November 1999

MICROSOFT SYSTEMS JOURNAL

MyBand is Your Band: More Reusable MFC Goodies with Band Objects and COMToys

Paul DiLascia

Band objects come in three flavors: desk, info, and comm bands. Desk bands live in the task bar; info and comm bands—also called Explorer Bars—live in Microsoft Internet Explorer. MyBands.dll implements all three kinds of band and demonstrates how easy it is to write a band object.

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

Code for this article: bandObj.exe (93KB)

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.*

Not too long ago someone asked me how to add an edit control to his Windows® task bar. He wanted to know how to get the HWND so he could add his control as a child window. I had to break the bad news: unfortunately, that sort of thing isn't done any more. You don't go mucking around the system the way you did in the old days. It's not allowed. Nowadays, you write COM interfaces so your extensions can communicate with the operating system in a polite, orderly fashion. May I have your menu, please? Why, certainly, here it is. Would you execute this command now, please? But of course. That's IContextMenu for you, just one of many little interfaces that pop up in various kinds of shell extensions.

But among all these interfaces, I couldn't find any way to add a window to the task bar. I was about to tell my poor reader "no can do," when I discovered band objects and IDeskBand. (They were new at the time.) Right away I knew IDeskBand was the kind of interface I could love: it has only one method! So I decided, why not: I'll write my own desk band. With only one function, how hard could it be?

Several months and thousands of code lines later (not to mention Advil tablets and extended therapy sessions), I'm here to report my results. It's not that band objects are so hard (they aren't), but my project grew and grew. It wasn't enough to write one band object; I had to build a framework. Then it wasn't enough to have the framework; I had to have my own general-purpose COM programming system. Reusability is the Holy Grail of programming, and like Indiana Jones, I'm always hunting it. In the end, I built my BandObj framework and some reusable COM goodies I called COMToys. I'll describe BandObj here and COMToys in the sequel.

Meet MyBands

The best way to understand any system is to understand the problem it was invented to solve. (Some systems weren't invented to solve any problem, and they're usually bad.) That problem is MyBands.

I really hate sample apps that paint some dopey text like "Hello [insert app name here]" on an ugly background and then call it a day. I figure, if I'm going to the trouble of writing a program, I may as well get something useful—or at least cute—out of it. So the first thing I had to do when I set out to write a desk band was invent a reason for doing so because, let's be honest, putting your own little window in the task bar is pretty frivolous. I chose the first, most obvious dopey Web idea that crossed my feeble mind, only two notches above "Hello, world": a Web Search Band (see **Figure 1**). Type something in the box, and there goes the browser to your favorite search engine. You can even drag it off the task bar, as in **Figure 2**.

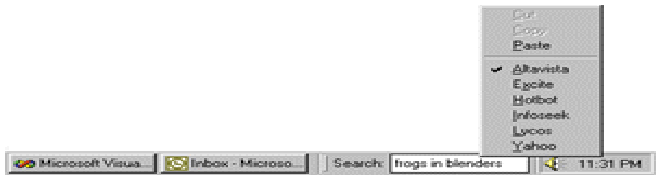


Figure 1 The Web Search Band

I implemented the Web Search Band in a DLL called MyBands.dll. Astute readers will notice the plural. In fact, band objects come in three flavors: desk, info, and comm bands. Desk bands live in the task bar; info and comm bands—also called Explorer Bars—live in

Microsoft® Internet Explorer (see **Figure 3**). The Internet Explorer History, Favorites, and Search windows are all info bands. MyBands.dll implements all three kinds of band, but the info and comm bands are of the dopey "Hello, world" variety.

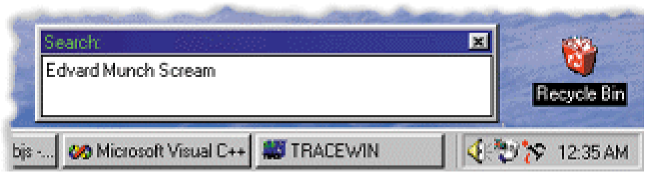


Figure 2 Standalone Search Band

To install MyBands, download the code (93KB), compile it, put the DLL somewhere, and type the following:

```
regsvr32.exe MyBands.dll
```

The info and comm bands will appear in the Toolbars menu the next time you run Internet Explorer. The desk band is a little trickier: you have to restart Explorer. Ctrl-Alt-Del, kill process, wait for rebirth. When you do, Web Search Band appears in the Toolbars menu as in **Figure 4**.

