December 1999

MICROSOFT SYSTEMS JOURNAL

More Reusable MFC Goodies: Simplify Your (Programming) Life with the COMToys Library

Paul DiLascia

Reusable classes are like little COM tinkertoys you can assemble in different ways in different situations to create more elaborate objects. But how would you go about creating them? I used MyBands and the BandObj framework as a test bed to develop a library called COMToys.

This article assumes you're familiar with C++, COM, Internet Explorer

Code for this article: ComToys.exe (81KB)

Paul DiLascia is the author of Windows ++: Writing Reusable Code in C++ *(Addison-Wesley, 1992) and a freelance consultant and writer-at-large. He can be reached at askpd@pobox.com or http://pobox.com/~askpd.*

Last month (November 1999) I showed you a program called MyBands that enables you to put your own edit control in the Windows® task bar. MyBands implements three kinds of Windows band objects, among them the Web Search band shown in **Figure 1**. To implement MyBands, I wrote a mini band object framework, BandObj, with classes called CBandObjDll, CBandObjFactory, and CBandObj that provide all the support you need for writing any kind of band object with little effort.

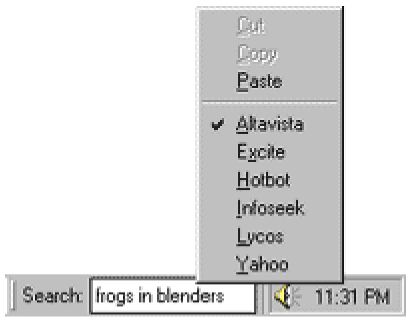

Figure 1 The Web Search Band

CBandObj has lots of COM code to implement various interfaces like IDeskBand, IOleWindow, IContextMenu, and so on, but the only one that has anything to do with band objects is IDeskBand. The others are wholly generic. The CBandObj implementation for IContextMenu doesn't use anything from band objects; all it needs is a menu. All IOleWindow needs is a window handle. And so on for the others. It should be possible to extract these interface implementations and encapsulate them in reusable classes—little COM tinkertoys you can assemble in different ways in different situations to create more elaborate objects. But how exactly would you go about it?

This month, I'll show you how I used MyBands and the BandObj framework as a test bed to develop a library called COMToys. COMToys isn't an all-embracing system like the ActiveX® Template Library (ATL) or MFC; it's more of an ad hoc collection of macros, functions, classes, and whatever else I could think of to make writing BandObj easy. But COMToys offers an approach to programming COM in C++ that works for any kind of COM object, whether or not you're using MFC. More than anything else, COMToys is an attitude—one that says programming COM in C++ doesn't have to be hard. Really!

COM: A C++ Programming Predicament

When it comes to COM, you have to pity the poor C++ programmer using the Java language and Visual Basic®. Programmers smile smugly as they write "form.color = red", while C++ hackers fret their instructions with QueryInterface and get_foo and set_mumble and—don't forget to check the HRESULT! Lord help you if you don't have a Release for every AddRef. In Visual Basic, you don't even have to type semicolons. C++ is far more powerful and leaner than Visual Basic and the Java language, but where COM is concerned, it seems hopelessly complex. How ironic when COM was designed for C++! After all, a COM object is just a C++ vtbl.

The problem isn't C++ and it isn't COM. The problem is that most C++ programmers don't know any better.

They cut and paste from the SDK samples—long lines of open code that are intended only to demonstrate the rawest, most bare-bones way of doing something, without any thought toward system design. Folks, that stuff isn't production code! Why do you think they call it samples? C++ lets you program down to the metal—that's its advantage. But no one who isn't a masochist programs that way on a daily basis. If you want easy, you can have easy. You just have to take a little time up front to build some tools that'll repay you tenfold later. Visual Basic and the Java language are easy because have considerable infrastructure built in. But you can write your own infrastructure that makes C++ just as easy.

The MyBands program from last month has a two-layer architecture, with MyBands (the app) built on BandObj (the framework). This two-layer architecture is an artificial system I created for educational purposes. In reality, I built BandObj using the three-layered architecture shown in **Figure 2**, with COMToys providing the base. Only after the fact did I remove COMToys from the picture so that I could focus on band objects. Now it's time to reveal the whole system.

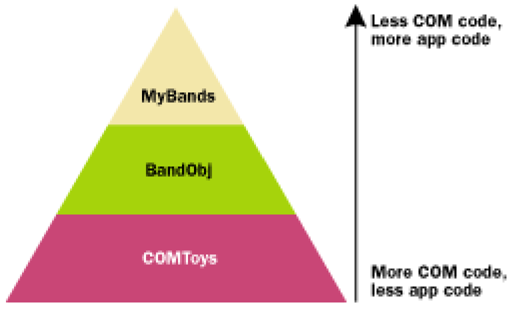

Figure 2 MyBands Architecture

As I was planning MyBands/BandObj/COMToys, I had to make a decision that confronts many programmers today: what programming system should I use to do COM? ATL? MFC? COM+ (whatever that is)? Or none of the above? I usually prefer to roll my own, but I have nothing against using someone else's wheel if it works and it's easy. So I considered both ATL and MFC.

ATL is lean and very clever, but who can read all those T's and angle brackets? Do you really know how much code ATL is generating? If the code is so generic that it fits in a template, why not use a simple class or subroutine instead? True, templates let you parameterize everything up the wazoo, which is cool. But all I want to do is drive to the grocery; I don't need or want a space shuttle for that. More important, ATL lacks the GUI support I want for shell extensions like band objects. On the up side, ATL's leanness, multiple inheritance model, and smart pointers are a big win, and you saw last month why the ATL registrar is the greatest thing since chocolate-covered espresso beans.

What about MFC? MFC has the GUI stuff in spades. Even if you don't need the doc/view architecture, there's all the command routing and ON_COMMAND_UPDATE_ UI handlers—essential stuff for any object with a UI. But when it comes to COM, MFC has its own infelicities. Most people worry about the elephantine DLL, but in my opinion that's a red herring. Yes, mfc42.dll is a sure candidate for the fat farm, but have you ever seen a Windows installation without it? For all practical purposes, mfc42.dll is part of the OS. (In fact, I just looked in my Windows 98 CAB and mfc42.dll is in win98_ 62.cab, so it is part of Windows—so what's to download?) When it comes to COM, MFC has an even bigger problem: its use of nested classes is downright demented. More on this shortly.

