

"Quite simply, this book is a must-have for any serious MFC developer."

—Dean McCrory, MFC Development Lead

MFC Internals

*Inside the Microsoft® Foundation
Class Architecture*

GEORGE SHEPHERD
SCOT WINGO





Digitized by the Internet Archive
in 2017

<https://archive.org/details/mfcinternalsinsi00shep>

MFC Internals

Inside the Microsoft® Foundation
Class Architecture

George Shepherd and Scot Wingo

Foreword by Dean D. McCrory



Addison-Wesley Developers Press

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Harlow, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan
Paris • Seoul • Milan • Mexico City • Taipei

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters or all capital letters.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

Shepherd, George (George R.), 1962-
MFC internals : inside the Microsoft Foundation class architecture
/ George Shepherd and Scot Wingo.
p. cm.
Includes index.
ISBN 0-201-40721-3
1. Microsoft Windows (Computer file) 2. Microsoft foundation
class library. 3. C++ (Computer program language) I. Wingo,
Scot. II. Title.
QA76.76.W56S4967 1996
005.26'2—dc20 95-52102
CIP

Copyright © 1996 by George Shepherd and Scot Wingo
A-W Developers Press is a division of Addison-Wesley Publishing Company.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Claire Horne
Project Manager: John Fuller
Production Coordinator: Ellen Savett
Cover design: Jean Seal
Set in 11-point Times by Rob Mauhar, CIP of Coronado

2 3 4 5 6 7 89-MA-0099989796
Second Printing, June 1996

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Find A-W Developers Press on the World-Wide Web at:
<http://www.aw.com/devpress/>

To Kris, for putting up with the long hours, CT years,
loud music, and me—and, most importantly, for being a great
friend and wife.

—S.W.

To my mother, Elizabeth Shepherd, and to the memory of
my father, George R. Shepherd.

—G.S.



Contents

<i>Foreword</i>	xvii
<i>Acknowledgments</i>	xix
<i>Introduction</i>	xxi
Why MFC Internals?	xxi
MFC Internals to the Rescue!	xxii
Who Will Benefit from This Book	xxiii
How to Use This Book	xxiii
A Word of Caution	xxiv
Contacting the Authors	xxv
 Chapter 1	A Conceptual Overview of MFC 1
Some Background on Object-Oriented Programming	1
OOP Terminology	2
Abstraction	2
Encapsulation	2
Inheritance	2
Polymorphism	3
Modularity	3
Objects in General	3
Objects and C++	4
Why Use OOP?	6
Application Frameworks and MFC	7
A Little History	7
Design Goals for MFC	8
The First Release	10

But That Wasn't Enough	10
Where Are We Now?	11
A Grand Tour of MFC	12
General-Purpose Classes	12
Windows API Classes	16
Application Framework Classes	21
High-Level Abstractions	22
Operating System Extensions	24
Conclusion	28
 Chapter 2 Basic Windows Support	29
MFC versus C/SDK	30
All That (Boilerplate) Code	31
Comparing the Code	32
The WinMain() Function	36
Initializing a Particular Instance of the Application	37
The Message Loop	37
Message Handling	38
Basic MFC Application Components	39
CWinApp: The Application Object	39
CWnd: The Base Window Class	42
Turning Window Handles into Window Objects	44
Attaching and Detaching Window Handles	46
Find WinMain() Now	46
MFC State Information	48
Back to WinMain()	51
Initializing the Framework: AfxWinInit()	52
Some Other Hidden Cool Stuff	54
Registering Window Classes	54
MFC's Windows Hooks	57
MFC's Message Pump: CWinApp::Run()	58
MFC's GDI Support	58
Device Contexts	59
Graphic Objects	61
Conclusion	63

Chapter 3	Message Handling in MFC	65
	CCmdTarget and Message Maps	65
	Window Messages	66
	Message Handling Using C and the SDK	67
	Windows and C++	69
	MFC Message-Mapping Internals	70
	The CCmdTarget Class	70
	Message Map Data Structures	70
	Message Map Macros	71
	How MFC Uses Message Maps	75
	How MFC Windows Become Wired to a WndProc	76
	Handling Messages	77
	Handling WM_COMMAND	81
	Handling Regular Window Messages	87
	Other Kinds of Messages	89
	Hooking into the Message Loop: PreTranslateMessage()	91
	Conclusion	92
Chapter 4	The MFC Utility Classes	93
	Simple Value Types	93
	Class CString: A char * on Steroids	94
	Other Simple Value Types	106
	MFC Collection Classes	107
	MFC Collection Shapes	108
	MFC Array Collections	108
	Lists	114
	MFC Map Collections	122
	The CFile Family: MFC Access to Files	129
	Using CFile	130
	CFile Internals	131
	CStdioFile	133
	CMemFile	136
	There Is Another . . .	141
	CFile Summary	142
	CException: Providing Better Error Handling	143
	CException Internals	145
	Conclusion	148

Chapter 5	All Roads Lead to CObject	151
Isn't That Expensive?		151
CObject Features		151
An Introduction to Macros		153
Run-Time Class Information		153
RTCI: How Does It Work?		155
Dynamic Creation		159
What's Left to Learn about CRuntimeClass?		161
Persistence in MFC		162
Adding Serialization to Your Classes		162
Serialization: How It Works		163
CObject and Serialization		169
Tracing a Write and a Read of a CObject Derived Object		170
Serialization Performance		173
A CRuntimeClass Status Update		174
CObject Diagnostic Support		174
Diagnostic Output		174
Run-Time Checking		175
Memory Diagnostics		177
Inside CObject Diagnostic Support		180
Diagnostic Output		180
Advanced Memory Diagnostics		187
Back to CMemoryState		192
Cool AFX Helper Functions		195
Putting It All Together		196
Putting It into Practice		197
Is It Worth It?		198
Conclusion		199
Chapter 6	MFC Dialog and Control Classes	201
CDialog: Modal and Modeless MFC Dialogs		201
Using CDialog		201
CDialog Internals		204
CDialog Control Initialization		212
DDX/DDV: CDialog Exchange and Validation		215
MFC Common Dialogs		225
Using the MFC Common Dialog Classes		226
Common Dialogs Internals		227