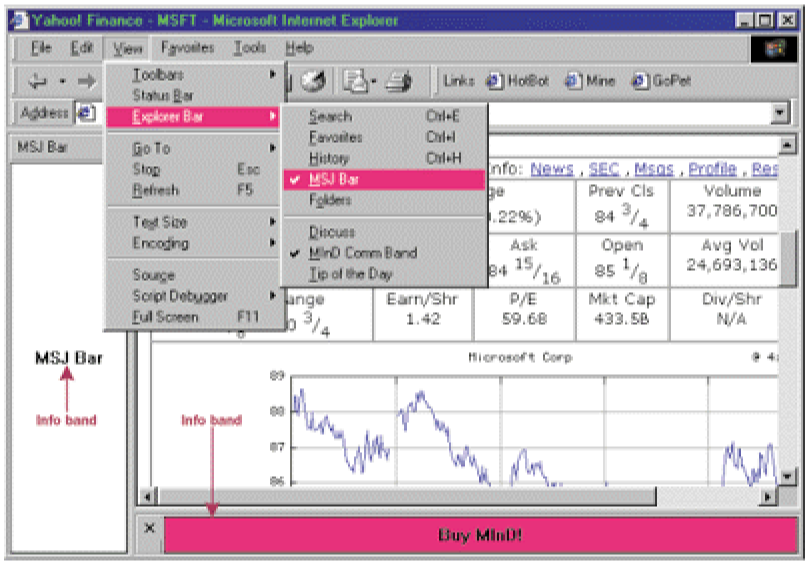


Figure 3 Info and Comm Bands in Internet Explorer

To see how easy it is to write a band object using BandObj, let's look at MyBands. MyBands is composed of several modules, but the only one that has anything to do with band objects is the main module, MyBands.cpp (see **Figure 5**). MyBands has an app class called CMyBandsDll, which is derived from CWinApp via CBandObjDll (BandObj) and COleControlModule (MFC). Like a normal MFC app, CMyBandsDll has an InitInstance function.

```
BOOL CMyBandsDll::InitInstance()
{
  AddBandClass(CLSID_MYINFOBAND,
    RUNTIME_CLASS(CMyInfoBand),
    CATID_InfoBand,
    IDR_INFOBAND);
  AddBandClass(CLSID_MYCOMMBAND, ...);
  AddBandClass(CLSID_MYDESKBAND, ...);
  return CBandObjDll::InitInstance();
}
```

This resembles doc/view, except instead of calling AddDocTemplate, you call AddBandClass. For each band class you must supply the class ID (GUID), MFC runtime class, category, and resource IDs. The category ID is just a GUID that tells Windows what kind of band your class is—info, comm, or desk. As you'd expect, MyBands uses a separate class for each kind of band. CMyInfoBand, CMyCommBand, and CMyDeskBand are each derived from CBandObj and each uses DECLARE/IMPLEMENT_DYNCREATE so MFC can create instances dynamically using its normal runtime mechanism and COleObjectFactory. They have constructors that initialize information about the band in a DESKBANDINFO struct, CBandObj::m_dbiDefault. For example, the desk band has a default width of 100 and variable height.

```
CMyDeskBand::CMyDeskBand()
  : CBandObj(CLSID_MYDESKBAND)
{
  m_dbiDefault.ptActual = CPointL(100,0);
  m_dbiDefault.dwModeFlags = DBIMF_VARIABLEHEIGHT;
}
```

The comm band has a fixed height of 30 pixels and no title.

```
CMyCommBand::CMyCommBand()
  : CBandObj(CLSID_MYCOMMBAND)
{
  m_strTitle.Empty();
  m_dbiDefault.ptMinSize = CPointL(0,30);
  m_dbiDefault.ptMaxSize = CPointL(-1,30);
}
```

Believe it or not, that's pretty much it as far as the band object part of MyBands goes. The rest of MyBands deals with implementing the band behavior and looks like a normal MFC app. For example, CMyDeskBand has an OnCreate handler that creates an edit control, and CMyCommBand has a WM_PAINT handler that draws a subtle advertising message.

```
void CMyCommBand::OnPaint()
{
  CPaintDC dc(this);
  dc.DrawText("Buy MInD!");
}
```

Band objects don't have top-level menus like frames, but they can have context menus if you like. CBandObj handles the dirty details. All MyBands has to do is create a menu with the same resource ID as the band class. Commands magically arrive at MyBands' ON_COMMAND handlers through all the normal channels. Except for the GUIDs and category IDs, you'd hardly even know MyBands is a COM object. BandObj hides all the handshaking, leaving you free to write your band. This is how life should be.

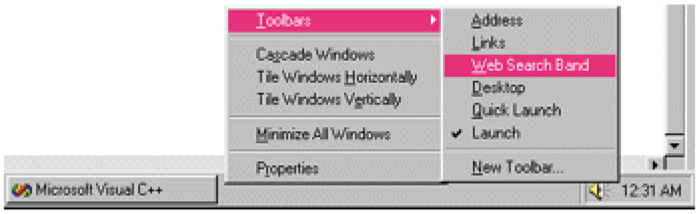


Figure 4 Desk Band in the Toolbars Menu

Meet BandObj

Now that you've seen you how trivial it is to write a band object using BandObj (define a GUID, call AddBandClass), let's dig down a layer and see how BandObj works. What I've been calling BandObj is actually three classes: CBandObjDll, CBandObjFactory, and CBandObj (see **Figure 6**). CBandObjFactory is the class factory that makes CBandObj objects. Generally, you never need to use it directly. When you call AddBandClass from your InitInstance function, CBandObjDll creates a new factory and adds it to a list.

```
BOOL CBandObjDll::AddBandClass(...)
{
  CBandObjFactory* pFact =
    OnCreateFactory(...);
  pFact->m_pNextBandFact =
    m_pBandFactories;
  m_pBandFactories = pFact;
  return TRUE;
}
```

OnCreateFactory is a virtual function that simply returns a new factory.

```
CBandObjFactory*      CBandObjDll::OnCreateFactory(...)
{
  return new CBandObjFactory(...);
}
```

