An introduction to COM connection points

devblogs.microsoft.com/oldnewthing/20130611-00

June 11, 2013

Raymond Chen

Last time, we saw how to enumerate all the Internet Explorer and Explorer Windows and see what they are viewing. But that program printed static information. It didn't track the changes to the windows if the user clicked to another Web page or navigated to a different folder.

In order to hook that up, we need to understand the connection point model and the way events are expressed in dispatch interfaces. First, let's look at the connection point model. These topics confused me when I first met them (in part because I didn't do a good job of mentally separating them into two topics and instead treated it as one big topic), so I'm going to spend a few days trying to explain how it works, and then later this week, we'll actually hook things up. (And actually hooking it up is a lot easier than explaining it.)

Today, the connection point model.

Suppose you have a widget which can have multiple clients. The clients can communicate with the widget by invoking methods on the widget, like `IWidget::SetColor`. but how does the widget communicate with its clients? Well, since this is COM, the first thing you need is an interface, say, `IWidgetClient`. The idea is that each client implements `IWidget-Client`, and when the widget needs to, say, notify each client that the color changed, it can invoke `IWidgetClient::OnColorChanged` on each one. Each client can register with the widget for notifications.

The COM interface for standardizing the registration mechanism is `IConnectionPoint`. A *connection point* acts as a middle-man between the widget and all its clients: Whenever the widget needs to notify all its clients, it tells the connection point to do it.

Widget		Connection Point	→	Client A
	→		→	Client B

			→	Client C

A client registers with a connection point by calling `IConnectionPoint::Advise`, and it unregisters by calling `IConnectionPoint::Unadvise`.

Okay, that’s great, but how do clients find the connection point so they can register with it?

The widget exposes an interface known as `IConnectionPointContainer` which provides access to an object’s connection points. The client can call the `IConnectionPointContainer::FindConnectionPoint` method to get access to a specific connection point.

Here’s how the pieces fit together:

```
// error checking elided for expository purposes


void IUnknown_FindConnectionPoint(IUnknown *punk,
                                  REFIID riid,
                                  IConnectionPoint **ppcp)
{
 // get the IConnectionPointContainer interface
 CComQIPtr<IConnectionPointContainer> spcpc(punk);


 // Locate the connection point
 spcpc->FindConnectionPoint(riid,  ppcp);
}


class CClient : public IWidgetClient
{
…
 IWidget *m_pWidget;
 DWORD m_dwCookie;
};


CClient::RegisterWidgetClient()
{
 // Find the IWidgetClient connection point
 CComPtr<IConnectionPoint> spcp;
 IUnknown_FindConnectionPoint(m_pWidget,
                              IID_IWidgetClient, &spcp);


 // register with it
 spcp->Advise(this, &m_dwCookie);
}


CClient::UnregisterWidgetClient()
{
 // Find the IWidgetClient connection point
 CComPtr<IConnectionPoint> spcp;
 IUnknown_FindConnectionPoint(m_pWidget,
                              IID_IWidgetClient, &spcp);


 // unregister from it
 spcp->Unadvise(m_dwCookie);
}
```

After registering as a widget client, the `CClient` object will receive method calls on its `IWidgetClient` until it unregisters.

Now the widget and clients have two-way communication. If the clients want to initiate the communuication, it can call a method on `IWidget`. If the widget wants to initiate the communication, it can call a method on `IWidgetClient`.

Note that we've created a giant circular reference. The widget has a reference to its connection point (so it can tell it to fire a notification to all its clients), and the connection point has a reference to the widget client (so it can forward the notification along), and the widget client has a reference to the widget in its `m_pWidget` member. In order to break this cycle, you have to remember to explicitly call `UnregisterWidgetClient` when you are no longer interested in receiving widget notifications.

Note that even though the arrows in the diagram above flow from left to right (from widget to clients), that doesn't mean that the information flow is strictly left-to-right. You can pass information in the other direction via return values or output parameters.

For example, there might be a method on the `IWidgetClient` interface called `GetColor`:

```
interface IWidgetClient : IUnknown
{
 …
 HRESULT GetColor([out] COLORREF *pclr);
 …
};
```

Since there can be multiple clients, the widget needs to have some sort of rule for deciding which client gets to choose the color. It might decide to ask each client in turn for a color, until one of them returns `S_OK`, and that client's color is used and no further clients are notified.

Or maybe there's a method called `OnSave`:

```
interface IWidgetClient : IUnknown
{
 …
 HRESULT OnSave([in] IPropertyStorage *pps);
 …
};
```

The convention here might be that all clients will be notified of the Save operation and they can write any additional information to the `IPropertyStorage` while handing the notification.

Those are just examples. Feel free to make up your own. The point is that just because the arrows go from the widget to the clients doesn't mean that information can't flow back the other way.

Most of the time, you have the simple case where a widget will expose a single connection point. In that case, the generality of the `IConnectionPointContainer` may seem unnecessary. But it allows you to add new connection points later. For example, you might have multiple client interfaces for different types of clients. You could have `IWidgetColorClient` for clients that are interested only in color changes, and `IWidgetNetworkClient` for clients that are interested only in monitoring the widget's network activity.

Or maybe you didn't plan on having multiple connection points originally, but in the second version of your product, you want to add additional methods to `IWidgetClient`, so you need to create `IWidgetClient2`, which means that you also need a new connection point for it.

Next time, a look at the special case where the client interface is a dispatch interface.

Raymond Chen

Follow