MFC Common Dialog Wrapup	235
OLE Dialogs	236
Using the OLE Dialogs	237
MFC OLE Dialog Class Internals	237
OLE Dialog Summary	239
Property Sheets (a.k.a. Tabbed Dialogs)	239
Using MFC Tabbed Dialogs	240
Property Sheet and Page Internals	241
Property Sheet Recap	249
MFC Control Classes	250
The "Old-Fashioned" Windows Control Classes	250
The "New-Fangled" MFC Windows Common Control Classes	256
Conclusion	257
 Chapter 7 MFC's Document/View Architecture	259
Why You Want Document/View	259
Other Reasons	260
The Old Way	260
The Architecture	261
Documents and Views	261
Document/View Components	262
CWinApp's Role	266
Inside the Document/View Architecture	267
The CWinApp/CDocTemplate Interface: CDocManager	267
CDocTemplate: CDocument, CView, and CFrameWnd Manager	273
CFrameWnd Internals	279
CDocument Internals	282
CView Internals	288
Document/View Internals Recap	289
Document/View Interdependencies	289
Creation Information	290
Putting It All Together	290
Conclusion	293
 Chapter 8 Advanced Document/View Internals	295
Mirror, Mirror, on the Wall . . .	295
Inside CMirrorFile	296

CView Printing	299
CView Printing Overview	300
CView Printing Internals	302
Inside CView Print Preview Support	307
CView::OnFilePrintPreview()	308
CPrintPreviewState	309
Inside DoPrintPreview()	309
CPreviewView: An Undocumented Print Preview CView Derivative	311
Inside CPreviewView::SetPrintView()	314
Print Preview Wrapup	316
CView Derivatives: CScrollView	317
How CScrollView Works	317
CScrollView Recap	323
CFormView: Forms in a View	324
CFormView Recap	327
Another CView Derivative: CCtrlView	327
CCtrlView: How It Works	327
CTreeView: An Example Control View/CCtrlView Derivative	329
CCtrlView Wrapup	330
Conclusion	331
 Chapter 9 MFC's Enhanced User-Interface Classes	 333
CSplitterWnd: MFC Splitter Windows	333
The Anatomy of a Splitter Window	334
Refresher: How to Use CSplitterWnd	335
Inside CSplitterWnd	336
CSplitterWnd Recap	363
The MFC CControlBar Architecture	365
Step Right Up to the CControlBar	365
CControlBar Persistence	382
CControlBar Layout Management	390
CControlBar Recap	391
CMiniFrameWnd	392
MFC MRU File List Implementation	394
How MFC Implements MRU File Lists	394
Conclusion	396

Chapter 10	MFC DLLs and Threads	399
Understanding States	399	
MFC States Explained	400	
How the MFC States Are Related	405	
MFC DLLs	407	
DLL Resource Issues	408	
Extension DLL Initialization and Cleanup	414	
AFXDLL and Macros	414	
MFC Threads	417	
MFC Worker Threads	417	
MFC User-Interface Threads	424	
A Tale of Threads, Handles, and Objects	429	
Conclusion	429	
For More Information	430	
Up Next	431	
Chapter 11	How MFC Implements COM	433
MFC and OLE	434	
The Component Object Model	434	
What Is a COM Class?	435	
Do You Speak I-Speak?	437	
Globally Unique Identifiers	439	
Exploring the Great IUnknown	439	
Object Lifetime Management	440	
Interface Negotiation	442	
A Peek at the Client Side: Call/Use/Release	443	
COM Object Servers	444	
In-Proc Servers (DLLs)	445	
Out-of-Proc Servers (EXEs)	445	
Class Factories	446	
Exposing the Class Factory in an In-Proc Server	449	
Exposing the Class Factory in an Out-of-Proc Server	449	
Unloading In-Proc Servers	450	
Unloading Out-of-Proc Servers	451	
Class Registration in the Registry	452	
Creating Instances of COM Classes	453	
COM Classes with Multiple Interfaces	454	
Implementing CoMath Using Multiple Inheritance	455	

Implementing CoMath Using Nested Classes	458
A Class Factory for CoMath	462
MFC COM Classes	464
Using MFC to Create CoMath	467
IUnknown and CCmdTarget	467
COM Aggregation	468
The Internal IUnknown Functions	473
The External IUnknown Functions	474
Multiple Interfaces through Nested Classes	474
The MFC COM and Interface Map Macros	475
Declaring the Nested Classes	475
Building the Interface Map	477
The CoMath Class Using MFC	479
MFC COM Classes and Inheritance	482
InternalQueryInterface() Revisited	484
Finishing the Server	485
Implementing the Interfaces	485
MFC Support for Class Factories	487
COleObjectFactory	487
Developer Tip: Registering Other Information	491
The Heart of COleObjectFactory: OnCreateObject()	493
COleObjectFactory and IClassFactory::LockServer()	493
Creating Class Factories within Your App	494
COleObjectFactory and Interface Maps	495
Exporting the Class Factory from a DLL	497
Conclusion	498
 Chapter 12 Uniform Data Transfer and MFC	501
Some History	501
The Old Way	502
Limitations of the Windows Clipboard and DDE	503
OLE and Uniform Data Transfer	504
Important Structures	504
The FORMATETC Structure	504
The STGMEDIUM Structure	506
The IDataObject Interface	507
IDataObject::GetData()	508