So which to use? In the end, I built my own system that uses both MFC and ATL. It saves me work, and dumps the parts of MFC and ATL that get in the way. COMToys uses the MFC class factories and IUnknown, but dumps nested classes. COMToys uses the ATL smart pointers, registrar, and multiple inheritance, but dumps the ponderous templates and confusing object model. This hybrid approach is not only practical, it's fun! (Plus, you always understand the code better when you write it yourself.)

I make no claim that COMToys is better than any other system; I simply offer it as something to think about. Mainly, I want to show that you don't have to use MFC, ATL, or any other system entirely. And how, by writing just a little infrastructure, you can make programming COM in C++ as easy as pie—even easier than Visual Basic!

Gnarly Nested Classes

Enough with the punditry; let's look at some code. To explain how COMToys works, the first thing I have to do is show you one of the problems it is designed to overcome: the MFC use of nested classes. So let's take a quick MFC/COM 101 review.

To write a COM class in MFC, you derive your class from CCmdTarget and use macros to implement your interfaces. For example, if your class implements IPersistFile, you'd write:

```
// in .h file
class MyComClass : public CCmdTarget {
  BEGIN_INTERFACE_PART(PersistFile, IPersistFile)
  STDMETHODIMP GetClassID(LPCLSID pClsID);
  STDMETHODIMP IsDirty(void);
.
.
.
  END_INTERFACE_PART(PersistFile)
  };
```

These macros declare a nested class called MyComClass::XPersistFile inside your main class and an instance called m_xPersistFile. MyComClass::XPersistFile implements the IPersistFile methods, including the ones inherited from IUnknown.

To make the interface known to MFC, you must create an interface map:

```
// in .cpp file
BEGIN_INTERFACE_MAP(CMyComClass, CCmdTarget)
  INTERFACE_PART(CMyComClass, IID_IPersistFile, PersistFile)
.
.
.
END_INTERFACE_MAP()
```

The macros generate a table with one entry for each COM interface your class supports. Each table entry stores the IID of the interface and the offset within the main class of the nested class that implements it.

```
{ &IID_IPersistFile, offsetof(CMyComClass, m_xPersistFile) }
```

Once you've declared and implemented the interface map, you have to implement the methods themselves, including AddRef, Release, and QueryInterface. Because the classes are nested, you have to write IUnknown for every interface your COM object supports, even though the implementations are identical. For example:

```
// ditto for AdRef and Release
STDMETHODIMP
CMyComClass::XPersistFile::QueryInterface(...)
{
  METHOD_PROLOGUE(CMyComClass, PersistFile)
  return pThis->ExternalQueryInterface(...);
}
```

METHOD_PROLOGUE is required for MFC to get its bearings (AFX_MANAGE_STATE is required at the start of any entry point to your DLL) and to set up pThis, which points to the parent class CMyComClass ("this" minus the nested offset). CCmdTarget::ExternalQueryInterface searches your interface map for an entry with an IID that matches the interface requested. When it finds one, it adds the offset to its this pointer and returns the result—which, if everything comes out OK, points to the nested object that implements the interface. It works, but it's highly cumbersome for several reasons.

First, implementing IUnknown for every interface your class supports is not only a pain in the carpal tunnel, it opens the potential for mistakes and contributes to code bloat by duplicating code needlessly.

Second, you can't access your class members directly from your interface methods; you have to access them through pThis—Hokey at best; grody at worst. Conceptually, the interface methods belong to the outer class, so why shouldn't they be able to access its members as first-class citizens?

Finally, and most annoying of all, the nested class approach doesn't let you override the interface methods in derived classes. Say you derive a new class, CMyComClass2, and all you want to do is override IPersistFile::SaveCompleted to set a flag, which one of your ON_UPDATE_COMMAND_UI handlers will examine to display "Saving..." until the save is complete. You can't do it. CMyComClass has no SaveCompleted function to override. The class that implements SaveCompleted, CMyComClass::XPersistFile, is nested inside; there's no way to override its methods. You have to reimplement the entire IPersistFile interface in your derived class, creating another nested class with methods that do nothing but call the base class, CMyComClass::m_xPersistFile, except for the one SaveCompleted override.

I ran into this problem last month with CBandObj. To let CBandObj-derived classes do something when the container sets the site, I had to provide a new virtual function, CBandObj::OnSetSite, and call it from CBandObj::XDeskBand::SetSite. And what about the other interface methods? Should I introduce OnXxx for every interface method CBandObj implements? I don't think so!

For these reasons, many C++ programmers—ATL coders included—use multiple inheritance to do COM. With multiple inheritance, you derive your COM class multiply from every interface it implements.

```
class CMyComClass :
  public IPersistFile,
  public IContextMenu, ...
{
  // IUnknown
  STDMETHOD_(ULONG, AddRef)();
•
•
•
// IPersistFile
  STDMETHODIMP GetClassID(LPCLSID pClsID);
•
•
•
// IContextMenu
  STDMETHOD (QueryContextMenu)(...);
•
•
•
};
```

All the methods belong to the main class so you can override them the normal way, and all the methods can access the class members directly; there's no need for pThis. What's more, through the magic of C++, you only have to implement AddRef, Release, and QueryInterface once, and the same implementation applies to all instances of IUnknown. This happens because of a C++ rule that says, "a pure virtual function is always redefined by any subclass that implements it." All the vtbls will have IUnknown slots that point to the same actual functions. The same is true for other interfaces that might be multiply inherited; for example, IPersist that's inherited from IPersistFile and IPersistStream. **Figure 3** illustrates this, which I'll call the Magic MI Rule.

