The BOCOLE Engine and OCF Class Library Make Programming OLE Easier

by Ted Faison

Ted Faison has written several books and articles on C++ and Windows. He is president of Faison Computing, a firm that develops C++ applications, class libraries and custom controls for Windows. Ted can be reached at tedfaison@msn.com.

It's safe to say now that C++ is in the mainstream of programming. As people developed their class hierarchies and objects, a problem has emerged: C++ is great for developing reusable objects, but only in the context of the same development environment. If you create objects using Visual C++ and MFC, you can't realistically mix them with objects developed with Borland's OWL or Smalltalk. The problem is that object-oriented programming languages support reusability at the source code level, not the binary level.

Consider creating a C++ object and putting it in a DLL. Great, but there are problems. DLLs export a list of callable function names, but C++ mangles these names. There are no name-mangling standards, so if the DLL was developed with Borland C++ it will have names mangled differently than if it was implemented with Visual C++. To be able to call the member functions exported from a DLL, the calling application must be implemented using the same vendor's C++ as the DLL. A pity, and also a source of headaches for companies with C++ custom control packages. OLE and VBX controls don't suffer from these limitations.

OLE rescues C++

But that's where the next level of standardization comes in: binary interoperability. OLE uses objects that have a predefined binary image. With binary objects, you can interact with objects whose internal structure and implementation may be entirely unknown to you. If you develop an OLE server object, its binary image and interface must be the same no matter how the object was created. How is this done ? Well, Microsoft developed the Component Object Model (COM), an object format standard that describes the exact binary interface of objects, and the way calling applications interact with these objects. OLE objects are COM objects that support certain basic operations, such as dragging, dropping, streamability, in-place activation, and so on.

The COM standard takes the concept of object from the source code level to the binary level — not exactly a trivial accomplishment. Using OLE objects, it no longer matters whether an object was created using C, Borland C++, Visual C++ ,or even Visual Basic. The external interface of an OLE object is completely language independent. If you create an OLE object using C++. you are required to give it a standard OLE interface that encapsulates the internal C++ interfaces, hiding language-specific details like data members and mangled function names from users of the object. With OLE, we're finally on the road to full object interoperability. But beware, there are still many obstacles on the road to full object-oriented application development, and it will be a few more years before these problems are solved.

C++ rescues OLE

All this wonderful new OLE functionality comes at a price: complexity. Anyone who has read anything on OLE programming will want to avoid dealing directly with the OLE API. There are

hundreds of functions, dozens of interfaces, problems with memory allocation, lists of registration parameters, the list goes on and on. OLE seems to be a technology that everyone needs but few find easy to implement.

Microsoft eased much of the pain of utilizing OLE by introducing MFC 2.5 in late 1993. Using the new MFC classes, developing an OLE server or container is relatively easy (see Paul DiLascia'a article "OLE Made Almost Easy: Creating Containers and Servers Using MFC 2.5", MSJ Vol 9 No 4) — especially with AppWizard. Borland C++ had no application framework support for OLE programmers until now. Borland C++ programmers can now also reap the benefits of OLE with Borland C++ 4.5.

Covering Up the Mess

Borland waited so long to bring OLE support to C++ because it took a lot of time to design a C++ package that would elegantly wrap up OLE, support all the OLE functionality, be easy to use, and be independent of OWL. No small order. Borland wanted to wait until they felt they had *done it right*. After spending a lot of time studying their design and code, I am ready to agree with them. Not only did they hide the tremendous complexity of OLE, but the resulting classes make good use of C++ constructs like multiple inheritance and polymorphism.

OLE objects don't support inheritance or polymorphism or member access specifiers. After all, OLE objects are not C++ objects, even though they might be implemented using C++. The Borland OLE classes provide a level of insulation code around the OLE API that makes it possible to use C++ constructs like inheritance with nonchalance. While C programmers are forced to learn an entirely new API for OLE programming, with all the complexity of new interfaces, Borland C++ programmers need only derive their OLE objects from an OLE class. They simply *inherit* OLE capabilities. Borland C++ 4.5 supports the OLE functionality that comes with Visual C++ 2.0, and in addition offers the following:

- Support for in-proc (DLL) OLE servers
- Support for OLE for non-OWL applications
- Support for self-registering servers

In MFC, OLE was implemented by adding new code and classes directly to MFC. If you want to develop an OLE object using C++ classes in Visual C++, you must use MFC. Not so for Borland C++. The classes that encapsulate OLE code are not part of OWL, but form an entirely new class hierarchy of their own called the ObjectComponents Framework (OCF). It is possible for programmers working in Visual C++ to use the OCF classes, a fact that could be of interest to developers writing OLE servers that have no user interface of their own, and require no MFC functionality.

The BOCOLE Engine

The first step in reducing the complexity of the OLE API was to encapsulate OLE in a package that made available to applications all the normal OLE interfaces, but also added a number of higher level ones. Borland christened the resulting product *BOCOLE* (Borland ObjectComponents OLE). BOCOLE ships as a 16 bit DLL bundled with Borland C++ 4.5. The DLL can be used by stand-alone applications that have nothing to do with Borland C++. In fact, WordPerfect 6.1 and Paradox 5.0 were both OLE-enabled through the use of BOCOLE. Borland C++ 4.5 ships with the full source code for BOCOLE.

Application developers often use high-level constructs such as the document/view model, in which data is separated from the code that displays the data in a window. Providing OLE

support to such applications requires the repeated use of groups of low-level OLE interfaces and functions. To simplify application development, Borland added a number of high-level interfaces to BOCOLE that correspond to some of the commonly used application objects like Documents and Views. BOCOLE therefore has high-level interfaces like IBWindow, IBDocument and IBView that are built on top of lower level OLE interfaces like IDataObject, IUnknown and IClassFactory. All the high level BOCOLE interfaces start with the letters IB (for *Interface BOCOLE*),. Applications can call functions in the BOCOLE interfaces the same way they would call functions in a native OLE interface.

BOCOLE defines internally a number of classes that correspond one-to-one to the interfaces supported. For example, the OLE IClassFactory interface is supported with the class IClassFactory, the IUnknown interface is supported by a class called IUnknown. The high level BOCOLE interfaces also have matching classes, so the IBWindow interface is implemented by an internal class called IBWindow.
 Figure 1 shows a list of the classes that encapsulate native OLE interfaces. Figure 2 shows the classes that support higher level BOCOLE interfaces, built on top of the lower level OLE interfaces.

OLE Interface	Purpose
IClassFactory	Creation of new objects
IDataObject	Data transfer for clipboard operations, drag and drop
IDispatch	Automation support
IEnumVARIANT	Accessing collections contained inside automated objects
IMalloc	Memory allocation
IStorage	Structured storage support
ITypeInfo	Description of objects in type libraries
ITypeLib	Description of objects in type libraries
IUnknown	Universal Interface

Figure 1 - OLE interfaces encapsulated by low level BOCOLE classes.

BOCOLE Interface	Purpose
IBApplication	Frame window support for container applications
IBClassMgr	Support for aggregate objects
IBContainer	Drag and drop with container windows
IBContains	Support for client containers
IBDataConsumer	Data sink support
IBDataNegotiator	Drag and drop support
IBDataProvider	Clipboard Support
IBDocument	General support for application documents
IBDropDest	Drag and drop
IBLinkable	Moniker Support

IBLinkInfo	Support for linked objects
IBPart	Used by OCF container applications to control embedded/linked objects
IBRootLinkable	Moniker support
IBService	Clipboard support
IBSite	Client container screen management for linked/embedded objects
IBWindow	Support for main and child windows in container applications

Figure 2 - High-level OLE interfaces supported by BOCOLE.

The beauty of using classes rather than interfaces is that you can use inheritance with classes, so all the usual notation of a C++ program can be used in an OLE context. In C-style OLE, if you have an object that supports the `IUnknown` interface and the `IDispatch` interface, you have to create an object that supports all the functions of both interfaces. With BOCOLE, the class `IDispatch` is derived from class `IUnknown`, so you only have to support the functions of the `IDispatch` interface – the others are inherited from the base class `IUnknown`. You only override those base class functions that you want to change. The name-mangling problem disappears altogether in OLE, because function calls made through OLE interfaces are internally made by ordinal number, not by name. OLE has a specification that spells out the order in which functions must appear in the virtual function table (vtbl) for OLE objects. As long as a class' vtbl is laid out according the the OLE spec, everything works. For example, consider the OLE `IUnknown` interface, which supports the functions `QueryInterface`, `AddRef` and `Release`. The OLE spec says that `IUnknown`'s vtbl must have `QueryInterface` as the first function, then `AddRef`, then `Release`. The C++ class defined like this:

```
class IUnknown {
  virtual HRESULT QueryInterface();
  virtual ULONG AddRef();
  virtual ULONG Release();
};
```

has a vtbl structure that matches the OLE spec. When an application obtains a pointer to an object of class `IUnknown` and calls the function `Release`, OLE actually calls the function referenced by the third entry in the vtbl. No actual function names are involved.

For each OLE interface, BOCOLE has a corresponding class, as discussed earlier. Some of the higher level BOCOLE interfaces (like `IBPart`) support multiple interfaces, so they are derived from other classes. Figure 3 shows the class hierarchy of the high-level interface classes of BOCOLE.
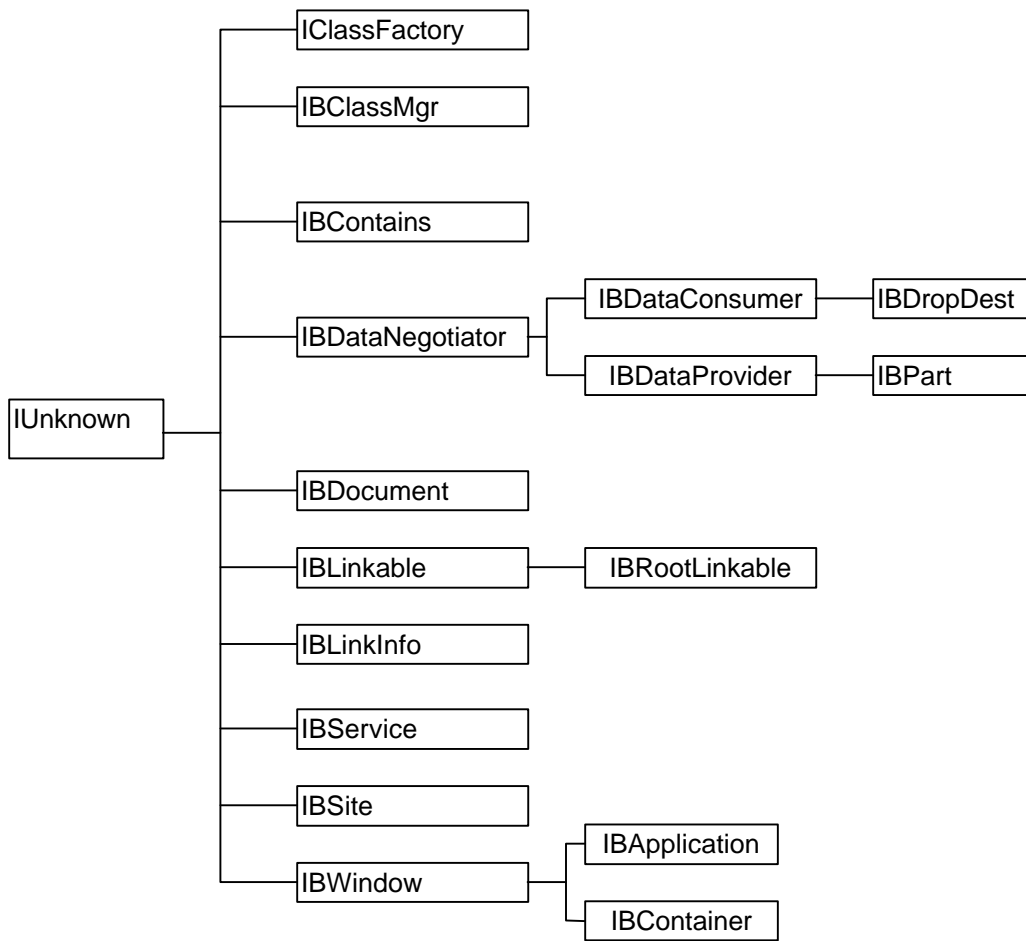
Figure 3 - The hierarchy of the BOCOLE high-level interface classes.

To see how inheritance works with BOCOLE classes, from the perspective of OLE interfaces, consider the layout of the vtbl for an object of type `IBWindow`, as shown in figure 4.

QueryInterface
AddRef
Release

○ IUnknown interface

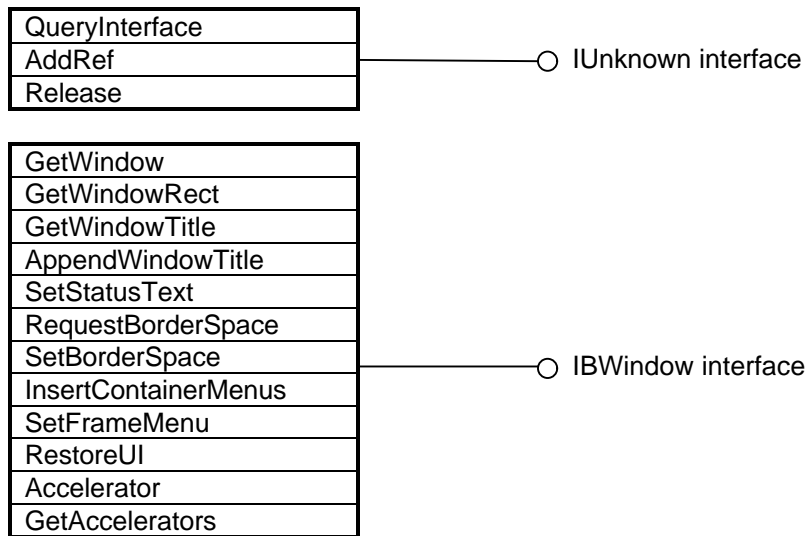| |
|---|
| GetWindow |
| GetWindowRect |
| GetWindowTitle |
| AppendWindowTitle |
| SetStatusText |
| RequestBorderSpace |
| SetBorderSpace |
| InsertContainerMenus |
| SetFrameMenu |
| RestoreUI |
| Accelerator |
| GetAccelerators |

○ IBWindow interface

Figure 4 - The memory layout of the vtbl of a BOCOLE object of class `IBWindow`, supporting the OLE `IUnknown` and BOCOLE `IBWindow` interfaces.

If an application calls the function `Release`, OLE internally calls the third function in the vtbl, which correctly calls `IUnknown::Release`. If an application calls `GetWindow`, then the fourth function in the vtbl will be correctly called. Inheritance is used extensively inside the BOCOLE code, but from an OLE perspective there are only interfaces and vtbls to call functions through. The BOCOLE code reaps all the benefits of C++ inheritance, remains completely compliant with the OLE interface specs, and adds a number of high level interfaces on top of the low-level OLE interfaces.

OCF = Object-Oriented OLE

The centerpiece of OLE programming in Borland C++ 4.5 is OCF, an entirely new C++ class hierarchy. OCF calls functions exported by the BOCOLE DLL to deal with OLE, avoiding the direct call of native OLE API functions. OCF is to OLE what OWL is to Windows. It wraps all the nasty stuff up, and presents to you a clean object-oriented environment. When you develop an application that requires OCF, you statically link the application code to the OCF library. Although OCF can be used with programs written in C or Visual C++, OCF is not sold separately, but bundled with Borland C++ 4.5, like BOCOLE. While BOCOLE deals primarily with OLE interfaces, OCF has C++ classes that deal with higher-level concepts such as messaging and callbacks. Figure 5 gives you a sense of how OCF fits into a typical application.
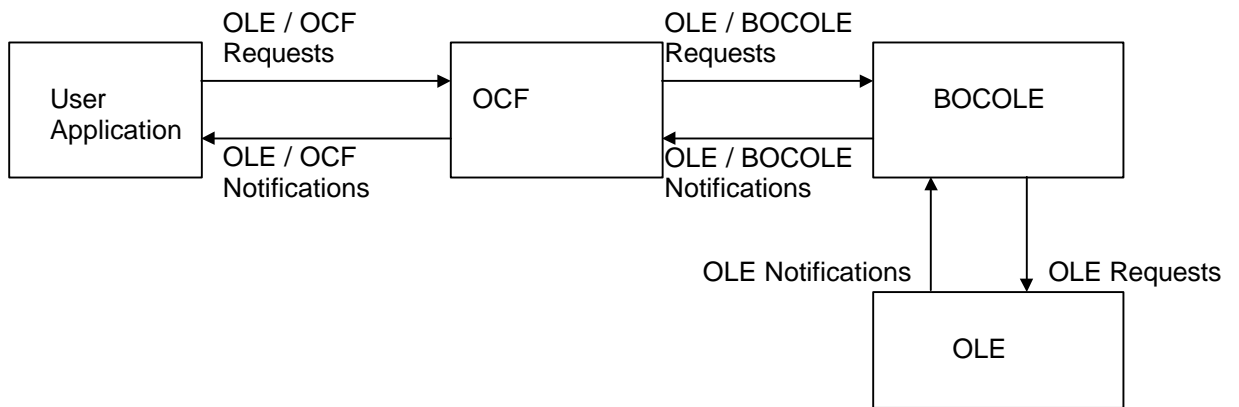
Figure 5 - The relationship between applications, OCF, BOCOLE and OLE.