IDataObject::GetDataHere()	508
IDataObject::QueryGetData()	508
IDataObject::GetCanonicalFormatEtc()	508
IDataObject::SetData()	508
IDataObject::EnumFormatEtc()	509
IDataObject::DAdvise()	509
IDataObject::DUnadvise()	509
IDataObject::EnumDAdvise()	509
The OLE Clipboard	509
MFC's IDataObject Classes	511
Transferring Data via the Clipboard	511
Delayed Rendering	515
MFC's IDataObject Classes in Detail	516
COleDataSource	517
COleDataObject	523
OLE Drag-and-Drop	526
IDropSource	526
IDropTarget	526
Implementing Drag-and-Drop Data Transfer Using MFC	528
Originating a Drag-and-Drop Transfer	528
Implementing a Drop Target	529
Inside MFC's Drag-and-Drop Classes	531
How MFC Drag-and-Drop Works	534
Conclusion	538
 Chapter 13 OLE Documents the MFC Way	539
OLE Documents 101	540
Linking and Embedding	540
Structured Storage, Compound Files, and Persistent Objects	542
In-Place Activation and Visual Editing	547
OLE Document Containers	548
OLE Document Servers	549
The OLE Document Protocol	550
MFC's Support for OLE Documents	550
The Base Classes: CDocItem and COleDocument	550
OLE Document Containers the MFC Way	552
COleLinkingDoc	554

OLE Document Servers the MFC Way	557
COleServerDoc	557
COleServerItem	561
The Container/Server Dance (Embedding)	561
The Container: Creating a New File	561
Adding Container Items	562
Deactivating the Item	575
Inside COleClientItem::Close()	575
Saving the Container's Document	577
Loading OLE Documents	578
Conclusion	580
 Chapter 14 MFC and Automation	581
The History of Automation	581
What Automation Can Do for You	582
Writing an MFC Automation Application	584
But How Does It All Work?	584
COM Interfaces versus Automation	585
COM Interfaces Reviewed	585
IDispatch: The Key to Automation	587
Implementing IDispatch by Hand	591
Another Way: Using Type Information	597
Type Information	597
Object Description Language	597
Implementing IDispatch Using Type Information	599
Recap: Automation in the Raw	601
MFC and Automation	602
Extensions to CCmdTarget	602
Dispatch Maps	604
CCmdTarget and GetIDsOfNames()	611
CCmdTarget and Invoke()	613
MFC and Type Information	614
Conclusion: The Consequences of "The MFC Way"	617
 Chapter 15 OLE Controls	619
VBXs and Their Shortcomings	620
OLE Controls	620

Writing a Control	621
Using OLE Controls in a Project	622
OLE Controls in a Dialog Box	622
Using an OLE Control in a View	623
So How Does It All Work?	624
MFC's OLE Control Classes	624
MFC and OLE Control Containers	628
COleControlContainer	628
COleControlSite	628
COccManager	629
A Day in the Life of an OLE Control	629
Control Creation	630
Inside the Control	632
Finishing Creating the Control	633
OLE Connections	634
OLE Connection Interfaces	635
Establishing a Connection	636
Establishing a Connection (Continued)	639
OLE Control Events	639
Firing Events	641
MFC OLE Control Events	641
How MFC Handles Events	642
Developer Tip: Adding an Event Sink to a View	643
Adding a Function to Handle the Event	644
Setting Up the Event Sink Map	645
Inside OLE Control Property Pages	646
Property Pages in a Nutshell	647
Programming the Property Page	648
Inside the Properties Verb	649
Inside COlePropertyPage	651
Accessing the Properties	652
Inside GetPropCheck()	653
Property Page Wrapup	653
Conclusion	654
 <i>Appendix A A Field Guide to the MFC Source Code</i>	 655
MFC Coding Techniques	656
Class Declaration Subsections	656

Variable Naming—Think Hungarian!	657
Symbol-Naming Conventions	658
The Proof Is in the Pudding!	658
Common MFC #defines	659
Granularity and Swap Tuning?!	660
Tools for Exploring MFC	661
The Visual C++ Browser	661
Visual C++ Find in Files	662
A Visual C++ Syntax-Coloring Tip	662
Commercial Products	662
A Guide to the MFC Source Code	662
MFC Directory Structure	662
MFC Header Files	664
MFC Inline Files	672
MFC Resources	673
MFC Source Files	674
Happy Trails . . .	687

<i>Appendix B The MFC Internals Floppy</i>	689
--	-----

<i>Index</i>	691
--------------	-----

As you can see, MFC's `IDataObject` classes (`COleDataSource` and `COleDataObject`) wrap Clipboard data transfers very well. They also do a great job implementing OLE drag-and-drop. In fact, doing drag-and-drop is very similar and only takes a little bit more work to implement.

OLE Drag-and-Drop

Like OLE Clipboard transfers, OLE drag-and-drop involves getting an `IDataObject` from a data producer to the consumer. Clearly, the data producer has to implement `IDataObject`, and the data consumer has to know how to deal with an `IDataObject` pointer.

In addition to `IDataObject`, OLE drag-and-drop involves two other interfaces: `IDropSource` and `IDropTarget`. Here's the `IDropSource` interface.