I provided OnCreateFactory in case you ever have the perverse desire to improve my code by deriving your own specialized factory class. If so, derive from CBandObjFactory and override OnCreateFactory to make BandObj use it. The args I omitted for the sake of clarity are the same ones you pass to AddBandClass: the class ID, MFC runtime class, category ID, and resource ID. CBandObjFactory passes the first two to MFC and keeps the last two itself.

```
CBandObjFactory::CBandObjFactory(REFCLSID clsid,
    CRuntimeClass* pClass,
    const CATID& catid, UINT nIDRes)
    :  COleObjectFactory(clsid, pClass, FALSE, NULL)
  {
    m_catid = catid;
    m_nIDRes = nIDRes;
  }
```

BandObj doesn't do factories quite the normal way. Normally when you write a COM object in MFC, you use DECLARE_OLECREATE and IMPLEMENT_OLECREATE, which create your COleObjectFactory as a static object. One problem with these macros is they have COleObjectFactory hardwired as the class, so you can't use any other. Another problem is that they create the factory as static data, again with the hardwired name CYourClass:: factory. So? Who says you have to use the macros? They're only provided for convenience. If I want to create my own factory on the heap instead of the stack, and using some other class, I'm allowed. As long as I derive from COleObjectFactory, MFC will find my factory when it's time to create objects because each COleObjectFactory adds itself to a master list MFC searches when COM calls my DLL to create objects. By creating the factories on the fly, BandObj is able to hide them from programmers.

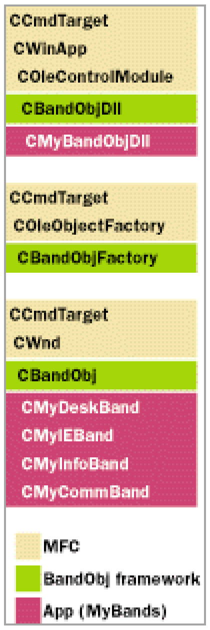


Figure 6 BandObj

Registering MyBands

Technically I still haven't even told you exactly what a band object is yet, but never mind—it's time to discuss registration. In COM, you can't even blow your nose without registering. Here are the registry entries for the Web Search Band:

```
HKEY_CLASSES_ROOT
  CLSID
    {4647E383-520B-11d2-A0D0-004033D0645D} = "&Web Search Band"
      InprocServer32 = MyBands.dll
        ThreadingModel=Apartment
      Implemented Categories
        {00021492-0000-0000-C000-000000000046}
```

CLSID, InprocServer32, ThreadingModel— it's all COM 101. The only thing that's new for bands is the previously mentioned category ID, which goes under the key Implemented Categories. In general, a COM object declares to the world what categories it implements by listing them under HKCR\CLSID\guid\Implemented Categories.

Of course, you don't register a COM object by hand—

COM objects are expected to register themselves. When you type

```
regsvr32.exe MyBands.dll
```

regsvr32 calls the special entry DllRegisterServer to register MyBands. If you add /u before the file name, it calls DllUnregisterServer. BandObj.cpp provides default implementations for these standard entries as well as the others, DllGetClassObject and DllCanUnloadNow. The default implementations call special MFC functions designed to do the right thing.

```
STDAPI DllRegisterServer()
{
  AFX_MANAGE_STATE(AfxGetStaticModuleState());
  return COleObjectFactory::UpdateRegistryAll(TRUE)
   ? S_OK : SELFREG_E_CLASS;
}
```

COleObjectFactory::UpdateRegistryAll loops over all the factories and calls UpdateRegistry(TRUE) for each one. DllUnregisterServer is similar, except it calls MFC with FALSE. COleObjectFactory::UpdateRegistry does a pretty good job registering controls, but it knows nothing about bands or categories, so it looks like I have to write some code to register the band.

Registration Frustration? Not

If you've ever tried to register an object using the Windows API, you know what a pain it is. RegCreateKey, RegSetValue, RegClose. . . I never get it right the first time. Fortunately, there's a better way. One of my favorite features in ATL is the Registrar. It makes registering COM objects a total snap. The ATL Registrar is like REGEDIT on steroids. It lets you load special .RGS scripts, similar to .REG files, that can not only add, but also delete registry entries. In fact, given a script, the registrar knows (more or less) how to undo—that is, unregister—it. This means you can use the same script to register as well as unregister, which is a good thing because unregistration is like an orphan child—most programmers are too lazy to do it, so apps end up polluting the registry.

Using the ATL registrar is really easy.

```
CComPtr<IRegistrar> ireg;
ireg.CoCreateInstance(CLSID_Registrar,
                      NULL, CLSCTX_INPROC);
ireg->FileRegister("foo.rgs");
```

The funny-looking thing with angle brackets is an ATL smart pointer (more on that later). CoCreateInstance creates a registrar object and you're ready to roll. Because I have a neurotic aversion to seeing angle brackets in my code except where I'm comparing things or doing a shift, I wrote a little COMToys class to simplify it even further.

```
CTRegistrar r;
r->FileRegister("foo.rgs");
```

The constructor calls CoCreateInstance so you don't have to. Just instantiate CTRegistrar and call the methods. IRegistrar has methods to register and unregister scripts from a file or—better yet—resource. **Figure 7**, from atliface.h, shows all the IRegistrar methods. CBandObjFactory::UpdateRegistry has a generic implementation that looks for a REGISTRY resource with the same ID as your factory and calls the registrar to load it.

```
BOOL CBandObjFactory::UpdateRegistry(BOOL bRegister)
{
  static const LPOLESTR RT_REGISTRY =
    OLESTR("REGISTRY");
  UINT nID = GetResourceID();
  if (!::FindResource(AfxGetResourceHandle(),
    MAKEINTRESOURCE(nID), CString(RT_REGISTRY)))
    return FALSE;
  CTRegistrar iReg;
  OnInitRegistryVariables(iReg); // see below
  LPOLESTR lposModuleName = /* get module pathname */
  HRESULT hr = bRegister ?
    iReg->ResourceRegister(lposModuleName, nID,
      RT_REGISTRY) :
    iReg->ResourceUnregister(lposModuleName, nID,
      RT_REGISTRY);
  return SUCCEEDED(hr);
}
```