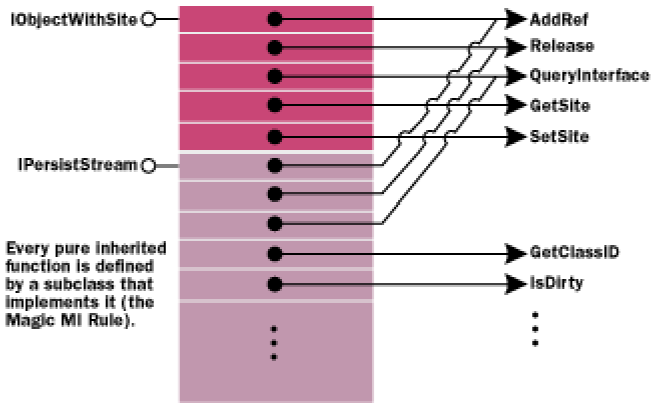
IObjectWithSite
IPersistStream
Every pure inherited function is defined by a subclass that implements it (the Magic MI Rule).
AddRef
Release
QueryInterface
GetSite
SetSite
GetClassID
IsDirty

Figure 3 Multiple Inheritance

If multiple inheritance is so great, why doesn't MFC use it? Because in the case of concrete classes—classes with real functions and data—multiple inheritance creates ambiguities. If you write x = m_foo, which m_foo do you mean, the one inherited from A or the one from B? And since MFC derives all its classes from CObject, using multiple inheritance would lead to the dreaded diamond hierarchy. You can use virtual base classes to overcome it, but things get messy real fast. So the founding fathers of MFC wisely decided to steer clear of multiple inheritance.

Meet COMToys

Well, just because MFC is burdened by its tree-shaped object model doesn't mean COMToys has to be, right? In fact, COMToys uses multiple inheritance to do COM. To see how it works, let's look at CBandObj, the new version built with COMToys. To the outside world—that is, to apps like MyBands—CBandObj is exactly the same as before. Internally, things are different. First off, CBandObj is multiply derived from CWnd and IDeskBand.

```
// in BandObj.h
class CBandObj :
  public CWnd, public IDeskBand
{
  DECLARE_IUnknown();
  // IDeskBand
```

```
    HRESULT GetBandInfo(...);
  .
  .
  .
  };
```

A band object is a window, so CBandObj is derived from CWnd. But it also implements IDeskBand, so it's derived from that too. DECLARE_IUnknown is a COMToys macro that declares AddRef, Release, and QueryInterface in one fell swoop. The macro reduces typing errors and trips to the orthopedist. Of course, if you declare an interface, you have to implement it, and COMToys has a macro for that, too.

```
  // in BandObj.cpp
  IMPLEMENT_IUnknownCT(CBandObj);
```

That's it. One line. CT stands for CCmdTarget, not COMToys; you should read the line as "implement IUnknown for a CCmdTarget." (Remember, CBandObj inherits CCmdTarget from CWnd.) IMPLEMENT_IUnknownCT generates stock implementations for AddRef, Release, and QueryInterface that call CCmdTarget:

```
  ULONG CBandObj::AddRef()
  {
    CMDTARGENTRYTR("CBandObj(%p)::AddRef,
      count=%d\n", this, m_dwRef+1);
    return ExternalAddRef();
  }
```

ExternalAddRef, ExternalRelease, and ExternalQueryInterface are the standard MFC implementation for IUnknown. They call internal versions that do the work—unless you're using aggregation, in which case they call the outer IUnknown. Don't get hung up on the internal/external thing; it's just an MFC implementation detail. The main point is that COMToys uses CCmdTarget to implement IUnknown. Hey—if it works, why fix it? CMDTARGENTRYTR is a COMToys macro that generates the standard entry for a CCmdTarget-derived class. It's the COMToys equivalent of METHOD_PROLOGUE. It initializes the state of MFC and generates a TRACE diagnostic.

```
  AFX_MANAGE_STATE(m_pModuleState);
  CTTRACEFN(...);
```

There's also CMDTARGENTRY without TRACE, and for non-CCmdTarget entries (such as externs like DllGetClassObject) COMToys has MFCENTRY and MFCENTRYTR, which initialize the state from AfxGetStaticModuleState instead of m_pModuleState. **Figure 4** shows the full macro expansion for

IMPLEMENT_IUnknownCT.

So far life is looking up. CBandObj has an implementation for IUnknown that requires typing one line. But there's a problem here. When the container calls QueryInterface asking for IID_IDeskBand, CBandObj passes the request to CCmdTarget—but how does MFC know what interface pointer (vtbl) to return? As I described in the preceding section, MFC looks for interface maps containing the offsets of nested classes. CBandObj doesn't have an interface map—COMToys doesn't do interface maps. So how can CCmdTarget find the interface when someone calls with a specific IID?

The missing piece of the puzzle is a CCmdTarget virtual function called GetInterfaceHook. The designers of MFC, due to their keen foresight, added an escape hatch. Before doing its interface map thing, CCmdTarget calls GetInterfaceHook.

```
// pseudo-code
HRESULT
CCmdTarget::InternalQueryInterface
  (REFIID iid, void** ppv)
{
  *ppv = GetInterfaceHook(iid);
  if (*ppv)
    return S_OK;
// search interface maps
.
.
.
}
```

In other words, to bypass the whole interface map thing, just implement GetInterfaceHook as CBandObj does.

```
LPUNKNOWN CBandObj::GetInterfaceHook(void* piid)
{
  REFIID iid = *((IID*)piid);
  if (iid==IID_IUnknown)
    return (IDeskBand*)this;
  if (iid==IID_IDeskBand)
    return (IDeskBand*)this;
  return NULL;
}
```

Whatever the interface requested, CBandObj returns the appropriate cast in standard multiple inheritance fashion. For IUnknown, you can't cast to IUnknown* because that would be ambiguous; which IUnknown do you mean, the one in IDeskBand, or the one in one of the other interfaces I haven't show you yet? So you have to cast to some singly inherited class that inherits IUnknown. It doesn't matter which one, since they all point to the same functions, as shown in **Figure 3**.

I thought of adding my own version of interface maps based on multiple inheritance, to use instead of the old-fashioned if statements—but I can't see the benefit. How hard is it to add a couple of lines for each new interface? Sometimes old-fashioned is simpler and less confusing.