IDropSource

`IDropSource` handles the user interface for drag-and-drop operations. It's the data producer's job to implement this interface. Here's the actual interface definition:

```
interface IDropSource : public IUnknown {
public:
    virtual HRESULT QueryContinueDrag(BOOL fEscapePressed,
                                      DWORD grfKeyState) = 0;
    virtual HRESULT GiveFeedback(DWORD dwEffect) = 0;
};
```

- `QueryContinueDrag()` determines whether to continue or to cancel a drag-and-drop operation.
- `GiveFeedback()` changes the mouse cursor according to the drop effect flags (`DROPEFFECT_NONE`, `DROPEFFECT_MOVE`, `DROPEFFECT_COPY`, and others).

IDropTarget

As its name indicates, `IDropTarget` handles the target end of a drag-and-drop operation. COM objects that wish to be drop targets implement `IDropTarget`. At the raw COM level, an OLE API function called `RegisterDragDrop()` associates a window handle to the COM object implementing `IDropTarget`. Once a drop target has been registered, OLE can notify the drop target whenever mouse dragging action is occurring

over the window specified in RegisterDragDrop through one of IDropTarget's four member functions, as shown in Listing 12-3.

Listing 12-3. The IDropTarget interface

```
interface IDropTarget : public IUnknown {
public:
    virtual HRESULT DragEnter(IDataObject *pDataObj,
                             DWORD grfKeyState,
                             POINTL pt,
                             DWORD *pdwEffect) = 0;

    virtual HRESULT DragOver(DWORD grfKeyState,
                             POINTL pt,
                             DWORD *pdwEffect) = 0;

    virtual HRESULT DragLeave( void);

    virtual HRESULT Drop(IDataObject *pDataObj,
                         DWORD grfKeyState,
                         POINTL pt,
                         DWORD *pdwEffect) = 0;
};
```

- OLE calls IDropTarget::DragEnter() whenever a mouse cursor involved in a drag-and-drop operation enters the window.
- OLE calls IDropTarget::DragOver() multiple times as the mouse cursor representing a drag-and-drop operation moves around over the window.
- OLE calls IDropTarget::DragLeave() whenever the drag-and-drop mouse cursor leaves the window.
- OLE calls IDropTarget::Drop() when the mouse button is released over a drop target window.

Finally, OLE has an API function called DoDragDrop(), which commences the drag-and-drop operation. Once called, DoDragDrop() enters a modal loop, monitoring changes in the mouse and keyboard (especially the ESCAPE key, which cancels it) until the user finishes the drag-and-drop operation by either releasing the mouse button or by pressing the Escape key.

MFC provides a very convenient way to do drag-and-drop transfers: you'll use COleDataSource, COleDataObject, and two other classes, called COleDropTarget and COleDropSource.

Implementing Drag-and-Drop Data Transfer Using MFC

Drag-and-drop is a shortcut for doing Clipboard transfers. Drag-and-drop data transfer is interactive, so its requirements are different from those of Clipboard data transfers. The basic idea behind drag-and-drop data transfers is that you can select the data you want by pressing the left mouse button. Then, as you continue to hold down the left button, you move the mouse cursor (dragging the data) into another window (the one to contain the new data). Once the mouse cursor is over the second window, you drop the data by releasing the left mouse button.

You can further control the kind of transfer by using one or more of the control keys:

<i>Key Pressed</i>	<i>Type of Transfer</i>
No control keys	Move data
CONTROL	Copy data
CONTROL-SHIFT	Link data
ALT	Move data

As with Clipboard data transfers, drag-and-drop transfers involve both a data source and a data destination—a data producer and a data consumer. In both cases, the data producer uses a `COleDataSource` to originate the data transfer and the data consumer uses a `COleDataObject` to retrieve the data. In fact, the code for pasting from the Clipboard and that for receiving data in a drag-and-drop operation are usually identical. Many times you can use the same code for retrieving data from a drag-and-drop operation as you would use for a Clipboard paste operation. However, there are a few subtle differences between Clipboard data transfers and drag-and-drop data transfers.

Originating a Drag-and-Drop Transfer

Drag-and-drop data transfers usually begin when the user presses the left mouse button. So, to initiate a drag-and-drop transfer, simply handle the `WM_LBUTTONDOWN` message.

The mechanics for the data producer are very similar: copy some data into a `COleDataSource` object using `CacheData()` or `CacheGlobalData()`. However, instead of calling `COleDataSource::SetClipboard()` to set the data to the Clipboard, call `COleDataSource::DoDragDrop()`, which handles the data transfer automatically. The following snippet demonstrates how to initiate a drag-and-drop operation:

```

COleDataSource cods;

LPCSTR source = GetString();
HGLOBAL h =
    GlobalAlloc(GHND | GMEM_SHARE, strlen(source) + 1);
strcpy(LPSTR(GlobalLock(h)), source);
GlobalUnlock(h);
cods.CacheGlobalData(CF_TEXT, h);

DROPEFFECT de =
    cods.DoDragDrop(DROPEFFECT_COPY | DROPEFFECT_MOVE);
if ( de == DROPEFFECT_MOVE ) {
    // Do any special stuff depending on drop effect
}

```

Notice how `DoDragDrop()` accepts a combination of `DROPEFFECT...` values. Those values indicate what kind of transfer operations are allowed. In this example, the program allows copy and move transfer operations.

By calling `COleDataSource::DoDragDrop()`, you hand off the data to OLE. `DoDragDrop()` does not return until the user has dropped the data somewhere else (by releasing the left mouse button) or has canceled the transfer by pressing the `ESCAPE` key.

Implementing a Drop Target

The data consumer of a drag-and-drop operation is called a drop target. Implementing a drop target using MFC is very straightforward. The two classes usually involved in creating one: `COleDropTarget` and `CView`. You can also use a `CWnd` for the target, but you get less default functionality, like no automatic scrolling, for example.

