch to another application, then pull down the Edit menu to choose Paste. Not a big deal, I imagined. After all, the clipboard has been the most common techniques used to transfer data between source and consumer applications, and that sequence of operations it what simply works.

I realized then, as others were doing at the same time, that the much more efficient way of performing the same Cut-Copy/Paste operations was to use a drag-drop technique. In drag-drop, the you select some data in the source application, *pick* it up with a mouse click, *drag* that picked up data from the source application's window to the consumer application's window, and *drop* that data into the consumer.

This sequence of operations I call *pick-drag-drop* does exactly what you today accomplish with the clipboard through the Copy/Cut-switch applications-do the hokey-pokey and turn yourself around-Edit/Paste sequence. The big difference is that drag-drop is direct and immediate, that is, the entire data transfer operation happens all at once. Clipboard transfers are most useful when you want to store data on the clipboard for an undetermined amount of time, and that data might not ever be consumed. With drag-drop, the source and consumer at linked together for the transfer without having the clipboard in between. In fact, it does not affect the contents of the clipboard in any way.

OLE 2.0 drag-drop,[2] like OLE 2.0's use of the clipboard, works through data objects. Drag-drop is therefore just another way to move a data object pointer, that is a pointer to IDataObject, from the source application to the consumer application. In order to facilitate this pointer exchange, the source application implements an object called the "drop source" that provides an IDropSource interface. The consumer application, which in drag-drop context is called a *target* application, implements a 'drop target' object which provides the interface IDropTarget. The first section in this chapter explains how these two objects are managed, how OLE 2.0 becomes aware of them, and how they communicate in a drag-drop operation.

Using the Schmoo application I'll then show how to implement simple drag-drop features into an application. This example will show about how little you can get away with to benefit from this OLE 2.0 feature as we'll only implement minimal user interface. This minimal implementation will lead into a more complicated implementation in Patron which will use drag-drop to not only accept data from other applications but will also use it to move tenants around within a single page, or between different open documents. Patron makes special considerations for scrolling a page while a drag-drop operation is happening and provides more precise user feedback.

I want to again stress that if you are going to implement drag-drop that you first convert your clipboard handling code from Windows APIs into OLE 2.0 APIs and make specific functions that paste or test pasting from any given data object as demonstrated in Chapter 7. Those efforts pay off here to the extent that a drag-drop implementation can be done in a matter of days, if not hours, depending on your desired complexity. (C'mon Kraig, you think we *desire* complexity?) People watching you demonstrations on fast-forward video won't know what hit them.

1In November of 1991, with OLE 1.0.

2Officially termed drag-**and**-drop by the terminology cops. However, I find that cumbersome and prefer to use the more terse drag-drop, as the OLE 2.0 design specification uses in its explanations.

Sources and Targets: The Drag-Drop Transfer Model

Let's begin with a typical situation where the end user conceptually has two applications running and wants to transfer some selection of data from the source application into the consumer application as represented in Figure 8-1. In reality the two applications may be two different documents in the same application, or may even be completely within one document of one application. In any case, there is a place we can label as the source and a place we can label as the consumer. The source of the data must implement a data object to represent the data being transferred. This can be exactly the same data object that you would create for a clipboard transfer.

Figure 8-1: The state of things before a drag-drop operation: the source has a data object attached to the affected data. There are not necessarily two applications: the source and consumer may be the same document in one application or perhaps two documents in the same application, not just two separate applications. With OLE 2.0 in the middle, who knows the difference?

So our problem now is getting that IDataObject pointer from the source to the target via drag-drop. In order to be the source of a drag-drop operation, an application must implement an object with the IDropSource interface as shown in Figure 8-2. The drop source object is a stand-alone object, that is, QueryInterface on it will only understand IDropSource and IUnknown. IDropSource only has two member functions (besides IUnknown members):

QueryContinueDrag	Determines what conditions cause a drop or cancellation of the operation.
GiveFeedback	Sets the mouse cursor depending on an "effect" flag that indicates what would happen if the data was dropped at the current location. Can also provide other user-interface within the source application.

Figure 8-2: Objects necessary in both source and target (consumer) to facilitate a drag-drop operation.

The target, in order to accept any data from a drag-drop operation, must implement a stand-alone object with the IDropTarget interface as also shown in Figure 8-2.

The drop target object is generally part of the consumer's document as a whole, that is, the drop target object is identified as belonging to a specific window. To attach the object to the document window, the consumer passes the IDropTarget pointer to the OLE 2.0 API function RegisterDragDrop as shown in Figure 8-3. **RegisterDragDrop** takes an HWND and an IDropTarget pointer and internally saves that pointer (after an AddRef of course!) as a property on the hWnd. At this point the window is open to accept data. When the consumer no longer wants to be a drop target, it calls the OLE 2.0 API function **RevokeDragDrop** with the same HWND as passed to RegisterDragDrop. Internally OLE 2.0 removes the property and releases the IDropTarget pointer.

The IDropTarget interface has four member functions outside IUnknown members:

DragEnter	Indicates that the mouse entered the window registered with this interface. The target generally initializes state variables that are meaningful during this operation and provides some visual indication of what might happen on a drop.
DragOver	Indicates that the mouse moved within the window or that the keyboard state changed. The target indicates what would happen if a drop occurred here using an "effect" flag as well as any visual feedback in its document.
DragLeave	Indicates that the mouse left the window. The target cleans up any state

	from DragEnter/DragOver including any visual feedback.
Drop	Indicates that the source's IDropSource::QueryContinueDrag said "drop" and so the target must paste the data and perform cleanup as in DragLeave.

Figure 8-3: A consumer registers itself as a drop target by passing a window handle and a pointer to its IDropTarget interface to RegsterDragDrop.
NOTE: Most of the IDropTarget member functions are passed a POINTL structure containing the current mouse position in **screen** coordinates. Be sure to call ScreenToClient before hit-testing the mouse position against other client-area coordinates.

The table is now set for the operation to commence. The source application must provide some sort of means to start a drag-drop operation which is typically a WM_LBUTTONDOWN on a particular region of the selected data such as the outer edge of a rectangle. When this event occurs, the source passes its IDropSource and IDataObject pointers to the OLE 2.0 API **DoDragDrop** as shown in Figure 8-4.

Figure 8-4: A source starts a drag-drop operation by passing its IDataObject and IDropSource to DoDragDrop.
Internally DoDragDrop enters a loop that watches the movements of the mouse and changes in the state of the keyboard. The loop executes a series of commands below as illustrated in Figure 8-5:

1. Call the Windows API WindowFromPoint using the current coordinates of the mouse cursor. If WindowFromPoint returns a valid window handle, then check that window for the property containing an IDropTarget pointer.

2. If there is an IDropTarget pointer for this window, call IDropTarget::DragEnter with the (marshaled) IDataObject pointer from the source. The target can check the available formats in the data object and returns an "effect" code that indicates what would happen if a drop occurred here, such as a Copy (Copy/Paste), Move (Cut/Paste), or nothing at all. If there is not an IDropTarget for the window, then DoDragDrop assumes that nothing would happen here.

3. Given the "effect" code from the target under the current cursor position (which is nothing if there is no target), DoDragDrop calls IDropSource::GiveFeedback which is responsible for changing the mouse cursor to an appropriate shape and providing whatever feedback is appropriate in the source depending on the effect.

4. Now, whenever the mouse moves it might leave the confines of one window and enter another. Whenever the mouse moves out of a target window, DoDragDrop calls IDropTarget::DragLeave, then calls the IDropTarget::DragEnter of the new target under the mouse. If the mouse moves but does move into a new target window, then the current target's IDropTarget::DragOver is called.

5. The DragEnter-DragOver-DragLeave loop continues as long as there is no change in the keyboard or mouse button states. Whenever there is a state change, however, DoDragDrop calls IDropSource::QueryContinueDrag which determines whether to continue the operation, cancel it, or indicate a drop. Canceling the operation is done whenever the user pressed the ESC key and dropping is typically indicated on an event like WM_LBUTTONUP since WM_LBUTTONDOWN typically starts the operation in the first place. Any other change allows the operation to continue. In addition, the

current IDropTarget::DragOver is also called on a keyboard state change even if the mouse does not move. By checking the key state the target indicates if it would do either a Move or Copy if a drop happened here, that is, it returns a new effect flag. DoDragDrop then passes this new effect flag to IDropSource::GiveFeedback which again, changes the mouse cursor.

6. If IDropSource::QueryContinueDrag indicates that the operation should be canceled, then DoDragDrop calls IDropTarget::DragLeave and finally exits with an HRESULT containing DRAGDROP_S_CANCEL. If QueryContinueDrag says "drop" then DoDragDrop instead calls IDropTarget::Drop (which does the paste) and exits with an HRESULT of NOERROR. DoDragDrop also returns the last effect code such that the source knows if it needs to cut the affected data (that is, in a move).

Figure 8-5: Execution of DoDragDrop to perform the entire drag-drop operation.

One potential problem is that the current visible area in the target document may not be where the end user wants to drop the data. How, then, if DoDragDrop is eating keyboard and mouse messages, can the end-user scroll the target document? The answer is that OLE 2.0 specifies that within each target window is an "inset region" of a set number of pixels within the border of the window as shown in Figure 8-6. If the mouse stays within an inset region for a given length of time the target should start scrolling the document. If the mouse then leaves this region the target stops scrolling.

This is not much different than the automatic scrolling that happens when you scroll something like and edit control while selecting text or when you select items in a listbox. However, the difference is that once the mouse goes outside the window, scrolling stops. The reason why OLE 2.0 drag-drop is specified in this manner is that a common operation is to drag data between applications. Therefore scrolling whenever the mouse is off the edge of any target document would cause just about every document in every application to scroll wildly. While this certainly obeys the Second Law of Thermodynamics, entropy is not user-friendly.[1] So instead, the mouse has to be in the inset region to affect scrolling. The time delay as well is specified such a quick drag out of one window doesn't start it scrolling just because the mouse passed through the inset region. By having this delay, then end-user must hold the mouse in the inset region for a short time, usually on the order of a few tenths of a second.