Let me summarize what I've shown you so far: the way you implement a basic COM object with COMToys is derive from CCmdTarget or a CCmdTarget-derived class like CWnd; declare IUnknown with DECLARE_IUnknown; implement it with IMPLEMENT_IUnknownCT; and write GetInterfaceHook to return the proper interface pointers. Oh, and of course, you have to implement the methods—in this case IDeskBand::GetBandInfo.

```
HRESULT CBandObj::GetBandInfo(...)
{
  // do it
}
```

Mixing in the Implementations

So far, all I've done is show you a way to use multiple inheritance instead of nested classes to implement COM objects in MFC. The basic idea is to ignore the MFC macros and interface maps and use GetInterfaceHook instead to return the interface pointer. This by itself certainly makes programming easier, but where's the reusability? The main reason for doing COMToys is to have reusable classes that implement common COM interfaces. For that, COMToys uses mix-in classes.

A mix-in class is designed to be mixed in with other classes using multiple inheritance. Usually the mix-in class is orthogonal to the main class, meaning it offers independent functionality and comes from an unrelated class hierarchy to avoid diamond-shaped derivation trees (pardon my oxymoron). Not surprisingly, CBandObj is actually a bit more complicated than I presented in the previous section. **Figure 5** shows the true picture. Yikes! Look at all those classes—it looks like ATL! Don't have a conniption. Let's examine it slowly.

```
class CBandObj : public CWnd,
  // interfaces
//public IOleWindow,     // inherited
//public IDockingWindow, // inherited
  public IDeskBand,...

  // implementations
  public CTOleWindow,
  public CTDockingWindow,...
{
  DECLARE_IUnknown();
  DECLARE_IOleWindow();
  DECLARE_IDockingWindow();
  // IDeskBand
  STDMETHOD(GetBandInfo)
    (DWORD, DWORD, DESKBANDINFO*);
.
.
.
}
```

CBandObj inherits IOleWindow and IDockingWindow from IDeskBand, so it must implement them. As with IUnknown, COMToys has macros to declare the methods. DECLARE_IOleWindow and DECLARE_IDockingWindow declare only the most-derived methods, not ones that are inherited (you'll see why in a minute), so you need both. There's no DECLARE_IDeskBand because COMToys doesn't have a class to implement it; IDeskBand is implemented by CBandObj.

As with IUnknown, COMToys has more macros to implement the interfaces.

```
// in BandObj.cpp
IMPLEMENT_IUnknownCT(CBandObj); // as before
IMPLEMENT_IOleWindow(CBandObj, CTOleWindow);
IMPLEMENT_IDockingWindow(CBandObj, CTDockingWindow);
```

The new macros implement IOleWindow using CTOleWindow, and implement IDockingWindow using CTDockingWindow. The IMPLEMENT_IWhatsIt macros link an interface definition (abstract class) with an interface implementation (concrete class) by generating boilerplate methods that call the implementation class.

```
// generated by IMPLEMENT_IOleWindow
HRESULT CBandObj::GetWindow(HWND* pHwnd)
{
  CMDTARGENTRYTR(
    _T("CBandObj(%p)::IOleWindow::GetWindow\n"),
    this);
  return CTOleWindow::GetWindow(pHwnd);
}
```

Once you've declared and implemented the interfaces, all that remains is to make them known to MFC by adding more lines to GetInterfaceHook, as before.

```
// in CBandObj::GetInterfaceHook
if (iid==IID_IOleWindow)
  return (IOleWindow*)this;
if (iid==IID_IDockingWindow)
  return (IDockingWindow*)this;
```

It's totally straightforward. If you understand how this works for IOleWindow, **Figure 5** is just more of the same. For each IWhatsIt interface your COM class implements, you derive from two classes: the COM interface itself (IWhatsIt) and the CT class that implements it (CTWhatsIt). The implementation class can be one that comes with COMToys or your own. Either way, interface and implementation are separate. This feels "right" because that's the foundational principle of COM.

So assuming you have a CTWhatsIt that implements some interface, to use it in your COM class you must do four things: derive from IWhatsIt and CTWhatsIt; declare the methods with DECLARE_IWhatsIt; implement with IMPLEMENT_IWhatsIt(CMyComClass, CTWhatsIt); and add a couple of lines in CMyComClass::GetInterfaceHook. That's about five lines of code. It's that easy.

Implementing the Implementation Classes

Of course, everything assumes you have a CTWhatsIt to implement the interface! But at least I've shown you how to localize the interface implementation to one C++ class—in other words, how to reuse interface implementations. To provide a little content with its form, COMToys comes with a bunch of interface implementations already built in. CTPersist, CTPersistStream, CTOleWindow, CTDockingWindow, CTInputObject and others. But how do they work?

The first surprising thing about the implementation classes is they're not derived from their corresponding interfaces! They're merely classes that have methods with the same names and signatures.

```
// not derived from IOleWindow!
class CTOleWindow {
public:
  // same macro as main class
  DECLARE_IOleWindow();
};
```

There's no real connection between the interfaces and implementations; the connection is purely lexical. That's another reason for using the DECLARE_IFoo macros: to make sure you get the names and signatures right. A typo in CTOleWindow wouldn't generate an error, it'd simply define a new function.

Another thing to notice about the mix-in classes is they each implement only the methods in the most-derived interface, not methods from inherited interfaces. IDockingWindow implements ShowDW, CloseDW and ResizeBorderDW—but not GetWindow or ContextSensitiveHelp, which it inherits from IOleWindow. So if you want to implement IDockingWindow, you have to mix in both CTDockingWindow and CTOleWindow. Likewise, CTPersistStream doesn't implement GetClassID, which it inherits from IPersist.

Why did I do it this way? For one thing, it lets you mix and match different implementation classes. For another, consider the alternative. Suppose, for example, I had derived CTPersistFile and CTPersistStream from CTPersist, or given them both a GetClassID function (as I originally tried to do). Now any class that uses both CTPersistFile and CTPersistStream has two copies of GetClassID, which creates a multiple inheritance ambiguity. The Magic MI Rule of **Figure 3**, whereby all multiply inherited interfaces get the same concrete function, works only for pure virtual functions, not concrete ones. It's conceivable you'd want IPersistFile and IPersistStream to have different implementations for GetClassID, but 99 percent of the time you don't. By implementing only the most-derived methods in CTPersistFile and CTPersistStream, COMToys lets you mix in the one-and-only implementation you want for CTPersist. When you write

```
IMPLEMENT_IPersist(CMyComClass, CTPersist);
```

the GetClassID method generated fills in the GetClassID for both IPersistStream and IPersistFile.