The steps necessary to enable a `CView` to be a drop target are these: (1) register a `COleDropTarget` with a `CView` and (2) override four virtual functions—`OnDragEnter()`, `OnDragOver()`, `OnDragLeave()`, and `OnDrop()`.

To register a view as a drop target, declare a member variable of type `COleDropTarget` in your view class. Then handle the view's `OnCreate()` method (called right after a view is created). Inside `OnCreate()`, call `COleDropTarget::Register()`, sending it the view's this pointer. Underneath the hood, `COleDropTarget()` calls the `RegisterDragDrop()` function, which registers a window handle as a drag-and-drop target within OLE.

Once a window is registered with OLE to be a drop target, it becomes eligible to receive notifications that correspond to `CView`'s `OnDragEnter()`, `OnDragOver()`, `OnDragLeave()`, `OnDrop()` functions. These functions are important because they allow the user interface to be updated during the drag-and-drop operation.

Drag-and-drop transfers involve changes in the user interface. The normal feedback for drag-and-drop operations is represented by one of several standard cursors. If a

window cannot accept a data drop, then the program should display a circle with a slash through it to show that the window is not a drop target. If the window can accept a drop and the data is being moved (as opposed to being copied), then the cursor should appear as the regular arrow cursor but with a small square (representing data) to the lower right of the arrow. If the window can accept a drop and the data is being copied, then the program should show the same cursor it does when data is being moved, except that there should be a small plus sign to the upper right of the arrow. Fortunately, MFC handles the cursor changes very easily through the return values of the drag-and-drop handler functions. Here's how the four drag-and-drop notification functions work.

Whenever the mouse cursor enters a window registered as a drop target, the framework calls `OnDragEnter()`. The framework calls the view's `OnDragOver()` function as the mouse cursor is dragged over a drop target.

`OnDragEnter()` and `OnDragOver()` are very closely related. Both accept three parameters: a pointer to a `COleDataObject`, a `DWORD` indicating the state of the control keys, and a `CPoint` object indicating the location of the mouse. Both functions return a `DROPEFFECT...` value, which indicates the type of data transfer occurring. There are five different `DROPEFFECT...` values, which indicate the following drag-and-drop transfer states:

- `DROPEFFECT_NONE`—A drop is not allowed.
- `DROPEFFECT_COPY`—A copy operation is pending.
- `DROPEFFECT_MOVE`—A move operation is pending.
- `DROPEFFECT_LINK`—A link from the dropped data to the original data is pending.
- `DROPEFFECT_SCROLL`—A drag/scroll operation is about to occur or is occurring in the target.

`OnDragEnter()` and `OnDragOver()` can use the information in the `COleDataObject`, the key press mask, and the `CPoint` parameters to determine what kind of operation is being performed. For example, if the drop target accepts only `CF_BITMAP`-formatted data and the Clipboard contains only `CF_TEXT`-formatted data (which you can find out by querying the `COleDataObject`), the view cannot accept the data. In this case, `OnDragEnter()` and `OnDragOver()` would return `DROPEFFECT_NONE` to indicate that the view cannot accept a drop. By returning the appropriate `DROPEFFECT...` value, these functions tell the framework which cursor to display.

In addition, the `OnDragEnter()` and `OnDragOver()` functions can check the key mask (the `DWORD` parameter) to find out if any control keys are pressed. To change the cursor, the drop target need only return the appropriate `DROPEFFECT...` value and MFC does the rest.

The view's `OnDragLeave()` function is called when the mouse cursor leaves the view. It takes no parameters and returns void. This is a good place to put any needed drag-and-drop clean-up code.

The final function, `OnDrop()`, is called whenever the user releases the left mouse button inside a view that is a drop target. `OnDrop()` takes the same parameters as `OnDragEnter()` and `OnDragOver()`: a pointer to a `COleDataObject`, a `DWORD` indicating the state of the control keys, and a `CPoint` object indicating the location of the mouse. `OnDrop()` also returns a `DROPEFFECT...` value that indicates the type of transfer operation that actually occurred. This is the value passed back to `COleDataSource::DoDragDrop()`. Again, the code for dropping data and that for pasting it from the Clipboard are virtually identical. Many developers decide to implement one body of code for both pasting from the Clipboard and accepting a data drop.

Inside MFC's Drag-and-Drop Classes

In summary, MFC's drag-and-drop data transfers are really an extension of the Clipboard transfer. As with Clipboard transfers, drag-and-drop transfers start off with the `COleDataSource`. A data producer either caches or promises a rendering later (using `DelayRenderData()`). However, instead of calling `SetClipboard()`, the data producer calls `DoDragDrop()`. On the data consumer side, the application registers one of its views with a `COleDropTarget`. Then the view receives drag-and-drop notifications. Here's a closer look at MFC's drag-and-drop-related classes.

COleDataSource

In addition to holding a data cache and providing support for Clipboard transfers, `COleDataSource` has a function called `DoDragDrop()`. `DoDragDrop()` wraps the OLE function `DoDragDrop()`, which has this prototype:

```
DROPEFFECT DoDragDrop(
    DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
    LPCRECT lpRectStartDrag = NULL,
    COleDropSource* pDropSource = NULL);
```

Notice that `COleDataSource::DoDragDrop()` takes three parameters: (1) a `DROPEFFECT` indicating which drag-and-drop operations are allowed, (2) a rectangle that determines when the drag-and-drop operation actually starts, and (3) a pointer to a `COleDropSource` that handles the user interface aspects of the drag-and-drop operation. If the last parameter is `NULL`, `DoDragDrop()` uses a standard `COleDropSource` implementation already provided by MFC.