§

Figure 8-6: The Inset region within a target document window.

From all this we can extract the specific responsibilities for both source and consumer in a drag-drop scenario:

Source:

1. Provides the IDataObject pointer for the affected data.

2. Implements an object with the IDropSource interface. An drop source object does not need to be instantiated until the source calls DoDragDrop and can be freed immediately after the operation is finished.

3. Calls DoDragDrop to start the operation and implements IDropSource::QueryContinueDrag to determine when to end the operation.

1OK, OK, so you didn't take Thermodynamics in college. The law states that the amount of entropy, ***S***, (disorder, chaos) in the universe is increasing, that is, D***S*** > 0.

4. Sets the appropriate mouse cursors for various effects in IDropSource::GiveFeedback.

Target:

1. Implements an object with the IDropTarget interface.

2. Registers the IDropTarget object with an appropriate window handle using RegisterDragDrop. Typically this is done whenever a document is created or opened.

3. Determines whether or not data is acceptable in IDropTarget::DragEnter. Determines the effect of a drop in IDropTarget::DragEnter and ::DragOver as well as provides user interface feedback appropriate to the effect. Since user feedback is optional I recommend that you skip implementing it until other target operations are working (that is, steps 1, 2, 5, and 6).

4. Scrolls the target document (if appropriate) when the cursor has been in an inset region long enough. As with step 3, you can initially skip this step, and in some cases it will never be a consideration. Concentrate on this step after all others, including 3, have been addressed.

5. Performs a Paste on IDropTarget::Drop.

6. Removes the IDropTarget pointer by calling RevokeDragDrop when the window is no longer a target, typically when the document is closed.

So let's see these responsibilities in code.

A Step-By-Step Drag-Drop Implementation: Schmoo

To demonstrate basic drag-drop implementation let's now add the feature to the Schmoo application (as well as to Component Schmoo, but I won't show any of its code here since it's identical to Schmoo). Since Schmoo's idea of a document only contains one Polyline figure, we don't have to worry about scrolling. Furthermore, providing some user feedback to indicate the drop target is simple as we can just draw an inverted outline around the Polyline. Since we already have a border around the Polyline, that makes a perfect region from which to start a drag; therefore we don't have to do any extra work to figure out when and where a pick happens.

A Schmoo document in this example will be both source and target for its private data format. This will typically be the case in your application as well since you can probably copy and paste your own data. But in drag-drop, you not only copy/paste without your own document, but the same exact implementation works between multiple documents in your same application as well as between different documents in multiple instances of the application. Drag-drop simply does not care: a source is a source and a target a target, independent of their relationship to one another, be it identical, sibling, distant cousin, or complete stranger.

Schmoo's drag-drop implementation follows six steps, three for the target, three for the source. The resulting code is shown in Listing 8-1. You'll notice that the steps implement the target side first. This is intentional since I want Schmoo to drag-drop to itself with its private data. In order to test Schmoo as a drag-drop source, I need at least one acceptable target, and Schmoo is the only application that accepts its own data. Therefore we follow simple target implementation first.

Target Steps:

1. Design and implement the necessary code to provide user feedback that indicates the result of a drop. This should be done first such that step #2 can use it.

2. Implement the IDropTarget interface on a drop target object. All four member functions must be fully implemented; part of their implementation is to use the code from Step #1 to provide user

interface.

3. Use the RegisterDragDrop and RevokeDragDrop functions to manage the lifetime of the drop target object implemented in step 2. These function calls are also accompanied by a special required use of CoLockObjectExternal.

Source Steps:

1. Design what user interface is necessary on the source while dragging. This can be as little as changing the mouse cursor which OLE 2.0 will perform for you.

2. Determine how a drag-drop operation starts, that is, what I call the 'pick' event, and what terminates it, either cancellation or a drop.

3. Implement the IDropSource interface on a drop source object. This implementation can be trivial for most circumstances.

4. Call DoDragDrop when the 'pick' event occurs and possibly cut the selected data from the source if the operation was a move instead of a copy.

Each of these are treated in a separate step-by-step sections below. The code basis for the steps is shown in Listing 8-1, modifications to Schmoo's SCHMOO.H and DOCUMENT.CPP as well as two new files, IDROPTGT.CPP and IDROPSRC.CPP, which implement the drop target and drop source objects, respectively.

DOCUMENT.CPP

```
[Unmodified code omitted from listing]

CSchmooDoc::CSchmooDoc(HINSTANCE hInst)
   : CDocument(hInst)
   {
   m_pPL=NULL;
   m_pPLAdv=NULL;
   m_uPrevSize=SIZE_RESTORED;

   m_pDropTarget=NULL;
   m_fDragSource=FALSE;
   return;
   }


CSchmooDoc::~CSchmooDoc(void)
   {
   if (NULL!=m_pDropTarget)
      {
      RevokeDragDrop(m_hWnd);

      //This lets OLE forget about the object.
      CoLockObjectExternal((LPUNKNOWN)m_pDropTarget, FALSE, TRUE);

      m_pDropTarget->Release();
      }

   ...

   return;
   }



/*
 * CSchmooDoc::FInit
 */
```

```
BOOL CSchmooDoc::FInit(LPDOCUMENTINIT pDI)
   {
   ...

   m_pDropTarget=new CDropTarget(this);

   if (NULL!=m_pDropTarget)
      {
      m_pDropTarget->AddRef();
      RegisterDragDrop(m_hWnd, (LPDROPTARGET)m_pDropTarget);

      CoLockObjectExternal((LPUNKNOWN)m_pDropTarget, TRUE, FALSE);
      }

   return TRUE;
   }



/*
 * CSchmooDoc::FMessageHook
 */

BOOL CSchmooDoc::FMessageHook(HWND hWnd, UINT iMsg, WPARAM wParam
   , LPARAM lParam, LRESULT FAR *pLRes)
   {

   [Handling of WM_SIZE]


   if (WM_LBUTTONDOWN==iMsg)
         {
         LPDROPSOURCE    pIDropSource;
         LPDATAOBJECT    pIDataObject;
         HRESULT         hr;
         SCODE           sc;
         DWORD           dwEffect;

         /*
          * The document has an 8 pixel border around the polyline
          * window where we'll see mouse clicks.  A left mouse button
          * click here means the start of a drag-drop operation.
          *
          * Since this is a modal operation, the IDropSource we need
          * here is entirely local.
          */

         pIDropSource=(LPDROPSOURCE)new CDropSource(this);

         if (NULL==pIDropSource)
            return FALSE;

         pIDropSource->AddRef();
         m_fDragSource=TRUE;

         //Go get the data and start the ball rolling.
         pIDataObject=TransferObjectCreate();

         hr=DoDragDrop(pIDataObject, pIDropSource
            , DROPEFFECT_COPY | DROPEFFECT_MOVE, &dwEffect);

         pIDataObject->Release();

         /*
          * When we return from DoDragDrop, we either cancelled or dropped.
          * First we can toss the IDropSource we have here, then bail out
          * on cancel, and possibly clear our data on a move drop.
          */

         pIDropSource->Release();

         /*
```

```
         * If there was a drop on the same document (determined using
         * this flag, then dwEffect will be DROPEFFECT_NONE (see
         * IDropTarget_Drop in IDROPTGT.CPP).  In any case, we need
         * to reset this since the operation is done.
         */

        m_fDragSource=FALSE;
        sc=GetScode(hr);

        if (DRAGDROP_S_DROP==sc && DROPEFFECT_MOVE==dwEffect)
           {
           m_pPL->New();
           FDirtySet(TRUE);
           }

        //On a canceled drop or a copy we don't need to do anything else
        return TRUE;
        }

   return FALSE;
   }


...


/*
 * CSchmooDoc::DropSelectTargetWindow
 *
 * Purpose:
 *  Creates a thin inverted frame around a window that we use to show
 *  the window as a drop target.  This is a toggle function:  it uses
 *  XOR to create the effect so it must be called twice to leave the
 *  window as it was.
 */

void CSchmooDoc::DropSelectTargetWindow(void)
   {
   HDC        hDC;
   RECT       rc;
   UINT       dd=3;
   HWND       hWnd;

   hWnd=m_pPL->Window();
   hDC=GetWindowDC(hWnd);
   GetClientRect(hWnd, &rc);

   //Frame this window with inverted pixels

   //Top
   PatBlt(hDC, rc.left, rc.top, rc.right-rc.left, dd, DSTINVERT);

   //Bottom
   PatBlt(hDC, rc.left, rc.bottom-dd, rc.right-rc.left, dd, DSTINVERT);

   //Left excluding regions already affected by top and bottom
   PatBlt(hDC, rc.left, rc.top+dd, dd, rc.bottom-rc.top-(2*dd), DSTINVERT);

   //Right excluding regions already affected by top and bottom
   PatBlt(hDC, rc.right-dd, rc.top+dd, dd, rc.bottom-rc.top-(2*dd), DSTINVERT);

   ReleaseDC(hWnd, hDC);
   return;
   }
```