As for the methods themselves, that's where I finally had to write some code, as opposed to performing semantic gymnastics. Lucky for me, it was mostly straightforward. Just take all the implementations from the nested classes in last month's version of BandObj and move them into their respective CT*Xxx* classes, converting any needed variables into data members initialized from constructor arguments. For example, here's CTPersist:

```
class CTPersist {
public:
  const CLSID& m_clsid;
  CTPersist(const CLSID& clsid) : m_clsid(clsid) { }
  STDMETHODIMP GetClassID(LPCLSID pClassID) {
    return pClassID ?
      (*pClassID=m_clsid, S_OK) : E_UNEXPECTED;
  }
};
```

CTPersist holds a reference to the class ID, not the class ID itself. That's a general principle in COMToys: implementation classes don't store real data, only pointers or references to outside objects. For CTPersist, it doesn't much matter since it would be pathological ever to change an object's class ID while it's running. But in general, data can change, so it's wise to let the parent class own the data so it can manipulate the data freely. Both CTPersistFile and CTPersistStream have a modified flag called m_bModified, but it's a reference to a BOOL, not a BOOL. If the main class changes the real flag, the implementation classes automatically get the change without you having to call some SetModified function or other. As a general programming principle, it's always better to use demand-pull rather than supply-push to operate your state, and there should be only one copy of each state variable throughout the entire system.

In a similar fashion, CTMfcContextMenu holds a reference to a CMenu, which the main class must supply at construction.

```
// in COMToys.h
class CTMfcContextMenu {
public:
  CCmdTarget* m_pCmdTarget;
  CMenu& m_ctxMenu;
  CTMfcContextMenu(CCmdTarget* pTarg, CMenu& menu)
    : m_pCmdTarget(pTarg), m_ctxMenu(menu) { }
.
.
.
};
// In BandObj.cppCBandObj::CBandObj(REFCLSID clsid) :
  CTMfcContextMenu(this, m_menu), ...
{
.
.
.
}
```

Because CTMfcContextMenu::m_ctxMenu is a reference, CBandObj can change the menu any way it likes without having to explicitly notify CTMfcContextMenu. CBandObj depends on this feature because it generates its context menu (see **Figure 6**) on the fly when the user right-clicks the band. As for the implementation itself, CTMfcContextMenu has all the goodies from last month.

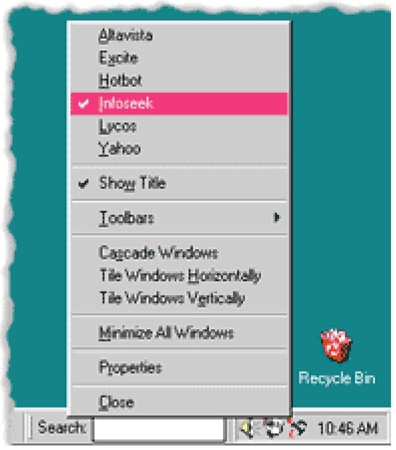

Figure 6 CBandObj Context Menu

When the container calls IContextMenu::QueryContextMenu to get the menu items, CTMfcContextMenu fills the menu with your CMenu items as you'd expect, but not before it creates a CCmdUI object and routes it through your CCmdTarget so all your ON_UPDATE_COMMAND_UI handlers can tweak the menu. Likewise, when the container calls InvokeCommand, CTMfcContextMenu sends the command to your command target's ON_COMMAND handlers. CTMfcContextMenu even does prompt strings by looking for string resources with IDs that match your menu items. In short, CTMfcContextMenu converts COM-speak to MFC-speak.

All you have to do is give it a CMenu and a command target (usually the COM class itself) and forget about it. All the command stuff happens as it would in a vanilla MFC app. You never need to implement IContextMenu again—just use COMToys.

A Problem in Paradise

When I first implemented COMToys, I ran into a minor snag. What happens if two COM interfaces have methods with the same name and signature? It's rare, but it does happen. For example, IPersistStream and IPersistFile both have IsDirty. So if you write

```
  DECLARE_IPersistFile();
  DECLARE_IPersistStream();
```

you'll end up with IsDirty declared twice. This isn't a show-stopper since you can always type the declarations manually, but the problem is somewhat more disconcerting for the implementations. To do them manually, you'd have to retype a bunch of code from IMPLEMENT_IPersistStream, which is pretty yucky. To avoid that, I introduced function-level macros.

```
// IPersistFile — whole interface
IMPLEMENT_IPersistFile(CMyClass,CTPersistFile);

// IPersistStream — each fn separate
// IMPLEMENT_IPersistStream_IsDirty();
IMPLEMENT_IPersistStream_Load(CMyClass,CTPersistStream);
IMPLEMENT_IPersistStream_Save(CMyClass,CTPersistStream);
IMPLEMENT_IPersistStream_GetSizeMax(CMyClass,CTPersistStream);
```

Note how once again, by virtue of the Magic MI Rule, there's only one IsDirty, which applies to both IPersistFile and IPersistStream. In most cases, this is what you want because the object is either dirty or it isn't—right? If for reasons best known to yourself you want two interfaces with the same function to have different implementations, you can always do it by fully qualifying the method name using normal C++ syntax: CMyComClass::IPersistStream::IsDirty and CMyComClass::IPersistFile::IsDirty. But at least with the function-level macros you don't have to retype—or even know—the implementations. The function-level macros also provide a way to override the standard implementation for one or some of the interface methods—though that's better achieved by deriving your own implementation class, CTMyPersistStream or whatever.

Class Factories and Registration

One important detail I've omitted from the discussion so far is object creation. That's because it's relatively straightforward. Last month I discussed class factories and registration fairly extensively. I showed you how BandObj has DLL entries and a class factory that uses the ATL Registrar to register COM objects from resource scripts. For COMToys, I simply moved the code from BandObj to COMToys.