When an application needs an OLE service (such as when embedding an OLE object), it calls an OCF function, which will typically result in multiple lower-level calls to BOCOLE. Conversely, when an application receives a notification message back from OCF, it needs to provide a handler for it. If you develop your application using OWL with AppExpert, then all the code to call OCF and handle OCF notifications is provided automatically. If you want to use OCF with an older project, or may be a Visual C++ project, then you will need to add a number of calls to OCF functions, and provide message handlers for notifications received from OCF.

Applications don't need to be concerned whether a feature is carried out by BOCOLE alone or OCF and BOCOLE together. In the remainder of this article, I will use the term *OCF* to indicate the combined functionality of ObjectComponents Framework and BOCOLE, unless otherwise specified. Most of the OLE functionality is handled automatically and transparently by OCF classes, so users of OCF classes don't need to be OLE API experts. OCF handles all the details for the following OLE operations:

- Linking
- Embedding
- Clipboard operations
- Drag-and-drop
- Compound files
- Support for server applications as EXEs
- Support for server applicatrions as DLLs
- Automation
- Type library creation
- OLE registration
- Localization

A really nice feature of OCF is that it can be used both in Borland C++ based and Visual C++ based applications. If you created a Visual C++ app using AppWizard without enabling OLE, there are only two ways you can add OLE to it (apart from the suicidal approach of low-level coding of the OLE API calls): you rebuild a new OLE-enabled app and transfer all your code into it, or you add OCF to the application. Using the second approach, you just add handlers for messages originated or delegated to you by OCF, and call OCF functions to handle user actions

that involve OLE operations. Borland provides a number of sample programs that show how to use OCF to OLE-enable a generic application written in plain C. The code shown can easily be adapted to applications using Visual C++.

OCF has classes that manage the functionality of OLE servers, containers, embedded objects, linked objects, automated objects and automation controllers. All the classes are wrapped up in a single package, and create the class hierarchy shown in figure 6. OCF class names normally begin with the letters TOc.
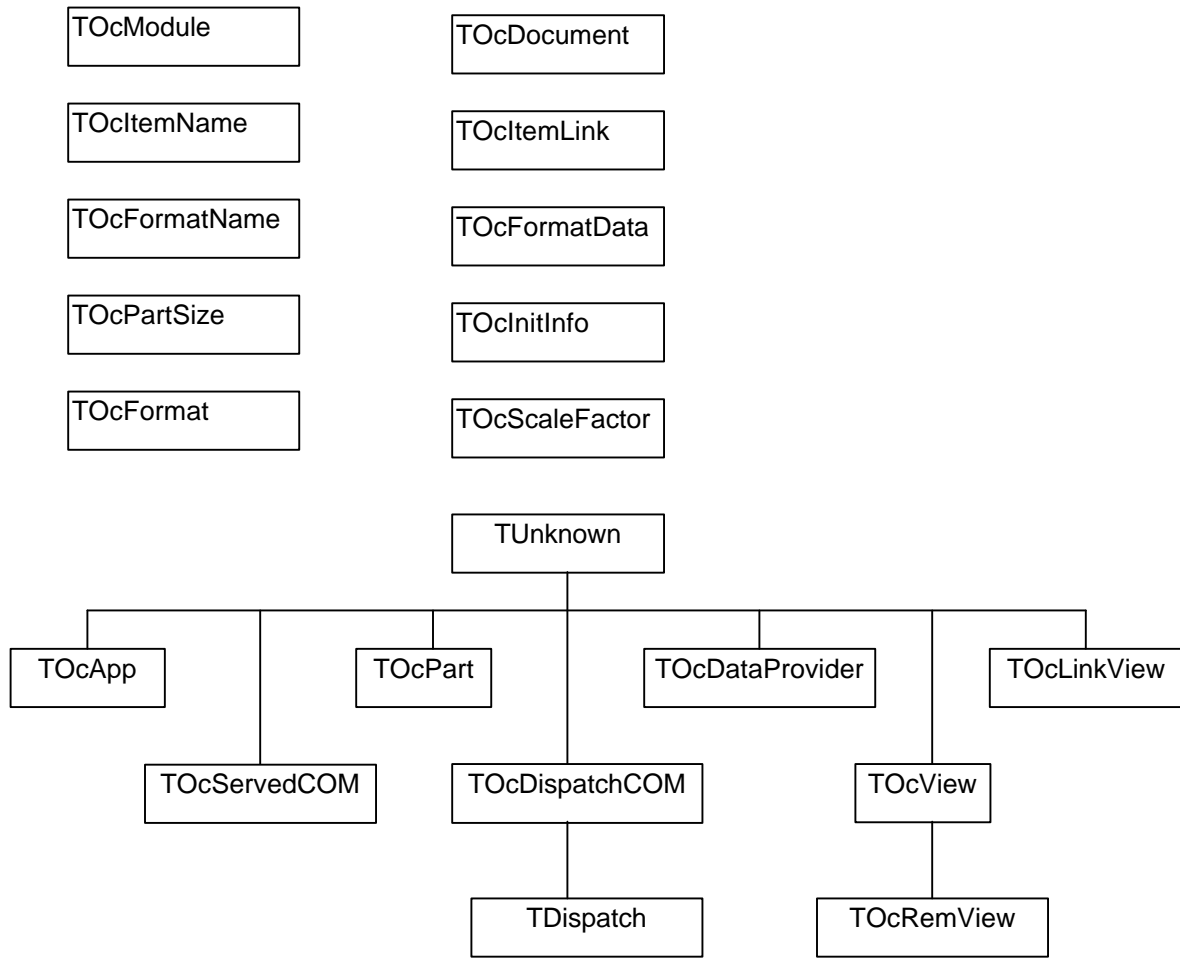
```
TOcModule              TOcDocument

TOcItemName            TOcItemLink

TOcFormatName          TOcFormatData

TOcPartSize            TOcInitInfo

TOcFormat              TOcScaleFactor


                    TUnknown

   TOcApp      TOcPart      TOcDataProvider      TOcLinkView

      TOcServedCOM    TOcDispatchCOM      TOcView

                    TDispatch      TOcRemView
```

Figure 6 - The complete OCF class hierarchy.

Connector Objects

If you embed or link an OLE object in a container, you want to be able to manipulate the object as a C++ object, avoiding all the low-level OLE stuff. When you are create an OLE server you also want to be able to handle painting in the view space of a client container using C++ objects. OCF defines the classes `TOcPart` and `TOcRemView` to support these two situations. These classes, along with a couple others, are called *connector classes* in OCF, because they connect your application directly with an underlying OLE object.

If you create an OLE container using OCF, when you link or embed OLE objects in it, a `TOcPart` is created dynamically for each object. Rather than interact directly with the low level OLE interfaces of the linked/embedded objects, OCF calls members of `TOcPart`, which in turn manipulate the OLE object connected to it. When you create a server object with OCF, an object of type `TOcRemView` is allocated to handle painting of the object in the view space of client containers. The server calls member functions of the `TOcRemView` connector object, which in turns makes calls to BOCOLE and OLE to carry out the painting. Figure 7 shows a diagram of the relationships between OCF connector objects, the user code and OLE.
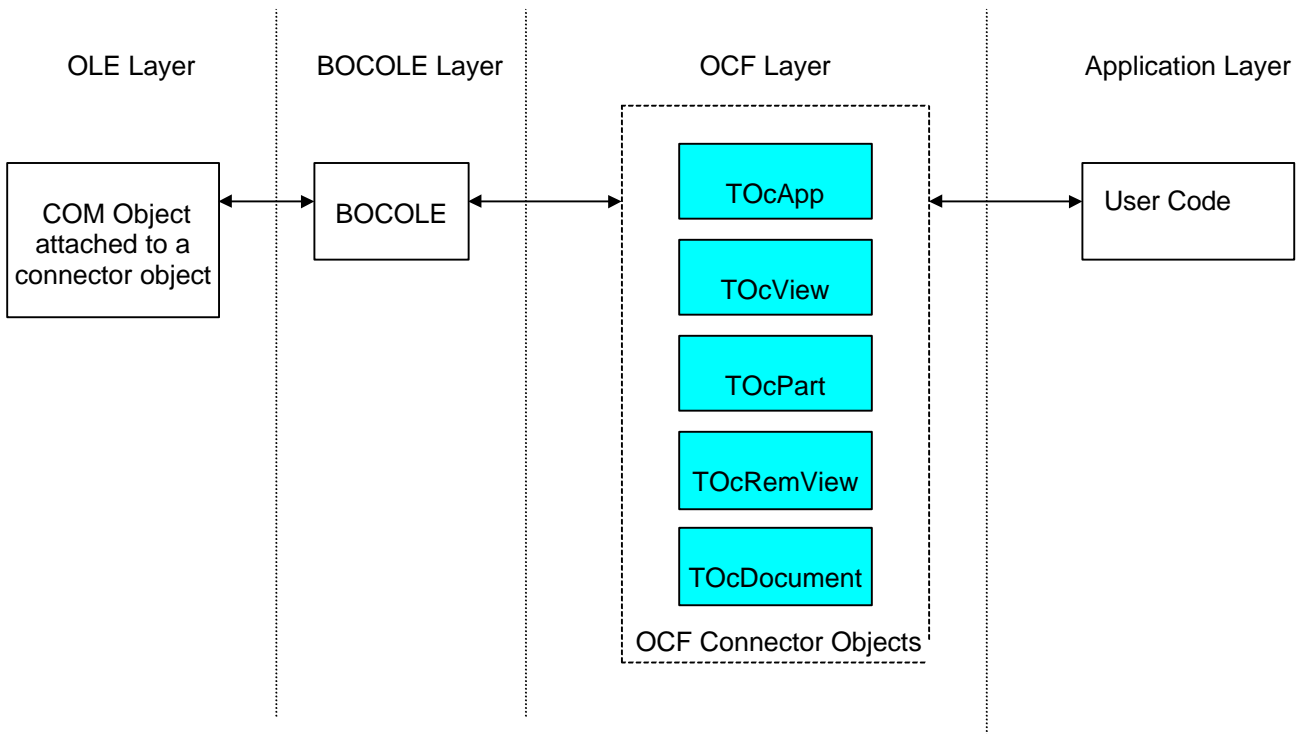
Figure 7 - The OCF connector classes, shown in light blue.

How OCF connector objects are created

To see how connector objects are used inside OCF, let's start by finding where connector objects are created, and how they are attached to OLE objects. Assume you have an OLE container application, that uses OCF. To embed an object, you use the **Insert Object** menu, which is a standard menu that all OLE containers should support. Using the **Insert Object** menu, you bring up a dialog box showing a list of all the OLE servers registered in your system. The dialog box might look as shown in figure 8.
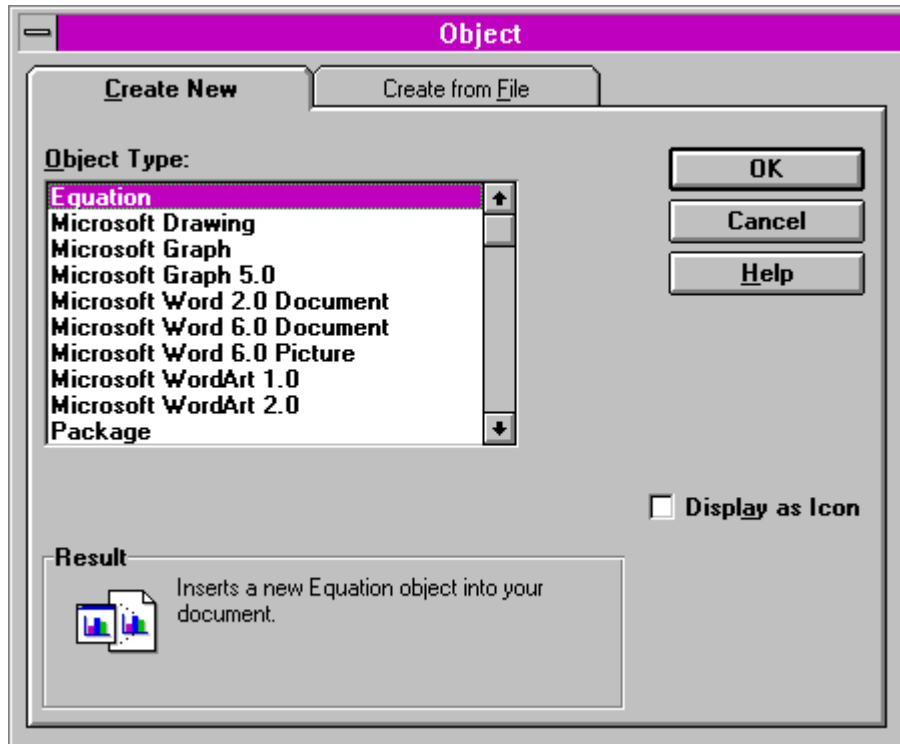

Figure 8 - The OLE **Insert Object** dialog box.

If you create the container using OWL with AppExpert, code is generated to automatically call all the OCF functions to display the **Insert Object** dialog box and embed a new object of the selected type. When you select an object type on the dialog box, OWL creates a `TOcPart` connector object with information about the new OLE object being embedded.The whole process is wrapped up inside the handler for the **Insert Object** command in the OWL class `TOleWindow`, which looks something like this:

```
// OWL handler for the Edit Insert Object command
void TOleWindow::CmEditInsertObject()
{
// initialize an OLE information object
  TOcInitInfo initInfo(OcView);

  if (OcApp->Browse(initInfo)) {
   // open the Insert Object dialog box and
   // get the class ID of the object selected

    // set a default position and size for
    // the object being embedded
    TRect rect(30, 30, 100, 100);
    GetInsertPositon(rect);
```

```
      // create an OCF connector object
      TOcPart* embeddedObject =
        new TOcPart(*OcDoc, initInfo, rect);

      // select the newly embedded object
      SetSelection(embeddedObject);
  }
}
```