SCHMOO.H

```
[Unmodified sections omitted from listing]

...
```

```
class __far CSchmooDoc : public CDocument
   {
   ...

   //These need access to FQueryPasteFromData, FPasteFromData
   friend class CDropTarget;
   friend class CDropSource;

   protected:
      ...

      class CDropTarget FAR *m_pDropTarget;  //Registered target.
      BOOL           m_fDragSource;      //Drag-drop source==target

   protected:
      ...
      void           DropSelectTargetWindow(void);

   ...
   }
...

//Drag-drop interfaces we need in the document
class __far CDropTarget : public IDropTarget
   {
   protected:
      ULONG             m_cRef;     //Interface reference count.
      LPCSchmooDoc      m_pDoc;     //Back pointer to the document

      LPDATAOBJECT      m_pIDataObject;  //Data object from DragEnter

   public:
      CDropTarget(LPCSchmooDoc);
      ~CDropTarget(void);

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

      STDMETHODIMP DragEnter(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
      STDMETHODIMP DragOver(DWORD, POINTL, LPDWORD);
      STDMETHODIMP DragLeave(void);
      STDMETHODIMP Drop(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
   };


typedef CDropTarget FAR * LPCDropTarget;


class __far CDropSource : public IDropSource
   {
   protected:
      ULONG             m_cRef;     //Interface reference count.
      LPCSchmooDoc      m_pDoc;     //Back pointer to the document

   public:
      CDropSource(LPCSchmooDoc);
      ~CDropSource(void);

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

      STDMETHODIMP QueryContinueDrag(BOOL, DWORD);
      STDMETHODIMP GiveFeedback(DWORD);
   };

typedef CDropSource FAR * LPCDropSource;
```

IDROPSRC.CPP

```
/*
 * IDROPSRC.CPP
 *
 * Implementation of a IDropSource implementation.
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "schmoo.h"

/*
 * CDropSource::CDropSource
 * CDropSource::~CDropSource
 *
 * Constructor Parameters:
 *  pDoc           LPSchmooDoc containing this interface.
 */

CDropSource::CDropSource(LPCSchmooDoc pDoc)
   {
   m_cRef=0;
   m_pDoc=pDoc;
   return;
   }

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



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

STDMETHODIMP CDropSource::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_IDropSource))
      *ppv=(LPVOID)this;

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

   return ResultFromScode(E_NOINTERFACE);
   }


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

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

   cRefT=--m_cRef;

   if (0L==m_cRef)
```

```
      delete this;

   return cRefT;
   }



/*
 * CDropSource::QueryDragContinue
 *
 * Purpose:
 *  Determines whether to continue a drag operation or cancel it.
 *
 * Parameters:
 *  fEsc           BOOL indicating that the ESC key was pressed.
 *  grfKeyState    DWORD providing states of keys and mouse buttons.
 *
 * Return Value:
 *  SCODE          DRAGDROP_S_CANCEL to stop the drag, DRAGDROP_S_DROP
 *                 to drop the data where it is, or NOERROR to continue.
 */

STDMETHODIMP CDropSource::QueryContinueDrag(BOOL fEsc, DWORD grfKeyState)
   {
   if (fEsc)
      return ResultFromScode(DRAGDROP_S_CANCEL);

   if (!(grfKeyState & MK_LBUTTON))
      return ResultFromScode(DRAGDROP_S_DROP);

   return NOERROR;
   }



/*
 * CDropSource::GiveFeedback
 *
 * Purpose:
 *  Provides cursor feedback to the user since the source task
 *  always has the mouse capture.  We can also provide any other
 *  type of feedback above cursors if we so desire.
 *
 * Parameters:
 *  dwEffect       DWORD effect flags returned from the last target.
 *
 * Return Value:
 *  SCODE          NOERROR if you set a cursor yourself or
 *                 DRAGDROP_S_USEDEFAULTCURSORS to let OLE do the work.
 */

STDMETHODIMP CDropSource::GiveFeedback(DWORD dwEffect)
   {
   return ResultFromScode(DRAGDROP_S_USEDEFAULTCURSORS);
   }
```

IDROPTGT.CPP

```
/*
 * IDROPTGT.CPP
 *
 * Implementation of the IDropTarget interface.
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#include "schmoo.h"

/*
 * CDropTarget::CDropTarget
 * CDropTarget::~CDropTarget
```

```
 *
 * Constructor Parameters:
 *  pDoc           LPCSchmooDoc of the document containing us.
 */

CDropTarget::CDropTarget(LPCSchmooDoc pDoc)
   {
   m_cRef=0;
   m_pDoc=pDoc;

   m_pIDataObject=NULL;
   return;
   }


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



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

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

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

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

   return ResultFromScode(E_NOINTERFACE);
   }


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

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

   cRefT=--m_cRef;

   if (0L==m_cRef)
      delete this;

   return cRefT;
   }


/*
 * CDropTarget::DragEnter
 *
 * Purpose:
 *  Indicates that data in a drag operation has been dragged over our
 *  window that's a potential target.  We are to decide if it's something
```

```
 *  we're interested in or not.
 *
 * Parameters:
 *  pIDataSource    LPDATAOBJECT providing the source data.
 *  grfKeyState     DWORD flags indicating states of keys and mouse buttons.
 *  pt              POINTL coordinates in the client space of the document.
 *  pdwEffect       LPDWORD into which we'll place the appropriate effect
 *                  flag for this point.
 *
 * Return Value:
 *  SCODE           NOERROR
 */

STDMETHODIMP CDropTarget::DragEnter(LPDATAOBJECT pIDataSource
    , DWORD grfKeyState, POINTL pt, LPDWORD pdwEffect)
    {
    HWND       hWnd;

    m_pIDataObject=NULL;

    if (!m_pDoc->FQueryPasteFromData(pIDataSource))
        {
        *pdwEffect=DROPEFFECT_NONE;
        return NOERROR;
        }

    //Default is move
    *pdwEffect=DROPEFFECT_MOVE;

    if (grfKeyState & MK_CONTROL)
        *pdwEffect=DROPEFFECT_COPY;

    m_pIDataObject=pIDataSource;
    m_pIDataObject->AddRef();

    hWnd=m_pDoc->Window();
    BringWindowToTop(hWnd);
    UpdateWindow(hWnd);
    m_pDoc->DropSelectTargetWindow();

    return NOERROR;
    }



/*
 * CDropTarget::DragOver
 *
 * Purpose:
 *  Indicates that the mouse was moved inside the window represented
 *  by this drop target.  This happens on every WM_MOUSEMOVE, so this
 *  function should be very efficient.
 *
 * Parameters:
 *  grfKeyState     DWORD providing the current keyboard and mouse states
 *  pt              POINTL where the mouse currently is.
 *  pdwEffect       LPDWORD in which to store the effect flag for this point.
 *
 * Return Value:
 *  SCODE           NOERROR
 */

STDMETHODIMP CDropTarget::DragOver(DWORD grfKeyState, POINTL pt
    , LPDWORD pdwEffect)
    {
    if (NULL==m_pIDataObject)
        {
        *pdwEffect=DROPEFFECT_NONE;
        return NOERROR;
        }

    //Since we can always drop, we just return effect flags based on keys.
```

```
   *pdwEffect=DROPEFFECT_MOVE;

   if (grfKeyState & MK_CONTROL)
      *pdwEffect=DROPEFFECT_COPY;

   return NOERROR;
   }



/*
 * CDropTarget::DragLeave
 *
 * Purpose:
 *  Informs the drop target that the operation has left its window.
 *
 * Return Value:
 *  SCODE          NOERROR
 */

STDMETHODIMP CDropTarget::DragLeave(void)
   {
   if (NULL==m_pIDataObject)
      return NOERROR;

   m_pDoc->DropSelectTargetWindow();
   m_pIDataObject->Release();
   return NOERROR;
   }



/*
 * CDropTarget::Drop
 *
 * Purpose:
 *  Instructs the drop target to paste the data that was just now dropped
 *  on it.
 *
 * Parameters:
 *  pIDataSource    LPDATAOBJECT from which we'll paste.
 *  grfKeyState     DWORD providing current keyboard/mouse state.
 *  pt              POINTL at which the drop occurred.
 *  pdwEffect       LPDWORD in which to store what you do with the data.
 *
 * Return Value:
 *  SCODE          NOERROR
 */

STDMETHODIMP CDropTarget::Drop(LPDATAOBJECT pIDataSource
   , DWORD grfKeyState, POINTL pt, LPDWORD pdwEffect)
   {
   BOOL       fRet=TRUE;

   *pdwEffect=DROPEFFECT_NONE;

   if (NULL==m_pIDataObject)
      return ResultFromScode(E_FAIL);

   m_pDoc->DropSelectTargetWindow();
   m_pIDataObject->Release();

   //No point in drag-drop to ourselves (for Schmoo, at least)
   if (m_pDoc->m_fDragSource)
      return ResultFromScode(E_FAIL);

   fRet=m_pDoc->FPasteFromData(pIDataSource);

   if (!fRet)
      return ResultFromScode(E_FAIL);

   *pdwEffect=DROPEFFECT_MOVE;
```

```
if (grfKeyState & MK_CONTROL)
   *pdwEffect=DROPEFFECT_COPY;

return NOERROR;
}
```

Listing 8-1: Modifications and additions to the Schmoo application to handle drag-drop.

Design and Implement Drop Target User Feedback

There are many ways through which you can indicate what might happen if a drop happens in your document. For example, a word processor might show a shaded caret at the point where text would be dropped:

§

An application like Patron, that pastes graphics, may show a rectangle showing the exact size and location that the dropped data would occupy:

§

For Schmoo, pasting any Polyline data obliterates the existing data in the window with the new data. Therefore what we can do is just highlight the Polyline window itself with inverted lines:

§

This feedback is generated in the new function CSchmooDoc::DropSelectTargetWindow which simply inverts a 3-pixel frame around the window. This function is a toggle such that once we 'turn on' this feedback in a function like IDropTarget::DragEnter we have to remember to turn it off in IDropTarget::DragLeave or IDropTarget::Drop.

This DropSelectTargetWindow function, and whatever type of function you need to implement, has two aspects. First you need to determine where to place the feedback. Here I can just use the rectangle of the Polyline window, but for applications like word processors you might need to calculate line and character nearest the mouse cursor, or for a spreadsheet you would have to determine the closest range of cells. The second aspect then is to draw the visuals at that location, as I do with PatBlt and DSTINVERT. Normally we use some sort of XOR operation such that we can show and remove it quickly and easily, but its always your choice.

