The ways people mess up IUnknown::QueryInterface, episode 4

devblogs.microsoft.com/oldnewthing/20110811-00

August 11, 2011

Raymond Chen

One of the rules for `IUnknown::QueryInterface` is so obvious that nobody even bothers to state it explicitly as a rule: "If somebody asks you for an interface, and you return `S_OK`, then the pointer you return must point to the interface the caller requested." (This feels like the software version of dumb warning labels.)

During compatibility testing for Windows Vista, we found a shell extension that behaved rather strangely. Eventually, the problem was traced to a broken `IUnknown::QueryInterface` implementation which depended subtly on the order in which interfaces were queried.

The shell asked for the `IExtractIconA` and `IExtractIconW` interfaces in the following order:

```
// not the actual code but it gets the point across
IExtractIconA *pxia;
IExtractIconW *pxiw;
punk->QueryInterface(IID_IExtractIconA, &pxia);
punk->QueryInterface(IID_IExtractIconW, &pxiw);
```

One particular shell extension would return the same pointer to both queries; *i.e.*, after the above code executed, `pxia == pxiw` even though neither interface derived from the other. The two interfaces are not binary-compatible, because `IExtractIconA::GetIconLocation` operates on ANSI strings, whereas `IExtractIconW::GetIconLocation` operates on Unicode strings.

The shell called `pxiw->GetIconLocation`, but the object interpreted the `szIconFile` as an ANSI string buffer; as a result, when the shell went to look at it, it saw gibberish.

Further experimentation revealed that if the order of the two `QueryInterface` calls were reversed, then `pxiw->GetIconLocation` worked as expected. In other words, the first interface you requested "locked" the object into that interface, and asking for any other

interface just returned a pointer to the locked interface. This struck me as very odd; coding up the object this way seems to be *harder* than doing it the right way!

```
// this code is wrong - see discussion above
class CShellExtension : public IExtractIcon
{
 enum { MODE_UNKNOWN, MODE_ANSI, MODE_UNICODE };
  HRESULT CShellExtension::QueryInterface(REFIID riid, void **ppv)
  {
   *ppv = NULL;
   if (riid == IID_IUnknown) *ppv = this;
   else if (riid == IID_IExtractIconA) {
    if (m_mode == MODE_UNKNOWN) m_mode = MODE_ANSI;
    *ppv = this;
   } else if (riid == IID_IExtractIconW) {
    if (m_mode == MODE_UNKNOWN) m_mode = MODE_UNICODE;
    *ppv = this;
   }
   if (*ppv) AddRef();
   return *ppv ? S_OK : E_NOINTERFACE;
  }
  ... AddRef / Release ...
  HRESULT GetIconLocation(UINT uFlags, LPTSTR szIconFile, UINT cchMax,
                          int *piIndex, UINT *pwFlags)
  {
   if (m_mode == MODE_ANSI) lstrcpynA((char*)szIconFile, "foo", cchMax);
   else lstrcpynW((WCHAR*)szIconFile, L"foo", cchMax);
   ...
  }
  ...
}
```

Instead of implementing both `IExtractIconA` and `IExtractIconW`, my guess is that they implemented just one of the interfaces and made it alter its behavior based on which interface it thinks it needs to pretend to be. It never occurred to them that the single interface might need to pretend to be two different things at the same time.

The right way of supporting two interfaces is to actually implement two interfaces and not write a single morphing interface.

```
class CShellExtension : public IExtractIconA, public IExtractIconW
{
  HRESULT CShellExtension::QueryInterface(REFIID riid, void **ppv)
  {
   *ppv = NULL;
   if (riid == IID_IUnknown ||
       riid == IID_IExtractIconA) {
    *ppv = static_cast<IExtractIconA*>(this);
   } else if (riid == IID_IExtractIconW) {
    *ppv = static_cast<IExtractIconW*>(this);
   }
   if (*ppv) AddRef();
   return *ppv ? S_OK : E_NOINTERFACE;
  }
  ... AddRef / Release ...
  HRESULT GetIconLocation(UINT uFlags, LPSTR szIconFile, UINT cchMax,
                          int *piIndex, UINT *pwFlags)
  {
   lstrcpynA(szIconFile, "foo", cchMax);
   return GetIconLocationCommon(uFlags, piIndex, pwFlags);
  }
  HRESULT GetIconLocation(UINT uFlags, LPWSTR szIconFile, UINT cchMax,
                          int *piIndex, UINT *pwFlags)
  {
   lstrcpynW(szIconFile, L"foo", cchMax);
   return GetIconLocationCommon(uFlags, piIndex, pwFlags);
  }
  ...
}
```

We worked around this in the shell by simply changing the order in which we perform the calls to `IUnknown::QueryInterface` and adding a comment explaining why the order of the calls is important.

(This is another example of how the cost of a compatibility fix is small potatoes. The cost of deciding whether or not to apply the fix far exceeds the cost of just doing it for everybody.)

A different shell extension had a compatibility problem that also was traced back to a dependency on the order in which the shell asked for interfaces. The shell extension registered as a context menu extension, but when the shell tried to create it, it got `E_NOINTERFACE` back:

```
CoCreateInstance(CLSID_YourAwesomeExtension, NULL,
                 CLSCTX_INPROC_SERVER, IID_IContextMenu, &pcm);
// returns E_NOINTERFACE?
```

This was kind of bizarre. I mean, the shell extension went to the effort of registering itself as a context menu extension, but when the shell said, “Okay, it’s show time, let’s do the context menu dance!” it replied, “Sorry, I don’t do that.”

The vendor explained that the shell extension relies on the order in which the shell asked for interfaces. The shell used to create and initialize the extension like this:

```
// error checking and other random bookkeeping removed
IShellExtInit *psei;
IContextMenu *pcm;
CoCreateInstance(CLSID_YourAwesomeExtension, NULL,
                 CLSCTX_INPROC_SERVER, IID_IShellExtInit, &psei);
psei->Initialize(...);
psei->QueryInterface(IID_IContextMenu, &pcm);
psei->Release();
// use pcm
```

We changed the order in a manner that you would think should be equivalent:

```
CoCreateInstance(CLSID_YourAwesomeExtension, NULL,
                 CLSCTX_INPROC_SERVER, IID_IContextMenu, &pcm);
pcm->QueryInterface(IID_IShellExtInit, &psei);
psei->Initialize(...);
psei->Release();
```

(Of course, it's not written in so many words in the code; the various parts are spread out over different components and helper functions, but this is the sequence of calls the shell extension sees.)

The vendor explained that their shell extension will not respond to any shell extension interfaces (aside from `IShellExtInit`) until it has been initialized, because it is at that point that they decide which extensions they want to support. Unfortunately, this violates the first of the four explicit rules for `IUnknown::QueryInterface`, namely that the set of interfaces must be static. (It's okay to have an object expose different interfaces conditionally, as long as it understands that once it says yes or no to a particular interface, it is committed to answering the same way for the lifetime of the object.)

Raymond Chen

Follow