A sample implementation of the weak reference pattern for COM callbacks

devblogs.microsoft.com/oldnewthing/20250213-00

February 13, 2025

Some time ago, I shared some patterns for implementing a COM object that hands out references to itself, and the first was something I called the “weak pattern”.

As a reminder, the weak pattern uses a separate callback object that holds a weak reference to the main object and forwards method calls to the main object.

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

Let’s flesh this out with an actual implementation.

For concreteness, let’s say that the main Widget object is implemented in C++/WinRT, and there is a callback interface called something like `IGadgetCallback`. (Applying the same logic to other libraries is left as an exercise.)

We start with the gadget and its callback.

```
// gadgets.idl
interface IGadgetCallback : IUnknown
{
    HRESULT ColorChanged([in] COLORREF oldColor,
                         [in] COLORREF newColor);
    HRESULT NameChanged([in] PCWSTR oldName,
                        [in] PCWSTR newName);
    HRESULT Refreshed();
};

interface IGadget : IUnknown
{
    [ other Gadget methods ]

    HRESULT SetCallback([in] IGadgetCallback* callback);
};
```

The idea is that after you create a Gadget, you can use the `SetCallback()` method to pass an object that receives gadget callbacks for various things that might happen.

Here's how we can create a Widget that sets a callback on the Gadget that calls back into the Widget, but without creating a circular reference.

```
namespace winrt::Contoso::implementation
{
    struct Widget : WidgetT<Widget>
    {
        void InitializeComponent();

        ⟦ other Widget methods ⟧

        // Used by WidgetGadgetCallback
        void ColorChanged(COLORREF oldColor,
                          COLORREF newColor);
        void NameChanged([PCWSTR oldName,
                          PCWSTR newName);
        void Refreshed();
    private:

        winrt::com_ptr<IGadget> m_gadget =
            winrt::create_instance(CLSID_Gadget);
    };

    struct WidgetGadgetCallback :
        winrt::implements<WidgetGadgetCallback,
                          IGadgetCallback>
    {
        WidgetGadgetCallback(implementation::Widget* widget);

        IFACEMETHODIMP ColorChanged(COLORREF oldColor,
                                    COLORREF newColor);
        IFACEMETHODIMP NameChanged(PCWSTR oldName,
                                   PCWSTR newName);
        IFACEMETHODIMP Refreshed();

    private:
        winrt::weak_ref<implementation::Widget> m_weakWidget;

        ⟦ more to come ⟧
    };
}
```

The `IGadgetCallback` is implemented not by the `Widget` but by the `WidgetGadgetCallback`, and the idea is that the `WidgetGadgetCallback` forwards the method calls to the corresponding methods on the `Widget`.

Let's hook them up. When the `Widget` initializes, it creates the callback object and sets it as the callback.

```
void Widget::InitializeComponent()
{
    winrt::check_hresult(m_gadget->SetCallback(
        winrt::make<WidgetGadgetCallback>(this).get());
}
```

The real work happens in the callback object.

The constructor takes the incoming `Widget*` and obtains a weak reference to it.

```
WidgetGadgetCallback::WidgetGadgetCallback(
        implementation::Widget* widget) :
    m_weakWidget(widget->get_weak()) {}
```

When a method on the callback interface is called, we want to forward the call back to the `Widget`. To reduce typing, we'll templatize the pattern.

```
struct WidgetGadgetCallback :
    winrt::implements<WidgetGadgetCallback,
                      IGadgetCallback>
{
    ⟦ existing public methods ⟧

private:
    winrt::weak_ref<implementation::Widget> m_weakWidget;

    template<auto Method, typename... Args>
    HRESULT Forward(Args&&... args) try
    {
        if (auto strong = m_weakWidget.get()) {
            ((*strong).*Method)(
                std::forward<Args>(args)...);
        }
        return S_OK;
    }
    catch (...)
    {
        return winrt::to_hresult();
    }
};
```

The pattern for forwarding the call is to try to resolve the weak reference to a strong reference, and if it succeeds, call the provided method while forwarding the parameters. We also do some adapting between the `HRESULT`-based COM call and the exception-based method back in the `Widget` object. (This was a quirk of this particular example. Your version may not require adapting.)

Now, we happen to know that all of the parameters to the COM callback methods are cheap to copy, so we could pass them along by value:

```
    template<auto Method, typename... Args>
    HRESULT Forward(Args... args) try
    {
        if (auto strong = m_weakWidget.get()) {
            ((*strong).*Method)(args...);
        }
        return S_OK;
    }
    catch (...)
    {
        return winrt::to_hresult();
    }
```

The last piece of the puzzle is telling each method to do the forwarding.

```
struct WidgetGadgetCallback :
    winrt::implements<WidgetGadgetCallback,
                      IGadgetCallback>
{
    ⟦ ... ⟧

    IFACEMETHODIMP ColorChanged(COLORREF oldColor,
                                COLORREF newColor)
    {
        return Forward<&implementation::Widget::ColorChanged>(
            oldColor, newColor);
    }

    IFACEMETHODIMP NameChanged(PCWSTR oldName,
                               PCWSTR newName);
    {
        return Forward<&implementation::Widget::NameChanged>(
            oldName, newName);
    }

    IFACEMETHODIMP Refreshed();
    {
        return Forward<&implementation::Widget::Refreshed>();
    }

    ⟦ ... ⟧
};
```

We invoke the `Forward` method with the function we want to forward to and the arguments we want to forward.

In our example, we gave the forwarded-to method the same name as the forwarded-from method, but there's no requirement that you do that. You just have to provide the correct method name to the `Forward` template function.