COMToys handles object creation, registration, and DLL entries using the classes CTModule, CTFactory, and a special file, DllEntry.cpp. CTModule is a typical "module" class, like the MFC CWinApp or ATL CComModule. It has the virtual functions OnGetClassObject, OnDllRegisterServer, and so on, which you can override. You must also #include DllEntry.cpp, which implements the entries.

```
extern "C"
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid,
                         LPVOID* ppv)
{
  MFCENTRY;
  return CTModule::GetModule()->
    OnGetClassObject(rclsid, riid, ppv);
}
```

Ditto for DllRegisterServer, DllCanUnloadNow, and the others. CTModule::GetModule is like AfxGetApp. It returns a pointer to the one-and-only global CTModule object. So all DllEntry.cpp does is convert hard-wired extern functions into soft-wired virtual ones, which are a little easier to override.

For MFC apps, COMToys has CTMfcModule, which is multiply derived from CTModule and COleControlModule, the MFC app class for COM. CTMfcModule has implementations for OnGetClassObject and so on that call the corresponding MFC functions to do the work. The reason for separating things this way is to put as much of the goodies as possible in code that doesn't depend on MFC. More to follow. **Figure 7** shows the relationships between the MFC and non-MFC classes.

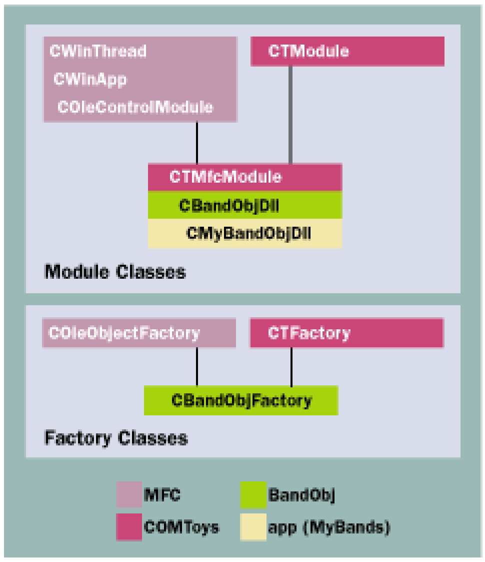

Figure 7 Class Relations

To do factories, COMToys has CTFactory. Mimicking MFC, CTModule::OnRegisterServer calls a static function, CTFactory::OnRegisterAll, which calls CTFactory::OnRegister for each factory. When you create a CTFactory, one of the parameters you must specify is a resource ID. By default, CTFactory::OnRegister looks for a REGISTRY resource (RGS script) with the same ID and invokes the ATL registrar to run it. But first it calls another virtual function, OnInitRegistryVariables, to initialize common registrar variables like %CLSID% and %ClassName%. **Figure 8** shows the full list; you can override OnInitRegistryVariables to add your own. All this is the same as it was last month, only now the code is abstracted out of band objects and into COMToys. The upshot is, to handle registration/unregistration, all you have to do is write a registry script and stick it in your resource file with the same ID as your COM class.

```
IDR_MYCOMCLASS REGISTRY DISCARDABLE "MyComClass.rgs"
```

I should point out that CTFactory is brain-damaged in one respect: it doesn't implement IClassFactory as it should. That's because I haven't gotten around to it yet. Since all the COM classes I've built so far use MFC, I've been using the MFC COleObjectFactory, which works fine. Thus, CBandObjFactory derives from COleObjectFactory and CTFactory. If I ever need a non-MFC class factory, I can implement IClassFactory for CTFactory. For now, CTFactory and COleObjectFactory exist side-by-side, with CTFactory providing all the easy-registration stuff and COleObjectFactory providing IClassFactory and the linkup with DllGetClassObject.

When a process like Windows Explorer wants to create an instance of your COM object—say it's a band object in MyBands.dll—it calls CoCreateInstance, which reads the registry to see which DLL to load and loads it, then calls DllGetClassObject (in DllEntry.cpp), which calls CTModule::OnGetClassObject. Since BandObj is an MFC-based COM object, it uses CTMfcModule, whose OnGetClassObject function calls AfxDllGetClassObject. MFC searches the list of factories and voilà! But wait a minute, where did the factory get created?

Normally, to create a class factory in MFC you use DECLARE_OLECREATE and IMPLEMENT_OLECREATE. These macros declare and instantiate a static COleObjectFactory object called CMyComClass:: factory. But as I described last month, BandObj doesn't use DECLARE/IMPLEMENT_OLECREATE, nor does COMToys. To create your factory, you must call new.

```
// in InitInstance
new CTBandObjFactory(MYGUID,
  RUNTIME_CLASS(CMyComClass),
  IDR_MAINFRAME);
```

When you create a new CTFactory, COMToys adds it to a master list; MFC does the same for COleObjectFactory. In other words, as I explained last month, you don't have to do anything special to make your factory known to MFC or COMToys. Just creating it is enough. You don't have to delete your factory, either; CTModule does it in ExitInstance. If you'd rather create your factory the old way, as a static global, you can do that too—but you have to set m_bAutoDel = FALSE in your factory's constructor. The main reason for dumping DECLARE/IMPLEMENT_OLECREATE is so I can derive my own factories when I want. In the case of BandObj, of course, you don't have to create a factory; you still call AddBandClass, as described in last month's installment.

So the bottom line is this. To handle object creation, all you have to do is derive your factory class from CTFactory and COleObjectFactory and remember to #include DllEntry.cpp somewhere. COMToys does the rest. To make your COM object self-registering, you have to write a resource script (RGS file) that registers your class, and add it as a REGISTRY resource with the same ID as your factory. For more details, download the source from the link at the top of this article.

But Wait, There's More!

That's the basic core of COMToys. Here's a list of other traits and features.

Smart Pointers I can't praise ATL smart pointers enough. If you aren't using some form of smart pointers, it's time to get out of the Stone Age. I mean it; stop with all the AddRef and Release already. Smart pointers ensure that all the reference counting drudgework happens correctly, regardless of how many exit paths your function has. They save you hours tracking down hard-to-find ref count bugs. **Figure 9** shows what'll happen to you if you don't use smart pointers. For those who wince at the sight of templates, COMToys has a macro to hide the angle brackets.

```
#define DECLARE_SMARTPTR(ifacename) \
typedef CComQIPtr<ifacename> SP##ifacename;
```

Now I can write

```
DECLARE_SMARTPTR(IPersist);
```

to define a new type SPIPersist, which I can use anywhere for a pointer to IPersist. SPIUnknown is declared specially since it requires CComPtr, not CComQIPtr. (There's no need to QueryInterface for IUnknown—every interface has it.) **Figure 10** shows some typical COM code written with and without smart pointers. Need I say more?

