Notes on COM aggregation: Obtaining a pointer to your aggregated partner without introducing a reference cycle

devblogs.microsoft.com/oldnewthing/20220210-00

February 10, 2022

Raymond Chen

COM aggregation allows multiple objects to work together so that they appear to be a single object. It is one of those advanced topics most people never deal with. It just works invisibly.

There are many other write-ups of COM aggregation, so I'll leave you to learn the basics somewhere else. The short version is that the two parties in COM aggregation are informally called the *outer* and *inner* objects. The object that is being aggregated is the *inner* object, and the object that is doing the aggregating is the *outer* object. The outer object formally goes by the name *controlling unknown*, a name with a rather nefarious secret-society ring that has not gone unnoticed.

Most of the responsibility for making aggregation work lies on the inner object. The inner object exposes a special non-delegating `IUnknown` to the outer object. This non-delegating `IUnknown` is a "for your eyes only" interface exclusively for the outer object. It is how the outer object accesses the interfaces of the inner object. The non-delegating `IUnknown` goes like this:

- QueryInterface for `IUnknown`: Return the non-delegating IUnknown and call its `AddRef` (which we will see below increments the reference count of the *inner* object).
- QueryInterface for anything else: Obtain an interface (if supported) from the inner object as a delegating interface (see below), and call the delegating interface's `AddRef`. (As we'll see below, this delegating `AddRef` increments the reference count of the *outer* object.)
- AddRef: Increment the reference count of the *inner* object.
- Release: Decrement the reference count of the *inner* object, and destroy the inner object if the reference count is zero.

Whenever the inner object hands out a non- `IUnknown` interface from its non-delegating QueryInterface, the `IUnknown` methods on the resulting interface are *delegating*:

- QueryInterface: Forward to the outer object.
- AddRef: Forward to the outer object.

- Release: Forward to the outer object.

It is this forwarding that makes the inner object appear to be part of the outer object. Whenever anybody (other than the outer object) asks for information about the inner object, the request is always forwarded to the outer object for consistent handling.

The outer object's `QueryInterface` method will look at the interface being requested and classify it in one of three buckets:

- Handled by the outer object: Return a pointer to the outer object's interface and increment the reference count of the outer object.
- Handled by the inner object: Query the non-delegating inner object for the requested interface. This will also increment the reference count of the outer object.
- Handled by neither: Fail.

If the outer object needs to obtain an interface from the inner object temporarily, it can query the inner object for that interface (which will increment the outer object's reference count), use the interface, and then release it (which will decrement the outer object's reference count). At the end of the sequence, everything has returned to normal.

Things get weird if the outer object wants to obtain an interface from the inner object for an extended period of time. This is common if the outer object intends to use the interface a lot, so it wants to query once and just cache the result. If it followed the "temporary" usage pattern above, it would end up with a reference cycle: Querying the inner object for the interface increments the outer object's reference count, so the object is indirectly holding a reference to itself, which means that the object will never destruct, even if all external clients release their references.

The fact that you have a circular reference is more obvious if you remember that the point of aggregation is to make two objects appear to be one, so what the outer object did was query *itself* for the interface, which naturally creates a reference cycle.

In order to break this reference cycle, you need to perform an artificial `Release` on the outer object to make the net change zero.

When you want to clean up that secret internal reference, you need to perform the steps in reverse: `AddRef` the outer object, and then `Release` the inner interface. This part of the trick requires the outer object to set an artificial reference count during destruction to avoid accidental double-destruction.

This same logic works in the other direction, too: The inner object can query its outer for an interface to use temporarily, releasing it when finished. If the inner object wants to query its outer for an interface and cache it, then it needs to perform a `Release` on the outer object (not on the queried interface) to counteract the reference count increment that resulted from the `QueryInterface`. When releasing the interface, the same reversal algorithm applies: `AddRef` the outer object and then release the queried interface.

The code sequence therefore goes like this:

```
// outer querying inner and caching the result

if (SUCCEEDED(m_inner->QueryInterface(IID_PPV_ARGS(&m_cachedInner))) {
    this->Release();
}

// outer releasing cached inner interface
this->AddRef();
m_cachedInner->Release();

// inner querying outer and caching the result

if (SUCCEEDED(m_outer->QueryInterface(IID_PPV_ARGS(&m_cachedOuter))) {
    m_outer->Release();
}

// inner releasing cached outer interface

m_outer->AddRef();
m_cachedOuter->Release();
```

This all looks great and seems to work, until you realize that there's a corner case you missed: Tear-offs.

We'll look more closely at the interaction between aggregation and tear-offs next time.

Bonus chatter: Weak `QueryInterface` in COM aggregation was the topic I was referring to when I mentioned that Don Box told me that it was too advanced even for his advanced book on COM.

Raymond Chen

Follow