Implement a Drop Target Object and the IDropTarget Interface

Schmoo implements a drop target through its class CDropTarget defined in SCHMOO.H and implemented in IDROPTGT.CPP. Note that this is not an interface implementation, it is a full object implementation since it implements an independent QueryInterface and controls its own reference count. In addition, this object frees itself in ::Release on a zero reference.

The only data fields in this drop-target object (besides a reference count and a back pointer to the document in which this object lives) is an LPDATAOBJECT, a pointer to an IDataObject:

```
class __far CDropTarget : public IDropTarget
  {
  protected:
    ULONG            m_cRef;     //Interface reference count.
    LPCSchmooDoc     m_pDoc;     //Back pointer to the document

    LPDATAOBJECT     m_pIDataObject;  //Data object from DragEnter

  ...
```

This data object pointer is used to store a data object we're given in IDropTarget::DragEnter so that if we need to examine the data again during IDropTarget::DragOver, which does not receive a data object pointer, we know what data object is involved.

Implementation of each member function in IDropTarget is very specific and is covered in the following sections.

IDropTarget::DragEnter

This function, as explained above, is called whenever the mouse moves into the window for which we registered this specific instance of IDropTarget. DragEnter receives the following parameters:

pIDataSource	(LPDATAOBJECT) a pointer to the data object involved in this operation.
grfKeyState	(DWORD) A mixture of Windows' MK_ flags containing the current state of the keyboard, that is, any combination of MK_LBUTTON MK_RBUTTON, MK_SHIFT, MK_CONTROL, and MK_MBUTTON
pt	(POINTL) The current location of the mouse in screen coordinates.
pdwEffect	(LPDWORD) A pointer to a DWORD in which the function stores the 'effect' flag that a drop would have here: DROPEFFECT_NONE: No drop can happen here. DROPEFFECT_COPY: A copy should occur DROPEFFECT_MOVE: A cut should occur DROPEFFECT_LINK: A Paste Link will occur DROPEFFECT_SCROLL: The target document is scrolling. With these parameters DragEnter has the following responsibilities:

1. Use *pIDataSource*->EnumFormatEtc and *pIDataSource*->QueryGetData to determine if it has usable data. The conditions for a drop are generally the same as for pasting. If pasting isn't possible, store DROPEFFECT_NONE in **pdwEffect* and return NOERROR.

2. Determine if you can drop on the point in *pt*. If not, store DROPEFFECT_NONE in **pdwEffect* and return NOERROR.

3. If you can drop at pt, determine what effect will occur and store that flag in **pdwEffect*.

4. If you want to access *pIDataSource* in ::DragOver, save it here and call *pIDataSource*->AddRef.

5. Provide some user feedback and return NOERROR.

You will notice that you generally return NOERROR even if you cannot accept the data involved. OLE 2.0 drag-drop is such that if you are a registered target, you're ::DragEnter is always called when the mouse moves into you, and your ::DragOver is always called regardless of what you returned as an effect here in ::DragEnter. That is, returning DROPEFFECT_NONE does not prevent further ::DragOver calls, so let's see how we need to handle that function.

For step #1, Schmoo simply calls its CSchmooDoc::FQueryPasteFromData function we implemented in Chapter 7. If you recall I strongly encouraged you to write a function that given and data object pointer would determine if you could paste, that is drop, from it. If you have such a function, this step is simple:

```
if (!m_pDoc->FQueryPasteFromData(pIDataSource))
  {
  *pdwEffect=DROPEFFECT_NONE;   //Drop not possible with this data
  return NOERROR;
  }
```

Schmoo does nothing for step #2 since any point in the document is valid for dropping. Other applications, like Patron, are more stingy as to valid drop points, so I'll defer further discussion of this

step until then.

Step #3 is fairly standard: the OLE 2.0 user interface specifications state that a drag-drop by default means "move," that is, Cut and Paste. If, however, the Control key is down, it means "copy," that is, Copy and Paste. To implement this, set **pdwEffect* to DROPEFFECT_MOVE by default and overwrite it with DROPEFFECT_COPY if *grfKeyState* contains MK_CONTROL:

```
*pdwEffect=DROPEFFECT_MOVE;

if (grfKeyState & MK_CONTROL)
   *pdwEffect=DROPEFFECT_COPY;
```

The Shift key, that is, MK_SHIFT, is used to effect a Paste Link instead of just a Paste when the drop occurs. But we won't deal with linking until Chapters 12 and 13, so I'll again defer this feature until then. That leaves us with two steps to go. Step #4, copying pIDataSource, means saving the pointer and calling AddRef through it as good citizens do:

```
m_pIDataObject=pIDataSource;
m_pIDataObject->AddRef();
```

This is not necessary if you are not going to use the data object in your IDropTarget::DragOver implementation because OLE will pass this same pointer to IDropTarget::Drop when the drop occurs.

Finally, for step #5 we just have to give some visual indication of the drop target. Schmoo uses its CSchmooDoc::DropSelectTargetWindow for this purpose, but to insure that document is visible, it first calls BringWindowToTop to force it in the users face:

```
hWnd=m_pDoc->Window();
BringWindowToTop(hWnd);
UpdateWindow(hWnd);
m_pDoc->DropSelectTargetWindow();

return NOERROR;
```

This use of BringWindowToTop has the nice effect that when you drag across any number of document windows in Schmoo each one is brought to the top as the mouse moves into it. That means the end-user can effectively switch document windows (as if they used the Window menu in an MDI application) during this operation with only the mouse.

IDropTarget::DragOver

This function is called very frequently from within the DoDragDrop function. Not only it is called whenever the mouse moves and whenever the keyboard state changes (for Ctrl, Shift, and mouse buttons), but also for every iteration through an internal loop in DoDragDrop. This latter behavior it necessary to support scrolling in a target document that we'll see later when I discuss Patron's drag-drop implementation.

In any case, your implementation of ::DragOver should be as optimized as possible to insure that the drag-drop operation is crisp. This function generally performs hit-testing on the mouse coordinates, provides any user feedback for the drop target, and returns the effect flag applicable to the current mouse position and keyboard state. OLE 2.0 passes only three parameters to this function: *grfKeyState*, *pt*, and *pdwEffect*. These have the same types and uses as the same named parameters in ::DragEnter (again, *pt* is in screen coordinates). You'll notice that OLE does not pass a data object pointer here, so if you want to do something like IDataObject::QueryGetData as the mouse moves, then you must save a copy of the pointer from ::DragEnter.

I would encourage you to at most use IDataObject::QueryGetData in your implementation here and not call IDataObject::GetData since the latter might be too expensive to insure a smooth operation. If you implementation of this function is too slow, then the user will see choppy mouse movements and a slow response when dragging into your application only. The user is naturally apt to compare your

response to other applications, and finding yours slower will probably wonder why.

In any case, IDropTarget::DragOver can be implemented in three steps:

1. Check if dropping is allowable at *pt* given *grfKeyState* and whatever information you want to obtain from the data object.

2. If you determine you can drop, store the effect flag in **pdwEffect* and provide some sort of visual indication in your window of the result of a drop. If the control key is down, the effect should be copy instead of move (cut). This is also the time to check for scrolling possibilities as we'll see with Patron later.

3. Return NOERROR.

Schmoo executes one quick trick to see if it's even interested in this ::DragOver notification. Remember that even if you return DROPEFFECT_NONE from ::DragEnter (or even if you returned an error in your HRESULT), OLE will still call your ::DragOver here. Therefore Schmoo NULLed its saved IDataObject pointer in ::DragEnter and if it's still NULL in ::DragOver, then ::DragEnter must have initially returned DROPEFFECT_NONE. Therefore we can just return that same effect here making for a very fast response in this function:

```
if (NULL==m_pIDataObject)
   {
   *pdwEffect=DROPEFFECT_NONE;
   return NOERROR;
   }
```

Otherwise Schmoo just checks the keyboard state again as in ::DragEnter and returns DROPEFFECT_MOVE or DROPEFFECT_COPY. Since any point in a Schmoo document is a valid drop point, there's no need to change the user feedback visuals here. So we're done:

```
*pdwEffect=DROPEFFECT_MOVE;

if (grfKeyState & MK_CONTROL)
   *pdwEffect=DROPEFFECT_COPY;

return NOERROR;
```

IDropTarget::DragLeave

All good things come to an end, and so as far as a particular target is concerned the drag-drop operation initiated with ::DragEnter is over when the mouse leaves the target window. In addition, the operation is over if the source instructed OLE to cancel the whole thing. In either of these cases OLE calls IDropTarget::DragLeave to give the target a chance to clean up whatever state it happens to be in. Note that if a drop occurs, however, IDropTarget::Drop is called (see the next section).

DragLeave takes no parameters, so it's fairly straightforward to understand what we must do here:

1. Remove any UI feedback in your window for potential drop results.
2. Release any IDataObject held from DragEnter.
3. Return NOERROR.

As far as Schmoo is concerned, it only needs to reverse the call to CSchmooDoc::DropSelectTargetWindow and to release the data object pointer its holding. Again, if Schmoo decided it could not use the data on ::DragEnter, then its stored IDataObject pointer is NULL and therefore implementation is trivial:

```
if (NULL==m_pIDataObject)
   return NOERROR;

m_pDoc->DropSelectTargetWindow();
m_pIDataObject->Release();
```

IDropTarget::Drop

This function is only called when DoDragDrop called IDropSource::QueryContinueDrag and that source function returned DRAGDROP_S_DROP. OLE then calls this function passing the same parameters as it did to IDropTarget::DragEnter (once again, *pt* is in screen coordinates). The value of *pIDataSource* will be exactly the same as in DragEnter, but the values of the other parameters will likely change (unless the mouse did not move).

This function must re-analyze the data and the drop point and seeing that the data is useful, must try to perform a paste operation, returning the effect flag for what actually happened. This results in the following steps:

1. Remove any UI feedback in your window for potential drop results and release any held IDataObject pointer as in DragLeave.