ColeDropSource

ColeDropSource implements the IDropSource interface. Listing 12-4 shows the definition of ColeDropSource.

Listing 12-4. The ColeDropSource class

```
class ColeDropSource : public CCmdTarget {
// Constructors
public:
    ColeDropSource();

// Overridables
    virtual SCODE QueryContinueDrag(BOOL bEscapePressed, DWORD dwKeyState);
    virtual SCODE GiveFeedback(DROPEFFECT dropEffect);
    virtual BOOL OnBeginDrag(CWnd* pWnd);

public:
    BEGIN_INTERFACE_PART(DropSource, IDropSource)
        INIT_INTERFACE_PART(ColeDropSource, DropSource)
        STDMETHOD(QueryContinueDrag)(BOOL, DWORD);
        STDMETHOD(GiveFeedback)(DWORD);
    END_INTERFACE_PART(DropSource)

    DECLARE_INTERFACE_MAP()

    CRect m_rectStartDrag;
    BOOL m_bDragStarted;
    DWORD m_dwButtonCancel;
    DWORD m_dwButtonDrop;

    // metrics for drag start determination
    static AFX_DATA UINT nDragMinDist;
    static AFX_DATA UINT nDragDelay;

    friend class ColeDataSource;
};
```

ColeDropSource implements the IDropSource interface via the normal MFC interface maps. IDropSource::QueryContinueDrag() and IDropSource::GiveFeedback() map directly to the virtual functions ColeDropSource::QueryContinueDrag() and ColeDropSource::GiveFeedback(). Because these functions are virtual, you can safely override them.

There are several reasons you might want to override MFC's default implementation of ColeDropSource. If you're unhappy with the default cursors provided by MFC, all you need to do is derive a class from ColeDropSource and override the GiveFeedback() function. In addition, ColeDropSource maintains variables for identifying the key for canceling the drag-and-drop operation (m_dwButtonCancel) and

for specifying the mouse button that identifies the drop operation (`m_dwButtonDrop`).

`COleDropSource` also has static members for specifying the time delay before the drag-and-drop operation starts (`m_nDragDelay`) and for specifying the number of pixels the mouse must move before the drag-and-drop operation starts. `COleDropSource`'s constructor initializes these variables. The drag distance and drag delay variables are read in from the `WIN.INI` file. The drag distance's key is "DragMinDist" and the drag delay's key is "DragDelay". If these entries aren't in the initialization file, then the drag distance's value is set to 2 and the drag delay's value is set to 0.

To change any of the user-interface characteristics for a given drag-and-drop operation, just derive your own class from `COleDropSource`. Then feed your newly derived class into `COleDataSource::DoDragDrop()`.

COleDropTarget

`COleDropTarget` wraps the `IDropTarget` interface. `COleDropTarget` is a real COM object implemented using MFC's interface maps. Listing 12-5 shows its declaration.

Listing 12-5. The `COleDropTarget` class

```
class COleDropTarget : public CCmdTarget {
// Constructors
public:
    COleDropTarget();

// Operations
    BOOL Register(CWnd* pWnd);
    virtual void Revoke(); // virtual for implementation

// Overridables
    virtual DROPEFFECT OnDragEnter(CWnd* pWnd, COleDataObject* pDataObject,
        DWORD dwKeyState, CPoint point);
    virtual DROPEFFECT OnDragOver(CWnd* pWnd, COleDataObject* pDataObject,
        DWORD dwKeyState, CPoint point);
    virtual BOOL OnDrop(CWnd* pWnd, COleDataObject* pDataObject,
        DROPEFFECT dropEffect, CPoint point);
    virtual DROPEFFECT OnDropEx(CWnd* pWnd, COleDataObject* pDataObject,
        DROPEFFECT dropDefault, DROPEFFECT dropList, CPoint point);
    virtual void OnDragLeave(CWnd* pWnd);
    virtual DROPEFFECT OnDragScroll(CWnd* pWnd, DWORD dwKeyState,
        CPoint point);

// Implementation
public:
    virtual ~COleDropTarget();

protected:
    HWND m_hWnd; // HWND this IDropTarget is attached to
```

```

LPDATAOBJECT m_lpDataObject; // != NULL between OnDragEnter, OnDragLeave
UINT m_nTimerID;             // != MAKEWORD(-1, -1) when in scroll area
DWORD m_dwLastTick;          // only valid when m_nTimerID valid
UINT m_nScrollDelay;         // time to next scroll

// metrics for drag-scrolling
static AFX_DATA int nScrollInset;
static AFX_DATA UINT nScrollDelay;
static AFX_DATA UINT nScrollInterval;

// implementation helpers
void SetupTimer(CView* pView, UINT nTimerID);
void CancelTimer(CWnd* pWnd);

// Interface Maps
public:
    BEGIN_INTERFACE_PART(DropTarget, IDropTarget)
        INIT_INTERFACE_PART(ColeDropTarget, DropTarget)
        STDMETHOD(DragEnter)(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
        STDMETHOD(DragOver)(DWORD, POINTL, LPDWORD);
        STDMETHOD(DragLeave)();
        STDMETHOD(Drop)(LPDATAOBJECT, DWORD, POINTL pt, LPDWORD);
    END_INTERFACE_PART(DropTarget)

    DECLARE_INTERFACE_MAP()
};

```

`ColeDropTarget`'s most important members are `Register()`, `DragEnter()`, `DragOver()`, `DragLeave()`, and `Drop()`.