The OCF function `TOcApp::Browse(TOcInitInfo& init)` calls into the BOCOLE code, with the code

```
BServiceI->Browse(&init) ;
```

which calls the BOCOLE function `IBService::Browse`, which calls the OLE function `OleUIInsertObject()`, which in turn displays the dialog box in figure 8. When the user selects an OLE object type and clicks the OK button, the call to `TOcApp::Browse(TOcInitInfo& initInfo)` returns, with the variable `initInfo` holding a valid OLE class ID. The code then invokes the constructor for `TOcPart`, passing the class ID in `initInfo`. The constructor does some internal initialization, then calls `TOcApp::BOleComponentCreate()` to create an OLE object to connect itself to. The latter function calls into BOCOLE to create the actual OLE object that the `TOcPart` object then connects itself to. The entire process is shown graphically in figure 9.
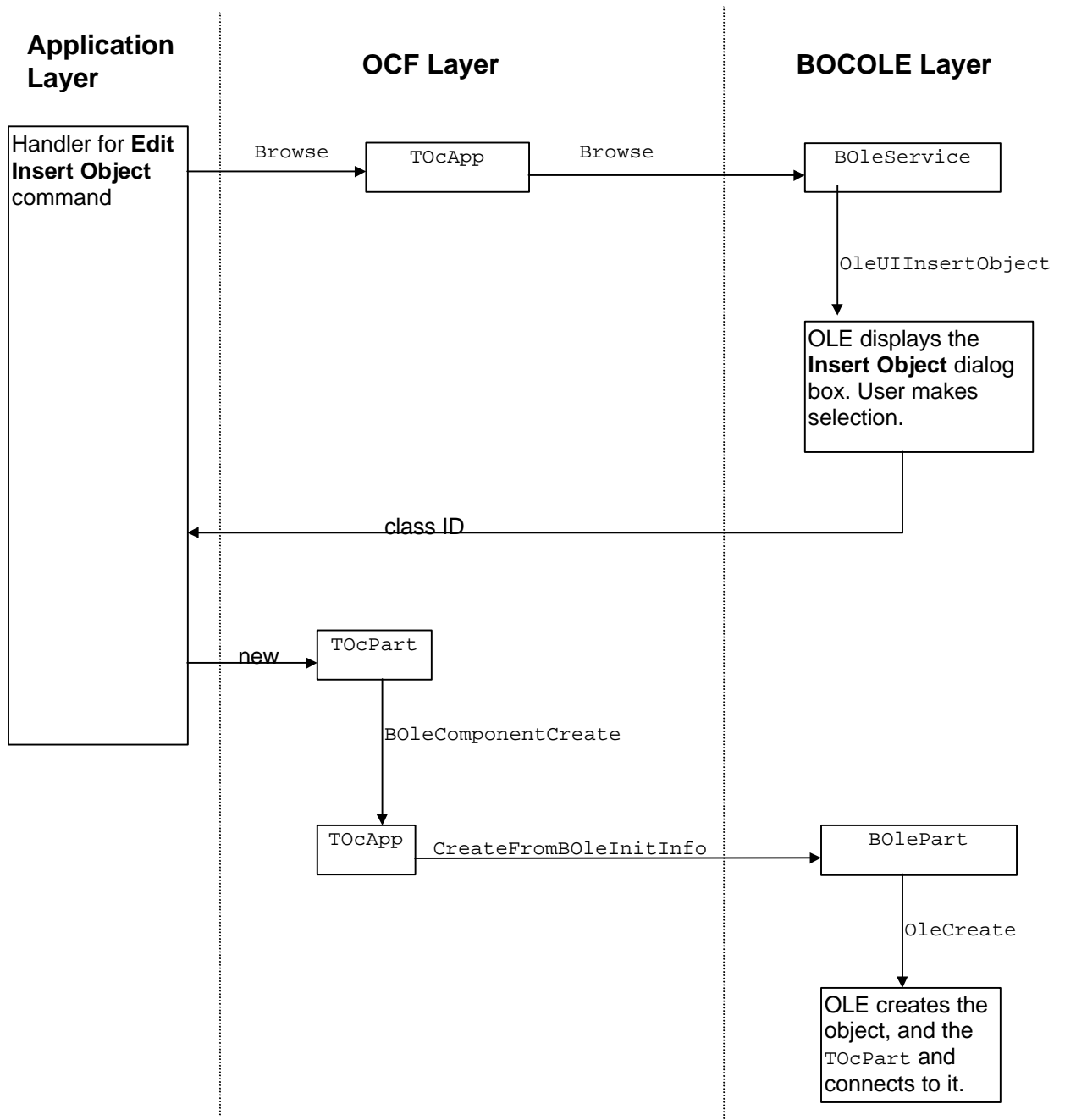
Application Layer   **OCF Layer**   **BOCOLE Layer**

Handler for **Edit Insert Object** command

Browse → TOcApp

Browse → BOleService

OleUIInsertObject

OLE displays the **Insert Object** dialog box. User makes selection.

class ID

new → TOcPart

BOleComponentCreate

TOcApp — CreateFromBOleInitInfo → BOlePart

OleCreate

OLE creates the object, and the TOcPart and connects to it.

Figure 9 - The activity involved in creating an ObjectComponents TOcPart connector object.

Does figure 9 look like something you want to be part of? I don't think so, but using OCF you don't need to know about the details that occur below the OCF layer. If you use AppExpert to create your application, you generally won't need to add additional code to support OLE.

Manipulating embedded objects through OCF connectors

Having created a `TOcPart` connector object in the previous section, I'll show how a container application would manipulate an embedded object through `TOcPart` member function calls. Containers must have a window in order to display embedded or linked objects. This window must be associated with a `TOcView` connector object, which knows how to paint the embedded or linked objects. The embedded or linked objects must be associated by connector objects of class `TOcPart`, allowing you to use standard C++ notation to manipulate the objects:

```
// assume myServer points to some
// kind of embedded or linked object
TOcPart* myServer;

if (!myServer->IsVisible() )

  // change the size of the object
  Tsize size(30, 30, 100, 100);
  myServer->SetSize(size);

  // make sure the server is visible
  myServer->Show(True);
}
```

If you use AppExpert, you won't need to write any code to manipulate OCF objects, because all the code is generated for you. You will need to create and manipulate OCF connector objects if your app was not created with AppExpert, or if was written in C or Visual C++. The Borland sample programs show exactly what OCF code needs to added, and where.

The big picture

The only way to fully grasp how OCF works is to study a complete example. In the next few sections I'll show how OWL servers and containers link through OCF, then BOCOLE down into OLE, and then how OLE links back to your application.

Since AppExpert creates OWL code that calls OCF code, I'll discuss the implementation of a simple OLE server app that uses OWL. Let's assume you want to create a small OLE server that works like a pocket calculator, as shown in figure 10.

Figure 10 - A simple calculator server.

You enter values into the calculator by clicking the buttons with the mouse. Results are displayed in the black area at the top of the calculator. The first step is to create an OCF-enabled application, so the server's application class will need to be multiply derived from the OWL class `TApplication` and the OCF class `TOcModule`. The server's main window must also be OLE-enabled, so it must be derived from the OWL class `TOleFrame`, which uses an internal OCF connector object of type `TOcApp` to handle OLE messages.

When you register the server with the system, your server's code lies dormant until a client container application invokes it (by embedding, linking or automation). Assuming the container embeds your server, the first flurry of activity involves the creation of the OLE server object in memory, which I outlined from the container's point of view in figure 9. But there is a whole bunch of activity that occurrs on the server object being embedded that doesn't show up in figure 9. The server code is only triggered into action with the `OleCreate()` call.

How OCF talks to OLE

When `OleCreate` is executed, the server app is run. When the container application manipulates the embedded server using a `TOcPart` connector object, several layers of code are called into action. Figure 11 shows what happens when the container does something like

```
myServer->Show(True);
```

Application Layer   **OCF Layer**   **BOCOLE Layer**

myServer

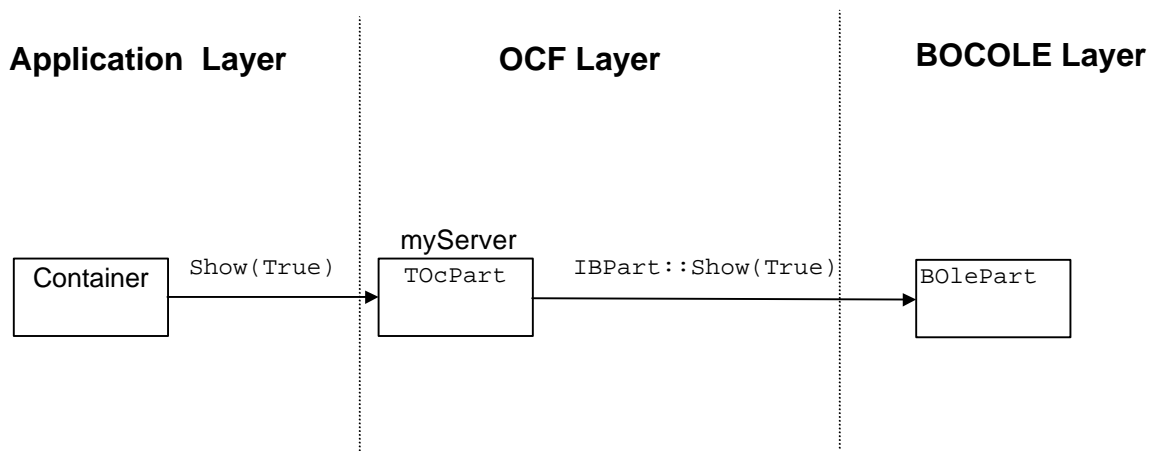| Container | Show(True) → | TOcPart | IBPart::Show(True) → | BOlePart |

Figure 11 - How a client container requests an embedded server to show itself.

The function `Show` is a very simple one, because it doesn't involve user application code. OCF and BOCOLE have code to completely process a `Show` command. The application code `Show(True)` invokes the OCF function `TOcPart::Show`, which invokes the BOCOLE function `IBPart::Show`. `IBPart` is one of the high-level interfaces supported by BOCOLE. Calling functions of `IBPart` generally results in several lower level standard OLE functions being called.

How OLE talks to OCF

Figure 11 isn't complete. When the `IBPart::Show` function is executed, OLE must vector the call to the appropriate function in the server. What `IBPart::Show` does is call the `DoVerb` function of the embedded object's `IOleObject` interface, passing the verb `OLEIVERB_SHOW`, which then makes the embedded object visible:

```
HRESULT BOlePart::Show (BOOL bShow)
{
  if (bShow) {
    HRESULT hRes = DoVerb(OLEIVERB_SHOW);
    pSite->Invalidate(BOLE_INVAL_VIEW);
    return hRes;
  }
  else {
    return pOleObject->DoVerb(OLEIVERB_HIDE,
                              NULL, this, 0,
                              pContainer->GetWindow(),
                              (LPRECT) 0);
  }
}
```

Calling `TOcPart::Show(bool)` is a simple way of controlling an embedded object, because the command can be entirely processed by low-level BOCOLE code.

A more complicated (and more typical) process is set into motion when doing things like telling an embedded object to paint itself. There are no default paint handlers in BOCOLE, so the request must filter up through the BOCOLE layer and connect to some C++ function in the server's OCF code. Assume the container wants the embedded object to paint itself in the container's view space. The container will call the function `TOcPart::Draw`, which filters down to BOCOLE and calls the `Draw` function of the server's `IViewObject` interface, which then calls the member function `Draw` of the server's OCF connector class `TOcRemView`. This `Draw` function does some internal processing, then sends a special WM_OCEVENT message to the view window.

WM_OCEVENT is an OCF-specific message that is used to pass OLE requests and information to the user application. Most operations performed on an embedded object result in some kind of WM_OCEVENT notification being sent to the server's view window. The WPARAM parameter carries an OCF notification code, and LPARAM contains additional notification data. Using Borland C++, messages are processed and dispatched using structures called *Event Response Tables* (EVTs), which are equivalent to Visual C++ message maps.To process WM_OCEVENT notifications, you add handlers to a window's EVT. To handle the OC_VIEWPAINT notification, you would add an EV_OC_VIEWPAINT entry:

```
DEFINE_RESPONSE_TABLE1(TMyOleContainerWindow, TOleContainerWindow)

  EV_OC_VIEWPAINT,

END_RESPONSE_TABLE;
```

When a WM_OCEVENT message is sent to the window with the notification code OC_VIEWPAINT in WPARAM, OCF will automatically call the handler `EvOcViewPaint`. Each notification code has a predefined handler name. WM_OCEVENT messages can be sent both to servers and containers. Figure 12 lists the types of WM_OCEVENT notifications that OCF can generate.

Notification Code	Description
OC_APPBORDERSPACEREQ	Requests a container for border space to display a frame
OC_APPBORDERSPACESET	Sets the amount of border space used to display a frame
OC_APPDIALOGHELP	Help button was pressed on a standard OLE dialog box
OC_APPFRAMERECT	Gets the inner rectangle of the main window
OC_APPINSMENUS	Inserts menus of embedded/linked object into main menu
OC_APPMENUS	Sets main menu of the application
OC_APPPROCESSMSG	Processes accelerator key messages
OC_APPRESTOREUI	Eliminates the toolbar and menu items added to the application for an activated linked/embedded object
OC_APPSHUTDOWN	Closes the main window
OC_APPSTATUSTEXT	Sets the text on the status line
OC_VIEWATTACHWINDOW	Server attaches itself to an owner window.
OC_VIEWBREAKLINK	Server breaks a link to an item.
OC_VIEWCLIPDATA	Server renders its data in a given format.
OC_VIEWCLOSE	Server closes its remote view in the container view space.
OC_VIEWDRAG	A linked/embedded object is being dragged. Give feedback to the container application and the user.
OC_VIEWDROP	An object is being embedded into a container by a drop operation.
OC_VIEWGETITEMNAME	Server names its contents or selection.
OC_VIEWGETPALETTE	Server returns the palette used to draw its view.
OC_VIEWGETSCALE	Asks a container for view scaling information
OC_VIEWGETSITERECT	Asks a container for the bounding rectangle of the view site
OC_VIEWINMENUS	Server inserts its own menus in the container's menu bar.
OC_VIEWLOADPART	Server loads its document data.
OC_VIEWOPENDOC	Asks a container to open an exisiting document
OC_VIEWPAINT	Server paints itself in the container's view space.
OC_VIEWPARTINVALID	A linked/embedded object was invalidated
OC_VIEWPARTSIZE	Requests the view extent of a linked/embedded object
OC_VIEWSAVEPART	Server saves itself in a compound file
OC_VIEWSCROLL	Requests a linked/embedded object to scroll its view
OC_VIEWSETLINK	Server establishes a link to an item.
OC_VIEWSETSCALE	Servers sets up scaling information.
OC_VIEWSETSITERECT	Requests a container to set the size of the bounding rectangle of the view of a linked/embedded object
OC_VIEWSETTITLE	Sets the window title
OC_VIEWSHOWTOOLS	Server displays its toolbar in the container's view space.
OC_VIEWTITLE	Requests the window title of a view

Figure 12 - The OCF notifications sent with WM_OCEVENT messages.

The handling of notifications through EVT dispatching fits very well into OWL, because it is an extension of the event-handling mechanism already used to dispatch child control notifications and regular Windows messages.

Persistence

Given the ability to embed server objects inside containers, the container must be able to serialize the embedded objects. For example, if I create a simple container with embedded Microsoft Word documents, my container should have the ability to save the Word documents to a file and later read those documents back in. OLE uses the *Structured Storage Model* and *compound files* to store documents that have embedded objects.

In a nutshell, a compound file is made up of storages and streams. A storage is equivalent to a sub-directory (inside the compound file) and the stream is equivalent to a file (again, inside the compound file). A compound file lets you save documents with objects embedded recursively to any depth.

Conceptually, compound files are simple, but using the low level OLE API for structured storage can be very complex. There are several interfaces, like `IStorage`, `IStream`, `ILockBytes`, and dozens of different functions. OCF and OWL makes working with compound files a breeze. When you create a server object, you derive its document class from `TOleDocument`. This class is derived from `TStorageDocument`, which is derived from the standard OWL document-handling class `TDocument`. Class `TOleDocument` uses an internal OCF object of type `TOcDocument` to handle basic document serialization support. Class `TStorageDocument` puts a C++ stream interface around the low-level structured storage OLE interfaces. To support serialization of a user document derived from `TOleDocument`, you must override the member functions `Commit` and `Open`, like this:

```
class MyServer: public TOleDocument {
  // declare some user data;
  int a;
  float b;
  .
  .
  .
};

bool MyServer::Commit(bool force)
{
  // get an output stream to use
  // with an OLE compound file
  TOutStream* os = OutStream(ofWrite);
  if (!os) return false;

  // save our document data
  *os << a << ' ' << b;

  TOleDocument::Commit(force);  // save all objects embedded
                                // in the server and commit the
                                // transacted storage


  delete os;
return true;
}

bool MyServer::Open(int mode, const char* path)
{
  TOleDocument::Open(mode, path);  // read all objects embedded
                                   // in the server

  if (GetDocPath()) {
    // get an input stream to use for reading
    // an OLE compound file
    TInStream* is = (TInStream*)InStream(ofRead);
    if (!is) return false;
```

```
    // read our user data from the compound file
    *is >> a >> b;
    delete is;
}
  return true;
}
```

The reading and writing is handled using standard C++ stream insertion and extraction operators. The base class `TStorageDocument` handles the instantiation of the OCF stream objects that support these operators. The input stream is of type `TStorageInStream`, the output stream of type `TStorageOutStream`. The stream objects are allocated dynamically by `TStorageDocument`, so you must delete them yourself.

Server objects linked or embedded in a client container don't read or write to compound files on their own. It is the client container that controls when its contained objects read or write themselves. Containers implemented using OWL use the same class to handle their document data as servers — a class derived from `TOleDocument`. Assume this class is called `TMyOleDocument`. When the user executes a **File Save** command for one of its windows containing linked/embedded objects, the container's document manager invokes `TOleMyDocument::Commit`. This function saves any private data the document may have, and invokes the base class to write all linked/embedded objects out to storage. The `Commit` code is essentially the same for containers and servers.

When the user opens a compound file, the application's document manager invokes the member function `TMyOleDocument::Open`, which reads in any private data you might have, and then invokes the base class to read in any linked/embedded objects that were saved. Again, the `Open` code is essentially the same for containers and servers. You aren't required to use OWL in your application, but it comes very naturally, since AppExpert generates OWL code with embedded calls to OCF objects that do the grunt work. If you use Visual C++ or C, then you will need to look at one of the Borland sample programs to see exactly where you need to add OCF objects to your app.