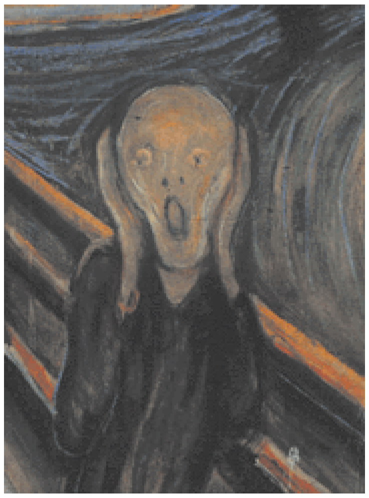

Figure 9 Aaaaah!

For an in-depth description of ATL smart pointers, see Don Box's "The Active Template Library Makes Building Compact COM Objects a Joy" (*MSJ*, June 1997). Don's article also describes the rare situations where smart pointers can cause trouble.

Multithreading COMToys was not written with every kind of COM object in mind, only shell extensions. It assumes single-threaded apartments (STA), where class members are automatically thread-safe and only globals need be protected. COMToys classes that have static members (globals) also have a critical section, g_mydata, that you should lock before accessing. COMToys has a little class called CTLockData to make it easy. The constructor locks the critical section and the destructor unlocks it, so all you have to do is write this:

```
{ // protected block of code
  CTLockData lock(g_mydata);
// do your worst
.
.
.
}
```

As with smart pointers, the benefit of CTLockData is you never have to remember to unlock, even if your function or code block has many exit paths. C++ makes sure the destructor will be called and the lock unlocked when control leaves scope. CTFactory uses CTLockData any time it needs to access its global factory list.

Debugging Diagnostics The best way to understand what goes on with any COM object is to generate diagnostics whenever an interface method is called. Without such diagnostics, I could never have figured out how band objects work. (See Part I in the November 1999 issue of *MSJ* for details.) COMToys has tracing built in. The IMPLEMENT_ IWhatsIt macros generate a TRACE statement at the top of every function, so if you turn tracing on (COMToys::bTRACE = TRUE), you can see what your object is doing. COMToys uses the CTraceFn class from my January 1997 *MSJ* article, "More Fun with MFC: DIBs, Pallettes, Subclassing, and a Gamut of Reusable Goodies."

COMToys has the ability to generate human-readable interface names. My diagnostic system uses an overloaded function called DbgName to get the "debug name" of various kinds of objects. DbgName (WM_DESTROY) returns WM_DESTROY; DbgName (SCODE) returns something like S_OK or E_OUTOFMEMORY; and DbgName(CWnd*) returns the name of the window.

As you'd expect, DbgName(REFIID) returns a human-readable GUID like

```
{EB0FE172-1A3A-11D0-89B3-00A0C90A90AC}
```

but in case you don't know off the top of your head that that's IDeskBand, COMToys has a way you can tell.

```
DEBUG_BEGIN_INTERFACE_NAMES()
  DEBUG_INTERFACE_NAME(IDeskBand)
DEBUG_END_INTERFACE_NAMES()
```

Now DbgName(IID_IDeskBand) returns IDeskBand instead of the hex gobbledygook. The macros generate a local table that's linked to a global one that

DbgName can search. For more information, download the source and look in debug.h and debug.cpp.
COMToys DLL The COMToys library you get when you download the source from this site can be built six ways: as a static library that statically links to MFC, as a static library that dynamically links to MFC, or as a DLL that dynamically links to MFC (extension DLL) all with debug and release versions.
Problems What system isn't without problems? As I said at the outset, COMToys isn't complete. I wrote only the parts I needed to do band objects. Thus, for example, CTPersistStream and CTPersistFile don't really do anything; all they have is a modified flag. As the Microsoft® bug docs are wont to say, "this behavior is by design." (When you can't fix it, call it a feature.) Another problem with COMToys is sometimes the macros are annoying—you can't step through them in the debugger. Fortunately, I hate debugging anyway.

COMToys and MFC

Before wrapping up, I should say a few words about COMToys' relationship to MFC. COMToys seems to rely heavily on MFC. That's no big deal for me since I like MFC and use it all the time. But some folks may consider MFC a liability. Without getting into the MFC-versus-ATL fray (I'm an atheist when it comes to religion), I'll only point out that COMToys' basic approach of mixing interfaces with concrete classes is independent of MFC.

As I was writing COMToys, I was careful to be aware of where I needed MFC and where I didn't. Thus CTOleWindow, CTDockingWindow, and CTInputObject use handles (HWND, HACCEL) instead of MFC objects like CWnd*, since they don't need the MFC wrappers for anything. On the other hand, CTMfcContextMenu requires CMenu and CCmdTarget because it does all the command routing stuff. Likewise, CTMfcModule calls MFC to implement standard DLL entries. In general, classes designed to work with MFC have Mfc in the name.

Aside from implementation classes like CTMfcContextMenu, there are only three places where COMToys really needs MFC. IMPLEMENT_IUnknownCT assumes a CCmdTarget-derived main class to implement IUnknown. CTFactory doesn't implement IClassFactory as it should, so you need a COleObjectFactory-derived factory class to create objects. CTModule doesn't implement module locking or DllGetClassObject and company, so you need CTMfcModule to get these important details. To completely divorce COMToys from MFC, you simply need to fill these gaps.

For IUnknown, you need a basic CTUnknown that implements AddRef, Release, and QueryInterface. AddRef/Release could simply call ++/-- on a data member m_dwRef, and QueryInterface could call a moral equivalent to GetInterfaceHook, which the outer class would have to supply. You could derive another implementation (call it CTUnknownMTA) that uses InterlockedIncrement and InterlockedDecrement instead of ++/--. With these implementations, your COM class would use IMPLEMENT_ IUnknown (CMyComClass, CTUnknown) instead of the current scheme that uses IMPLEMENT_IUnknownCT for a CCmdTarget. It's starting to resemble ATL—but without templates.

