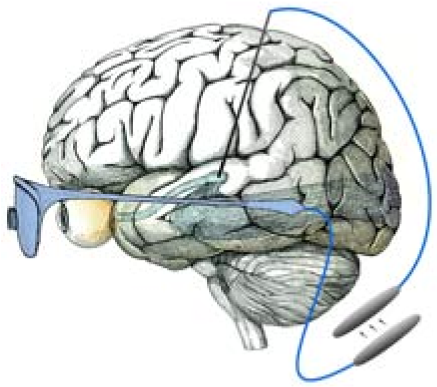
Introduction to Attached Behaviors in WPF

Josh Smith, 30 Aug 2008

Explains the concept of attached behaviors and shows how to use them in the context of the MVVM pattern.

Download demo project (requires Visual Studio 2008) - 21.3 KB



Introduction

This article explains what an attached behavior is, and how you can implement them in a WPF application. Readers of this article should be somewhat familiar with WPF, XAML, attached properties, and the Model-View-ViewModel (MVVM) pattern. I highly recommend that you also read my 'Simplifying the WPF TreeView by Using the ViewModel Pattern' article, because the material here is an extension of the material presented in it.

Background

Back in May of 2008, I published an article called 'Simplifying the WPF TreeView by Using the ViewModel Pattern'. That article focused on the MVVM pattern. This morning, I woke up to find that a fellow by the name of Pascal Binggeli had asked an excellent question on that article's message board.

Pascal wanted to know how to scroll a `TreeViewItem` into the viewable area of the `TreeView` control when its associated ViewModel object selects it. That seems simple enough, but upon further examination, it is not quite as straightforward as one might initially expect. The objective, and problem, is to find the proper place to put code that calls `BringIntoView()` on the selected `TreeViewItem`, such that the principles of the MVVM pattern are not violated.

For example, suppose that the user searches through a `TreeView` for an item whose display text matches a user-defined search string. When the search logic finds a matching item, the matching ViewModel object will have its `IsSelected` property set to `true`. Then, via the magic of data binding, the `TreeViewItem` associated with that ViewModel object enters into the selected state (i.e., its `IsSelected` property is set to `true`, too). However, that `TreeViewItem` will not necessarily be in view, which means the user will not see the item that matches their search string. Pascal wanted a TreeViewItem brought into view when the ViewModel determines that it is in the selected state.

The ViewModel objects have no idea that a TreeViewItem exists, and is bound to them, so it does not make sense to expect the ViewModel objects to bring TreeViewItems into view. The question becomes, now, who is responsible for bringing a TreeViewItem into view when the ViewModel forces it to be selected?

We certainly do not want to put that code into the ViewModel because it introduces an artificial, and unnecessary, coupling between a ViewModel object and a visual element. We do not want to put that code in the code-behind of every place a TreeView is bound to a ViewModel, because it reintroduces some of the problems that we avoid by using a ViewModel in the first place. We could create a TreeViewItem subclass that has built-in support for bringing itself into view when selected, but, in the WPF world, that is definitely a heavy-handed solution to a lightweight problem.

How can we elegantly solve this problem in a lightweight and reusable way?

Attached Behaviors

The solution to the problem explained above is to use an *attached behavior*. Attaching a behavior to an object simply means making the object do something that it would not do on its own. Here is the explanation of attached behaviors that I wrote in my 'Working with CheckBoxes in the WPF TreeView' article:

> The idea is that you set an attached property on an element so that you can gain access to the element from the class that exposes the attached property. Once that class has access to the element, it can hook events on it and, in response to those events firing, make the element do things that it normally would not do. It is a very convenient alternative to creating and using subclasses, and is very XAML-friendly.

In that article, the demo application uses attached behaviors in complicated ways, but in this article, we will keep it simple. Enough with the background and theory, let's see how to create an attached behavior that solves the problem posed by our friend Pascal.