Inheritance with OCF

Since OCF is a C++ class hierarchy, it follows that you can use standard C++ inheritance to derive new classes from OCF classes. As mentioned earlier, to create a server you multiply derive a class from the classes `TApplication` and `TOcModule`, like this;

```
class TMyServer: public TApplication, public TOcModule {...};
```

Don't get too excited: inheritance is limited to building OLE objects from C++ classes. It isn't currently possible to use inheritance to derive one OLE object from another. For example, given an OLE server called `Furniture`, you can't create a new `Chair` server by deriving it from `Furniture`. Obviously, if you have the source code for `Furniture`, then you can use it to derive some kind of `Chair` class:

```
class Chair: public Furniture {...};
```

but you still have to transform the new class into an OLE object.I It is possible to create a system in which objects dynamically inherit from other objects at run-time, but OLE doesn't attempt to deal with such problems.

There is a way to put objects together that's reminiscent of inheritance known as *aggregation*. Using aggregation, you can basically *chain* together separately allocated and implemented objects, so that one object appears to have inherited functionality from the other.

OCF defines the class `TUnknown` that can be used to encapsulate an OLE object's `IUnknown` interface, with support for aggregated objects. Objects aggregated inside `TUnknown` must also have TUnknown encapsulation. Figure 13 shows the relationship between aggregate and aggregated OLE objects in `TUnknown`.
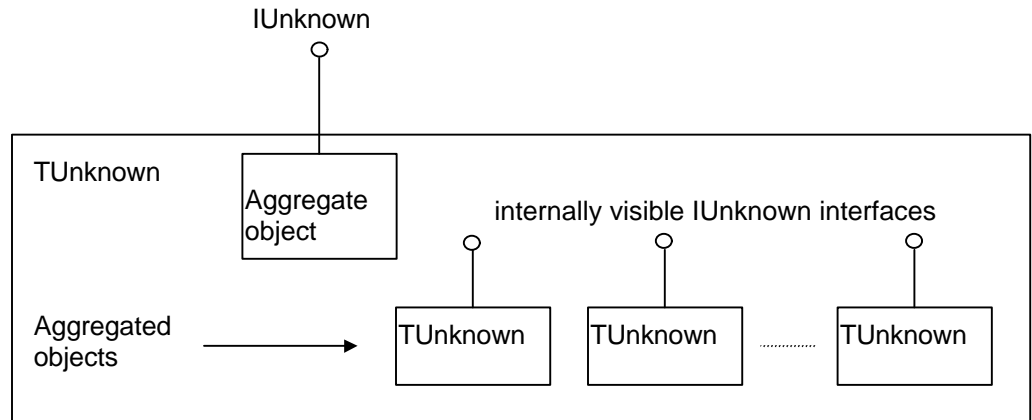

Figure 13 - A `TUnknown` object containing aggregated TUnknown objects inside it.

From the outside, OLE see only one `IUnknown` interface. The `IUnknowns` of the aggregated objects are hidden inside the aggregate object. When the user requests the aggregate object for an interface it doesn't support directly, the request is passed on to each of the aggregate objects. If one of the objects has the requested interface, it is returned to the caller. To OLE, it appears that the interfaces of all the aggregated objects are supported by the single OLE object. OLE is unaware that class TUnknown even exists. It isn't exactly inheritance, but the results are similar.

Creating an OLE Container

Now that most of the small details have been taken care of, it's time to see some code in action. Borland C++ 4.5 makes it very easy to create an OLE container, because AppExpert is now OLE-aware. Using the AppExpert (which is similar to the Visual C++ AppWizard), click on the OLE options you want, hit the Generate button, and you wind up with a complete project to start off your OLE container application. AppExpert is essentially a one-shot deal: once you generate an application, you can't add features at a later time by running AppExpert on it again. AppExpert doesn't know how to take an existing project (regardless of whether it was written in Visual C++, Borland C++ or other), and add new features to it. It creates a brand new project every time.

 To demonstrate everything, I created an app called CONTAIN using AppExpert. To launch AppExpert, you select the **Project AppExpert** command from the main menu (see figure 14).
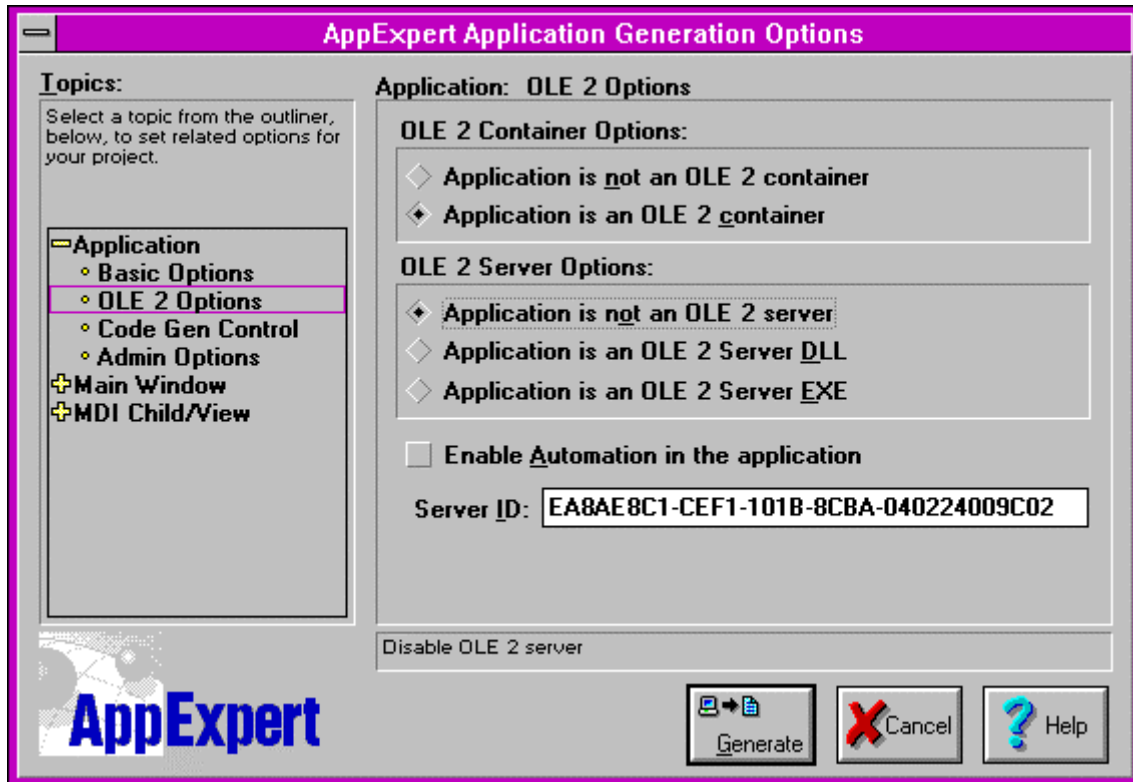

Figure 14 - The AppExpert window, showing the OLE options.

I clicked the Container option, gave the project a name, chose the MDI app type and that's about it. What I wound up with were seven files with roughly 1400 lines of source code. The simple app supported basic things such as **File New** and **File Open**, but most importantly, the **Edit** menu had an item called **Insert Object**. As mentioned earlier, this command allows you to insert OLE objects into an application, and displays a standard OLE **Insert Object** dialog box with a list of the OLE objects registered in the system.

To see how inserting an OLE object works, I ran CONTAIN and selected **Edit Insert Object**. I chose the object type *Microsoft Word 6.0 Document*, and wound up with a small Word object in an MDI child window of CONTAIN. I double-clicked the Word object to in-place activate Word, and wound up with an empty Word window with a ruler (see figure 15).
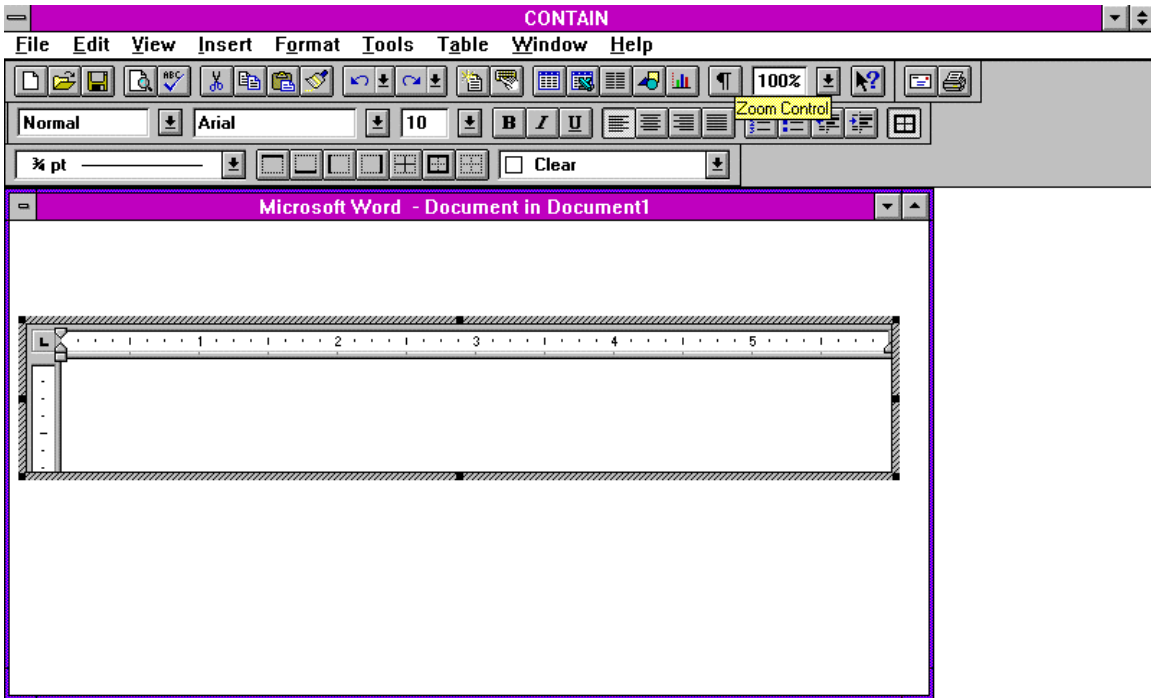
Figure 15 - An in-place activated Word 6.0 object.

I typed some text, added an embedded picture and some formatted text, and closed Word, to see if all the features would show up unchanged in CONTAIN. Sure enough they did, and I wound up with a window that looks like figure 16.
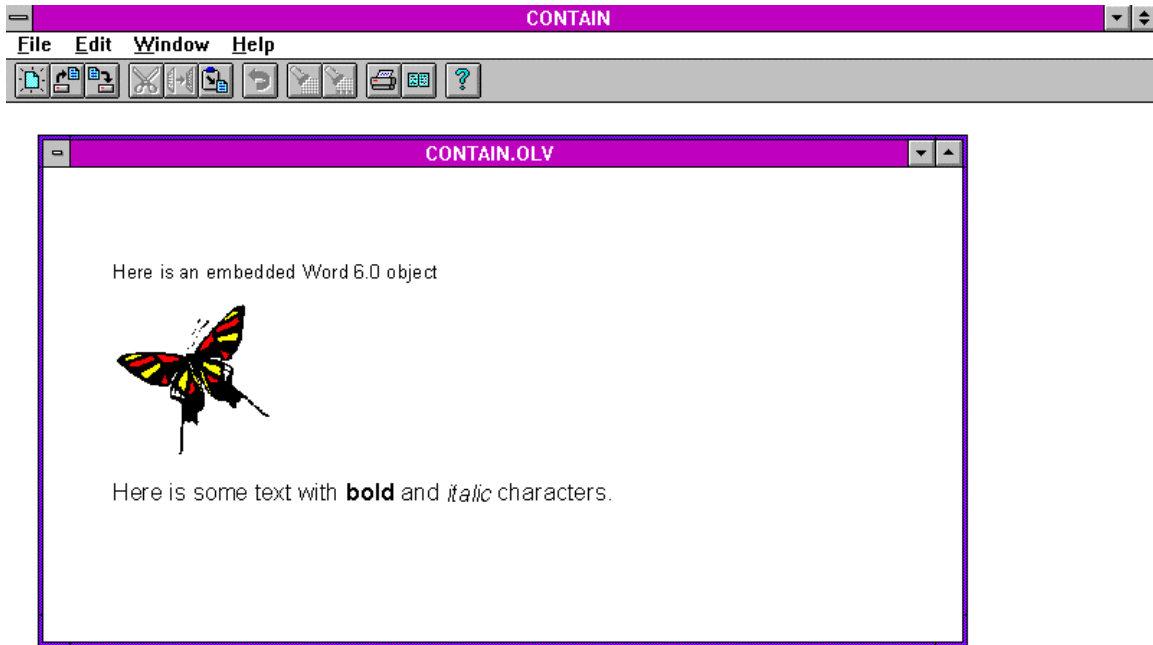
Figure 16 - The de-activated Word object, showing an embedded picture and formatted text.

One of the less-obvious features of an OLE container app is the ability to save embedded objects in a file. The new structured storage of OLE 2.0 is completely supported by the OCF. I put CONTAIN to the test. I selected **File Save As**, and entered a filename. I closed the child window with the embedded Word object, then used the **File Open** command. The child window returned, with its embedded document. There's an awful lot going on to make all this happen!

Creating an OLE Server

Using AppExpert, it is just as easy to create a server or even a container-server as it is to create a container. AppExpert lets you create both EXE and DLL servers. For example, I brought up AppExpert and selected the options MDI and OLE EXE Server. After a few seconds, I had the source code for an application called SERVER. First I tested SERVER by running it from the File Manager, as a standalone Windows app. When you run a server built with Borland C++ 4.5, there is no need to manually register it: servers register themselves. SERVER is very simple, just supporting plain empty MDI child windows (see figure 17).
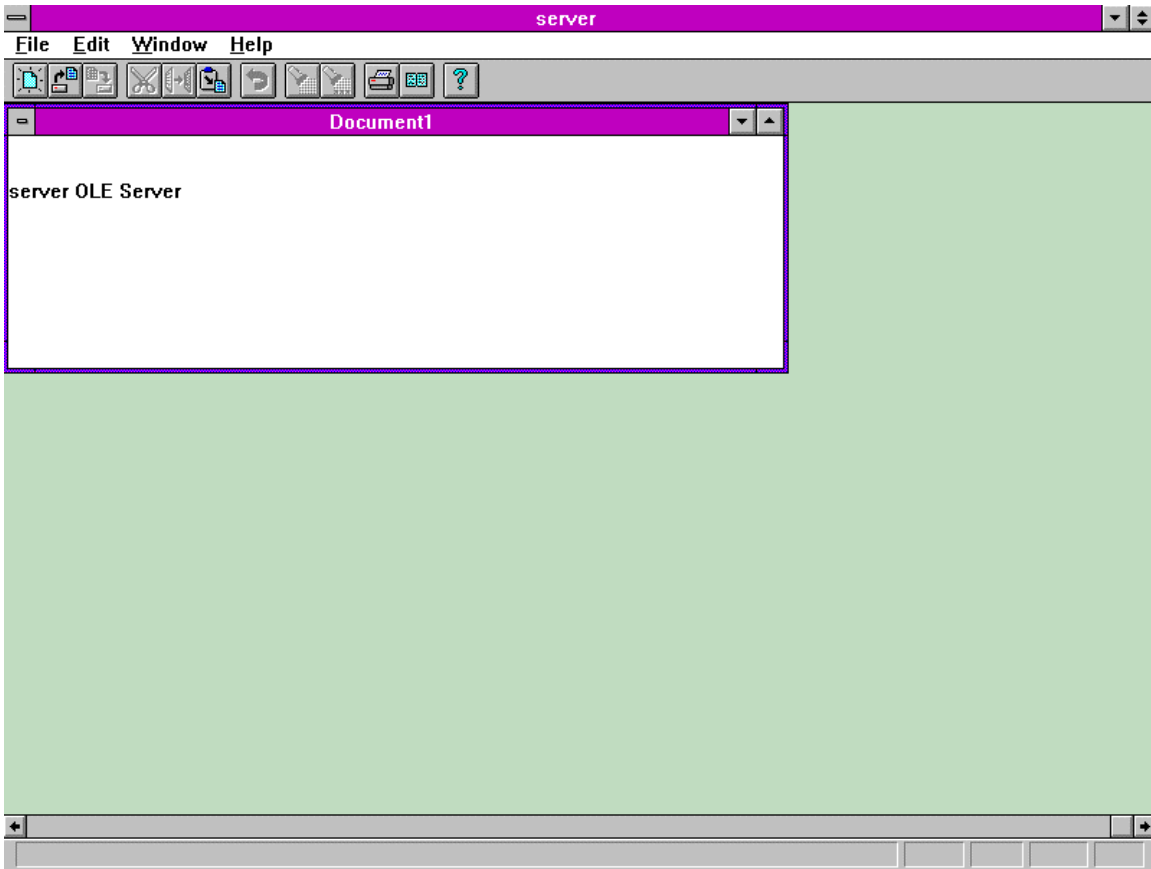
Figure 17 - How SERVER looks, after opening a child window.