For class factories, you need to implement IClassFactory for CTFactory. The implementation could call a virtual function named something like OnCreateObject that each COM class would have to provide by returning "new CMyComClass". (COleObjectFactory uses the MFC runtime system to create objects from scratch.)

To hook the factories up, you need to implement CTModule::OnDllGetClassObject. The implementation would search the list of factories for one whose CLSID matches the request, and then call its CreateInstance method.

Finally, there are a couple of other nits. You need some code in CTModule to do module locking (CTModule::OnDllCanUnloadNow) and a DllMain that initializes or terminates the module when it gets DLL_PROCESS_ATTACH/DETACH.

With these modifications, you could use COMToys to write COM objects that are just as lean or leaner than objects written with ATL. COMToys would separate into two layers: a foundation layer independent of MFC and higher-level classes that use it. I never bothered to make the split because I haven't had the need. When I do, I will. No doubt I'll write about it, too.

Terrifying Anecdote from Hell

No project is complete without at least one horror story, and I had one building COMToys. As I was testing the registration code, I noticed IRegistrar didn't completely remove everything when I called ResourceUnregister. I guess it likes to play it safe. I decided to add a few lines to delete HKCR\CLSID\clsid when a class is unregistered. I poked around MFC for a function to do it, and found just what I needed: AfxOleUnregisterClass. This function takes a class ID and a ProgID and removes it from the registry. So I added this line:

```
AfxOleUnregisterClass(m_clsid, GetProgID());
```

To test my code, I ran regsvr32.exe /u, then popped over to REGEDIT, which was already open, and pressed F5 to refresh. **Figure 11** shows the result. You will note, please, that HKEY_CLASSES_ROOT is fully expanded. That's it. That's all there was. My entire HKEY_CLASSES_ROOT gone, except for the empty key CLSID.

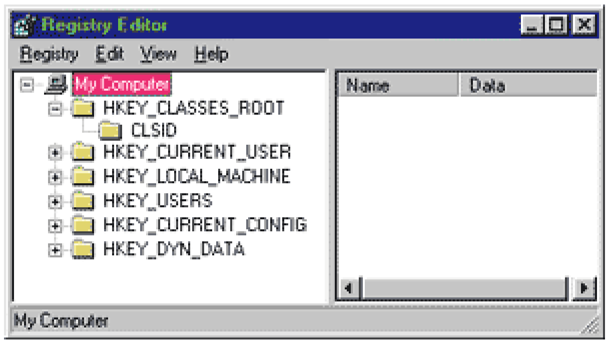

Figure 11 Yikes! Where Did Everything Go?

This can't be; REGEDIT must be confused (stage one, denial). So I quit REGEDIT and reran it from the Start menu. Instead of REGEDIT, I got a little dialog box. "This program refers to a nonexistent link..." or some such message that just as blandly understated the enormity of my situation. Everything on my desktop was wiped out. Windows Explorer? Gone. Microsoft Internet Explorer? Gone. Every icon replaced with the generic I-don't-understand icon. I couldn't even start an MS-DOS window because that link was gone too. I almost threw the keyboard through my monitor (stage two, anger). I rebooted with fingers crossed. No luck. I was hosed. MFC had completely erased my registry. I stared dumb-eyed at the screen (depression), trying to remember when I'd made the last backup (acceptance).

Fortunately, I use a commercial program that automatically backs up my registry every day. But how to run it when the link was gone? In the end I was able to click Run from the Start menu, then run\command.com to get a shell. I went to the ConfigSafe directory and ran it to restore my registry to the previous day's configuration. Whew!

A couple of cardiac pills and some postmortem spelunking revealed the events that led to my system's near-death experience. Since band objects aren't insertable, they don't need a ProgID. The registry string for MyBands has no ProgID, so CTFactory::GetProgID naturally returned an empty CString. Empty, mind you, not NULL. I passed this empty string along to AfxOleUnregisterClass, which has the following lines:

```
if (pszProgID != NULL)
  _AfxRecursiveRegDeleteKey(HKEY_CLASSES_ROOT,
    (LPTSTR)pszProgID);
```

No need to explain further; the word "recursive" says it all. MFC blithely deleted everything under

HKEY_CLASSES_ROOT. Would somebody in Redmond please fix this!

There are several lessons here. Always be extremely careful doing anything with the registry. If you're writing code that manipulates keys, add lots of extra safety checks against NULL arguments and empty strings. If you aren't using a program to back up your registry every day, do it now. Don't trust MFC. Don't trust anybody. And don't be a programmer if you have heart trouble. (Just kidding about the cardiac pills.)

Conclusion

Whether or not you decide to use COMToys in your next project, I hope at least you now realize that using ATL, COM, or any other system descended from Above isn't an either/or, all-or-nothing proposition. You can take a one-from-column-A, two-from-column-B Chinese menu approach. I like to think of COMToys as MFC with multiple inheritance, or ATL without templates. The point is, you're not stuck with any one system. You can have it your way.

I also hope you can see how, by taking the time to build a little infrastructure (COMToys is only 2400 lines or so, including implementation classes), you can vastly simplify your C++ COM programming life. Macros, smart pointers, tracing—these simple tools go a long way to make programming easier. The final COMToys library is too much to print here, but you can download it from the link at the top of this article. Writing BandObj with COMToys is just a matter of gluing together prebuilt components and writing the part that's new. This isn't far from the Holy Reusability Grail. BandObj is half the size it used to be, and all the code that moved to COMToys can now be used in other apps. There's more to life than writing IPersistStream for the nth time.

COMToys makes no claims to perfection; no system ever can. But so far it works well in practice, and I've used it to build band objects and a quick view file viewer. I'll continue to refine COMToys as time goes on, so stay tuned for the latest developments and source code. Until then, happy programming! See you in the next millennium.

For related information see:
Creating Custom Explorer Bars, Tool Bands, and Desk Bands at:
http://msdn.microsoft.com/library/psdk/shellcc/shell/Bands.htm.
Also check http://msdn.microsoft.com for daily updates on developer programs, resources and events.

From the December 1999 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.