This is another place where, following MFC tradition, BandObj makes good use of resource IDs. All you have to do to register or unregister your band object is write a registration script—you don't have to write any code! And all I had to write was this simple function. Actually, you don't even have to write a registry script because BandObj comes with one that works for any band object. All you have to do is use it.

```
// in MyBands.rc
IDR_INFOBAND REGISTRY DISCARDABLE "BandObj.rgs"
IDR_COMMBAND REGISTRY DISCARDABLE "BandObj.rgs"
IDR_DESKBAND REGISTRY DISCARDABLE "BandObj.rgs"
```

But wait a minute. How can three different COM objects possibly use the same registration script? Don't they each have different names and class IDs? This is where IRegistrar really shows its mettle. Take a look at BandObj.rgs in **Figure 8**. What are those funny tags %CLSID%, %ClassName%, and %MODULE%? Those are variables. Before processing your script, the registrar replaces %CLSID%, %ClassName%, and %MODULE% with the actual class ID, class name, and module name. How does it know what values to use? Because you tell it—or rather, BandObj does. You may have noticed the call to OnInitRegistryVariables in UpdateRegistry. That's where BandObj defines its variables.

```
BOOL CBandObjFactory::
  OnInitRegistryVariables(IRegistrar* pReg)
{
  USES_CONVERSION;
  pReg->AddReplacement(OLESTR("CLSID"),
    StringFromCLSID(m_clsid));
  pReg->AddReplacement(OLESTR("MODULE"),
    T2OLE(GetModuleName()));
  pReg->AddReplacement(OLESTR("ClassName"),
    T2OLE(GetClassName()));
  return TRUE;
}
```

Here's a full list of variables I built into CBandObjFactory, automatically defined by BandObj.

```
%CLSID%     = class ID (GUID)
              (COleObjectFactory::m_clsid)
%MODULE%    = full pathname of DLL
%Title%     = title (resource substring 0)
%ClassName% = human-readable COM class name (resource substring 1)
%ProgID%    = ProgID (resource substring 2)
```

To add your own, such as %TimeStamp% or % MyReleaseVersion%, just derive a new factory class and override OnInitRegistryVariables. Don't forget to call the base class! Since MFC calls UpdateRegistry for each factory, the variables get reinitialized for each class. So for MyBands, %CLSID% is CLSID_MYINFOBAND for the first factory, CLSID_MYCOMMBAND for the second, and CLSID_MYDESKBAND for the third. The same script works for all three. It's way cool.

IRegistrar is so cool I wrote my own command-line utility, RGSRUN, to load RGS files. It's great for testing scripts and debugging, or just deleting extra junk from your registry, something REGEDIT can't do. You can download RGSRUN from the link at the top of this article. **Figure 9** shows the whole program. I use RGSRUN in my autoexec.bat with a file, autoexec.rgs, that sets various Explorer settings Windows seems to insist on bashing every time it boots. Take that, Windows!

Categorical Imperative

If you were reading carefully, you noticed that the script in **Figure 8** doesn't have a key for Implemented Categories. Why? And where does BandObj register its categories? I could've done categories using another variable like %catid%, but COM has an interface for every situation, and categories are no exception. The official way to register categories is through ICatRegister. By adhering to State-Approved Mechanisms, you indemnify yourself against future liability (in theory). No problem, here's the code.

```
BOOL CBandObjFactory::UpdateRegistry(
 BOOL bReg)
{
```

```
// as before
.
.
.
  // register/unregister categories using ICatRegister
  CTCatRegister iCat;
  REFIID clsid = m_clsid;
  hr = bRegister ?
    iCat->RegisterClassImplCategories(clsid, 1,           &m_catid) :
    iCat->UnRegisterClassImplCategories(clsid, 1,         &m_catid);
    // return, bypassing MFC
    return hr==S_OK;
 }
```

Once again, ATL smart pointers and a little COMToys class, CTCatRegister, make programming easy and bulletproof. Just instantiate and go.

Getting to the Bottom of Band Objects

Now that MyBands is registered and you have the factories to create them, it's time to ask: what exactly *is* a band object, anyway?

A band object is a window that sits inside the task bar or Internet Explorer. That much I hope you figured out by now. A band object is also a COM object that implements three interfaces: IDeskBand, IObjectWithSite and IPersistStream. Optional interfaces include IInputObject if your brand can accept input focus, and IContextMenu if you have a context menu. **Figure 10** tells the story. Man, I thought all I had to do was write one function! On top of that, IDeskBand is derived from IDockingWindow, which is derived from IOleWindow. That's two more interfaces you have to implement. What do all these interfaces do?

