The "Third Way" of Change Notification: the ValueChangedEventManager and How to Clean Up its Memory Leaks

by R. Stacy Smyth

When change notification is required by a binding but not implemented by the developer, sometimes the framework steps in and implements it automatically, via the ValueChangedEventManager. In terms of supplying the required functionality, this is great. For memory management, it's a disaster. This article explains how it works, and what to do about it.

We all know how change notification is supposed to be implemented:

- Create a dependency object and implement the properties that need change notification as custom dependency properties, or
- Implement INotifyPropertyChanged on a POCO ("plain old CLR object") and raise OnPropertyChanged in the setter of each property requiring change notification.

If you bind to a source property that doesn't use either of these two approaches, you probably won't be surprised to find that change notification isn't working: you change the source property, and the target property stays the same. Specifically, this is what happens if the source object of the binding is a dependency object but the source property is a regular CLR property (not a dependency property) — change notification simply doesn't work.

Surprisingly, though, if the source object of the binding is a POCO instead of a dependency object, and the source property is a plain old CLR property (not a dependency property), change notification *does* work, even when INotifyPropertyChanged isn't implemented — if you change the value of the property, the binding target is updated, just as it should be.

The obvious questions is, “How?” (Which I'll answer shortly.) The not-at-all-obvious question is, “What do we do about it?” Because this kind of change notification causes big problems for memory management.

The "how" is a barely documented change notification mechanism that exists in internal Microsoft code, in addition to the documented mechanisms of dependency properties and INotifyPropertyChanged.

First I'll describe how this barely-mentioned change notification system works, then I'll describe the consequences for memory management. (The "barely mentioned" part is Microsoft Knowledge Base Article 938416, "A memory leak may occur when you use

data binding in Windows Presentation Foundation".)

How ValueChanged Notification Works

When WPF detects that you are setting up a binding that should have change notification from the source (i.e. not a one time-binding or one-way-to-source) but doesn't have change notification implemented, it adds change handling automatically for you.

This is done by adding a change handler to the PropertyDescriptor for the property that needs to generate the change notifications. (The PropertyDescriptor is available through the type object describing the object on which the property resides.) Specifically, the change handler is added via PropertyDescriptor.AddChangeHandler, which takes as its parameters the object on which the property resides and the handler that should be invoked when a change occurs.

However, the intent of this change handler is not merely to provide change notification to this one particular listener (the binding, or rather a PropertyPathWorker within it), but rather to enable change notification in general for this property. As such, the change handler it adds to the PropertyDescriptor is not the ultimate change handler associated with the binding, but rather is an instance of MS.Internal.Data.ValueChangedEventManager.ValueChangedRecord — which in turn can store a whole list of listeners that should be notified whenever the property changes. Not surprisingly, the ultimate change handler associated with the binding is added to this list.

So far, this system seems like a sensible way to do things. That's only because we haven't talked about memory leaks yet.

How ValueChanged Notification Creates Memory Leaks

The irony of what comes next is that it involves the "weak event pattern", which was created specifically to avoid the memory leaks that are commonly associated with event handling.

With typical event handling, handlers are added something like this:

```
someSource.someEvent += someListener.someHandler;
```

This sort of event handling creates a hard reference from the invocation list of someEvent to someListener, with the result that if you never unwire the handler (with -=), someListener is forced to stay in memory as long as someSource has not been garbage collected. It's probably the most common source of memory leaks in WPF.

The weak event pattern was designed to avoid these leaks by creating a way of wiring

up events so that event listeners would only be referenced through weak references, thereby allowing them to have life spans that are unaffected by the sources of the events they are listening to.

If we wanted to use the weak event pattern for a particular event ourselves (the "whatever" event, for example), we would:

1. Derive a new class from WeakEventManager, called WhateverEventManager.
2. Implement IWeakEventListener on the class we want to have listen for this event.
3. Override AddListener and RemoveListener in our new manager class.

When we call AddListener to associate a listener with a source, the manager stores the reference to the listener as a weak reference, so that even if our event source is never garbage collected, the listener can be garbage collected if it is otherwise eligible.

