Simplifying context menu extensions with IExecuteCommand

devblogs.microsoft.com/oldnewthing/20100312-01

March 12, 2010

Raymond Chen

The `IExecuteCommand` interface is a simpler form of context menu extension which takes care of the annoying parts of `IContextMenu` so you can focus on your area of expertise, namely, doing the actual thing the user selected, and leave the shell to doing the grunt work of managing the UI part.

I've never needed a scratch shell extension before, so I guess it's time to create one. This part is completely boring, and those of you who have written COM inproc servers can skip over it.

```
#include <windows.h>
#include <new>
LONG g_cObjs;
void DllAddRef() { InterlockedIncrement(&g_cObjs); }
void DllRelease() { InterlockedDecrement(&g_cObjs); }
// guts of shell extension go in here eventually
class CFactory : public IClassFactory
{
public:
 // *** IUnknown ***
 STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
 STDMETHODIMP_(ULONG) AddRef() { return 2; }
 STDMETHODIMP_(ULONG) Release() { return 1; }
 // *** IClassFactory ***
 STDMETHODIMP CreateInstance(IUnknown *punkOuter,
                             REFIID riid, void **ppv);
 STDMETHODIMP LockServer(BOOL fLock);
};
CFactory c_Factory;
STDMETHODIMP CFactory::QueryInterface(REFIID riid, void **ppv)
{
 IUnknown *punk = NULL;
 if (riid == IID_IUnknown || riid == IID_IClassFactory) {
  punk = static_cast<IClassFactory*>(this);
 }
 *ppv = punk;
 if (punk) {
  punk->AddRef();
  return S_OK;
 } else {
  return E_NOINTERFACE;
 }
}
STDMETHODIMP CFactory::CreateInstance(
 IUnknown *punkOuter, REFIID riid, void **ppv)
{
 *ppv = NULL;
 if (punkOuter) return CLASS_E_NOAGGREGATION;
 CShellExtension *pse = new(std::nothrow) CShellExtension();
 if (!pse) return E_OUTOFMEMORY;
 HRESULT hr = pse->QueryInterface(riid, ppv);
 pse->Release();
 return hr;
}
STDMETHODIMP CFactory::LockServer(BOOL fLock)
{
 if (fLock) DllAddRef();
 else       DllRelease();
 return S_OK;
}
STDAPI DllGetClassObject(REFCLSID rclsid,
                         REFIID riid, void **ppv)
```

```
{
 if (rclsid == CLSID_ShellExtension) {
  return c_Factory.QueryInterface(riid, ppv);
 }
 *ppv = NULL;
 return CLASS_E_CLASSNOTAVAILABLE;
}
STDAPI DllCanUnloadNow()
{
 return g_cObjs ? S_OK : S_FALSE;
}
```

I'm assuming that the above code is all old hat. Consider it a prerequisite.

Okay, now the good stuff.

The `IExecuteCommand` interface is used when you create a static registration for a shell verb but specify `DelegateExecute` in the command. Our sample shell extension will be active on text files, and all it'll do is print the file names to the debugger.

Since we're a COM server, we need to register our CLSID. This should also be very familiar to you.

```
[HKEY_CLASSES_ROOT\CLSID\{guid}\InProcServer32]
@="C:\path\to\scratch.dll"
"ThreadingModel"="Apartment"
```

Here's where we register our object as a verb for text files, specifying that it should be invoked via `DelegateExecute`:

```
[HKEY_CLASSES_ROOT\txtfile\shell\printnamestodebugger]
@="Print names to debugger"
[HKEY_CLASSES_ROOT\txtfile\shell\printnamestodebugger\command]
"DelegateExecute"="{guid}"
```