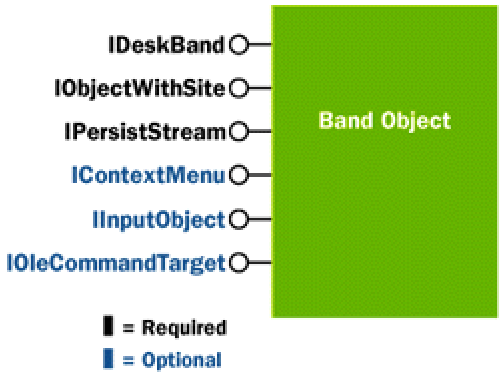


Figure 10 Band Object Interfaces

The best way to understand band objects—or any COM object for that matter—is to examine the sequence of events as the object starts up—in this case when a user selects your band from the Toolbars menu until the time they close it. The best way to do that is to start with some sample code that works and add a lot of TRACE diagnostics. BandObj has the diagnostics built in, using one of my goodies from a previous article: TRACEFN. TRACEFN uses a special class and my own version of AfxTrace to generate indented diagnostic output, so you can see the stack. **Figure 11** shows the output of a typical MyBands session, beginning when the user selects Web Search Band from the Toolbars menu. Here's the play-by-play.

Windows (Explorer for desk bands, Internet Explorer for info and comm bands) discovers your band object by looking for COM objects that implement CATID_DeskBand, CATID_InfoBand, or CATID_CommBand, and adds the name of your band to its Toolbars menu as in **Figure 4**.

When the user selects your band from the menu, Windows calls CoCreateInstance or its equivalent. COM calls your DLL's DllGetClassObject, which calls AfxDllGetClassObject. MFC searches for a factory with the right ID and returns it. COM then calls IClassFactory::CreateInstance and voilà—there's your band object! More COM 101.

Next, Windows queries for IDeskBand and IObjectWithSite. CBandObj implements these interfaces in the normal MFC way—using nested classes, interface maps, and BEGIN/END_INTERFACE_PART—so MFC returns the right pointers.

Windows calls IObjectWithSite::SetSite to give you a pointer (IUnknown*) back to the container running your show. CBandObj::XObjectWithSite::GetSite calls the virtual function CBandObj::OnSetSite, converting the nested class method to a parent class virtual function call so you can override it easily. The default implementation stores the site in m_spSite. You can QueryInterface m_pSite for any interface the container implements. CBandObj uses it to get the HWND of its parent window.

```
CComQIPtr<IOleWindow> spOleWin =
  m_spSite;
if (!spOleWin)
  return E_FAIL;
HWND hwndParent = NULL;
spOleWin->GetWindow(&hwndParent);
if (!hwndParent)
  return E_FAIL;
```

When Windows calls SetSite, it expects you to create your window. This isn't obvious from the spec; I only found out the hard way. CBandObj::OnSetSite calls a virtual function OnCreateWindow to do it. My default implementation registers and creates a generic invisible window.

```
BOOL CBandObj::OnCreateWindow(CWnd* pParent,
  const CRect& rc)
{
  static BOOL bRegistered = FALSE;
  static CCriticalSection cs; // protection
  CTLockData lock(cs);
```

```
    // register window class if not already
    if (!bRegistered) {
      AfxRegisterClass(...);
      bRegistered = TRUE;
    }
    return CWnd::Create(BANDOBJCLASS,...);
  }
```

Band objects are apartment threaded, so it's important to protect globals. It's always a good idea to keep globals as close as possible to the functions that use them. In this case, OnCreateWindow is the only function that uses bRegistered, so it's a static function. You can override OnCreateWindow to create your own window class, and/or override PreCreateWindow to change some properties. Just remember: your window should be invisible at this point, so make sure to create it without WS_VISIBLE.

Next, Windows calls IOleWindow::GetWindow to get your window's HWND. (This is why you have to create your window in SetSite.) CBandObj returns m_hWnd.

After that, Windows calls IDeskBand::GetBandInfo requesting information about your band such as how big it wants to be, whether it has variable or fixed height, and the background color and title. CBandObj stuffs the DESKBANDINFO with a combination of defaults—which you can set in your band object's constructor—and information from your resource file. For example, it gets the title from your resource string. Next, Windows calls IDockingWindow:: ShowDW to show your window. CBandObj calls CWnd:: ShowWindow. It's alive!

Windows queries for IInputObject if the user types a key and IContextMenu if the user right-clicks for a context menu. If you implement these interfaces, Windows will use them. CBandObj has default implementations that grab the accelerators and menu from your resource file. All you have to do is add the resources to your project. The menu itself is stored in a data member, m_contextMenu, which you are free to manipulate.

CMyDeskBand doesn't have a resource menu; instead it creates the menu dynamically after reading the user's profile settings (see **Figure 12**). Before appending the menu to the container (when the container calls IContextMenu::QueryContextMenu), CBandObj routes it through the MFC command highway so all your ON_UPDATE_COMMAND_UI handlers work. This is how CMyDeskBand puts a checkmark next to the currently selected search engine in **Figure 12**.

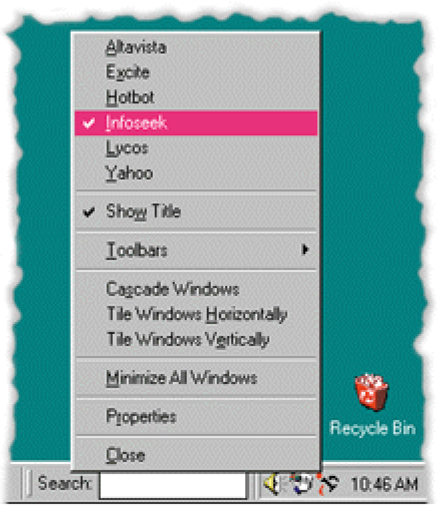


Figure 12 Instant Menu

If the user invokes a command by typing an accelerator key or selecting a menu item, Windows calls IContextMenu::InvokeCommand. CBandObj does the same MFC routing as for initializing the menu, so the command magically arrives at your ON_COMMAND handlers in the usual way.