The other important class in MFC's drag-and-drop implementation is `CView`. `ColeDropTarget` works in concert with `CView` to make it very easy to program OLE drag-and-drop.

CView

You may remember that `CView` also has `OnDragEnter()`, `OnDragLeave()`, `OnDragOver()`, and `OnDrop()` functions. But wait—these are normally handled by `IDropTarget`. What are they doing in `CView`? How are these functions ever called? Let's follow MFC's drag-and-drop operation from start to finish to see exactly what happens.

How MFC Drag-and-Drop Works

The best way to see how MFC does OLE drag-and-drop is to examine each side separately: the data producer side and the data consumer side. On the producer side, there's `ColeDataSource` and `ColeDropSource`. On the data consumer side, there's `ColeDataObject` and `ColeDropTarget`. Let's examine the data producer side first.

The Data Producer

To be a drag source, a data producer has only to either cache the data (or use `DelayRenderData()` to set up delayed rendering) and then call `DoDragDrop()`.

At the surface, `COleDataSource::DoDragDrop()` appears to be more or less a wrapper for the API function `::DoDragDrop()`. However, `COleDataSource::DoDragDrop()` does a bit more than just act as a wrapper for `::DoDragDrop()`.

When the data producer calls `COleDataSource::DoDragDrop()`, MFC examines the `COleDropSource` passed in by the caller. If the caller has provided a `COleDropSource`, `DoDragDrop()` uses that `COleDropSource` to manage the user interface (that is, change the cursors and determine under which conditions the drag-and-drop operation can continue or is canceled). If the caller doesn't provide a `COleDropSource`, `DoDragDrop()` uses the default implementation of `COleDropSource` provided by MFC.

If the caller provides a rectangle parameter, `DoDragDrop()` copies that rectangle into its own member rectangle (called `m_rectStartDrag`). `COleDataSource` uses this rectangle to know when to start dragging. That is, the actual dragging won't commence until the cursor leaves this rectangle. If the data producer passes a `NULL` pointer, `DoDragDrop()` creates its own empty rectangle around the current position of the cursor.

Before starting the drag operation, `COleDataSource::DoDragDrop()` calls `OnBeginDrag()`. This function is virtual, so you can override it if you'd like. The default implementation of `OnBeginDrag()` first captures mouse input and then sets up a loop that acts as a timer. `OnBeginDrag()` watches the tick count go down.

Then `COleDataSource::DoDragDrop()` uses `GetInterface()` to retrieve the `IDataObject` and the `IDropSource` interfaces from the `COleDropSource` that was passed in. `COleDataSource::DoDragDrop()` uses these interface pointers to call the API function `::DoDragDrop()`.

Earlier versions of MFC called `DoDragDrop()` directly. However, now MFC maintains a list of pointers to functions that implement OLE-specific features. For example, here's the code MFC uses to call `DoDragDrop()`:

```
#ifdef _AFXDLL
    _afxOLE.pfnDoDragDrop[0](lpDataObject, lpDropSource, dwEffects,
                             &dwResultEffect);
#else
    ::DoDragDrop(lpDataObject, lpDropSource, dwEffects, &dwResultEffect);
#endif
```

In MFC4, Microsoft combined all the separate MFC DLLs into one monolithic DLL. If the applications linked directly to various DLLs (like `OLE32.DLL`) the application would load OLE even for very simple apps. For this reason, Microsoft loads all non-essential DLLs dynamically using a very clever technique. Most of these instances

are hidden by macros, except where not possible, as in this case. MFC40.DLL only binds directly to KERNEL32.DLL, GDI32.DLL, USER32.DLL, and MSVCRT20.DLL. All other DLLs are loaded at runtime when the first function in them is called. OLE32.DLL, OLEAUT32.DLL, SHELL32.DLL, ADVAPI32.DLL, COMCTL32.DLL, ODBC32.DLL, WINSPOOL.DLL, are all examples of DLLs that are loaded at runtime. This allows MFC to maintain one giant DLL but with no performance hit for applications that don't use all these DLLs.

The Data Consumer

Again, like Clipboard data transfers, drag-and-drop operations culminate in the production of a COleDataObject. However, instead of pulling an IDataObject pointer off the Clipboard using OleGetClipboard(), MFC provides a fully loaded COleDataObject whenever data is dropped on a view.

Registering with OLE

To receive drag-and-drop notifications, a view must first register itself with a drop target. That is, the view must declare an instance of COleDropTarget and call the Register() function. The standard place for calling Register() is during the view's OnCreate() function, because that's the first place you can provide any initialization for a window. The following code snippet shows how to do this:

```
class CMyView: public CView {
...
    COleDropTarget m_dt;
...
};

CMyView::OnCreate() {
    m_dt.Register(this);
}
```

So just what does Register() do? Before doing anything, Register() calls CoLockExternalObject() to place a strong lock on the drop target object. That way, the drop target object is guaranteed to remain in memory even if the data consumer calls Release() enough times to balance the AddRef(s). If CoLockExternalObject() weren't called, the drag-and-drop operation would work once but fail every time thereafter.

Next, Register() calls the OLE API function RegisterDragDrop(). The API function takes two parameters: (1) the handle of the window that is to receive drag-and-drop notifications and (2) a pointer to an IDropTarget interface. Once this information is registered with OLE using RegisterDragDrop(), then OLE will be able to call the

IDropTarget functions (that is, OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop()) as the mouse cursor moves in and around the window specified by the window handle. Before leaving, COleDropTarget::Register() saves both the CWnd object and the window handle passed to Register(). COleDropTarget uses the CWnd object to implement OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop(). COleDropTarget needs the window handle to implement its Revoke() member function.