Next, I tried to embed two separate objects in an MDI child window of CONTAIN, a Word 6.0 object and a SERVER object. Figure 18 shows how the child window looked with the two embedded objects.
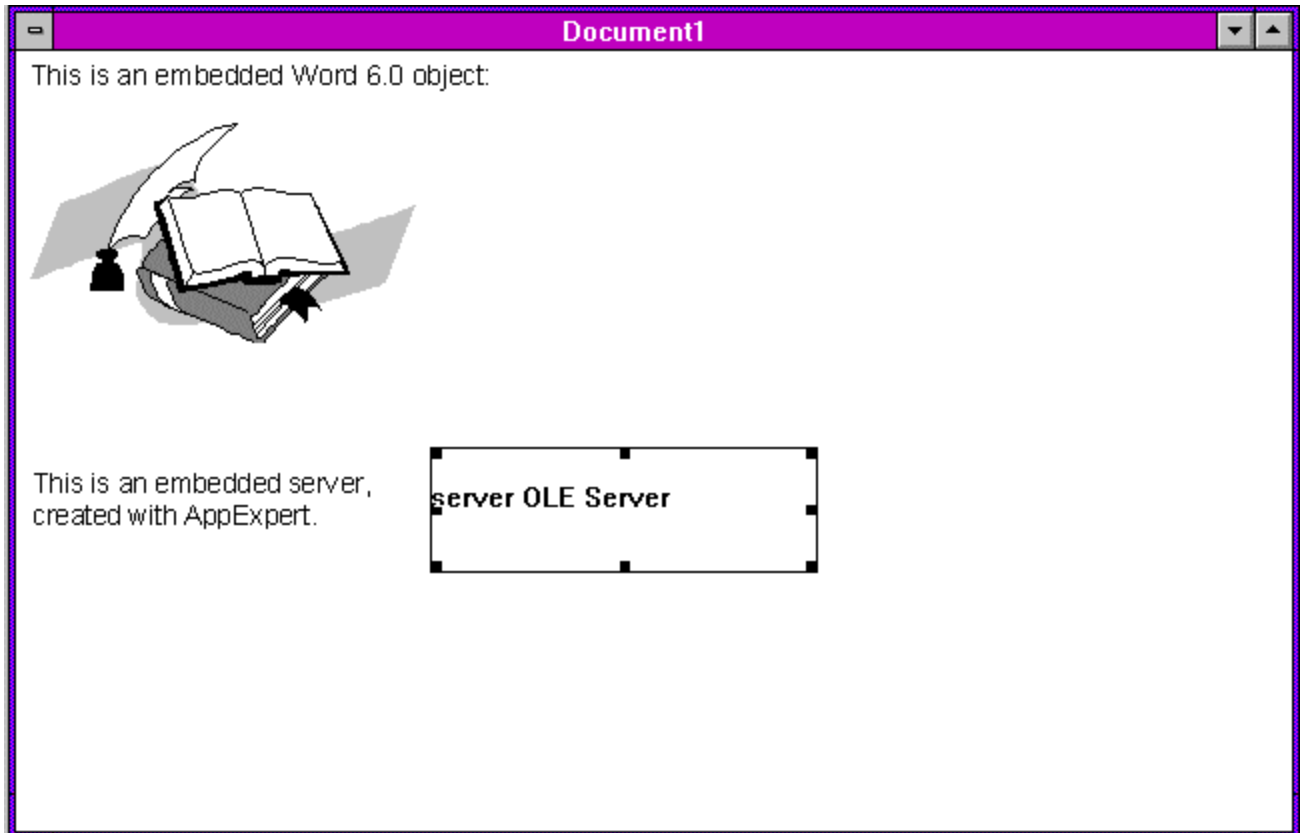
Figure 18 - A child window of CONTAIN, showing an embedded Word object, and an embedded SERVER object created with AppExpert.

Automation Support

Conceptually, an automated object is simply an OLE object that exposes some of its methods and properties to OLE containers. The exposed items can then be referenced from OLE containers that link to or embed the automated objects. The OLE automation technology does not rely on linking or embedding to work, but automation is still a rather complex beast. It requires stuff like registration attributes, type libraries, OLE storage allocators, macros for declaring and exposing methods and properties, and the list goes on... (For more details see the article "Application Interoperability with Visual Basic for Applications and OLE 2.0" by Joshua Trupin, MSJ Vol 9, No 2). Fortunately, Borland C++ acts as a liaison between you and the OLE automation details -- AppExpert creates the groundwork, and ClassExpert fills in the remaining details.

Exposing functionality

An automation controller can't necessarily get to ALL methods or properties of an automated object, only to those methods and properties that have been exposed. OCF uses macros to expose member functions (methods) and properties (data members).

Consider creating the old pocket calculator in figure 9 as an automated server. If you want clients (automation controllers) to be able to press the buttons on the calculator, you might expose a method called `Press` that takes a `string` parameter and returns `void`. Say you also want your

automated calculator to expose a property, such as the value displayed in the result window. The first step is to use a couple of macros: one for declaration and one for implementation purposes, much like the macros used to set up event response tables.

The ClassExpert that comes with Borland C++ 4.5 fully supports automation code. To automate a class, you simply turn on ClassExpert, select a class in the Class Browser Pane, right-click the mouse on the class, and select the **Automate class** command from the popup menu (see figure 19).
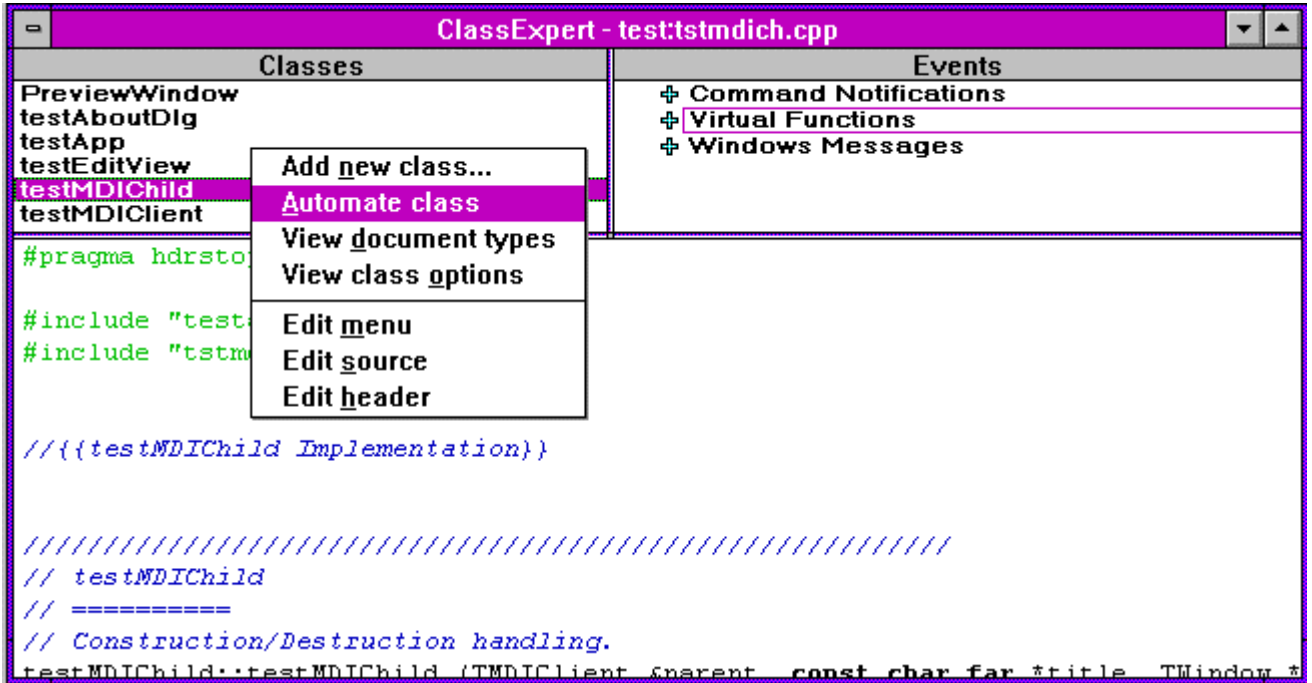
Figure 19 - Using the Borland ClassExpert to automate a class.

One of the neatest tricks Borland added to the new ClassExpert is the ability to add automation to an application that had been created without it. ClassExpert is able to detect applications that are being upgraded to automation, and add all the necessary code for you. The only proviso is that you created the original app using AppExpert.

Once you have enabled automation for a class, you can use the ClassExpert to expose properties and methods. Figure 20 shows the Events Pane, with a new entry called Automation, supporting the addition and deletion of properties and methods.

Figure 20 - Adding an automation method using the ClassExpert.

Although ClassExpert handles all the details for you, it is interesting to see what actually happens to your code. If you automated a calculator application that exposed the `Total` property and the `Press` method, your application class would be declared something like this:

```
class  TMyCalculator : public Tapplication, public TOcModule {

  .
  .
  .
  // automation methods and properties exposed
DECLARE_AUTOCLASS(TMyCalculator)
  AUTOFUNC1V(PressButton, Press, string)
  AUTODATA(Result, Total, double)
};
```
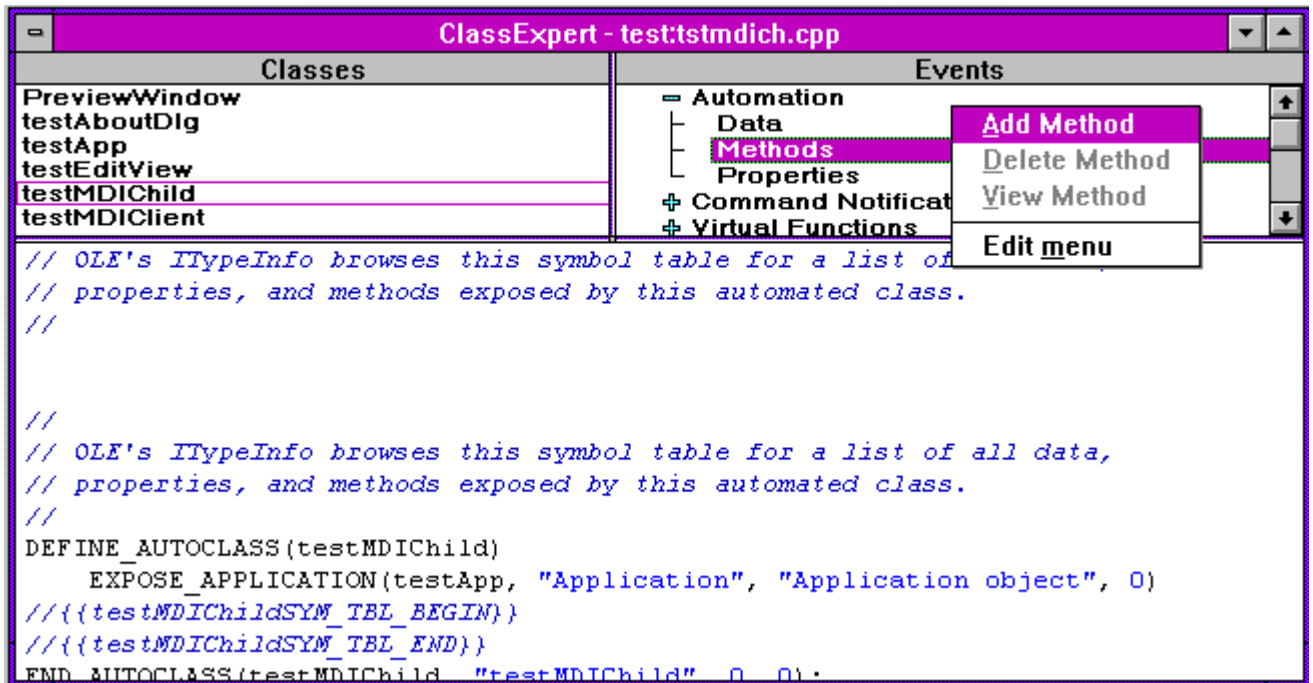
The `DECLARE_AUTOCLASS` macro supports a number of different line-item macros to declare methods and properties. For example, the macro `AUTOFUNC1V` above declares a method with the following characteristics:

- Internal name `TMyCalculator::PressButton`
- External name: `Press`
- Number of parameters accepted: 1
- Parameter type: `string`
- Return type: `void`

The internal name of a method or property is the identifier used in the source code. The external name is the identifier you want users to see through OLE. For example, if you exposed a method with the external name `DoSomething`, then `DoSomething` would be listed as one of the verbs available for your automated object. The internal and external names of properties and methods don't need to be the same, although frequently they are. The macro `AUTOFUNC1V` exposes a `void` function taking a single parameter of type `string`, and is defined like this:

```
#define AUTOFUNC1V(name, func, type1, defs) \
  AUTOFUNC_(name, This->func(Arg1);, defs, \
  DEFARGS1(type1), BLDARGS1(type1), CTRARGS1(type1), SETARGS1)
```

There is an elaborate succession of macros that call one another. The final result is a list of type identifiers, that are subsequently used to create a data structure that defines the exposed methods and properties.The macro `AUTODATA` shown above exposes a property with the following characteristics:

- Internal name: `Result`
- External name: `Total`
- Type: `double`

After declaring exposed items, you need to add another set of macros to implement the code that will hook up with OLE to physically make the items available to OLE. Again, ClassExpert does all the work for you. Using the calculator example, the macros might look like this:

```
DEFINE_AUTOCLASS(TMyCalculator)
  EXPOSE_PROPRW(Result, TAutoDouble, "Total",
                "The value shown in the result window", 0)
  EXPOSE_METHOD(PressButton, TAutoVoid, "Press",
                "Press one of the calculator buttons", 0)
    REQUIRED_ARG(TAutoString, "keys")
  EXPOSE_APPLICATION(TMyCalculator, "Application",
                     "TMyCalculator Application Object", 0)
END_AUTOCLASS(TMyCalculator, "TMyCalculator",
              "Automated OLE Calculator", 0)
```

The `EXPOSE_PROPRW` exposes a property of type `double` that can be both read and written by client controllers. The internal name of the property is `Result` and the external name is `Total`. There are equivalent macros for exposing read-only and read-write properties of all types. The fourth parameter passed to `EXPOSE_PROPRW` is a description string, displayed when the user browses the automated object from an automation controller. The last parameter in the `EXPOSE` macros is a help context value. The example above has zeros, but if you supply a nonzero value, users will be able to invoke Windows Help regarding the exposed method or property, to get information on what the method/property does and how to usw it. Obviously adding a help context requires you to also provide a help file.

Exposing a method (member function) is a bit more complicated than exposing a property, because you have to define the return type and the number/type of the arguments. The `EXPOSE_METHOD` macro shown above defines a method whose internal name is `PressButton`, returning `void`. The external name made available to OLE is `Press`, and the method requires a single parameter of type `TAutoString`, which is a standard C++ `string`.

The last macro used above – `EXPOSE_APPLICATION` – registers some additional information about the automated object, allowing other applications to use the OLE `IClassFactory` interface to create new instances of the object. The use of this macro is recommended to adhere to OLE conventions.