When the user closes your band, Windows calls IDockingWindow::CloseDW. CBandObj sends a WM_CLOSE and—poof—your window is gone. But the CBandObj object lives on until Windows releases it. It's all very straightforward and logical—*if* you use TRACE!

Band Object Bugaboos

What I've just described is the vanilla operation of a basic band object. But in the real world, nothing ever works quite the way it's supposed to, so it's time to let you know the things Mama didn't tell you.

The docs say a desk band needs IPersistStream to save any persistent data. What data? My trace diagnostics reveal that, in the case of desk bands, Explorer doesn't even query for IPersistStream, though it does query for IPersistStreamInit. So what's IPersistStream for? A comment buried in some Microsoft sample code explains the deal: IPersistStream is required to let the user drag the desk band off the task bar, as in **Figure 2**. If you don't want or need this feature, you don't have to implement IPersistStream.

TRACE also reveals that Windows queries for IOleCommandTarget and IDiscardableBrowserProperty. The latter is an interface I can *really* love; it has no functions! How can an interface have no functions? IDiscardableBrowserProperty is just a way to tell Internet Explorer, "It's OK to toss my properties if you visit another page. They're expendable." For more information, see "Discardable Properties for Your Web

Pages in Internet Explorer 4.0" in the MSDN™ Library.
Finally, TRACE again reveals that Windows queries for yet another interface, the secret interface {EA5F2D61-E008-11CF-99CB-00C04FD64497}. This ID doesn't appear in any document, source file or registry entry I can find. If you know what this interface does, please contact your local authorities.
When you first create your desk band, you may have trouble getting Windows to recognize it. For info and comm bands, all you have to do is restart Internet Explorer. The same is true for Explorer, the only trick there being you have to Ctrl-Alt-Del to kill it. But when I recently got a new machine with Windows 98 Second Edition installed, my desk band mysteriously stopped working. No matter how many times I registered the DLL and killed and rekilled Explorer, my band refused to appear. The same thing happened on Windows 2000.
After much fretting and frustration, and nearly throwing my brand-new $2700 18" liquid crystal monitor out the window, I called my trusty editor (this is where it helps to have connections). He relayed the problem to the friendly Redmondtonians, who promptly pointed me to Knowledge Base article Q214842, which explains the mystery. Windows 2000 (and apparently also Windows 98 Second Edition) keeps a category cache that it refreshes only "if it senses that an installation application is run or if the cache location in the registry is not present." Windows "senses" an installation. Is it psychic? The article goes on to explain two ways to make Explorer rebuild its category cache: install from a program named setup.exe or install.exe (psychic powers explained—is that hokey or what?) or delete the registry key HKEY_CLASSES_ROOT\Component Categories\{00021492-0000-0000-C000-000000000046}\Enum, which is the cache. ({00021492...} is CATID_DeskBand.) I modified BandObj.rgs to always delete this key. Problem solved. But you still have to restart the shell to make Windows generate a new cache.
The next problem seemed like three problems, but they were all caused by the same bug. Whenever I tried to recompile my code, I got "Cannot open MyBands.dll for writing." Apparently Explorer keeps its band objects alive as long as it's running. Internet Explorer does the same thing: when the user hides and shows your band, Internet Explorer calls IDockingWindow::HideDW and ShowDW. Internet Explorer doesn't release the object until it quits. In the case of desk bands, this means you have to Ctrl-Alt-Del again with every build cycle. Annoying, but not a big deal.
When I tried to debug my desk band using the technique in the Visual C++® Knowledge Base article

"HOWTO: Debug a Windows Shell Extension" (the Ctrl+Alt+Shift trick, then run the debugger with explorer.exe as the debug process), I couldn't because I always ended up with a lock on my DLL. This time you have to reboot—yuck! Fortunately, I hate debugging anyway. I much prefer TRACE. But the one or two times I resorted to debugging, I had to use a system-level debugger.

When I first implemented MyBands, it failed to remember which search engine the user had selected. That's because I was saving the state in my object's destructor—but the destructor was never called because, as I just told you, Windows doesn't destroy the object, it only closes the window. So I moved my save code to PostNcDestroy, which fixed the problem. But wait a minute. If the object is still alive even after the user closes it, there should be no need to save settings because they should still be there in memory, right?

