If you’re going to wrap a Windows Runtime event, you may as well let the wrapped event source manage the token

devblogs.microsoft.com/oldnewthing/20221208-00

December 8, 2022

Raymond Chen

Suppose that you have a Windows Runtime object with an event that is just a proxy for an event on some other object. For example, maybe we have this:

```
runtimeclass Widget
{
    event Windows.Foundation.TypedEventHandler<Widget, Object> Changed;

    /* ... other members ... */
}

runtimeclass Dashboard
{
    event Windows.Foundation.TypedEventHandler<Dashboard, Object> Changed;

    /* ... other members ... */
}
```

Secretly, a `Dashboard` uses a `Widget` to do the real work, and the `Changed` event on the `Dashboard` is just a wrapper around the `Widget` ‘s `Changed` event. The `Dashboard` registers for the `Widget` ‘s `Changed` event, and the event handler turns around and raises the `Dashboard` ‘s `Changed` event.

```
namespace winrt::Namespace::implementation
{
    struct Dashboard : DashboardT<Dashboard>
    {
        Namespace::Widget const m_widget;
        event<TypedEventHandler<Namespace::Dashboard, IInspectable> m_changed;

        Dashboard()
        {
            m_widget.Changed({ get_weak(), &Dashboard::OnChanged });
        }

        auto Changed(TypedEventHandler<Namespace::Dashboard, IInspectable> const&
handler)
        {
            return m_changed.add(handler);
        }

        void Changed(event_token token)
        {
            return m_changed.remove(token);
        }

        void OnChanged(Namespace::Widget const&, IInspectable const&)
        {
            m_event(*this, nullptr);
        }
    };
}
```

The idea here is that the `Dashboard` listens on the `Changed` event of its private `Widget`, and when the `Widget` changes, it turns around and raises its own `Changed` event to anybody who is listening.

One problem with this approach is that it registers a `Changed` event on the `Widget` even if there is nobody listening to the `Dashboard` ‘s `Changed` event. This can be a problem if the `Widget` ‘s `Changed` event is expensive to subscribe to. For example, detecting changes to a `Widget` may require the creation of a “watcher” object which monitors some system state. If nobody has subscribed to the `Dashboard` ‘s `Changed` event, then there’s no need to run the `Widget` change watcher.

Some people solve this problem by registering on the `Widget` ‘s `Changed` event only when somebody registers for the `Dashboard` ‘s `Changed` event.

```
namespace winrt::Namespace::implementation
{
    struct Dashboard : DashboardT<Dashboard>
    {
        Namespace::Widget const m_widget;
        event<TypedEventHandler<Namespace::Dashboard, IInspectable> m_changed;
        std::once_flag m_subscribe_once;

        Dashboard()
        {
            // m_widget.Changed({ get_weak(), &Dashboard::OnChanged });
        }

        auto Changed(TypedEventHandler<Namespace::Dashboard, IInspectable> const&
handler)
        {
            std::call_once(m_subscribe_once, [&]() {
                m_widget.Changed({ get_weak(), &Dashboard::OnChanged });
            });
            return m_changed.add(handler);
        }

        void Changed(event_token token)
        {
            return m_changed.remove(token);
        }

        void OnChanged(Namespace::Widget const&, IInspectable const&)
        {
            m_event(*this, nullptr);
        }
    };
}
```

This at least delays the registration until needed, but it doesn’t remove the registration when no longer needed. For that, we would have to add some more synchronization:

```
namespace winrt::Namespace::implementation
{
    struct Dashboard : DashboardT<Dashboard>
    {
        Namespace::Widget const m_widget;
        event<TypedEventHandler<Namespace::Dashboard, IInspectable> m_changed;
        std::mutex m_lock;
        winrt::event_token m_token{};

        Dashboard()
        {
        }

        auto Changed(TypedEventHandler<Namespace::Dashboard, IInspectable> const&
handler)
        {
            std::lock_guard guard(m_lock);
            if (!m_changed) {
                m_token = m_widget.Changed({ get_weak(), &Dashboard::OnChanged });
            }
            return m_changed.add(handler);
        }

        void Changed(event_token token)
        {
            std::lock_guard guard(m_lock);
            m_changed.remove(token);
            if (!m_changed) {
                m_widget.Changed(std::exchange(m_token, {}));
            }
        }

        void OnChanged(Namespace::Widget const&, IInspectable const&)
        {
            m_event(*this, nullptr);
        }
    };
}
```

But there’s an easier way.

Just get out of the event business entirely, and let the registrations go all the way to the `Widget`. Let the event registration token be the one from the `Widget`‘s `Changed` event. All you have to do is convert the event parameters when the `Widget`‘s `Changed` event is raised.

```
namespace winrt::Namespace::implementation
{
    struct Dashboard : DashboardT<Dashboard>
    {
        Namespace::Widget const m_widget;
        // event<TypedEventHandler<Namespace::Dashboard, IInspectable> m_changed;

        Dashboard()
        {
        }

        auto Changed(TypedEventHandler<Namespace::Dashboard, IInspectable> const&
handler)
        {
            return m_changed.add([weak = get_weak(), handler](auto&&, auto&&)
            {
                if (auto strong = weak.get()) handler(*strong.get(), nullptr);
            });
        }

        void Changed(event_token token)
        {
            m_widget.Changed(token);
        }

        // void OnChanged(Namespace::Widget const&, IInspectable const&)
        // {
        //     m_event(*this, nullptr);
        // }
    };
}
```

When a client wants to register a `Dashboard` `Changed` event handler, we wrap it inside a `Widget` `Changed` event handler and let the `Widget` deal with it. The wrapper captures a weak reference to the `Dashboard` to avoid a circular reference. When the wrapper is called, it recovers the original `Dashboard` from the weak reference so it can use it as the `sender` when calling the client's original handler.

This approach gives the `Widget` full insight into the clients, so it can perform whatever optimizations it likes. And it's less work for you, too, since you've delegated all the event bookkeeping to the `Widget`.

Raymond Chen

Follow