Automation Proxy Objects

Okay, so I created an automated object with a whole bunch of macros (actually, ClassExpert did the work). But how does a controller application control the automated object? What you'd like to do is manipulate the automated object like a regular C++ object, something like this:

```
// define a C++ calculator OLE object
TMyAutomatedCalculator calculator;

// connect the calculator object to an OLE object
  .
  .
  .

// and use the calculator as a regular C++ object...

// find the value of 95 / 13
calculator.Press("95");
calculator.Press("/");
calculator.Press("13");
double result = calculator.Total();
```

Using Borland C++ (with or without OWL), you can make things work pretty much this way, but only after enlisting the services of a C++ class that connects to the automated object's OLE interface. Such a class is called an *automation proxy* class in Borland C++. `TMyAutomatedCalculator`, above is a proxy class. Proxy classes are declared the same way as any other class, except they are derived from `TAutoProxy`. What's different about Proxy classes is that you don't write them yourself. The Borland utility program `AUTOGEN` creates the class declaration and source code for you. There are actually a couple of steps in developing proxy classes. First you create an automated object, with the `EXPOSE` macros described earlier. Then you run the object from the File Maanager, passing the command line argument `-TypeLib`. The OCF code in the object detects this special command string, and automatically generates a type library, with an OLB file extension. Then you run the `AUTOGEN` utility, passing it the name of the OLB type library, which then creates the proxy class. The proxy class for the sample calculator described earlier would look something like this:

```
class TMyAutomatedCalculator: public TAutoProxy {

public:

  TMyAutomatedCalculator() : TAutoProxy(0x409) {}
  double GetTotal();
  void SetTotal(double);
  void Press(TAutoString keys);
};
```

The hex value passed by the constructor to the base class represents the locale setting to use. The value 0x409 is the locale ID for American English. `AUTOGEN` determines the locale ID from the system setting.

Each read-write property gets two access functions: one to read and one to write the property. OLE function names cannot be overloaded, because name mangling is not supported.

Proxy classes don't have data members – just member functions. To access properties in an automated object, you use the access member functions in the proxy object. The class declaration for a proxy object requires that each member function be connected to the OLE interface of the connected OLE automated object, but `AUTOGEN` does this for you. The calculator code would look something like this:

```
double TMyAutomatedCalculator::GetTotal()
{
  AUTONAMES0("GetTotal")
  AUTOARGS0()
  AUTOCALL_PROP_GET
}

void TMyAutomatedCalculator::SetTotal(double value)
```

```
{
  AUTONAMES0("SetTotal")
  AUTOARGS1(value)
  AUTOCALL_PROP_SET
}

void TMyAutomatedCalculator::Press(TAutoString keys)
{
  AUTONAMES0("Press")
  AUTOARGS1(keys)
  AUTOCALL_METHOD_VOID
}
```

Now you have almost everything needed to use the `TMyAutomatedCalculator` proxy object in an OLE automation controller. The last thing is to connect the proxy object to a physical OLE object. The proxy class has no knowledge of what class ID that underlying OLE object has, so you need to supply the necessary information. Simply provide the OLE object's program ID as registered in the OLE registry. If the calculator server was registered with the ID `MyCalculator.Application`, use code like this:

```
// define a proxy object
TMyAutomatedCalculator calculator;

// bind the object to an OLE object
calculator.Bind("MyCalculator.Application");

// use the calculator...
```

You don't have to unbind the proxy object from the OLE object when the proxy object goes out of scope, because the its destructor does the unbinding automatically and deletes the OLE object.

Ugly, Huge, Complex, but Great

OLE is ugly. OLE is huge. OLE is complex. But OLE is also great, and it promises to make program development much simple, with frameworks like MFC and OWL. There are several key technologies that still have to be invented or stabilized before you can use off-the-shelf objects to build complete OLE programs with minimal coding. One essential piece missing is the ability to embed or link to objects over a network. Another is the ability to store and retrieve objects from databases, in a manner that allows any program to use them, without knowledge of their internal structure. A final piece is a standard for defining generic high-level software interfaces, akin to the IC standards long used by hardware engineers.

Borland C++ 4.5 helps tremendously in building OLE servers and containers. I took you through a pretty fast tour of the OLE support Borland added to its flagship C++ compiler. There are many features of the package I didn't discuss in detail.

I was slightly disappointed to see that support for OLE Custom Controls was not included in version 4.5, but Borland says it will be in the next release. The OCX SDK was still in beta when 4.5 was being developed. Overall, I found Borland C++ 4.5 to be a great improvement over its predecessor. Using the OCF class `TUnknown`, it is now possible to develop aggregate objects that are C++ on the inside and OLE on the outside. Such objects are like software components that can reuse each other at runtime, taking us one step closer to full component-based software development. Apart from the OCX aspect, Borland C++ 4.5 support for OLE is not only complete, but also very well-thought-out and elegant.

SideBar - Using OCF with MFC

OCF is not tied to OWL in any way, allowing MFC programmers and others to incorporate OCF in their programs. Having said this, I don't necessarily recommend MFC programmers to flock to Borland for OLE support, since MFC provides all the necessary functionality almost for free. Even though OCF can be added to an MFC app, the task isn't exactly trivial. In any event, it is interesting to see how an MFC program would incorporate OCF, so I'll describe the steps to create an OLE MDI container application with MFC using OCF. I won't show all the code, because the details would obscure the overall discussion.

The MDI application's main window and its MDI child windows need to create an OCF object, known as a *helper* object, to handle all OLE-related events. To keep things tidy, the best approach is to derive classes from `CMDIFrameWnd` and `CMDIChildWnd`, adding the necessary code to create, destroy and handle the OCF helper objects. You then derive your own MDI frame window and child windows from the new OCF-empowered classes. Here's how you would declare the main OCF-aware window, from which your own main window would be derived:

```
class CMDIFrameOCF : public CMDIFrameWnd {

  protected:

    TOleFrameWin*   OCFHelper;

    afx_msg int     OnCreate(LPCREATESTRUCT);
    afx_msg void    OnDestroy();
    afx_msg LRESULT OnOcEvent(WPARAM, LPARAM);

    // OCF event handlers
    BOOL OnOcAppFrameRect(HWND hwnd, TRect far* rect);
    BOOL OnOcAppInsMenus(HWND hwnd, TOcMenuDescr far* sharedMenu);
    BOOL OnOcAppMenus(HWND hwnd, TOcMenuDescr far* menuDescr);
    BOOL OnOcAppProcessMsg(HWND hwnd, MSG far*);
    BOOL OnOcAppBorderSpaceReq(HWND, TRect far*);
    BOOL OnOcAppBorderSpaceSet(HWND hwnd, TRect far* rect);
    void OnOcAppStatusText(HWND, const char far* text);
    void OnOcAppRestoreUI(HWND hwnd);};

    DECLARE_MESSAGE_MAP()
};
```

The OCF helper object is created and destroyed in the `OnCreate` and `OnDestroy` functions. The member function `OnOcEvent` handles all `WM_OCEVENT` messages generated by OCF and dispatches them to the specialized event handlers like `OnOcAppFrameRect`. For each OCF notification listed in figure 12 there can be a specialized handler. My example only has 8 handlers, but you get the idea. `OnOcEvent` cracks the message parameters, calling handlers with arguments of the correct type. The handlers in my examples mostly return `TRUE` or `FALSE` values, making OCF carry out default actions. The code for `CMDIFrameWnd` is shown in figure 21.

```
BEGIN_MESSAGE_MAP(CMDIFrameOCF, CMDIFrameWnd)
  ON_WM_CREATE()
  ON_WM_DESTROY()
  ON_MESSAGE(WM_OCEVENT, OnOcEvent)
END_MESSAGE_MAP()


int CMDIFrameOCF::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  int val = CMDIFrameWnd::OnCreate(lpCreateStruct);

  OCFHelper = new TOleFrameWin(m_hWnd);
  OCFHelper->OnCreate();
  return val;
}


void CMDIFrameOCF::OnDestroy()
{
  CMDIFrameWnd::OnDestroy();

  OCFHelper->OnDestroy();
  delete OCFHelper;
  OCFHelper = 0;
}


bool CMDIFrameOCF::OnOcAppFrameRect(HWND hwnd,
                                    TRect far* rect)
{
  GetClientRect(*((CRect*)rect));
  return true;
}


bool CMDIFrameOCF::OnOcAppInsMenus(HWND hwnd,
                                   TOcMenuDescr far* sharedMenu)
{
  return false;
}


bool CMDIFrameOCF::OnOcAppMenus(HWND hwnd,
                                TOcMenuDescr far* menuDescr)
{
  return false;
}


bool CMDIFrameOCF::OnOcAppProcessMsg(HWND, MSG far*)
{
  return false;
}


bool CMDIFrameOCF::OnOcAppBorderSpaceReq(HWND, TRect far*)
{
  return true;
}


bool CMDIFrameOCF::OnOcAppBorderSpaceSet(HWND hwnd,
                                         TRect far* rect)
{
  return false;
}
```

```
void CMDIFrameOCF::OnOcAppStatusText(HWND,
                                     const char far* text)
{
}


void CMDIFrameOCF::OnOcAppRestoreUI(HWND hwnd) {}


LRESULT CMDIFrameOCF::OnOcEvent(WPARAM wParam, LPARAM lParam)
{
  switch (wParam) {

    case OC_APPFRAMERECT:
       return (LRESULT) (UINT)
               OnOcAppFrameRect(m_hWnd, (TRect far*)(lParam));

    case OC_APPBORDERSPACESET:
       return (LRESULT) (UINT)
               OnOcAppBorderSpaceSet(m_hWnd, (TRect far*)(lParam));

    case OC_APPBORDERSPACEREQ:
       return (LRESULT) (UINT)
               OnOcAppBorderSpaceReq(m_hWnd, (TRect far*)(lParam));

    case OC_APPINSMENUS:
       return (LRESULT) (UINT) (BOOL)
               OnOcAppInsMenus(m_hWnd, TOcMenuDescr far*)(lParam));

    case OC_APPMENUS:
       return (LRESULT) (UINT) (BOOL)
               OnOcAppMenusnd(m_hWnd, (TOcMenuDescr far*)(lParam)));

    case OC_APPRESTOREUI:
       return OnOcAppRestoreUI(m_hWnd, 0L));

    default:
      return 0;
  }
}
```

Figure 21 - The implementation of an MFC MDI main frame window with an OCF helper object.

To support MDI child windows, you need to create a class equivalent to `CMDIFrameOCF`, but derived from `CMDIChildWnd`, like this:

```
class CMDIChildOCF : public CMDIChildWnd {

  protected:

    TOleWin*  OCFHelper;

    // message handlers
    afx_msg int     OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void    OnDestroy();
    afx_msg void    OnLButtonDown(UINT nFlags, CPoint pt);
    afx_msg void    OnLButtonDblClk(UINT nFlags, CPoint pt);
    afx_msg void    OnSize(UINT type, int cx, int cy);
    afx_msg LRESULT OnOcEvent(WPARAM wp, LPARAM lp);
    afx_msg void    OnUpdateInsertObject(CCmdUI* pCmdUI);
    afx_msg void    OnInsertObject();
```

```
      DECLARE_MESSAGE_MAP()
};
```

Class `CMDIChildOCF` is a little different from `CMDIFrameOCF`, because instead of having handlers for OCF events, it has handlers for standard Windows messages, like `WM_LBUTTONDOWN` and `WM_SIZE`. Class `CMDIFrameOCF` doesn't have any such handlers because it is only capable of responding to OCF notifications (via OnOcEvent), while class `CMDIChildOCF` is also able to initiate OCF actions in response to certain Windows messages. For example if the user resizes an MDI child window that contains embedded OLE objects, then the window will need to notify OCF in response to a `WM_SIZE` message, so OCF can go off and resize the embedded objects. Class `CMDIChildOCF` also needs to handle direct OLE operations like **Insert Object**, delegating the call to the OCF helper object. To handle OCF notifications, class `CMDIChildOCF` simply passes the messages along to the helper object. The code for `CMDIChildOCF` is shown in figure 22.

```
BEGIN_MESSAGE_MAP(CMDIChildOCF, CMDIChildWnd)
  ON_WM_CREATE()
  ON_WM_DESTROY()
  ON_WM_LBUTTONDOWN()
  ON_WM_LBUTTONDBLCLK()
  ON_WM_SIZE()
  ON_UPDATE_COMMAND_UI(CM_EDITINSERTOBJECT, OnUpdateInsertObject)
  ON_COMMAND(CM_EDITINSERTOBJECT, OnInsertObject)
  ON_MESSAGE(WM_OCEVENT, OnOcEvent)
END_MESSAGE_MAP()


int CMDIChildOCF::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  OCFHelper = new TOleWin(m_hWnd);
  OCFHelper->OnCreate();

  return CMDIChildWnd::OnCreate(lpCreateStruct);
}


void CMDIChildOCF::OnDestroy()
{
  CMDIChildWnd::OnDestroy();

  OCFHelper->OnDestroy();
  delete OCFHelper;
  OCFHelper = 0;
}


void CMDIChildOCF::OnLButtonDown(UINT nFlags, CPoint pt)
{
  if (!OCFHelper->OnLButtonDown(pt.x, pt.y, nFlags))
    CMDIChildWnd::OnLButtonDown(nFlags, pt);
}


void CMDIChildOCF::OnLButtonDblClk(UINT nFlags, CPoint pt)
{
  if (!OCFHelper->OnLButtonDblClk(pt.x, pt.y, nFlags))
    CMDIChildWnd::OnLButtonDblClk(nFlags, pt);
}


void CMDIChildOCF::OnSize(UINT type, int cx, int cy)
{
  OCFHelper->OnSize();
```

```
    CMDIChildWnd::OnSize(type, cx, cy);
}


void CMDIChildOCF::OnUpdateInsertObject(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(GetOcApp() && OCFHelper->GetOcView());
}


void CMDIChildOCF::OnInsertObject()
{
   OCFHelper->EditInsertObject();
}


LRESULT CMDIChildOCF::OnOcEvent(WPARAM wParam, LPARAM lParam)
{
   return OCFHelper->OnOcEvent(wParam, lParam);
}
```

Figure 22 - The implementation of the OCF-enabled class CMDIChildOCF.

With the two classes CMDIFrameOCF and CMDIChildOCF, now you derive you own classes from them, like this:

```
class CMainFrame : public CMDIFrameOCF {
   // ...
};

class CMyChildWnd : public CMDIChildOCF {
   // ...
};
```

and use them without worrying about OLE. All the OLE-related stuff is caught and handled by the base classes. Let's take a look at what is inside the OCF helper objects. Class CMDIFrameOCF uses a helper of type TOleFrameWin, declared like this:

```
class TOleFrameWin {

public:

    TOleFrameWin(HWND);

    void     OnCreate();
    void     OnDestroy();
    void     OnActivateApp(BOOL);
    operator HWND() const {return Hwnd; };

protected:

    HWND     Hwnd;
};
```

The purpose of this class is to create an object to encapsulate OLE frame windows, allowing straightforward access to OCF. The member functions of TOleFrameWin use the global function GetOcApp to obtain an application-specific OCF object of type TOcApp, to which Create, Destroy and Activate events are dispatched. The code for TOleFrameWin is shown in figure 23:

```
TOleFrameWin::TOleFrameWin(HWND hwnd) : Hwnd(hwnd) {}

void TOleFrameWin::OnCreate()
```

```
{
  // Hand our frame HWND to OCF so it may send
  // OLE notifications/requests
  if (GetOcApp() ) GetOcApp()->SetupWindow(Hwnd);
}


void TOleFrameWin::OnDestroy()
{
  // Release the OCF Application helper object
  if (GetOcApp() ) GetOcApp()->Release();
}


void TOleFrameWin::OnActivateApp(BOOL active)
{
  GetOcApp()->EvActivate(active);
}
```

Figure 23 - The code for OCF helper objects used in MDI frame windows.