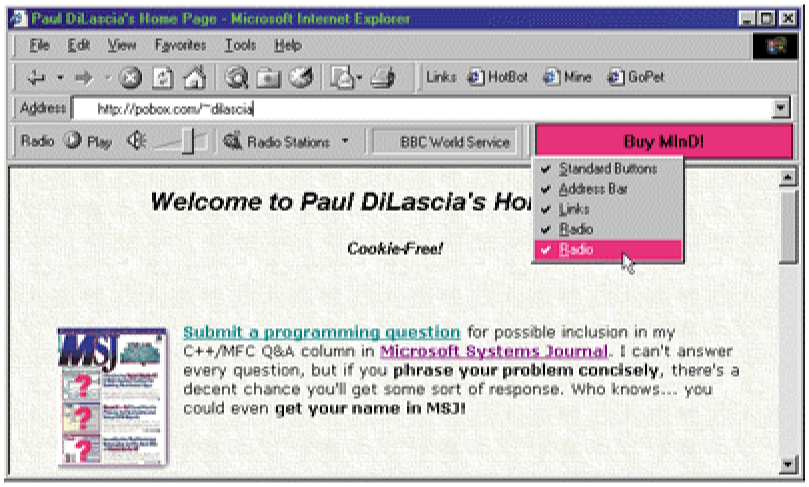
All these little problems were caused by one big problem, revealed instantly by the TRACE output: every time you hide or show a desk band, Explorer creates a new band object. It never releases the old one! Let's see. If CMyDeskBand is 916 bytes, and I have 128MB of RAM, how many times do I have to hide or show before Windows runs out of memory? The Redmondtonians acknowledge their booboo and assure it'll soon be fixed.

While I'm on the subject of booboos, I told you at the outset there are three kinds of bands. Well, I lied. There's a fourth kind: tool bands. My head is spinning just trying to keep all the terminology straight. A tool band lives in the Internet Explorer rebar. The Radio in Internet Explorer 5.0 is an example of a tool band. (You didn't know Internet Explorer has a radio now? That's nothing; 6.0 may even have HDTV.)

Writing a tool band is easy. It's just an info/comm band with a special registration; add your band's class ID (GUID) as a string under HKEY_LOCAL_MACHINE\Software\ Microsoft\Internet Explorer\Toolbar. I modified BandObj.rgs to add this value, and called it ExplrBar.rgs. Then, I modified MyBands.rc to use it.

```
IDR_COMMBAND REGISTRY DISCARDABLE "ExplrBar.rgs"
```

There's just one problem with tool bands. When you right-click in Explorer to show your band, it has the name Radio (see **Figure 13**). Actually, it has the name of the first tool band registered. This is a known bug—see Knowledge Base article Q231621.


Paul DiLascia's Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links HotBot Mine GoPet
Address http://pobox.com/~dilascia
Radio Play Radio Stations BBC World Service
Buy MInD!
Standard Buttons
Address Bar
Links
Radio
Radio

Welcome to Paul DiLascia's Ho

Cookie-Free!

Submit a programming question for possible inclusion in my C++/MFC Q&A column in Microsoft Systems Journal. I can't answer every question, but if you **phrase your problem concisely**, there's a decent chance you'll get some sort of response. Who knows... you could even **get your name in MSJ!**

Figure 13 Will the Real Radio Please Stand Up!

Back to MyBands

You're probably sick of band objects by now; you probably hope you never see another band object in your lifetime. Well, that's good because I've said all I have to say about them. Before I go, let me tell you just a few more things about MyBands—the part that actually implements the bands. It's mostly straightforward, but there were some slings and arrows worth mentioning.

First, CMyCommBand and CMyInfoBand are each derived from a common base class, CMyIEBand. CMyIEBand implements a common feature: when you click on the band, it sends the browser to the *MSJ* or *MIND* home page. To do so, the band needs an IWebBrowser2 interface so it can call IWebBrowser2::Navigate. Getting the Web browser proved to be something of a minor mystery. Obviously, it has to come from the site, so my first attempt was to write

```
CComQIPtr<IWebBrowser2> iwb = m_pSite;
```

which of course does a QueryInterface for IWebBrowser2. But QueryInterface fails with E_NOINTERFACE and a NULL pointer. The container doesn't implement IWebBrowser2. So where do you get it? Here's the required voodoo.

```
CComQIPtr<IServiceProvider> sp = m_pSite;
m_spWebBrowser2 = NULL;
if (sp) {
  sp->QueryService(IID_IWebBrowserApp,
                   IID_IWebBrowser2,
                   (void**)&m_spWebBrowser2);
}
```

IServiceProvider is a general-purpose way for COM objects to supply interfaces implemented by other objects they know about. At first it seems the same as

QueryInterface, but there's a subtle difference: the container itself doesn't implement IWebBrowser2, but it knows how to get an object that does.

Next, let's look at CMyDeskBand (the Web Search Band). The interesting thing here is the edit control, implemented in a class CEditSearch. To receive a WM_CHAR message when the user presses Enter, I had to add a WM_GETDLGCODE handler that returns DLGC_WANTALLKEYS. See? All that Windows 3.1 knowledge is still useful. Now when the user presses Enter, CEditSearch gets it and OnChar calls the DoSearch function to build a URL from the user's input. This requires replacing spaces with +, so if you search Yahoo for "beanie baby sex", the URL would be http://ink.yahoo.com/bin/query?p=beanie+baby+sex&z= 2&hc=0&hs=0, which CEditSearch passes to IWebBrowser2::Navigate. CEditSearch has the URLs for several search engines built in, but you can add more by editing MyBands.ini (see **Figure 14**). The easiest way to get the URL is to go to your favorite portal page, search for "MYSEARCH", and copy the resulting URL from the Address bar into MyBands.ini.