Finally, CWnd has a pointer to a COleDropTarget member. In addition to saving the CWnd object and the window handle, COleDropTarget::Register() sets the CWnd's COleDropTarget member to point to the COleDropTarget's this pointer. CWnd uses the pointer to the drop target to revoke drag-and-drop registration when the window is destroyed.

Revoking Drag-and-Drop Registration

COleDropTarget contains a complementary function that revokes a window's drag-and-drop registration. The function is called Revoke() (surprisingly enough!). COleDropTarget::Revoke() calls the OLE API function RevokeDragDrop() and frees the external lock on the COleDropTarget. As a developer, you'll rarely find yourself calling this function. As part of its OnNcDestroy() handler, CWnd calls COleDropTarget::Revoke() if the CWnd has a pointer to the COleDropTarget.

Handling the Notification

Once a window is registered with OLE to receive drag-and-drop notifications, it'll start receiving those messages as soon as the mouse cursor enters the boundaries of a window. In the case of MFC, the drag-and-drop notifications are handled by the CView class. So, though you can pass a plain-vanilla CWnd when registering a window to be a drop target, it's the CView that already has handlers for OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop(). MFC's drag-and-drop won't work if you register the drop target with something other than a CView.

Whenever the mouse cursor passes over a window that is registered to receive drag-and-drop notifications, OLE starts sending notifications to the drop target. The first message called during dragging is OnDragEnter(). Remember, COleDropTarget implements IDropTarget. If the window and the IDropTarget pointer are registered with OLE, OLE calls IDropTarget::DragEnter(). When OLE calls the IDropTarget's DragEnter() function, OLE passes the fully loaded IDataObject (the one provided by the data producer) and the window handle of the window over which the cursor was dragged. COleDropTarget's implementation of DragEnter() attaches the loaded IDataObject to a COleDataObject and looks up the window handle by calling CWnd::FromHandle(). COleDropTarget's DragEnter() simply delegates to the OnDragEnter() member function passing the CWnd and the COleDataObject. Inside

`COleDropTarget::OnDragEnter()` checks to see if the window attached is a view. If the attached window isn't a view, then `OnDragEnter()` exits, returning `DROPEFFECT_NONE`. Otherwise, `COleDropTarget::OnDragEnter()` tries to cast the `CWnd` pointer to a `CView`. If the cast is successful, `COleDropTarget::OnDragEnter()` delegates to `CView::OnDragEnter()`. At that point, the target view can examine the data, determine if it's useful, and update the user interface accordingly.

The other `IDropTarget` functions work in a similar way. When OLE calls the `IDropTarget` interface function represented by `COleDropTarget`, `COleDropTarget` takes the `IDataObject` pointer, attaches it to a `COleDataObject`, finds the `CWnd` object representing the window, casts it to a view, and calls the corresponding `CView` function (that is, `OnDragOver()`, `OnDragLeave()`, or `OnDrop()`).

As you can see, a lot of work goes into implementing drag-and-drop in a safe, robust way. Fortunately, MFC's implementation does most of the work for you.

Conclusion

Uniform Data Transfer in OLE is simpler than you might expect—especially when you use the MFC wrappers for `IDataObject`. Programming data transfers using the Microsoft Foundation Classes means that you can cut down your exposure to the OLE gunk (the low-level API). The information provided in this chapter gives you a good start in understanding how to use UDT to implement Clipboard and drag-and-drop data transfer techniques. In the next chapter, we cover OLE documents. For more information, be sure to check the `DNEDIT.EXE` example on the diskette. `DNEDIT` shows how to use UDT within the context of a simple Notepad-type text editor.

MFC Internals

Inside the Microsoft® Foundation Class Architecture

"This book is definitely not a rehash of existing documents. It is not a 'how-to' book—it is a 'how does it work' book." —Dean McCrory, MFC Development Lead

Finally, a book on MFC that fills the gaps between "Using the Wizards" Visual C++ books, product documentation, and MFC source code. *MFC Internals* is a guide to what goes on inside the Microsoft® Foundation Classes: It gives you unique and in-depth information on undocumented MFC classes, utility functions and data members, useful coding techniques, and critical analysis of the way various MFC classes work and fit together.

The first half of the book covers core Windows® graphical user interface classes and their supporting classes; the second half covers subjects like OLE that are extensions to the basic Windows support. You'll become an expert at understanding MFC implementation details by:

- exploring under the hood of MFC's document/view architecture to learn about view synchronization, printing, and even print preview
- diving deep into undocumented aspects of MFC serialization and undocumented classes like CPreview, CPreviewDC, CMirrorFile, CDockBar, and so on
- *finally* learning how MFC and OLE work together, and how OLE controls are implemented
- building the skills that help you investigate and understand MFC source code on your own.

MFC Internals focuses on MFC 4.0 for Windows 95 and Windows NT. Most key "internal" concepts also apply to previous versions, but where they don't the authors warn you with a version note. The book's disk contains example code and the MFC FAQ file. Be sure to check out Appendix A, a handy MFC source code field guide.

MFC Internals is an essential guide to tapping MFC's rich and robust application framework and applying advanced MFC knowledge in world-class Windows applications.

George Shepherd is a senior computer scientist with DevelopMentor, where he develops and delivers courseware for developers using MFC and OLE. **Scot Wingo** is a cofounder of Stingray Software, an MFC extension company. He is also the maintainer of the MFC FAQ <http://www.unx.com/~scot/mfc_faq.html>

Cover design by Jean Seal in the style of Mark Rothko

Addison-Wesley Publishing Company, Inc.
Find A-W Developers Press on the World-Wide Web at:
<http://www.aw.com/devpress/>