The OCF helper for MDI child windows is of type `TOleWin`, and is much more complicated then its main frame helper counterpart. The class declaration is shown in figure 24.

```
class TOleWin : public TOcMenuDescr {
  public:
    TOleWin(HWND hwnd);
    virtual ~TOleWin();

    operator        HWND() const {return Hwnd; };
    void            SetWinMenu(const TOcMenuDescr& menuDescr);
    HMENU           GetWinMenu() const {return HMenu; };
    HMENU           GetMergedMenu() const {return MergedMenu; };

    //
    // Windows message handlers
    //
    void            OnCreate();
    void            OnDestroy();
    void            OnSize();
    void            OnPaint(HDC hdc, PAINTSTRUCT& ps);

    bool            OnCommand(UINT);
    void            OnInitMenuPopup(HMENU hMenu, int item);
    void            OnInitMenu(HMENU hMenu);

    bool            OnLButtonDown(int x, int y, UINT keyFlags);
    bool            OnLButtonUp(int x, int y, UINT keyFlags);
    bool            OnLButtonDblClk(int x, int y, UINT keyFlags);
    bool            OnRButtonDown(int x, int y, UINT keyFlags);

    void            OnActivate(bool);
    void            OnSetFocus(HWND lostFocus);

    //
    // OCF event handlers
    //
    const char far* OnOcViewTitle();
    void            OnOcViewSetTitle(const char far*);
    bool            OnOcViewSetSiteRect(TRect far*);
    bool            OnOcViewGetScale(TOcScaleFactor far*);
    bool            OnOcViewPartInvalid(TOcPartChangeInfo far*);
    bool            OnOcViewGetSiteRect(TRect far*);
    bool            OnOcViewDrop(TOcDragDrop far*);
    LRESULT         OnOcEvent(WPARAM wParam, LPARAM lParam);
```

```cpp
        //
        // The following events are actually sent to the frame
        // However, the latter can delegate to the OLE Window
        //
        bool            OnOcAppInsMenus(TOcMenuDescr far* sharedMenu);
        HMENU           OnOcAppMenus(TOcMenuDescr far* menuDescr);
        HMENU           OnOcAppRestoreUI();

        //
        // OLE UI Dialog methods
        //
        void            EditInsertObject();
        void            EditPasteSpecial();
        void            EditConvert();
        void            EditLinks();

        //
        // OCF Helper functions
        //
        bool            Deactivate();
        void            DeactivateParts();
        void            SetSelection(TOcPart*);
        bool            Select(UINT, TPoint& pt);
        void            GetInsertPosition(TRect& rect);
        void            InvalidatePart(TOcInvalidate invalid);
        bool            PaintParts(HDC dc, bool erase,
                                  TRect& rect, bool metafile);
        void            GetLogPerUnit(TSize&);
        void            SetupDC(HDC dc, bool scale = true);
        bool            OleShutDown();
        TOcView*        CreateOcView(TRegLink*, bool isRemote,
                                     IUnknown* outer);

        //
        // Accessors
        //
        TOcDocument*    GetOcDoc() {return OcDoc;}
        TOcView*        GetOcView() {return OcView;}
        TOcRemView*     GetOcRemView()
                        {return TYPESAFE_DOWNCAST(OcView, TOcRemView);}
        bool            SelectEmbedded();

        bool            IsOpenEditing() const;
        bool            IsRemote() const { return Remote; }

        //
        // extra user info
        //
        void*           GetUserInfo() {return UserInfo;}
        void*           SetUserInfo(void* ui);
    protected:
        HWND            Hwnd;
        TOcDocument*    OcDoc;
        TOcView*        OcView;
        TOcScaleFactor  Scale;
        TOcPart*        DragPart;
        bool            Remote;
        TRect           Pos;
        bool            ShowObjects;
        HMENU           MergedMenu;
        void*           UserInfo;
        OleClientDC*    DragDC;
        TRect           DragRect;
};
```

Figure 24 - The OCF helper class used in MDI child windows.

Class `TOleWin` is derived from the OCF class `TOcMenuDescr`, which handles most of the details of OLE menu merging. `TOleWin` has a member function for every Windows message that can affect OLE objects linked or embedded in a window, plus functions to deal with every possible OCF notification sent to containers. `TOleWin` also has the responsibility for instantiating the OCF connector objects that manage linked/embedded objects, so it has data members pointing to `TOcDocument`, `TOcView`, and `TOcPart` objects. The class also knows how to merge its menus with those of in-place activated objects. If you think the class declaration was large, take a look at its implementation (see figure 25). Hey, I didn't say using OCF with MFC was trivial. I only said it was *possible*!

```
TOleWin::TOleWin(HWND hwnd) : Hwnd(hwnd),
                              Pos(0, 0, 0, 0),
                              Remote(false),
                              ShowObjects(true)
{
  OcDoc     = 0;
  OcView    = 0;
  DragPart  = 0;
  DragDC    = 0;

  //
  // Init menu related variables
  //
  MergedMenu = 0;
  HMenu      = 0;
  memset(Width, 0, sizeof(Width));
}


TOleWin::~TOleWin()
{
  if (OcDoc)
    OcDoc->Close();

  if (OcView && !IsRemote()) {
    OcView->ReleaseObject();
    OcView = 0;
  }

  delete OcDoc;
}


void TOleWin::OnCreate()
{
  // Create OcDoc/OcView pair
  if (!OcView)  CreateOcView(0, false, 0);

  // Hand OCF our window handle so it
  //may send OLE notifications/requests
  if (OcView)  OcView->SetupWindow(Hwnd, IsRemote());
}


void TOleWin::OnDestroy()
{
  OleShutDown();
}


// Create View/Doc pair associated with OLE window
TOcView* TOleWin::CreateOcView(TRegLink* link,
                               bool isRemote,
                               IUnknown* outer)
```

```
{
  if (!OcDoc) OcDoc = new TOcDocument(*GetOcApp());

  if (!OcView) {
    TRegList* regList = link ? &link->GetRegList() : 0;

    // Create a remote view on the server document
    // if it embeded, else make a normal, container view
    if (isRemote)
      OcView = new TOcRemView(*GetOcDoc(), regList, outer);
    else
      OcView = new TOcView(*GetOcDoc(), regList, outer);

    Remote = isRemote;
  }
  return OcView;
}


void TOleWin::OnSize()

{
  // Inform the OCF View helper that we've been resized
  if (OcView)  OcView->EvResize();
}


void TOleWin::OnPaint(HDC dc, PAINTSTRUCT& ps)
{
  bool metafile = GetDeviceCaps(dc, TECHNOLOGY) == DT_METAFILE;
  SetupDC(dc, !metafile);

  // make parts draw themselves
  PaintParts(dc, ps.fErase, TRect(ps.rcPaint), metafile);
}


// Method to call when OLE Window receives a WM_COMMAND with an
// id it does not support. This method handles the various
// OLE UI commands such as InsertObject or PasteSpecial
bool TOleWin::OnCommand(uint cmdID)
{
  switch (cmdID) {
    case CM_EDITINSERTOBJECT:
      EditInsertObject();
      break;

    case CM_EDITPASTESPECIAL:
      EditPasteSpecial();
      break;

    case CM_EDITCONVERT:
      EditConvert();
      break;

    case CM_EDITLINKS:
      EditLinks();
      break;

    case CM_EDITSHOWOBJECTS:
      ShowObjects = !ShowObjects;
      InvalidateRect(Hwnd, 0, TRUE);
      break;

    default:
      return false;        // We did not handle the command!
```

```
  }
  return true;               // We handled the command
}


void TOleWin::OnInitMenu(HMENU hMenu)
{
  EnableMenuItem(hMenu, CM_EDITINSERTOBJECT, MF_BYCOMMAND |
                  ((GetOcApp() && GetOcView()) ? MF_ON : MF_OFF));
  EnableMenuItem(hMenu, CM_EDITCONVERT,
              MF_BYCOMMAND|((DragPart != 0) ? MF_ON : MF_OFF));

  uint mask = GetOcApp()->EnableEditMenu(meEnablePaste |
                                          meEnablePasteLink |
                                          meEnableBrowseClipboard |
                                          meEnableBrowseLinks,
                                          GetOcView());
  EnableMenuItem(hMenu, CM_EDITPASTE,
              MF_BYCOMMAND|((mask & meEnablePaste) ?
              MF_ON : MF_OFF));
  EnableMenuItem(hMenu, CM_EDITPASTELINK,
              MF_BYCOMMAND|((mask & meEnablePasteLink) ?
              MF_ON : MF_OFF));
  EnableMenuItem(hMenu, CM_EDITPASTESPECIAL,
              MF_BYCOMMAND|((mask & meEnableBrowseClipboard) ?
              MF_ON : MF_OFF));
  EnableMenuItem(hMenu, CM_EDITLINKS,
              MF_BYCOMMAND|((mask & meEnableBrowseLinks) ?
              MF_ON : MF_OFF));

  CheckMenuItem(hMenu, CM_EDITSHOWOBJECTS,
                  MF_BYCOMMAND |(ShowObjects ?
                  MF_CHECKED : MF_UNCHECKED));
}


// Method to call when OLE Window receives a WM_INITMENUPOPUP message
// (which is typically a hand-down from the frame window).
// Method takes care of enabling/disabling/checking etc. OLE related
// commands.
void TOleWin::OnInitMenuPopup(HMENU hMenu, int /*item*/)
{
  OnInitMenu(hMenu);
}


// Method to call when OLE Window receives a WM_LBUTTONDOWN message.
// Deactivates any active parts.
bool TOleWin::OnLButtonDown(int x, int y, UINT keyFlags)
{
  if (Deactivate())
    return true;          // We handled the message...

  if (!DragDC)
    DragDC = new OleClientDC(*this);

  TPoint pt(x, y);
  DPtoLP(*DragDC, &pt, 1);
  return Select(keyFlags, pt);
}


bool TOleWin::Select(UINT KeyFlags, TPoint& point)
{
  if (!DragPart || !DragPart->IsVisible(TRect(point, TSize(1, 1))))
    SetSelection(GetOcDoc()->GetParts().Locate(point));
```

```
    if (DragPart) return true;
    return false;
}


// Method to call when OLE Window receives an WM_LBUTTONDBLCLK message.
// Activates the part if the mouse was clicked on an inactivate part.
bool TOleWin::OnLButtonDblClk(int x, int y, UINT modKeys)
{
  TPoint point(x, y);
  OleClientDC dc(*this);
  DPtoLP(dc, &point, 1);

  TOcPart* p = GetOcDoc()->GetParts().Locate(point);

  if (modKeys & MK_CONTROL) {
    if (p)
      p->Open(true);
  } else {
    SetSelection(p);

    if (p && p == GetOcView()->GetActivePart())
      p->Activate(false);

    GetOcView()->ActivatePart(p);
  }

  return true;
}


// Method to call when OLE window receives a WM_LBUTTONUP message
bool TOleWin::OnLButtonUp(int x, int y, UINT modKeys)
{
  if (DragPart) {
    TOcPartChangeInfo changeInfo(DragPart, invView | invData);
  }

  if (DragDC) {
    delete DragDC;
    DragDC = 0;
    DragRect.SetNull();
  }

  return false;
}


// Method to call when OLE window receives a WM_ACTIVATE or
// WM_MDIACTIVATE message.
void TOleWin::OnActivate(bool active)
{
  GetOcView()->EvActivate(active);
}


void TOleWin::OnSetFocus(HWND /*hwndLostFocus*/)
{
  GetOcView()->EvSetFocus(true);
}


// Method to call when OLE Window receives a WM_RBUTTONDOWN message.
// Implements the Local Object menu (with VERBS) if the mouse was
// clicked on a part.
bool TOleWin::OnRButtonDown(int /*x*/, int /*y*/, uint /*keyFlags*/)
{
```

```
      return true;
}


// Method to call when OLE Window receives a WM_OCEVENT message.
// This method invokes a separate handler for the various OCF
// events. These in turn provide various default behaviours.
LRESULT TOleWin::OnOcEvent(WPARAM wParam, LPARAM lParam)
{
  switch (wParam) {
    case OC_VIEWTITLE:
          return (LRESULT)OnOcViewTitle();

    case OC_VIEWSETTITLE:
          OnOcViewSetTitle((const char far*)lParam);
          return (LRESULT)true;

    case OC_VIEWSETSITERECT:
          return (LRESULT)
                  OnOcViewSetSiteRect((TRect far*)lParam);

    case OC_VIEWGETSCALE:
          return (LRESULT)
                  OnOcViewGetScale((TOcScaleFactor far*)lParam);

    case OC_VIEWPARTINVALID:
          return (LRESULT)
                  OnOcViewPartInvalid((TOcPartChangeInfo far*)lParam);

    case OC_VIEWGETSITERECT:
           return (LRESULT)OnOcViewGetSiteRect((TRect far*)lParam);

    case OC_VIEWDROP:
          return OnOcViewDrop((TOcDragDrop far*)lParam);

    default:
      return LRESULT(false);
  }
}


const char* TOleWin::OnOcViewTitle()
{
  static char title[100];

  if (IsWindow(Hwnd))
    GetWindowText(Hwnd, title, sizeof(title));
  return title;
}


void TOleWin::OnOcViewSetTitle(const char far* text)
{
  if (IsWindow(Hwnd))  SetWindowText(Hwnd, text);
}


bool TOleWin::OnOcViewGetScale(TOcScaleFactor* scaleInfo)
{
  if (scaleInfo)  *scaleInfo = Scale;
  return true;
}


bool TOleWin::OnOcViewSetSiteRect(TRect far* rect)
{
  OleClientDC dc(*this);
```

```
        return DPtoLP(dc, (POINT*)rect, 2);
}


bool TOleWin::OnOcViewGetSiteRect(TRect far* rect)
{
  OleClientDC dc(*this);
  return LPtoDP(dc, (POINT*)rect, 2);
}


bool TOleWin::OnOcViewPartInvalid(TOcPartChangeInfo far* changeInfo)
{
  TRect rect(changeInfo->GetPart()->GetRect());
  rect.right++;
  rect.bottom++;

  OleClientDC dc(*this);
  LPtoDP(dc, (TPoint*)&rect, 2);
  InvalidateRect(Hwnd, &rect, true);
  InvalidatePart((TOcInvalidate)changeInfo->GetType());
  return true;
}


bool TOleWin::OnOcViewDrop(TOcDragDrop far*)
{
// we'll take any drop
return true;
}


// Displays the InsertObject dialog and allows user to
// embed an object [newly created or created from file]
void TOleWin::EditInsertObject()
{
  try {
    TOcInitInfo initInfo(OcView);
    if (GetOcApp()->Browse(initInfo)) {
      TRect rect;
      GetInsertPosition(rect);
      SetSelection(new TOcPart(*GetOcDoc(), initInfo, rect));

      OcView->Rename();
      InvalidatePart(invView);
    }
  }
  catch (TXBase& xbase) {
    MessageBox(Hwnd, xbase.why().c_str(), "EXCEPTION: InsertObject", MB_OK);
  }
}


// Displays PasteSpecial dialog and allows use to paste/pasteLink
// object current on the clipboard.
void TOleWin::EditPasteSpecial()
{
  try {
    TOcInitInfo initInfo(GetOcView());

    if (GetOcView()->BrowseClipboard(initInfo)) {
      TRect rect;
      GetInsertPosition(rect);
      new TOcPart(*GetOcDoc(), initInfo, rect);
      initInfo.ReleaseDataObject();
    }
  }
```

```
    catch (TXBase& xbase) {
      MessageBox(Hwnd, xbase.why().c_str(),
                 "EXCEPTION: PasteSpecial", MB_OK);
    }
}


void TOleWin::EditConvert()
{
  GetOcApp()->Convert(DragPart, false);
}


void TOleWin::EditLinks()
{
  GetOcView()->BrowseLinks();
}


bool TOleWin::SelectEmbedded()
{
  return DragPart != 0;
}


void TOleWin::InvalidatePart(TOcInvalidate invalid)
{
  if (GetOcRemView())
    GetOcRemView()->Invalidate(invalid);
}


bool TOleWin::Deactivate()
{
  if (DragPart && DragPart->IsActive()) {
    SetSelection(0);
    return true;
  } else {
    return false;
  }
}


void TOleWin::DeactivateParts()
{
  // Deactivate the embedded parts
for (TOcPartCollectionIter i(OcDoc->GetParts()); i; i++) {
    TOcPart& p = *i.Current();
    p.Activate(false);
  }
}


bool TOleWin::OleShutDown()
{
  if (IsRemote()) {
    TOcRemView* ocRemView = GetOcRemView();
    if (IsOpenEditing())
      ocRemView->Disconnect();
  } else {
    if (OcView)  OcView->EvClose();
  }
  return true;
}


void TOleWin::SetSelection(TOcPart* part)
```

```
{
  if (part == DragPart)
    return;

  // Invalidate old part
  TOcPartChangeInfo changeInfo(DragPart, invView);
  if (DragPart) {
    DragPart->Select(false);
    DragPart->Activate(false);
    OnOcViewPartInvalid(&changeInfo);
  }

  // Select and invalidate new one
  DragPart = part;
  changeInfo.SetPart(DragPart);
  if (DragPart) {
    part->Select(true);
    OnOcViewPartInvalid(&changeInfo);
  }
}


void TOleWin::GetInsertPosition(TRect& rect)
{
  OleClientDC dc(*this);

  // Default to 0.5" from viewport origin
  rect.left  = rect.top = 0;
  rect.right = GetDeviceCaps(dc, LOGPIXELSX) >> 1;
  rect.bottom= GetDeviceCaps(dc, LOGPIXELSY) >> 1;

  LPtoDP(dc, (TPoint*)&rect, 2);

  // Embedded rect is in pixels
  rect.left  = rect.Width();
  rect.top   = rect.Height();
  rect.right = rect.bottom = 0;
}


// Get the logical unit per inch for document
void TOleWin::GetLogPerUnit(TSize& logPerUnit)
{
  HDC dc = GetWindowDC(0);  // Screen DC
  logPerUnit.cx = GetDeviceCaps(dc, LOGPIXELSX);
  logPerUnit.cy = GetDeviceCaps(dc, LOGPIXELSY);
  ReleaseDC(0, dc);
}


void TOleWin::SetupDC(HDC dc, bool scale)
{
  SetMapMode(dc, MM_ANISOTROPIC);
  TPoint scrollPos(0, 0);

  // Adjust for scrolling here
  // ...

  SetWindowOrgEx(dc, scrollPos.x, scrollPos.y, 0);
  if (!scale)  return;
  SetViewportOrgEx(dc, Pos.left, Pos.top, 0);


  TSize ext;
  GetLogPerUnit(ext);
  SetWindowExtEx(dc, ext.cx, ext.cy, 0);
```

```
      ext.cx = GetDeviceCaps(dc, LOGPIXELSX);
      ext.cy = GetDeviceCaps(dc, LOGPIXELSY);


      ext.cx = (int)(((uint32)ext.cx * Scale.SiteSize.cx +
                      (Scale.PartSize.cx >> 1)) / Scale.PartSize.cx);
      ext.cy = (int)(((uint32)ext.cy * Scale.SiteSize.cy +
                      (Scale.PartSize.cy >> 1)) / Scale.PartSize.cy);

      SetViewportExtEx(dc, ext.cx, ext.cy, 0);
    }


bool TOleWin::PaintParts(HDC dc, bool, TRect&, bool metafile)
{
   TRect clientRect;
   TRect logicalRect;

   GetClientRect(Hwnd, &logicalRect);

   if (IsRemote()) {
     GetWindowRect(Hwnd, &clientRect);
     clientRect.Offset(-clientRect.left, -clientRect.top);
   } else {
     clientRect = logicalRect;
   }

   TPoint scrollPos(0, 0);

   if (!metafile)
     DPtoLP(dc, (TPoint*)&logicalRect, 2);

   for (TOcPartCollectionIter i(GetOcDoc()->GetParts()); i; i++) {
     TOcPart& p = *i.Current();
     if (p.IsVisible(logicalRect) || metafile) {
       TRect r = p.GetRect();
       r.Offset(-scrollPos.x, -scrollPos.y);
       p.Draw(dc, r, clientRect, asDefault);

       if (metafile)
         continue;

       // Paint selection
       //
       if ((p.IsSelected() || ShowObjects) &&
            r.Width()>0 && r.Height()>0) {
         TPoint pts[5] = {TPoint(r.left, r.top),
                          TPoint(r.right, r.top),
                          TPoint(r.right, r.bottom),
                          TPoint(r.left, r.bottom),
                          TPoint(r.left, r.top)};
         Polyline(dc, pts, sizeof(pts)/sizeof(pts[0]));
       }
     }
   }

   return true;
}


// Sets and returns previous user-defined pointer which
// can be stored along with the OCF helper.
void* TOleWin::SetUserInfo(void* ui)
{
   void* previous = UserInfo;
   UserInfo = ui;
   return previous;
```

```
}


// Hands a menu handle and menu description to the OCF helper
// object so the latter can handle OLE menu merging on behalf
// of the window.
void TOleWin::SetWinMenu(const TOcMenuDescr& menuDescr)
{
  HMenu = menuDescr.HMenu;
  if (HMenu)
    memcpy(Width, menuDescr.Width, sizeof(Width));
  else
    memset(Width, 0, sizeof(Width));
}


// Merge the container's menu in OLE's menu handle
bool TOleWin::OnOcAppInsMenus(TOcMenuDescr far* sharedMenu)
{
  // If we don't have a menu handle to merge, return false
  if (!HMenu) return false;

  // If we've merged into OLE's menu already, don't bother remerging
  if(MergedMenu) return true;

  // Merge the container menu in OLE's menu
  MergeContainerMenu(*sharedMenu, *this);
  return true;
}


// Handles OC_APPMENUS, a request to set a new menu bar
// We return the handle so the frame may set the menu.
HMENU TOleWin::OnOcAppMenus(TOcMenuDescr far* menuDescr)
{
  // If OLE does not pass a handle, hand our original menu
  if (!menuDescr->HMenu) {
    MergedMenu = 0;
    return HMenu;
  }

  return MergedMenu ? MergedMenu : menuDescr->HMenu;
}


// When asked to restore the UI, we'll return our own menu handle and
// clear out any memory of the merged OLE menu handle.
HMENU TOleWin::OnOcAppRestoreUI()
{
  MergedMenu = 0;
  return HMenu;
}


// Merges source menu popups into the destination menu.
// SrcOwnsEven determines whether the source or destination
// own the even or odd entries within the menu.
static bool MergeMenu(MenuDescr& dst, const MenuDescr& src,
                      bool srcOwnsEven)
{
  int dstOff = 0;
  int srcOff = 0;

  for (int i=0; i<MENU_GROUP_COUNT; i++) {
    if ((!(i%2) && srcOwnsEven) || ((i%2) && !srcOwnsEven)) {
      // If the current location is owned by the source, there should
      // not be anything there from the destination already.
```

```
      if (dst.Width[i]) {
        // However, if the entries are occupied,
        // should they be removed/deleted - or
        // should we just append ours?
      }

      // If source has something to merge
      if (src.Width[i]) {
        // Iterate through the source's menu items
        int insertCount = 0;
        for (int j=0; j<src.Width[i]; j++) {
          uint state = GetMenuState(src, srcOff+j, MF_BYPOSITION);

          // Validate source menu item
          if (state == uint(-1))
            break;

          // Retrieve string
          char str[256];
          GetMenuString(src, srcOff+j, str, sizeof(str),
                        MF_BYPOSITION);

          // If it's a popup, we'll share the handle
          uint idOrPopup = 0;
          if (state & MF_POPUP) {
            state &= (MF_STRING|MF_POPUP);
            HMENU sub = GetSubMenu(src, srcOff+j);
            idOrPopup = UINT(sub);
          } else {
            idOrPopup = GetMenuItemID(src, srcOff+j);
          }

          // Insert destination menu
          if (GetMenuItemCount(dst) <= dstOff+j)
            AppendMenu(dst, state|MF_BYPOSITION, idOrPopup, str);
          else
            InsertMenu(dst, dstOff+j, state|MF_BYPOSITION,
                       idOrPopup, str);

          insertCount++;
        }

        // Update dst. width by # of new entries we've added
        dst.Width[i] += insertCount;
      }
    } else {
      // Here, if src.Width[i] == -1, we
      // could delete the destinations entries
    }

    // Update offset by width of entries just processed.
    srcOff += src.Width[i];
    dstOff += dst.Width[i];
  }

  return true;
}


// Merge so that source (container) owns the even entries
bool MergeContainerMenu(TOcMenuDescr& dst,
                        const TOcMenuDescr& src)
{
  return MergeMenu(*((MenuDescr*)&dst),
                   *((MenuDescr*)&src), true);
}
```

```cpp
// Merge so that source (server) owns the odd entries
bool MergeServerMenu(TOcMenuDescr& dst, const TOcMenuDescr& src)
{
  return MergeMenu(*((MenuDescr*)&dst), *((MenuDescr*)&src), false);
}


// Prompts user for a filename using Common Dialog
bool GetOpenFileName(HWND owner, const char* filter, char* fileName,
                int size, DWORD flags)
{
  OPENFILENAME ofn;
  memset(&ofn, 0, sizeof(ofn));
  ofn.lStructSize   = sizeof(ofn);
  ofn.hwndOwner     = owner;
  ofn.lpstrFilter   = filter;
  ofn.lpstrFile     = fileName;
  ofn.nMaxFile      = size;
  ofn.Flags         = flags;
  return GetOpenFileName(&ofn);
}


// Prompts user for a filename using Common Dialog
bool GetSaveFileName(HWND owner, const char* filter, char* fileName,
                int size, DWORD flags)
{
  OPENFILENAME ofn;
  memset(&ofn, 0, sizeof(ofn));
  ofn.lStructSize   = sizeof(ofn);
  ofn.hwndOwner     = owner;
  ofn.lpstrFilter   = filter;
  ofn.lpstrFile     = fileName;
  ofn.nMaxFile      = size;
  ofn.Flags         = flags;
  return GetSaveFileName(&ofn);
}


// Array of OCF dispatch Ids and string
#define OC_AND_STR(oc)  {oc, #oc}
static struct {
  WPARAM ocId;
  char*  ocStr;
} OcEventStr [] = {
  OC_AND_STR(OC_APPINSMENUS),
  OC_AND_STR(OC_APPMENUS),
  OC_AND_STR(OC_APPPROCESSMSG),
  OC_AND_STR(OC_APPFRAMERECT),
  OC_AND_STR(OC_APPBORDERSPACEREQ),
  OC_AND_STR(OC_APPBORDERSPACESET),
  OC_AND_STR(OC_APPSTATUSTEXT),
  OC_AND_STR(OC_APPRESTOREUI),
  OC_AND_STR(OC_APPDIALOGHELP),
  OC_AND_STR(OC_APPSHUTDOWN),
  OC_AND_STR(OC_VIEWTITLE),
  OC_AND_STR(OC_VIEWSETTITLE),
  OC_AND_STR(OC_VIEWBORDERSPACEREQ),
  OC_AND_STR(OC_VIEWBORDERSPACESET),
  OC_AND_STR(OC_VIEWDROP),
  OC_AND_STR(OC_VIEWDRAG),
  OC_AND_STR(OC_VIEWSCROLL),
  OC_AND_STR(OC_VIEWPARTINVALID),
  OC_AND_STR(OC_VIEWPAINT),
  OC_AND_STR(OC_VIEWLOADPART),
  OC_AND_STR(OC_VIEWSAVEPART),
```

```
   OC_AND_STR(OC_VIEWCLOSE),
   OC_AND_STR(OC_VIEWINSMENUS),
   OC_AND_STR(OC_VIEWSHOWTOOLS),
   OC_AND_STR(OC_VIEWGETPALETTE),
   OC_AND_STR(OC_VIEWCLIPDATA),
   OC_AND_STR(OC_VIEWPARTSIZE),
   OC_AND_STR(OC_VIEWOPENDOC),
   OC_AND_STR(OC_VIEWATTACHWINDOW),
   OC_AND_STR(OC_VIEWSETSCALE),
   OC_AND_STR(OC_VIEWGETSCALE),
   OC_AND_STR(OC_VIEWGETSITERECT),
   OC_AND_STR(OC_VIEWSETSITERECT),
   OC_AND_STR(OC_VIEWGETITEMNAME),
   OC_AND_STR(OC_VIEWSETLINK),
   OC_AND_STR(OC_VIEWBREAKLINK),
   OC_AND_STR(OC_VIEWPARTACTIVATE),
   OC_AND_STR(OC_VIEWPASTEOBJECT)
};


char* GetOcString(WPARAM id)
{
  for(int i=0; i<sizeof(OcEventStr)/sizeof(OcEventStr[0]); i++)
    if (id == OcEventStr[i].ocId)
      return OcEventStr[i].ocStr;

  return "OC_????";
}
```