Speaking of INI files—MyBands uses one! The registry is so cumbersome for storing simple human-editable configuration settings that I implemented a little class called CIniFile to help me avoid it.

```
BOOL CMyBandsDll::InitInstance()
{
// SetRegistryKey(_T("MSJ")); // Not!
  CIniFile::Use(this, CIniFile::LocalDir); // Yo!
}
```

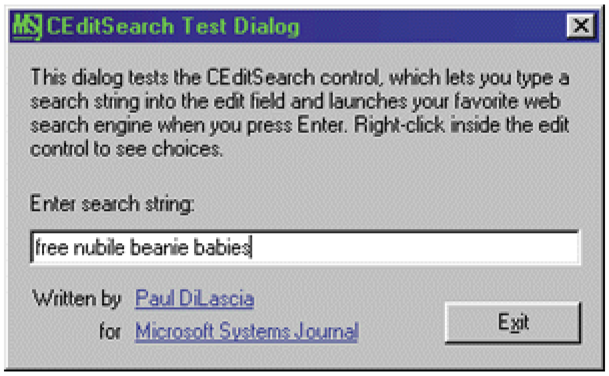
This tells your app to put the INI file in the same directory as your program or DLL, not \Windows. MFC will use an INI file by default if you don't call SetRegistryKey, but it puts the file in \Windows. I like to keep config files with their programs, so you can delete the whole directory to remove it without worrying about extra crud floating around. CIniFile gives you the option to use \Windows if you like, or specify a different file name. It works by setting your app's m_pszRegistryKey to NULL and m_pszProfileName to the name of the INI file. Of course, if you use an INI file, you don't get automatic user-specific settings on a multiuser machine the way you do with registry settings. So sue me. Or use SetRegistryKey.

Getting the context menu (see **Figure 12**) to work in the edit control required a little fidgeting. When the user right-clicks an edit control, the edit control tries to display its own Cut/Copy/Paste menu. No problem. Just override WM_CONTEXTMENU. But none of the MFC command routing stuff works because the MFC OnInitMenuPopup handler is implemented in CFrameWnd, not CWnd—even though any window can display a menu. This problem comes up from time to time in other situations. What you need to fix it once and for all is a little gadget you can drop in any CWnd that routes WM_INITMENUPOPUP through the system. I implemented one called CPopupMenuInitHandler, derived from the CSubclassWnd (née CMsgHook) class I originally described in my "More Fun with MFC" article in the March 1997 issue of *MSJ*, and which continually reappears in my columns.

CSubclassWnd lets you dynamically subclass any CWnd to trap messages. CPopupMenuInitHandler dynamically subclasses the edit control to trap WM_INITMENUPOPUP. When it sees WM_INITMENUPOPUP, it invokes another class/function, CPopupMenuInit::Init, to do the work, which is mostly copied from CFrameWnd. CPopupMenuInit::Init creates a CCmdUI object for each menu item and routes it around the system to all the ON_UPDATE_COMMAND_UI handlers. You can use it independently of CPopupMenuInitHandler any time you want to initialize a menu the MFC way. The code is in MenuInit.cpp.

Finally, CEditSearch mimics the activation dynamics found in the Internet Explorer address bar; when you first click it, all the text is selected. Click again to position the cursor. Implementing this simple feature requires handling several messages—WM_MOUSEACTIVATE, WM_ SETFOCUS, WM_LBUTTONDOWN, and any mouse or keyboard message—so I did the whole thing in CEditSearch::WindowProc. Sometimes the old pre-C++ way of doing things is a lot easier than message maps. See the source for details.

One final note. Since debugging desk bands is such a pain, I wrote a standalone program unsurprisingly called TestEditSrch to test CEditSearch before I put it in the desk band (see **Figure 15**). I strongly encourage you to do the same. Get your code to work standalone, then move it to your band.


CEditSearch Test Dialog

This dialog tests the CEditSearch control, which lets you type a search string into the edit field and launches your favorite web search engine when you press Enter. Right-click inside the edit control to see choices.

Enter search string:

free nubile beanie babies

Written by Paul DiLascia
for Microsoft Systems Journal

Figure 15 TestEditSrch

The code for MyBands/BandObj is too long to print in the magazine—and much of it is just tedious MFC nested class COM code.

Conclusion

The essence of reusable programming is identifying common behavior and encapsulating it into classes or subroutines that can be used repeatedly in different situations. In the case of band objects, BandObj reduces the entire band object to just four variables: the class ID (GUID), MFC class, resource ID, and category ID. Plus related resources that go with the ID. Everything else—all the COM interfaces and handshaking—is the same for every band object. So why should you have to write all that code? With BandObj, all you have to do is derive from the framework classes and call AddBandClass. Some programmers will object that BandObj relies on MFC and its giant MFC42.DLL. True. But I don't consider that much of a liability, at least not for shell extensions. MFC42.DLL can for all practical purposes be considered part of Windows.

If you look at CBandObj, you'll see that so much of it is generic—not just for band objects, but for any COM class. CBandObj implements IContextMenu using a CMenu, and IInputObject using an accelerator table. What does that have to do with band objects? Its implementation of IOleWindow and IDockingWindow requires only a CWnd. IObjectWithSite holds a pointer and that's it. The only interface that has anything to do with band objects is IDeskBand and the one function GetBandInfo. Everything else is boilerplate. It should be possible to capture these interface implementations in an even deeper layer of abstraction, in little classes you can assemble easily and reuse over and over again in all kinds of situations. Shell folders, file viewers, ActiveX® controls—whatever. Wouldn't it be great if all you had to do to build one of these COM objects was assemble a few pieces and compile?

Stay tuned. In my next article (December 1999), I'll show you how.

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

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

Manage Your Profile | Legal | Contact us | MSDN Flash Newsletter

Microsoft