2. Check if a drop is valid at this point (*pt*) and that you can use the formats available in *pIDataSource*. If you cannot drop here, store DROPEFFECT_NONE in **pdwEffect and* return an HRESULT with E_FAIL.

3. Perform a paste operation from the data available in *pIDataSource*, returning DROPEFFECT_NONE in **pdwEffect* and an HRESULT with E_FAIL if the paste does not work.

4. Otherwise store the appropriate effect in **pdwEffect* and return NOERROR.

The first step here is to perform exactly the same cleanup as we did in DragLeave, so I won't belabor that point again. The fourth step for Schmoo is the same old story of checking the control key and storing either DROPEFFECT_MOVE or DROPEFFECT_COPY in **pdwEffect.*

Schmoo both checks if it can paste and actually does the paste if possible by calling its function CSchmooDoc::FPasteFromData:

```
fRet=m_pDoc->FPasteFromData(pIDataSource);

if (!fRet)
   {
   *pdwEffect=DROPEFFECT_NONE;
   return ResultFromScode(E_FAIL);
   }
```

Again, I encourage you to make a nice function that performs a paste given any data object as I suggested in Chapter 7. With such a function, pasting from IDropTarget::Drop is trivialized.

Schmoo's implementation of ::Drop also has one other oddity:

```
if (m_pDoc->m_fDragSource)
   return ResultFromScode(E_FAIL);
```

This code handles the situation where the drop target object is related to the exact same document that is also the drop source. Drag-drop is essentially a system modal operation, that is, only one such operation can be happening in the system at any given time since it takes over the mouse capture. Therefore when we start a drag from a document we set its m_fDragSource flag to indicate that this document is the source. If a drop occurs on the same document, the drop target can check this flag, since it belongs to the same document, and if TRUE, fails the drop, that is, nothing happens. It makes little sense to drag and drop the same data to the same location from whence it comes, so this code handles that special case.

Register and Revoke the Drop Target Object

The last thing to do to make Schmoo a drop target is to insure that the drop target object, that is, its IDropTarget pointer, is registered with OLE. That means that when the document is ready to accept a

drop, typically on creation, it must instantiate its drop target object and call RegisterDragDrop:

```
m_pDropTarget=new CDropTarget(this);

if (NULL!=m_pDropTarget)
   {
   m_pDropTarget->AddRef();
   RegisterDragDrop(m_hWnd, (LPDROPTARGET)m_pDropTarget);
   CoLockObjectExternal((LPUNKNOWN)m_pDropTarget, TRUE, FALSE);
   }
```

The hWnd passed to RegisterDragDrop is the window of the document window to which the drop target is related. If we fail to create the drop target, or RegisterDragDrop fails, it's really of no consequence; it just means that we won't be a target. Therefore I don't see enough reason to fail document creation if it cannot be a drop target. It can always use the clipboard.

Now what in tarnation is that little CoLockObjectExternal all about?. This, I agree, seems incredibly out of place here in drag-drop code. In my opinion this is a design fault in OLE2.DLL, but what I've been told is you must use this heavy-duty locking function on any object that sits around for a long time and has an interface that is frequently used for short periods of time. IDropTarget is just such an interface (those with linking are affected as well, see Chapters 12 and 13). If you don't do this locking trick, then after you registered your drop target it will receive one drop, and one drop only, because after that first operation OLE internally loses track of the IDropTarget pointer. Therefore the next time DoDragDrop attempts to call your IDropTarget::DragEnter, it finds a bogus pointer and terminates the entire operation. Urk. CoLockObjectExternal prevents the loss of the pointer. Really. Trust me. It works. No, I'm not selling used late-seventies model Ford Pintos. Try taking it out of Schmoo and see what happens...

Anyway, we have to execute the reverse of all this when we no longer want the document to be a target, typically when the document is destroyed. This means calling RevokeDragDrop as a reversal to RegisterDragDrop, CoLockObjectExternal with opposite flags, and releasing the drop target object:

```
if (NULL!=m_pDropTarget)
   {
   RevokeDragDrop(m_hWnd);
   CoLockObjectExternal((LPUNKNOWN)m_pDropTarget, FALSE, TRUE);

   m_pDropTarget->Release();
   }
```

Design and Implement Drop Source User Feedback

Congratulations, you are now more than halfway to a drag-drop implementation. Implementing the drop target side of the picture is much more work than the source side: the source has half the number of interface functions and one third the number of API functions to deal with. In fact, you almost don't have to think to implement this part if you've already implemented OLE 2.0 clipboard handling and have a nice convenient function to create a data transfer object.

That aside for a moment, the first step as a source is to determine what, if any, user feedback you want to provide. You may, for example, wish to visually indicate that your selection will be copied instead of moved. Most of what you need, however, is encompassed in your one **requirement** as a source: to show the appropriate cursor based on the effect flags generated in the target:

§

Changing the cursor is the responsibility of your IDropSource::GiveFeedback implementation, and as we'll see, OLE2.DLL will do all of the work for you.

If you feel that these cursors by themselves are not sufficient to describe what's happening, you may do what you will. For example, you could draw a big skull-and-crossbones across the selected data for

DROPEFFECT_MOVE and draw a camera for DROPEFFECT_COPY.

Determine the Pick Event

What I describe as the 'pick event' is that mouse click or keystroke that puts the source into drag-drop mode, that is, causes a call to DoDragDrop. I bring this up here because you have to know what starts your drag-drop operation in order to implement IDropSource, whose QueryContinueDrag member function determines when the stop the operation.

Most often the pick event will be a WM_LBUTTONDOWN message on some meaningful point. Schmoo, for example, has an 8-pixel border around the Polyline window that I define as the "pick area" such that any mouse down here starts the drag-drop. Conveniently I don't have to do any hit testing because this region is the only portion of the document window's client area that shows–the rest is covered by the Polyline window. Therefore my pick event is any WM_LBUTTONDOWN in the document window.

Now that I know what starts a drag-drop, I know that something like a WM_LBUTTONUP will end it, that is, will try to affect a drop, if possible. But to control that I need to implement IDropSource, which I must also do before attempting to call DoDragDrop.

Implement a Drop Source Object and the IDropSource Interface

SCHMOO.H defines a C++ object class called CDropSource, implemented in IDROPSRC.CPP. Again, this object stands alone like the drop target: it controls its own QueryInterface (which knows IDropSource and IUnknown) and frees itself when Release decrements the reference count to zero. This object needs no data members except a reference count, but I have also included a back pointer to the document just out of habit. I may need it some day if I want a different implementation of IDropSource.[1]

But for Schmoo today, implementation is trivial. First, IDropSouce::QueryContinueDrag is called with two parameters:

fEsc	Indicates that the ESC key is down, usually meaning cancel the drag-drop operation.
grfKeyState	Contains the status of the Shift and Control keys as well as the mouse buttons.

You then use these to determine whether to cancel the operation, continue, or to drop the data:

1. If *fEsc* is TRUE, cancel by returning an HRESULT with DRAGDROP_S_CANCEL. Return the same value for any other canceling condition.

2. Test the opposite condition of the pick event. If you started the operation on a mouse click, test that button state in *grfKeyState*. If the opposite state is TRUE, then cause a drop by returning and HRESULT with DRAGDROP_S_DROP.

3. Otherwise continue dragging by returning NOERROR.

Schmoo's implementation, which will probably be the most common in all applications, is only a few short lines, where a left mouse button down was the pick event:

```
if (fEsc)
   return ResultFromScode(DRAGDROP_S_CANCEL);

if (!(grfKeyState & MK_LBUTTON))
   return ResultFromScode(DRAGDROP_S_DROP);

return NOERROR;
```

Now if you think that's trivial, IDropSource::QueryContinueDrag, which takes only a DWORD *dwEffect* flag, is even simpler if all you want to change is the mouse cursor (your only requirement

1I have a few ideas in mind for later work that will require this, so if you'll excuse the excess...

here). To do that, just tell OLE2.DLL to do it for you:

```
return ResultFromScode(DRAGDROP_S_USEDEFAULTCURSORS);
```

You are, of course, free to do anything else you desire here. If you need to know where the mouse is, then call GetCursorPos. Just remember, however, that this is called just as often as IDropTarget::DragOver, so keep this function as optimized as possible.

Call DoDragDrop

Finally we are ready to make it all work by starting the drag-drop operation. The source in this scenario performs the following steps when the pick event occurs, such as in processing WM_LBUTTONDOWN in Schmoo:

1. Instantiate a drop source object. If this fails, the operation fails.

2. Set a flag indicating that this document is currently the source of the drag to prevent unnecessary action if you drag onto yourself.

3. Create the data object to use in the transfer.

4. Call DoDragDrop, passing the data object, the drop source object, the allowable effects, and a pointer to a DWORD in which to store the final effect. DoDragDrop will not return until your IDropSource::QueryContinueDrag says DRAGDROP_S_CANCEL or DRAGDROP_S_DROP. The return value of DoDragDrop is the same as the return for QueryContinueDrag.

5. Clean up your data object and your drop source object as they are no longer needed. Also reset the source flag set in step 2.

6. If and only if DoDragDrop returned DRAGDROP_S_DROP and if the final effect flag was DROPEFFECT_MOVE, delete the data that was used in this operation. Be sure to set your dirty flag here as well since you are modifying the source document.

