Dispatch interfaces as connection point interfaces

devblogs.microsoft.com/oldnewthing/20130612-00

June 12, 2013

Raymond Chen

Last time, we learned about how connection points work. One special case of this is where the connection interface is a dispatch interface.

Dispatch interfaces are, as the name suggests, COM interfaces based on `IDispatch`. The `IDispatch` interface is the base interface for OLE automation objects, and if you want your connection point interface to be usable from script, you probably should make it a dispatch interface.

I'm assuming you know how `IDispatch` works. The short version is that script that wants to invoke a method or property calls `GetIDsOfNames` to get the *dispatch ID* for the method or property it wants to access, and it uses the type library to figure out things like the parameters and return value. Once the scripting engine figures out how the method or property expects to be called, it can call `IDispatch::Invoke` passing the dispatch ID and a `DISPPARAMS` structure that holds the parameters.

Nowadays, this sort of thing goes by the fancy name of *reflection*, but back in the OLE Automation days, it was simply all in a day's work. *You kids think you invented everything*.

Just like as with regular connection point interfaces, a dispatch interface used as a connection point interface consists of events which are formally implemented as methods.

```
dispinterface DWidgetEvents
{
 [id(WDISPID_RENAMED)]
 HRESULT Renamed([in] BSTR oldName, [in] BSTR newName);
…
};
```

You declare that your object is a source of events for this interface by noting it in your object declaration. (Thanks, Medinoc for noting the error in the original version of this article.)

```
coclass Widget
{
 [default] interface IWidget;
 [default, source] dispinterface DWidgetEvents;
}
```

A client registers with the connection point with the `DIID_DWidgetEvents` interface. By convention, dispatch interfaces usually end with the word `Events` and are often prefixed with the letter `D`, and the interface ID symbol begins with `DIID` rather than simply `IID`. These conventions are not universally adhered-to, so don't freak out if you see people who don't follow them. (If you declare your dispatch interface in an IDL file, then the MIDL compiler will generate the dispatch interface ID with the `DIID` prefix for you.)

Now, formally, when the connection point wants to invoke the `Renamed` method, it calls `GetIDsOfNames` to get the ID for the method called `L"Renamed"`, and asks for the type library to figure out what the parameters are. But this is frequently just pointless busy-work: The connection point often already knows the answer, since the connection point already knows what interface it is talking to. It doesn't need to do any "reflection" since the connection point already knows what the IDs and calling conventions are. In the same way, your C# code doesn't need to use reflection to call a method on an object whose assembly you already have referenced in your project. (The `GetIDsOfNames` exists not for connection points, but rather to assist dynamically-typed languages, where you can try to invoke any method on any object, and the method is looked up at run time.)

In other words, the connection point already knows that the ID for the method `Rename` is `WDISPID_RENAMED`, and that it takes two `BSTR` parameters, because that was part of the contract for registering with the connection point in the first place.

This means that in practice, the only method on the client that is ever called is `IDispatch::Invoke`.