Demonstration

This article's demo app, which is available for download at the top of this page, uses the Text Search demo provided by the 'Simplifying the WPF TreeView by Using the ViewModel Pattern' article. I made a few changes, such as adding more items to the TreeView, increasing the font size, and adding an attached behavior. The attached behavior is in a new static class called TreeViewItemBehavior. That class exposes a Boolean attached property that can be set on a TreeViewItem, called IsBroughtIntoViewWhenSelected. Here is the TreeViewItemBehavior class:

```
/// <summary>
/// Exposes attached behaviors that can be
/// applied to TreeViewItem objects.
/// </summary>
public static class TreeViewItemBehavior
{
    #region IsBroughtIntoViewWhenSelected

    public static bool GetIsBroughtIntoViewWhenSelected(TreeViewItem treeViewItem)
    {
        return (bool)treeViewItem.GetValue(IsBroughtIntoViewWhenSelectedProperty);
    }

    public static void SetIsBroughtIntoViewWhenSelected(
      TreeViewItem treeViewItem, bool value)
    {
        treeViewItem.SetValue(IsBroughtIntoViewWhenSelectedProperty, value);
    }

    public static readonly DependencyProperty IsBroughtIntoViewWhenSelectedProperty =
        DependencyProperty.RegisterAttached(
        "IsBroughtIntoViewWhenSelected",
        typeof(bool),
        typeof(TreeViewItemBehavior),
        new UIPropertyMetadata(false, OnIsBroughtIntoViewWhenSelectedChanged));

    static void OnIsBroughtIntoViewWhenSelectedChanged(
      DependencyObject depObj, DependencyPropertyChangedEventArgs e)
    {
        TreeViewItem item = depObj as TreeViewItem;
        if (item == null)
            return;

        if (e.NewValue is bool == false)
            return;

        if ((bool)e.NewValue)
            item.Selected += OnTreeViewItemSelected;
        else
            item.Selected -= OnTreeViewItemSelected;
    }

    static void OnTreeViewItemSelected(object sender, RoutedEventArgs e)
    {
        // Only react to the Selected event raised by the TreeViewItem
        // whose IsSelected property was modified. Ignore all ancestors
        // who are merely reporting that a descendant's Selected fired.
        if (!Object.ReferenceEquals(sender, e.OriginalSource))
            return;

        TreeViewItem item = e.OriginalSource as TreeViewItem;
        if (item != null)
            item.BringIntoView();
    }

    #endregion // IsBroughtIntoViewWhenSelected
}
```

The attached behavior seen above is basically just a fancy way of hooking the `Selected` property of a `TreeViewItem` and, when the event is raised, calling `BringIntoView()` on the item. The final piece of this puzzle is seeing how the `TreeViewItemBehavior` class gets a reference to every `TreeViewItem` in the `TreeView`. We accomplish that by adding a `Setter` to the `Style` applied to every item in the `TreeView`, as seen below:

```
  <TreeView.ItemContainerStyle>
   <Style TargetType="{x:Type TreeViewItem}">
     <!--
     This Setter applies an attached behavior to all TreeViewItems.
     -->
     <Setter

       Property="local:TreeViewItemBehavior.IsBroughtIntoViewWhenSelected"

       Value="True"

       />

     <!--
     These Setters bind a TreeViewItem to a PersonViewModel.
     -->
     <Setter Property="IsExpanded" Value="{Binding IsExpanded, Mode=TwoWay}" />
     <Setter Property="IsSelected" Value="{Binding IsSelected, Mode=TwoWay}" />
     <Setter Property="FontWeight" Value="Normal" />
     <Style.Triggers>
       <Trigger Property="IsSelected" Value="True">
         <Setter Property="FontWeight" Value="Bold" />
       </Trigger>
     </Style.Triggers>
   </Style>
 </TreeView.ItemContainerStyle>
```

When the demo application loads up, the search text will be set to the letter Y automatically. Click the Find button a few times, and you will see that each time an item is selected, it will contain the letter Y and will scroll into view. The fact that it scrolls into view upon being selected means that the attached behavior is working properly.

Conclusion

Hooking an event on an object and doing something when it fires is certainly not a breakthrough innovation, by any stretch of the imagination. In that sense, attached behaviors are just another way to do the same old thing. However, the importance of this technique is that it has a name, which is probably the most important aspect of any design pattern. In addition, you can create attached behaviors and apply them to any element without having to modify any other part of the system. It is a clean solution to the problem raised by Pascal Binggeli, and many, many other problems. It's a very useful tool to have in your toolbox.

References

- The Attached Behavior Pattern – John Gossman
- Simplifying the WPF TreeView by Using the ViewModel Pattern – Josh Smith
- Working with CheckBoxes in the WPF TreeView - Josh Smith

Revision History

- August 30, 2008 – Created the article.

License

About the Author

Josh Smith

Software Developer (Senior) Black Pixel
United States

Josh creates software, for iOS and Windows.

He works at Black Pixel as a Senior Developer.

Read his iOS Programming for .NET Developers[^] book to learn how to write iPhone and iPad apps by leveraging your existing .NET skills.

Use his Master WPF[^] app on your iPhone to sharpen your WPF skills on the go.

Check out his Advanced MVVM[^] book.

Visit his WPF blog[^] or stop by his iOS blog[^].

See his website Josh Smith Digital[^].

Comments and Discussions

106 messages have been posted for this article Visit **https://www.codeproject.com/Articles/28959/Introduction-to-Attached-Behaviors-in-WPF** to post and view comments on this article, or click **here** to get a print view with messages.