Figure 25 - The code for the OCF helper class used by an MDI child window.

Class `TOleWin` uses 2 smaller helper classes: `OleClientDC` and `MenuDescr`, shown in figure 26.

```
// Encapsulates a clientDC configured for OLE operations
class OleClientDC {

public:

    OleClientDC(TOleWin&, bool scale = true);
   ~OleClientDC() {ReleaseDC(HWnd, Hdc);}

    // HDC conversion operator for convenience
    operator HDC () const {return Hdc;}

protected:

    HDC              Hdc;
    HWND             HWnd;
};

inline OleClientDC::OleClientDC(TOleWin& win, bool scale)
                   :HWnd(win), Hdc(GetDC(win))
{
  win.SetupDC(*this, scale);
}



// Class encapsulating menu description for menu merging purposes
class MenuDescr : public TOcMenuDescr {

  MenuDescr(const TOcMenuDescr&);

public:
```

```
    MenuDescr(HMENU,
              int fileGrp = 0, int editGrp = 0,
              int contGrp = 0, int objGrp = 0,
              int winGrp  = 0, int hlpGrp = 0);

    // HMENU conversion operator for easy handling
    operator HMENU() const {return HMenu;}
};


inline MenuDescr::MenuDescr(HMENU menu, int fileGrp, int editGrp,
                            int contGrp, int objGrp,
                            int winGrp, int hlpGrp)
{
  HMenu = menu;
  Width[0] = fileGrp;
  Width[1] = editGrp;
  Width[2] = contGrp;
  Width[3] = objGrp;
  Width[4] = winGrp;
  Width[5] = hlpGrp;
}
```

Figure 26 - A couple of minor helper helpers.

We're still not done. The application object, derived from `CWinApp`, also uses a helper object to register the app, to set OLE to use the default `IMalloc` memory allocator and handle a few other details. The application helper is of class `TOleInit` (see figure 27).

```
class TOleInit : public TOleAllocator {

public:

    TOleInit(TRegList& regInfo,
             TComponentFactory callback,
             string& cmdLine, TRegLink* linkHead = 0,
             HINSTANCE hInst = ::_hInstance);
    ~TOleInit();

protected:

    TOcRegistrar*   OcRegistrar;
    TOcApp*&        OcApp;
};


// Initializes OLE to use default IMalloc allocator.
// Creates OCF helper objects
TOleInit::TOleInit(TRegList& regInfo,
                   TComponentFactory callback,
                   string& cmdLine, TRegLink* linkHead,
                   HINSTANCE hInst) :
          TOleAllocator(0), OcRegistrar(0), OcApp(TheOcApp)
{
  // Initialize OCF objects: Create TOcRegistrar & TOcApp
  OcRegistrar = new TOcRegistrar(regInfo, callback, cmdLine,
                                 linkHead, hInst);
  OcRegistrar->CreateOcApp(OcRegistrar->GetOptions(), OcApp);
}


// Cleans up the OCF helper objects
TOleInit::~TOleInit()
{
  // Free the Registrar
  //
```

```
  delete OcRegistrar;
}
```

Figure 27 - The application helper object.

Having seen the code for the helpers, let's go back to where your application creates its main frame window. Having the new `CMDIFrameOCF`-derived class, you will create an instance of it your app's `InitInstance` function, like this:


```
BOOL CMyMdiApp::InitInstance()
{
  // create main MDI Frame window
  CMainFrame* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
  m_pMainWnd = pMainFrame;
  return TRUE;
}
```

The main frame window will in turn instantiate windows of class `CMyChildWin`:

```
void CMainFrame::OnFileNew()
{
  CMyChildWin* pChildWin = new CMyChildWin;
  pChildWin->Create("An OLE container window",
                    WS_CHILD | WS_VISIBLE |
                    WS_OVERLAPPEDWINDOW |
                    WS_CLIPSIBLINGS,
                    rectDefault, this);
}
```

The application class will need a couple of OCF helpers to finish the job, to support OLE registration and OLE memory allocator initialization. For OLE registration you must set up a table with basic information that OLE will store in the system registry. Using OCF you set this list up using some pre-defined macros, like this:

```
// OLE Registration information
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{2C27A720-D323-101B-96B9-7A117CB6AF38}")
  REGDATA(progid,      "MFCMDI.ProgID.1")
  REGDATA(description, "MFC Sample using OCF")
END_REGISTRATION
```

Then you must setup a couple of OCF variables to manage the registration information and the app's OCF helper. Your app will need to instantiate a helper of type `TOleInit`, deleting it in the destructor (see figure 28).

```
// OCF variables
TRegLink* RegLinkHead = 0;
TOleInit* OleInit     = 0;


BEGIN_MESSAGE_MAP(CMdiApp, CWinApp)
  //{{AFX_MSG_MAP(CMdiApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
CMdiApp::CMdiApp()
{
  try {
    OleInit = new TOleInit(::AppReg,
                           /*Factory Callback*/0,
                           string(m_lpCmdLine),
                           ::RegLinkHead,
                           m_hInstance);
  }
  catch (TXBase& xbase) {
    MessageBox(GetFocus(), "OCF helper initialization error",
               "OLE error", MB_OK);
  }
}

CMdiApp::~CMdiApp()
{
  delete OleInit;
}
```

Figure 28 - The OLE registration information and OCF helpers in an MFC application.

And there you have it: MFC with OCF, in a nutshell (some nutshell!). Don't get the idea that Borland created OCF with the purpose of seducing MFC programmers. The point is that OCF is uncoupled from OWL to the extent that it *can* be used without OWL. Even though it is non-trivial to use OCF without OWL, there still may be reasons to use OCF in a Visual C++ app. Remember that to get OLE support with Visual C++ you *must* develop code using MFC. Assume you are writing a DLL that needs to be very small and also support OLE. You don't want to create a full-blown MFC program, so OCF may be the way to go. Borland has tons of sample programs that show how to use OCF both with OWL and without OWL, to ease some of the pain.