Here is a template base class that I use for my connection point interface implementations of dispatch interfaces. We'll discuss the pieces afterwards:

```
template<typename DispInterface>
class CDispInterfaceBase : public DispInterface
{
public:
 CDispInterfaceBase() : m_cRef(1), m_dwCookie(0) { }


 /* IUnknown */
 IFACEMETHODIMP QueryInterface(REFIID riid, void **ppv)
 {
  *ppv = nullptr;
  HRESULT hr = E_NOINTERFACE;
  if (riid == IID_IUnknown || riid == IID_IDispatch ||
      riid == __uuidof(DispInterface))
  {
   *ppv = static_cast<DispInterface *>
          (static_cast<IDispatch*>(this));
   AddRef();
   hr = S_OK;
  }
  return hr;
 }


 IFACEMETHODIMP_(ULONG) AddRef()
 {
  return InterlockedIncrement(&m_cRef);
 }


 IFACEMETHODIMP_(ULONG) Release()
 {
  LONG cRef = InterlockedDecrement(&m_cRef);
  if (cRef == 0) delete this;
  return cRef;
 }


 // *** IDispatch ***
 IFACEMETHODIMP GetTypeInfoCount(UINT *pctinfo)
 {
  *pctinfo = 0;
  return E_NOTIMPL;
 }


 IFACEMETHODIMP GetTypeInfo(UINT iTInfo, LCID lcid,
                            ITypeInfo **ppTInfo)
 {
  *ppTInfo = nullptr;
  return E_NOTIMPL;
 }
```

```
 IFACEMETHODIMP GetIDsOfNames(REFIID, LPOLESTR *rgszNames,
                              UINT cNames, LCID lcid,
                              DISPID *rgDispId)
 {
  return E_NOTIMPL;
 }


 IFACEMETHODIMP Invoke(
    DISPID dispid, REFIID riid, LCID lcid, WORD wFlags,
    DISPPARAMS *pdispparams, VARIANT *pvarResult,
    EXCEPINFO *pexcepinfo, UINT *puArgErr)
 {
  if (pvarResult) VariantInit(pvarResult);
  return SimpleInvoke(dispid, pdispparams, pvarResult);
 }


 // Derived class must implement SimpleInvoke
 virtual HRESULT SimpleInvoke(DISPID dispid,
    DISPPARAMS *pdispparams, VARIANT *pvarResult) = 0;


public:
 HRESULT Connect(IUnknown *punk)
 {
  HRESULT hr = S_OK;
  CComPtr<IConnectionPointContainer> spcpc;
  if (SUCCEEDED(hr)) {
   hr = punk->QueryInterface(IID_PPV_ARGS(&spcpc));
  }
  if (SUCCEEDED(hr)) {
  hr = spcpc->FindConnectionPoint(__uuidof(DispInterface), &m_spcp);
  }
  if (SUCCEEDED(hr)) {
  hr = m_spcp->Advise(this, &m_dwCookie);
  }
  return hr;
 }


 void Disconnect()
 {
  if (m_dwCookie) {
   m_spcp->Unadvise(m_dwCookie);
   m_spcp.Release();
   m_dwCookie = 0;
  }
 }
```

```
private:
 LONG m_cRef;
 CComPtr<IConnectionPoint> m_spcp;
 DWORD m_dwCookie;
};
```

First, a distraction: Our `QueryInterface` implementation performs a double-cast of `this` to `IDispatch`, then to the templated interface. This ensures that the templated interface pointer and `IDispatch` are compatible. It would be bad if somebody tried to use this `QueryInterface` implementation with something unrelated to `IDispatch`. (Yes, I could've used `std::is_base_of`, but I'm an old-timer who grew up before TR1.)

The bulk of the class merely stubs out all the methods of `IDispatch`, save for `IDispatch::Invoke`, which does a little grunt work (initializing the result `VARIANT`) and then leaves the derived class to do the heavy lifting.

Finally, there are two public methods `Connect` and `Disconnect`. These perform the `Advise` and `Unadvise` calls we saw yesterday. To simplify things for our caller, we save the `IConnectionPointer` we registered against so that the caller doesn't have to pass it back in when disconnecting.

Exercise: Is the `m_spcp.Release()` call in `Disconnect` really necessary? (Assuming that `Connect` is called at most once.)

This helper template class makes writing dispatch interface connection point clients really simple, since all you have to do is implement `SimpleInvoke` in the form of a `switch` statement on the dispatch IDs you care about:

```
class CWidgetClient : public CDispInterfaceBase
{
public:
 CWidgetClient() { }


 HRESULT SimpleInvoke(
    DISPID dispid, DISPPARAMS *pdispparams, VARIANT *pvarResult)
{
 switch (dispid) {
 case WDISPID_RENAMED:
  HeyLookItGotRenamed(pdispparams->rgvarg[0].bstrVal,
                      pdispparams->rgvarg[1].bstrVal);
  break;
 }
 return S_OK;
};
```

In the `SimpleInvoke` method, we switch on the dispatch ID, and if we see one we like, we extract the parameters from the `pdispparams`.

Update: Commenter parkrrr points out a huge gotcha with the `DISPPARAMS` structure: The parameters are passed in *reverse* order. I don't know why. They just are.

Next time, we'll start hooking up events to our Little Program so it can update when the user navigates an Explorer or Internet Explorer window.

Warning! Managed code! The CLR understands the connection point/dispatch interface convention and exposes a dispatch event to managed code in the form of a CLR event and corresponding delegate. For example, our `Renamed` event is exposed as an event called `Renamed`, with delegate type `DWidgetEvents_RenamedEventHandler`. You can listen on the event the way you listen to any other CLR event: `widget.Renamed += this.OnRenamed;`.

Note: I completely ignored the subject of dual interfaces. You can read about those if you like, but we won't need to know about them for the job at hand.

Raymond Chen

Follow