Schmoo's implementation looks like this, which can be found inside CSchmooDoc::FMessageHook:

```
if (WM_LBUTTONDOWN==iMsg)
   {
   LPDROPSOURCE    pIDropSource;
   LPDATAOBJECT    pIDataObject;
   HRESULT         hr;
   SCODE           sc;
   DWORD           dwEffect;

   pIDropSource=(LPDROPSOURCE)new CDropSource(this);

   if (NULL==pIDropSource)
      return FALSE;

   pIDropSource->AddRef();
   m_fDragSource=TRUE;

   pIDataObject=TransferObjectCreate();

   hr=DoDragDrop(pIDataObject, pIDropSource
      , DROPEFFECT_COPY | DROPEFFECT_MOVE, &dwEffect);

   pIDataObject->Release();
   pIDropSource->Release();
   m_fDragSource=FALSE;

   sc=GetScode(hr);
   if (DRAGDROP_S_DROP==sc && DROPEFFECT_MOVE==dwEffect)
      {
      m_pPL->New();    //Clears the current Polyline
      FDirtySet(TRUE);
      }
```

```
return TRUE;
}
```

And with all that accomplished, we can now move or copy Polyline figures between different documents in one instance of Schmoo or between document in different instances of Schmoo. But there is one final note to make before looking at a more complicated implementation in Patron.

DoDragDrop internally executes a large loop that first calls various member functions of IDropSource and IDropTarget, then calls PeekMessage to look for and remove any keyboard or mouse (including non-client) messages without yielding. If there is a message then it continues the loop immediately. What can happen here is that if you have a fast keyboard repeat on your machine, and hold down a key like Control during an operation, you may generate a WM_KEYDOWN message for VK_CONTROL more often than DoDragDrop can remove them. That is, the time it takes to execute one loop inside DoDragDrop is longer than your keyboard repeat time. So every time DoDragDrop looks for a message, one is there and it continues to sit in the loop. This means that when you might drag from a source to a target with the Control key held down and then let up on the mouse button, a drop doesn't happen! DoDragDrop is stuck inside its loop until you release the Control key, in which case a move happens instead of a copy. You can even click the mouse button again to no avail. This problem will be addressed in future versions of OLE

Intermission

Ole MacDonald's document was
E-I-E-I-E-OLE
In it was a data source and
E-I-E-I-E-OLE
(With) drag-drop here
And drag-drop there
And drag-drops going everywhere...
Ole MacDonald's document was
E-I-E-I-E-OLE!

Sung to "Ode to Joy" Theme, Beethoven Symphony #9, Fourth Movement
Lyrics inspired by Robert Fulghum

Advanced Drag-Drop: Feedback and Scrolling in Patron

If you've been playing with the version of Patron we developed in Chapter 7, you've probably noticed that it's highly useful except for one major flaw: there's no way to move a tenant around on a page short of enlarging it in from one corner and shrinking it from the opposite one. That's hardly what I'd call a usability feature. No doubt, as someone like Donald Norman[1] would point out, it's rather absurd to ask a user to perform two *sizing* tasks where one *moving* task would be more appropriate.

With OLE 2.0 drag-drop we can now allow you to move a tenant in one swift stroke. This essentially means making a page both a drop source and a drop target and actually doing something when a drag-drop happens on the same page. In that case the page can just change the rectangle for a tenant and repaint. But while we're at it, we might as well be able to drop a graphic from another application into a page, or be able to drag tenants between pages in any open Patron document, in the same application or otherwise. This is exactly why I did not bother to implement any kind of move functionality into Patron in Chapter 7.

1Read The Design of Everyday Things, formerly The Psychology of Everyday Things by Donald Norman, Basic Books, Inc., New York

Patron implements all the same steps as we did for Schmoo above, instantiating drop target and drop source objects, calling the appropriate APIs, and so on. The entire implementation of CDropSource is exactly the same as in Schmoo except that I eliminated the extra back pointer in the drop source object since I have no use for it here. My implementation of IDropTarget is generally the same except for some extensive additions to handle drawing more user feedback and scrolling the page during a drag-drop operation. For these two interfaces I've added two files, IDROPSRC.CPP and IDROPTGT.CPP, to Patron, as well as a third, DRAGDROP.CPP, that contains a few support functions. In addition, I had to made a modification to Patron's CPage::OnLeftDown function (PAGEMOUS.CPP) to start the drag-drop operation. Since the code is too lengthy to list off here, I will include the important fragments in the discussions that follow.

Patron has three main augmentations to drag-drop in Schmoo. The first is that not all points in Patron's documents are "droppable," since some are off the page boundaries or within the unusable page margins. Therefore Patron will have to hit-test the mouse coordinates passed to IDropTarget::DragEnter, ::DragOver, and ::Drop. Secondly, Patron shows more detailed user feedback: when dragging a tenant around on a page, Patron displays an outline of that tenant showing the exact rectangle that tenant would then occupy when dropped. This applies to any tenant that might be dropped on the page, regardless of whether that tenant came from the same document or another instance of Patron. The third change is that Patron provides for scrolling the page while a drag-drop operation is happening.

Patron's notion of the pick region is also more complex than Schmoo's, so let's examine that first before dealing with the more complex features.

Tenant Pick Regions and Drop Sourcing

In Patron I've defined the outer rectangular boundary of any tenant as the 'pick region,' excluding those areas already occupied by sizing handles. The width and height of the region are the same as the dimensions of sizing handles.

§

In Chapter 7 we added the function CPage::OnNCHitTest (PAGEMOUS.CPP) to determine when the mouse was over a sizing handle. We used the return value from this function in CPage::SetCursor to show an appropriate mouse cursor for each sizing handle. For this chapter, we need to first modify OnNCHitTest to check for the boundary, then modify SetCursor to show a four-pointed arrow (obtained from BTTNCUR.DLL) that means "move":

§

When OnNCHitTest determines that the mouse is within a pick region, it stores the code HTCAPTION in CPage::m_uHTCode which is later used by SetCursor to show the cursor above. In addition, if m_uHTCode contains HTCAPTION when WM_LBUTTONDOWN is processed in CPage::OnLeftDown, then Patron drops (pun intended) into the new function CPage::DragDrop. This function creates the data object for use in the operation using CPage::TransferObjectCreate that we added in Chapter 7 to handle clipboard operations. It then calls DoDragDrop, and when that function returns it handles the move or copy operations appropriately.

One special case in CPage::DragDrop is when a tenant is moved within the same page. Here I use the same trick as in Schmoo, setting a flag m_fDragSource to indicate that this document is the current drag-drop source. I also have a flag called m_fMoveInPage which is initially FALSE. Now take a look in IDROPTGT.CPP at IDropTarget::Drop. If m_fDragSource is set, and the last effect was DROPEFFECT_MOVE, then we can set m_fMoveInPage to TRUE. In all other cases it remains FALSE.

```
//In CPage::DragDrop
m_pPG->m_fDragSource=TRUE;
m_pPG->m_fMoveInPage=FALSE;
hr=DoDragDrop(...);

//In CDropTarget::Drop
if (m_pDoc->m_pPG->m_fDragSource && !(grfKeyState & MK_CONTROL))
   {
   *pdwEffect=DROPEFFECT_MOVE;
   m_pDoc->m_pPG->m_fMoveInPage=TRUE;
   m_pDoc->m_pPG->m_ptDrop=po.ptl;
   return NOERROR;
   }
```

When DoDragDrop returns and m_fMoveInPage is TRUE, then we only have to change the tenant's rectangle, repainting the old position and making sure that the new position is properly clipped to the page boundaries. This avoids unnecessarily creating a new tenant in the new position using the data from the original tenant, then destroying the original tenant:

```
//In CPage::DragDrop
if (m_pPG->m_fMoveInPage)
   {
   m_pTenantCur->Invalidate();

   [Code to calculate the new rect and clip it to the page boundaries]

   m_pTenantCur->RectSet(&rcl, TRUE);
   m_pTenantCur->Repaint();
   return TRUE;
   }
```

If the data object is from a different source (m_fDragSource is FALSE), or if we're copying a tenant in the same page (by holding down the Control key during the operation), then IDropTarget::Drop calls CPatronDoc::FPasteFromData. Regardless of where the data actually came from FPasteFromData will create a new tenant with the graphic in the data object. What works so beautifully here is that FPasteFromData doesn't know the difference between copying a tenant already on the current page and a graphic coming from a completely different application. Therefore we don't have to make any special cases to copy a tenant within the same page.

```
//In CDropTarget::Drop
m_pDoc->m_pPG->m_fMoveInPage=FALSE;
fRet=m_pDoc->FQueryPasteFromData(pIDataSource, &fe, &tType);

if (fRet)
   {
   po.fe=(m_pDoc->m_cf==fe.cfFormat) ? m_fe : fe;
   fRet=m_pDoc->FPasteFromData(pIDataSource, &fe, tType, &po, 0);
   }

if (!fRet)
   return ResultFromScode(E_FAIL);

*pdwEffect=DROPEFFECT_MOVE;

if (grfKeyState & MK_CONTROL)
   *pdwEffect=DROPEFFECT_COPY;

return NOERROR;
```

Finally, CPage::DragDrop must be sensitive to whether the operation was a copy or a move (that is, a move from a source other than the same page). When a move occurs, the tenant must be removed from the current page, and so DragDrop calls CPage::TenantDestroy:

```
if (DROPEFFECT_MOVE==dwEffect)
   {
   TenantDestroy();
   return TRUE;
```

```
}
```

More Advanced Drop Target Hit-Testing

The implementation of Schmoo earlier in this chapter had no real considerations for un-droppable points in a document: essentially everywhere was valid. Patron documents, however, show a page surrounded by unusable page margins and a border completely outside the page. Since these are unusable locations, nothing can be dropped there.