That was the easy part. Now to roll up our sleeves and write the shell extension.

```
#include <shobjidl.h>
CLSID CLSID_ShellExtension = { ...guid... };
class CShellExtension
 : public IExecuteCommand
 , public IInitializeCommand
 , public IObjectWithSelection
{
public:
 CShellExtension();
 // *** IUnknown ***
 STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
 STDMETHODIMP_(ULONG) AddRef();
 STDMETHODIMP_(ULONG) Release();
 // *** IInitializeCommand ***
 STDMETHODIMP Initialize(PCWSTR pszCommandName, IPropertyBag *ppb);
 // *** IObjectWithSelection ***
 STDMETHODIMP SetSelection(IShellItemArray *psia);
 STDMETHODIMP GetSelection(REFIID riid, void **ppv);
 // *** IExecuteCommand ***
 STDMETHODIMP SetKeyState(DWORD grfKeyState) { return S_OK; }
 STDMETHODIMP SetParameters(LPCWSTR pszParameters) { return S_OK; }
 STDMETHODIMP SetPosition(POINT pt) { return S_OK; }
 STDMETHODIMP SetShowWindow(int nShow) { return S_OK; }
 STDMETHODIMP SetNoShowUI(BOOL fNoShowUI) { return S_OK; }
 STDMETHODIMP SetDirectory(LPCWSTR pszDirectory) { return S_OK; }
 STDMETHODIMP Execute();
private:
 ~CShellExtension();
private:
 LONG m_cRef;
 IShellItemArray *m_psia;
};
CShellExtension::CShellExtension()
 : m_cRef(1), m_psia(NULL)
{
 DllAddRef();
}
CShellExtension::~CShellExtension()
{
 if (m_psia) m_psia->Release();
 DllRelease();
}
```

I've written this all out longhand; I'm trusting that you're using some sort of framework (like, say, ATL) which avoids all this tedium, but since different people may choose different frameworks, I won't choose a framework here. Instead, we just have the boring `IUnknown` methods.

```
STDMETHODIMP CShellExtension::QueryInterface(
 REFIID riid, void **ppv)
{
 IUnknown *punk = NULL;
 if (riid == IID_IUnknown || riid == IID_IExecuteCommand) {
  punk = static_cast<IExecuteCommand*>(this);
 } else if (riid == IID_IInitializeCommand) {
  punk = static_cast<IInitializeCommand*>(this);
 } else if (riid == IID_IObjectWithSelection) {
  punk = static_cast<IObjectWithSelection*>(this);
 }
 *ppv = punk;
 if (punk) {
  punk->AddRef();
  return S_OK;
 } else {
  return E_NOINTERFACE;
 }
}
STDMETHODIMP_(ULONG) CShellExtension::AddRef()
{
 return ++m_cRef;
}
STDMETHODIMP_(ULONG) CShellExtension::Release()
{
 ULONG cRef = --m_cRef;
 if (cRef == 0) delete this;
 return cRef;
}
```

Whew. Up until now, it's just been boring typing that you have to do for any shell extension. Finally we can start doing something interesting. Windows 7 will initialize your shell extension with information about the command being executed. For this particular shell extension, we'll just print the command name to the debugger to prove that something happened. (In real life, you might use the same `CShellExtension` to handle multiple commands, and this lets you determine which command you're being asked to execute.)

```
STDMETHODIMP CShellExtension::Initialize(
 PCWSTR pszCommandName,
 IPropertyBag *ppb)
{
 OutputDebugStringW(L"Command: ");
 OutputDebugStringW(pszCommandName);
 OutputDebugStringW(L"\r\n");
 return S_OK;
}
```

The shell will give you the items on which to execute in the form of an `IShellItemArray`:

```
STDMETHODIMP CShellExtension::SetSelection(IShellItemArray *psia)
{
 if (psia) psia->AddRef();
 if (m_psia) m_psia->Release();
 m_psia = psia;
 return S_OK;
}
STDMETHODIMP CShellExtension::GetSelection(
 REFIID riid, void **ppv)
{
 if (m_psia) return m_psia->QueryInterface(riid, ppv);
 *ppv = NULL;
 return E_NOINTERFACE;
}
```

The shell will then call a bunch of `IExecuteCommand::SetThis` and `IExecuteCommand::SetThat` methods to inform you of the environment in which you have been asked to execute. We just ignored them all for simplicity, but in practice, you may want to pay attention to some of them, particularly `IExecuteCommand::SetPosition`, `IExecuteCommand::SetShowWindow`, and `IExecuteCommand::SetNoShowUI`.

