A very brief introduction to patterns for implementing a COM object that hands out references to itself

devblogs.microsoft.com/oldnewthing/20211025-00

October 25, 2021

Raymond Chen

A common scenario for a COM object is that it needs to register itself as a callback or otherwise hand out references to itself. There are a few patterns for this.

One of the first things you have to decide is whether the reference to the main object should be strong (keep the main object alive) or weak (not be considered for deciding whether the main object should be kept alive).

Let's look at the weak pattern first.

If you don't want the callback registration to keep the object alive, then you typically give the callback object a weak reference to the main object. When the callback object is invoked, it converts the weak reference to a strong reference and calls the main object. Since the reference is weak, the existence of the callback object has no effect on the lifetime of the main object.

					Callback			
					IUnknown	ICallback	←	event source
		Widget			refcount			
client	→	IUnknown	IWidget	←	weak reference			
		refcount						
		state data						

In this model, the only reference counts on the main Widget object come from the client. When the client releases their last reference, the main Widget destructs, and the main Widget object unregisters the callback from the event source, thereby causing the Callback object to destruct as well.

Another pattern is to factor the *state data* into a shared object:

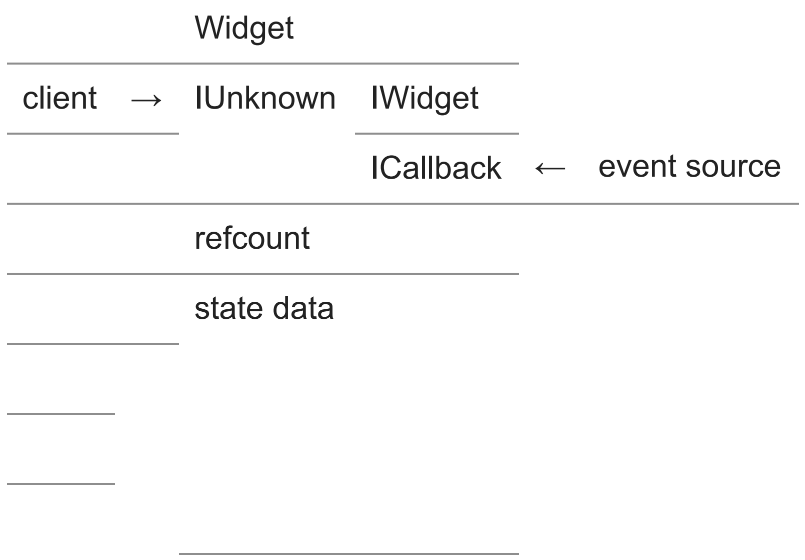
		Widget					Callback			
client	→	IUnknown	IWidget				IUnknown	ICallback	←	event source
		refcount					refcount			
		shared_ptr		→	state data	←	some_ptr			

If the `some_ptr` is a weak reference, then we have the same pattern as before. But if you make it a `shared_ptr`, then the callback will keep the state alive. This might be useful if the Callback is used to signal the completion of some sort of activity, and you need to keep the state data alive until the activity is complete, but you might want the destruction of the Widget to issue a Cancel request to the event source to accelerate the completion of the activity.

In the strong pattern, the callback registration keeps the main object alive. The usual way of doing this is to just upgrade our first diagram to use a strong reference from the Callback object to the main object.

					Callback			
					IUnknown	ICallback	←	event source
		Widget			refcount			
client	→	IUnknown	IWidget	←	strong reference			
		refcount						
		state data						

You might then realize that there's no point creating a separate callback object: The main object can be its own callback.



This is a very common pattern, but it does introduce a few problems compared to the separate-callback-object pattern.

One problem is that the client can `QueryInterface` for `ICallback` and use that to manipulate the Widget in ways that weren't intended, since it can invoke the callback from the client and feed it fake data. (Conversely, the event source can `QueryObject` for `IWidget`, but that is unlikely to occur in practice, since the event source doesn't care about widgets.) With the separate callback, the `QueryInterface` method main object responds only to `IWidget`, and the `QueryInterface` method callback object responds only to `ICallback`. The client has no way to access the callback object, and the event source has no way to access the main object.

A second problem is if the main object needs to register for multiple event sources, all of which use the same `ICallback` interface. There is only one `ICallback` in the main object, so it has no choice but to pass the same `ICallback` to both event sources, even though we may want the two callbacks to do different things. Again, the separate callback object avoids this problem because each callback object can do something different when it is called.

But there's still a solution to this problem without having to go back to the separate callback objects. We'll look at this trick next time, and fleshing out the idea will give us a tour through a lot of C++ features, so it might be interesting even if you don't care about COM.

Raymond Chen

Follow