How to support a COM interface conditionally in C++/WinRT

devblogs.microsoft.com/oldnewthing/20231026-00

October 26, 2023

Raymond Chen

We saw some time ago that the C++/WinRT library provides an extension point for adding additional interfaces that aren't declared in your `implements` template. But what about the reverse case? What if you want to *remove* an interface that was listed in your `implements` template?

You might want to do this if you want to support an interface only conditionally. Just be careful to follow the rules: Every attempt to query for a specific interface from an object must return a consistent result (either always succeed for that interface or always fail for that interface).

The extension point for removing an interface in C++/WinRT is the `find_interface` virtual method: We can override it to filter out interfaces we don't like.

```
struct Widget : winrt::implements<
    Widget, winrt::IWidget, winrt::IStringable>
{
    void* find_interface(winrt::guid const& id)
        const noexcept override
    {
        // If "Stringable" is not enabled,
        // then don't support IStringable.
        if (id == winrt::guid_of<winrt::IStringable>() &&
           !is_stringable_enabled())
        {
            return nullptr;
        }
        return implements::find_interface(id);
    }

    // Implement IWidget methods
    void WidgetMethod();

    // Implement IStringable methods
    winrt::hstring ToString();
};
```

When a query comes in for `IStringable`, we check whether `IStringable` support is enabled. If not, then we return `nullptr` immediately, which causes the `QueryInterface` to fail with `E_NOINTERFACE`. Otherwise, we forward the call to the base class to continue normally.

As I noted before, according to COM rules, once you decide whether or not `IStringable` is supported, you have to stick with that decision for the lifetime of the object. The imaginary `is_stringable_enabled()` function should be based on some immutable state, or at least state which *becomes* immutable once the `is_stringable_enabled()` function is called.

Now, there is also a method on `IInspectable` called `GetIids` which returns a list of the Windows Runtime interfaces implemented by an object. Shouldn't we also remove `IStringable` from that list?

I guess you could do that, but it wouldn't help much. The `GetIids` method is used for runtime reflection. However, a much richer way to get the Windows Runtime interfaces implemented by an object is to ask the object for its runtime class name (`GetRuntimeClassName`) and then look up that name in the Windows metadata (winmd) file. The winmd file will tell you all the interfaces which the object implements, and it will be in the form of strings, not IIDs. You can then look up those interfaces in Windows metadata files to see what their methods are. On the other hand, `GetIids` just gives you a bunch of IIDs, and there's no practical way to get an interface's name and methods from its IID.

Even though we could remove the IID from those reported by `GetIids`, we can't remove it from the interfaces reported by the Windows metadata files, since those are just static data files. We may as well just allow the interface to be reported and fail the query later. A debugger might show the methods on that interface, and then when the developer tries to call them, the call will fail. Not the best experience, but not the end of the world.

Bonus chatter: A template for this pattern exists in the Windows Implementation Library under the name `wil::winrt_conditionally_implements`.