After all the `IExecuteCommand::SetXxx` methods have been called, it's show time:

```
STDMETHODIMP CShellExtension::Execute()
{
 HRESULT hr;
 if (m_psia) {
  IEnumShellItems *pesi;
  if (SUCCEEDED(hr = m_psia->EnumItems(&pesi))) {
   IShellItem *psi;
   while (pesi->Next(1, &psi, NULL) == S_OK) {
    LPWSTR pszName;
    if (SUCCEEDED(psi->GetDisplayName(SIGDN_FILESYSPATH,
                                      &pszName))) {
     OutputDebugStringW(L"File: ");
     OutputDebugStringW(pszName);
     OutputDebugStringW(L"\r\n");
     CoTaskMemFree(pszName);
    }
    psi->Release();
   }
   pesi->Release();
   hr = S_OK;
  }
 } else {
  hr = E_UNEXPECTED;
 }
 return hr;
}
```

All we do is enumerate the contents of the `IShellItemArray` and print their file names (if they have one). Instead of `IEnumShellItems`, you can use `IShellItemArray::GetCount` and `IShellItemArray::GetItemAt`. Or, if you are porting an existing context menu that uses `IDataObject`, you can call `IShellItemArray::BindToHandler(BHID_DataObject)` to turn your `IShellItemArray` into an `IDataObject`.

Install this shell extension, right-click on a text file (or a bunch of text files), and select *Print names to debugger*. If all goes well, the debugger will report `Command: printnamestodebugger` followed by paths of the files you selected.

But wait, there's more. The `IPropertyBag` passed to `IInitializeCommand::Initialize` contains additional configuration options taken from the registry. You can use this to customize the behavior of the shell extension further. Put the bonus information under the command key like this:

```
[HKEY_CLASSES_ROOT\txtfile\shell\printnamestodebugger]
"extra"="Special"

STDMETHODIMP CShellExtension::Initialize(
 PCWSTR pszCommandName,
 IPropertyBag *ppb)
{
 OutputDebugStringW(L"Command: ");
 OutputDebugStringW(pszCommandName);
 OutputDebugStringW(L"\r\n");
 if (ppb) {
  VARIANT vt;
  VariantInit(&vt);
  if (SUCCEEDED(ppb->Read(L"extra", &vt, NULL))) {
   if (SUCCEEDED(VariantChangeType(&vt, &vt, 0, VT_BSTR))) {
    OutputDebugStringW(L"extra: ");
    OutputDebugStringW(vt.bstrVal);
    OutputDebugStringW(L"\r\n");
   }
   VariantClear(&vt);
  }
 }
 return S_OK;
}
```

This updated version of `CShellExtension` looks for that registry value `extra` we set above and if found prints its value to the debugger.

Okay, so it looks like a lot of typing, but most of that was typing you have to do for any shell extension. The part that is specific to `IExecuteCommand` is not that bad, and it certainly avoids having to mess with `IContextMenu::QueryContextMenu` and the fifty bajillion

variations on `IContextMenu::InvokeCommand`. Furthermore, the shell doesn't even load your `IExecuteCommand` handler until the user selects your command, so switching to a static registration also gives the system a bit of a performance boost.

Bonus tip: You can combine the `IExecuteCommand` technique with *Getting Dynamic Behavior for Static Verbs by Using Advanced Query Syntax* and *Using Item Attributes* to specify the conditions under which you want your verb to appear without having to write a single line of C++ code. Choosing a Static or Dynamic Shortcut Menu Method provides additional guidance on choosing among the various methods for registering verbs.

One nice thing about `IExecuteCommand` is that it supports out-of-proc activation (i.e., local server rather than in-proc server). This means that it supports cross-bitness shell extensions: If you don't have the time to port your 32-bit shell extension to 64-bit, you can register it as an out-of-proc `IExecuteCommand`. When running on 64-bit Windows, the 64-bit Explorer will launch your 32-bit server to handle the command. Conversely, if your `IExecuteCommand` is a 64-bit local server, a 32-bit application can still invoke it.

(We'll see more about local server shell extensions in a few months. This was just foreshadowing.)

Raymond Chen

Follow