The function CPages::UTestDroppablePoint in DRAGDROP.CPP handles this more complex hit-testing, returning a code from the UDROP_* values I've defined in PAGES.H (these are not part of OLE 2.0). For now, the only values of interest are UDROP_NONE (can't drop here) and UDROP_CLIENT (drop is allowed). The other values are related to scrolling discussed in "Scrolling During Drag-Drop" below. UTestDroppablePoint returns UDROP_CLIENT if the point its given is within the intersection of the document's client area and the client-relative rectangle of the usable page regions. Otherwise it returns UDROP_NONE.

```
UINT CPages::UTestDroppablePoint(LPPOINTL pptl)
  {
  POINT      pt;
  RECT       rc, rcT, rcC;
  UINT       uRet;

  POINTFROMPOINTL(pt, *pptl);
  ScreenToClient(m_hWnd, &pt);

  CalcBoundingRect(&rc, FALSE);

  GetClientRect(m_hWnd, &rcC);
  IntersectRect(&rcT, &rc, &rcC);

  //Check for at least a client area hit.
  if (!PtInRect(&rcT, pt))
    return UDROP_NONE;

  uRet=UDROP_CLIENT;

  [Code here for scrolling considerations]

  return uRet;
  }
```

UTestDroppablePoint is first called from IDropTarget::DragEnter to set the initial effect. It's then called on entry into IDropTarget::DragOver to set the effect as well as to determine if further checks for feedback and scrolling are necessary (see the next sections). Finally, it's called again from IDropTarget::Drop to insure that we don't attempt to perform a paste on an invalid drop point.

A Feedback Rectangle

The Chapter 7 version of Patron defined a private clipboard format called "Patron Object" that contained necessary information about the original position of an object within a page:

```
typedef struct tagPATRONOBJECT
  {
  POINTL     ptl;       //Location of object
  POINTL     ptlPick;   //Pick point from drag-drop operation.
  SIZEL      szl;       //Extents of object (absolute)
  FORMATETC  fe;        //Actual object format.
  } PATRONOBJECT, FAR * LPPATRONOBJECT;
```

If Patron found this data on the clipboard, it would attempt to paste whatever graphic (metafile, DIB, or bitmap) was on the clipboard at the position described in this structure. For drag-drop, we'll use this same structure to know how large the graphic is when we see it as a drop target. That means that as a target, whenever we see a data object in IDropTarget::DragEnter, we can check if this format is present.

If it is, then we know exactly how large to draw a feedback rectangle. If it's not, then we punt and show some sort of feedback although it obviously will not be true to the size of the graphic. This avoids possibly having to render the graphic just to determine its size.

As for being a drop source, Patron defines the edge of a tenant's rectangle (excluding those areas with sizing handles) as the 'pick' region. What we want to happen here is that when the end user picks up a tenant, the feedback rectangle is shown with the same relative position to the mouse cursor as when the user picked it up:

§

In order for Patron to know this relative location on any tenant, it stores (as a source) the offset of the mouse cursor from the upper left corner of the tenant in the *ptlPick* field of the PATRONOBJECT structure above. The function CPage::TransferObjectCreate (PAGE.CPP) that we created in Chapter 7 was designed with this in mind. All uses of this function for clipboard transfers store (0,0) in *ptlPick*. When we create a data object for a drag-drop transfer (in CPage::DragDrop) we pass CPage::TransferObjectCreate the real coordinates to store in *ptlPick* (all other calls in Patron to TransferObjectCreate pass a NULL):

```
//In CPage::DragDrop, x & y are mouse coordinates
m_pTenantCur->RectGet(&rcl, TRUE);
ptl.x=x+m_pPG->m_xPos-rcl.left;
ptl.y=y+m_pPG->m_yPos-rcl.top;
pIDataObject=TransferObjectCreate(&ptl);
```

Therefore, as a drop target, Patron will see these coordinates and can then place the feedback rectangle relative to the mouse such that it looks exactly the same as when it was picked up. In CDropTarget::DragEnter, Patron will look for the PATRONOBJECT structure first to determine the real size of the tenant:

```
if (fe.cfFormat==m_pDoc->m_cf)  //m_cf registered using "Patron Object"
   {
   if (SUCCEEDED(pIDataSource->GetData(&fe, &stm)))
      {
      LPPATRONOBJECT  ppo;
      RECT          rc;

      ppo=(LPPATRONOBJECT)GlobalLock(stm.hGlobal);

      SetRect(&rc, (int)ppo->szl.cx, -(int)ppo->szl.cy, 0, 0);
      RectConvertMappings(&rc, NULL, TRUE);
      SETSIZEL(m_szl, rc.left, rc.top);

      m_ptPick=ppo->ptlPick;
      m_fe=ppo->fe;

      GlobalUnlock(stm.hGlobal);
      ReleaseStgMedium(&stm);
      }
   }
else
   {
   SETSIZEL(m_szl, 30, 30);
   m_ptPick.x=0;
   m_ptPick.y=0;
   m_fe.cfFormat=0;
   }
```

During the drag-drop operation m_ptPick will contain the offset of the feedback rectangle from the mouse and m_szl will contain the dimensions of the rectangle. If the PATRONOBJECT structure was not available, then we default to a small rectangle to at least show something. CDropTarget::DragEnter is the first to use m_ptPick and is also the first to call a new function CPages::DrawDropTargetRect (see DRAGDROP.CPP, where this pretty much calls the Windows function DrawFocusRect), passing to it

the upper-left corner of the rectangle as well as the rectangle dimensions. The upper-left corner is simply the mouse coordinates minus the offset of the pick point:

```
//In CDropTarget::DragEnter
pt.x-=m_ptPick.x;
pt.y-=m_ptPick.y;

m_ptLast=pt;
m_fFeedback=TRUE;
m_pDoc->m_pPG->DrawDropTargetRect(&pt, &m_szl);
```

DrawDropTargetRect is a toggle function since DrawFocusRect is an XOR operation. Therefore in order to remove it inside the other CDropTarget functions, we have to remember where the rectangle is and whether it's currently visible. That's the purpose of m_ptLast and m_fFeedback. Early in ::DragOver, ::DragLeave, and ::Drop, we remove the old rectangle if it's showing:

```
if (m_fFeedback)
   ppg->DrawDropTargetRect(&m_ptLast, &m_szl);
```

::DragOver does not make this call if the mouse position did not change, that is, if the new mouse coordinates, offset for the pick location, match m_ptLast:

```
//In CDropTarget::DragOver
if ((pt.x-m_ptPick.x==m_ptLast.x) && (pt.y-m_ptPick.y==m_ptLast.y)
   return NOERROR;
```

This optimization is necessary because CDropTarget::DragOver will be called repeatedly even if the mouse position does not change. Without this special case, the feedback rectangle would continually and annoyingly flicker.

Scrolling the Page

I saved this topic for last simply because it involves a few tricks. To be honest, it took me about 6 working days to figure out Patron's scrolling code (not to mention I had a vacation in the middle here to think about it). I hope my experience will save you some time, because on the surface, scrolling seems simple enough as it's described in less than half a page in the OLE 2.0 specifications.[1] It boils down to two parts:

1. The target defines an 'inset region' or 'hot-zone' within the boundaries of its document *window* (not the usable page area). Any time the mouse cursor is within this region the target ORs in DROPEFFECT_SCROLL with whatever other effect flag is appropriate.

2. When the cursor has remained in the inset region for a given length of time, scrolling starts and continues until the mouse leaves the inset region. That is, you may move the mouse within the inset region and scrolling continues.

The first part is what precipitates the additional code in CPages::UTestDroppablePoint as described before. Not only does this function test whether a point is on a droppable region on the page, but also checks if the point is within the inset region on all sides of the window. But first, we need to know the inset width in pixels, which is done in CPages::CPages:

```
m_uScrollInset=GetProfileInt("windows", "DragScrollInset",
   DD_DEFSCROLLINSET);
```

Until there's another revision of Windows itself, there will not be an entry for "DragScrollInset" in WIN.INI unless an end-user adds one manually, so the inset region will be DD_DEFSCROLLINSET, defined in OLE2.H as 11. UTestDroppablePoint then uses m_uScrollInset to check if the mouse coordinates are within the inset region of the pages window:

```
UINT   uRet;
RECT   rcC;
```

1See *OLE 2 Design Specification*, 15 April 1993, page 282. Note that there are some inaccuracies in this part of the spec. For instance, it uses DRAGEFEECT_* instead of DROPEFFECT_* and incorrectly uses IDropTarget::GiveFeedback which should be IDropSource::GiveFeedback.

```
GetClientRect(m_hWnd, &rcC);

[Code to store UDROP_NONE or UDROP_CLIENT in uRet]

//Scroll checks happen on client area
if (PtInRect(&rcC, pt))
   {
   //Check horizontal inset
   if (pt.x <= rcC.left+(int)m_uScrollInset)
      uRet |= UDROP_INSETLEFT;
   else if (pt.x >= rcC.right-(int)m_uScrollInset)
      uRet |= UDROP_INSETRIGHT;

   //Check vertical inset
   if (pt.y <= rcC.top+(int)m_uScrollInset)
      uRet |= UDROP_INSETTOP;
   else if (pt.y >= rcC.bottom-(int)m_uScrollInset)
      uRet |= UDROP_INSETBOTTOM;
   }
```

The addition UDROP_* flags shown here are mutually inclusive with UDROP_NONE and UDROP_CLIENT, and UDROP_INSETLEFT/RIGHT are inclusive with UDROP_INSETTOP/BOTTOM although LEFT is exclusive of RIGHT as it TOP from BOTTOM. What this means is that we may need to simultaneously scroll both horizontally and vertically if the mouse is with both a horizontal and vertical inset region.

As mentioned before, UTestDroppablePoint is called each time in CDropTarget::DragEnter, ::DragOver, and ::Drop. The important call is the one in ::DragOver where a variable *uRet* contains the current UDROP_* combination and *m_uLastTest* (in the current CPages structure) contains the code from the last cycle through ::DragOver (or the one from ::DragEnter, as the case may be). So on any pass through ::DragOver, which again is called from DoDragDrop on each iteration of its internal loop, we know if we were outside the inset region and moved in, if we were the inset region and moved out, or if we haven't changed from the last pass, in or out of the region.