Microsoft has already done this with the ValueChangedEventManager class; they just haven't released any documentation on how it works. But if you dig into it with reflection or a memory profiler, you find out that:

- The ValueChangedEventManager is used to give property descriptors, which usually use AddValueChanged and RemoveValueChanged to manage change notification in a very standard, hard-reference-based way, access to the weak event pattern.
- Internally, all of the weak event managers are using the same big static hash table (the WeakEventTable) to keep track of what listeners are listening to what sources. The type of the weak event manager associated with a particular key-value pair is part of the key for this table.
- The other part of the key for the WeakEventTable is the event source. The event source is stored via a weak reference, so entries in the WeakEventTable won't keep sources from being garbage collected.
- The type of value stored in the WeakEventTable by a particular manager is dependent on the type of event that the manager manages. The ValueChangedEventManager class uses ValueChangedEventManager.ValueChangedRecord.
- All of the values in the weak event table store lists of listeners, referenced through weak references. Things are a little more complicated in the case of managers where source objects can have multiple properties producing change notifications (i.e., the PropertyChangedEventManager and the ValueChangedEventManager), but the complexity doesn't matter for this discussion.

So far, all of this sounds quite good: property descriptors usually use a hard-reference-based change notification system via AddValueChanged and RemoveValueChanged, but that system can produce memory leaks where the source keeps the listener in memory. In the case of automatically implemented change notification (the kind where the system adds it for us), the system shouldn't be making assumptions about the lifetimes of the sources and the listeners, so instead of using the hard-reference-based system, the system uses the ValueChangedEventManager to give property descriptors access to the weak event pattern to avoid the possibility of leaks. The WeakEventTable uses weak references inside the key values referencing the sources, and also uses weak references to store the lists of references, so everything ought to be fine.

All well and good, except that each value that the ValueChangedEventManager stores in the WeakEventTable — of type ValueChangedEventManager.ValueChangedRecord — *stores a hard reference to its source object.*

As nearly as I can tell, the way the WeakEventTable is *intended* to work is analogous to garbage collection — periodically, it scans itself and finds things to get rid of. If an entry is keyed (via a weak reference) to a source object that has been garbage collected, the entry is deleted, as that source can no longer generate events. If a listener has been garbage collected, it is removed from the listener lists, with the possible consequence of removing entire WeakEventTable entries if particular sources no longer have any listeners. The fact that reflection reveals methods named "Cleanup", "Purge", and "ScheduleCleanup" gives credence to this interpretation.

The problem is that since the WeakEventTable values for the ValueChangedEventManager contain a hard reference to the source objects, the source objects are never garbage collected, and that means that the periodic scans conducted by the WeakEventTable never show that these entires should be removed — with the result that the hard references, and therefore the source objects, remain in memory indefinitely. In other words, a memory leak.

In a strange inversion of the usual memory leaks associated with event handling, *this means that it is the source objects, not the listeners, that are trapped indefinitely in memory.*

Solution

Based on the architecture described above, here is the code that unhooks a source from the entire ValueChangedWeakEventManager system, so that the source can be garbage collected. It removes the source first from the WeakEventTable, then from the lists of ValueChanged handlers on the property descriptors of the type to which the source belongs.

```
public static void RemoveSourceFromValueChangedEventManager(object source)
{
    // Remove the source from the ValueChangedEventManager.
    Assembly assembly = Assembly.GetAssembly(typeof(FrameworkElement));
```

```
    Type type = assembly.GetType("MS.Internal.Data.ValueChangedEventManager");
    PropertyInfo propertyInfo = type.GetProperty("CurrentManager",
        BindingFlags.NonPublic | BindingFlags.Static);
    MethodInfo currentManagerGetter = propertyInfo.GetGetMethod(true);
    object manager = currentManagerGetter.Invoke(null, null);

    MethodInfo remove = type.GetMethod("Remove", BindingFlags.NonPublic |
        BindingFlags.Instance);

    remove.Invoke(manager, new object[] { source });

    // The code above removes the instances of ValueChangedRecord from the
    // WeakEventTable, but they are still rooted by the property descriptors of
    // the source object. We need to clean them out of the property descriptors
    // as well, to allow them to be garbage collected. (Which is necessary
    // because they contain a hard reference to the source, which is what we
    // really want garbage collected.)

    FieldInfo valueChangedHandlersInfo = typeof(PropertyDescriptor).GetField
        ("valueChangedHandlers", BindingFlags.Instance | BindingFlags.NonPublic);

    PropertyDescriptorCollection pdc = TypeDescriptor.GetProperties(source);
    foreach (PropertyDescriptor pd in pdc)
    {
        Hashtable changeHandlers =
(Hashtable)valueChangedHandlersInfo.GetValue(pd);
        if (changeHandlers != null)
        {
            changeHandlers.Remove(source);
        }
    }
}
```