These cases are handled a little later in CDropTarget::DragOver which is invariably tied to the second part of scrolling: the delay. The delay exists because OLE 2.0 drag-drop is a mechanism for dragging from any arbitrary source to any arbitrary target and that involves a much mouse motion across the edges of windows. If scrolling began as soon as the mouse hit an inset region, then it would be almost impossible for the end user to drag out of a window without scrolling it a little. Therefore the mouse must remain in the inset region for a short period of time before scrolling begins. This delay, in milliseconds, is also a new OLE 2.0 addition to WIN.INI which you again retrieve with GetProfileInt:

```
m_uScrollDelay=GetProfileInt("windows", "DragScrollDelay",
   DD_DEFSCROLLDELAY);
```

The default value for DD_DEFSCROLLDELAY is unfortunately defined as 50 (ms) which most of you know is shorter than the 55ms timer resolution in Windows. Therefore you can't really time this value, and my experience with trying suggests that it's too short anyway when you don't have any mouse acceleration. I recommend you create a line in WIN.INI that reads "DragScrollDelay=200" for testing purposes.

In any case, you now have the problem of timing subsequent calls to ::DragOver to detect when the delay has elapsed. There are two ways to do this: use a timer, or use GetTickCount. A timer might seem the easiest way to go first, but it's really a pain because the code to handle the timer expiration will be in a different function (NOT fun for C++ programmers) and because a WM_TIMER message (or a call to a timer proc) will only be generated if you have a chance to pass through your message loop. As I found out, my message loop may not always run when I'm inside DoDragDrop, especially when I'm dragging within the same document. Since this does not yield, my message loop is locked down in

DispatchMessage on WM_LBUTTONDOWN. Therefore I will not receive any messages, no timer events.

That leaves me with the better alternative anyway, which keeps all the timing within CDropTarget::DragOver. When I first detect that I have entered the inset region, I call the Windows function GetTickCount to read the current system time, storing it in a variable *m_dwTimeLast*. OK, fine. The next time through ::DragOver, I may or may not still be in the inset region. If I am, then I call GetTickCount, subtract from it the time I read when first moving into the region, and if the difference is greater than the delay time, I can start scrolling. Otherwise, if I moved out of the inset region, I reset *m_dwTimeLast* to zero (meaning 'no scroll under any circumstances'). This is all wrapped up in some repetitive-looking code in ::DragOver, where *uLast* is the value in *m_uLastTest* and *ppg* is the current CPages pointer:

```
if ((UDROP_INSETHORZ & uLast) && !(UDROP_INSETHORZ & uRet))
   ppg->m_uHScrollCode=0xFFFF;

if (!(UDROP_INSETHORZ & uLast) && (UDROP_INSETHORZ & uRet))
   {
   ppg->m_dwTimeLast=GetTickCount();
   ppg->m_uHScrollCode=(0!=(UDROP_INSETLEFT & uRet))
      ? SB_LINELEFT : SB_LINERIGHT;   //Same as UP & DOWN codes.
   }

if ((UDROP_INSETVERT & uLast) && !(UDROP_INSETVERT & uRet))
   ppg->m_uVScrollCode=0xFFFF;

if (!(UDROP_INSETVERT & uLast) && (UDROP_INSETVERT & uRet))
   {
   ppg->m_dwTimeLast=GetTickCount();
   ppg->m_uVScrollCode=(0!=(UDROP_INSETTOP & uRet))
      ? SB_LINEUP : SB_LINEDOWN;
   }

if (0xFFFF==ppg->m_uHScrollCode && 0xFFFF==ppg->m_uVScrollCode)
   ppg->m_dwTimeLast=0L;

//Set the scroll effect on any inset hit.
if ((UDROP_INSETHORZ | UDROP_INSETVERT) & uRet)
   *pdwEffect |= DROPEFFECT_SCROLL;
```

This first block of code shows how Patron handle each case where we moved into or out of an inset region, that is, the conditions changed, as well as where it ORs in DROPEFFECT_SCROLL. The CPages variables *m_uHScrollCode* and *m_uVScrollCode* contain the code to send with WM_HSCROLL and WM_VSCROLL messages, where 0xFFFF us my flag to mean 'no scrolling.'

CDropTarget::DragOver then checks for expiration of the time delay, and if it has elapsed, then it sends the appropriate scroll messages:

```
if (ppg->m_dwTimeLast!=0
   && (GetTickCount()-ppg->m_dwTimeLast) > (DWORD)ppg->m_uScrollDelay)
   {
   if (0xFFFF!=ppg->m_uHScrollCode)
      {
      m_fPendingRepaint=TRUE;
      SendMessage(ppg->m_hWnd, WM_HSCROLL, ppg->m_uHScrollCode, 0L);
      }

   if (0xFFFF!=ppg->m_uVScrollCode)
      {
      m_fPendingRepaint=TRUE;
      SendMessage(ppg->m_hWnd, WM_VSCROLL, ppg->m_uVScrollCode, 0L);
      }
   }
```

This will send both WM_HSCROLL and WM_VSCROLL messages in the same pass through ::DragOver if necessary. This now brings us to the final area of consideration: repainting.

What we want to happen in Patron is that drag-drop scrolling was fast, not requiring a repaint on every scroll. With many tenants on a page, each scroll would be painfully slow. To prevent the repaints we need the *m_fPendingRepaint* flag that's FALSE unless a scroll has occurred in which case it's set to TRUE. This flag is used in ::DragOver, ::DragLeave, and ::Drop to repaint the page when scrolling has stopped. The last two cases are obvious: moving out of the window or dropping stops scrolling.
In ::DragOver, however, we have to determine if the last SendMessage did, in fact, change the scroll position of the page. Therefore before executing the code above, we save the current scroll positions in some local variables:

```
xPos=ppg->m_xPos;
yPos=ppg->m_yPos;
```

Then after we possibly sent WM_*SCROLL messages, we check the previous scroll positions against the new ones, and if they are the same, and there's a pending repaint, then repaint:

```
if (xPos==ppg->m_xPos && yPos==ppg->m_yPos && m_fPendingRepaint)
   {
   UpdateWindow(ppg->m_hWnd);
   m_fPendingRepaint=FALSE;
   }
```

::DragLeave and ::Drop will always call UpdateWindow if m_fPendingRepaint is TRUE.

There is one last consideration that caused me much consternation. When I first implemented Patron's drag-drop scrolling, I mostly tested it by moving tenants around on the same page, or between different documents in the MDI version. Everything worked great. Then I tried to drag something in from another application, like Schmoo. Things fell apart, because since there was LRPC going on, there were plenty of yields happening and my message loop had a chance to iterate. This didn't happen before because there's no yielding when both source and target are the same application!

The result was that Patron would receive WM_PAINT messages, since scrolling, of course, invalidates regions of my client area. Normally this would not have been a problem except for my little feedback rectangle. What would happen is this: my CDropTarget::DragOver was called, I removed the previous feedback rectangle, I scrolled the page, then I drew the new feedback rectangle over a possibly invalid region of the window. When WM_PAINT came along, it repainted that invalid region, erasing parts of my feedback rectangle. Then I came back into ::DragOver and attempted to erase the old feedback rectangle again. Since part of it was already gone, and since my rectangle drawing is based on an XOR, I ended up rectangle fragments left on the screen. U-G-L-Y. I tried a number of things, like ignoring the WM_PAINT messages (which didn't work at all, I should know better!), and finally arrived at a solution after a few more days of going absolutely nowhere. I maintain a flag in CPages::m_fDragRectShown which is modified only in DrawDropTargetRect: TRUE if the rectangle is visible, FALSE otherwise. If this flag is set when processing WM_PAINT (PagesWndProc in PAGEWIN.CPP) I call DrawDropTargetRect with NULLs[1] to erase the current rectangle, do the painting as usual, then call DrawDropTargetRect again to reinstate the feedback. Now everything came out clean, and I could finally move Patron into Chapter 9. Yes, even us authors still struggle with some aspects of programming for Windows!

Summary

OLE 2.0's drag-drop mechanism is a streamlining of clipboard operations. Instead of having the end user select data in the source application, choose Edit/Copy or Edit/Cut, switch to the target or consumer application, and select Edit/Paste, the end user can simply pick the data up in the source application, drag it to the intended target, and drop it, all in one swift stroke. You can, as with the clipboard, transfer any data using drag-drop, as the mechanism uses a data object.

The two agents in a drag-drop operation are the source and target, which may or may not be the

1DrawDropTargetRect internally tracks the last rectangle position in static variables, and uses these if passed NULLs.

same application, the same document, or different applications. Since OLE 2.0 sits between source and target, the two are unaware of each other. Therefore a target consumes data from any source, even if that source is itself (effectively a copy/paste to yourself), and a source provides data without knowing the eventual target. Applications can, of course, maintain state flags to detect operations in which the source and target documents are identical.

The source of the drag-drop operation is responsible for providing the data object with the data, for implementing a drop source object with the IDropSource interface, for calling the DoDragDrop API in OLE2.DLL, and handling the impact of a move (cut) operation where the data in the source is removed once it has been transferred to the target.

The target of a drag-drop operation is responsible for implementing a drop target object with the IDropTarget interface, which is typically part of a document. This drop target object is registered for the document window using the OLE 2.0 API RegisterDragDrop when the document is available as a target. When the document is no longer available, it must remove the drop target object by calling RevokeDragDrop. In addition, the document must call CoLockObjectExternal to insure that OLE 2.0 does not internally lost track of the drop target object. Once registered, the drop target waits around for calls into its IDropTarget interface, at which time it determines if it can use the data, decides what will happen if the data is dropped at the current mouse coordinates, provides user feedback as to what might happen, and possibly scrolls the target document.

Both Schmoo and Patron applications are modified to support drag-drop in this chapter. Schmoo is a simple case whereas Patron has a number of complexities due to more advanced features, such as scrolling. Drag-drop in Schmoo allows convenient transfer of its figures between instances of itself as well as convenient transfers of metafiles and bitmaps of those figures into other applications. Drag-drop in Patron not only allows it to accept graphics from other applications (like Schmoo) but also allows us to add the capability to move a tenant within a page or between separate instances of Patron. Adding the ability to drag and drop compound document objects will be icing on the cake, as we'll see in the next chapter.