WPF and user Interactivity experience part II: Attached Property

Posted by Bechir Bejaoui in Articles | WPF on April 27, 2010
Tags: User Interactivity in WPF, Attached properties in WPF, WPF design, WPF
In this article, I will show how to profit of the attached property concept to reach the same goal but with more clean and well organized approach.

Source: http://www.c-sharpcorner.com/UploadFile/yougerthen/wpf-and-user-interactivity-experience-part-ii-attached-property/

Reader Level:

Download Files:

- **Graphics02.zip**

In a previous article, I've exposed some user experience scenarios to deal with objects and shapes within a WPF scene using mouse hit testing. In this article, I will show how to profit of the attached property concept to reach the same goal but with more clean and well organized approach.
The idea behind using the attached properties is to provide reusability of the code as it will be leveraged within an independent class scope that could be then used across the different windows or event across different projects when the given class resides within a separate assembly. The fact of choosing attached property over the dependency property is that the first one is more convenient and suitable to leverage business logic across the children and even the parents elements and it demonstrates more flexibility, that is, it could be applied independently of the target element, I mean the element that holds it.

First, let's add a new class to the project

```
namespace Graphics04.AttachedProperties
{
   public class HitTest
{
//TO DO: Add code here to provide the business logic
}

}
```

The second step consists of adding the attached property type of booelan into the scope of the class.

```
namespace Graphics04.AttachedProperties
{
   public class HitTest
{
//The attached property get wrapper method
       public static bool GetHitTestEnabled(DependencyObject obj)
       {
           return (bool)obj.GetValue(HitTestEnabledProperty);
       }
       //The attached property set wrapper method
       public static void SetHitTestEnabled(DependencyObject obj,
                                            bool value)
       {
           obj.SetValue(HitTestEnabledProperty, value);
       }

       // Using a DependencyProperty as the backing store for HitTestEnabled.  This enables
to activate or not the hit testing
       public static readonly DependencyProperty HitTestEnabledProperty =
           DependencyProperty.RegisterAttached("HitTestEnabled",
             typeof(bool), typeof(HitTest),
                          new  UIPropertyMetadata(false));}
}
```

Then the next step is to add our business logic that consists of displaying the shape type that the mouse hits on the scene. To add that business logic, we have to write down our proper UIPropertyMetadata.

```
static UIPropertyMetadata metadata = new UIPropertyMetadata(false,
             new PropertyChangedCallback(OnValueChangedCallBack));
static Canvas _Container;
static void OnValueChangedCallBack(DependencyObject d,
                      DependencyPropertyChangedEventArgs args)
       {
           if (d is Canvas)
           {
               _Container = d as Canvas;
               if ((bool)args.NewValue == true)
               {
                  _Container.MouseLeftButtonDown += new
                  System.Windows.Input.MouseButtonEventHandler
                            (_Container_MouseLeftButtonDown);
               }
           }
       }
       static void _Container_MouseLeftButtonDown(object sender,
                                         MouseEventArgs args)
       {
           Point HitPoint = args.GetPosition(_Container);
           HitTestResult result = VisualTreeHelper.HitTest(_Container,
                  HitPoint);
           if (result!=null)
           {
               MessageBox.Show(result.VisualHit.GetType().ToString());
```

```
        }
    }
```

Let's explain what this above code is consisting of. First, the property Meta data must provide a default value which is false. Second, it provides a PropertyChangedCallBack that points to the below method

```
static void OnValueChangedCallBack(DependencyObject d,

static void OnValueChangedCallBack(DependencyObject d,
                    DependencyPropertyChangedEventArgs args)
    {
        if (d is Canvas)
        {
            _Container = d as Canvas;
            if ((bool)args.NewValue == true)
            {
              _Container.MouseLeftButtonDown += new
              System.Windows.Input.MouseButtonEventHandler
                        (_Container_MouseLeftButtonDown);
            }
        }
    }
```

What is happening within the scope of that method is testing if the scene container is type of canvas, by the way, the canvas it more convenient in this case as it is more generalized than the other most frequently used containers like the Grid for example. If it is true then the attached property value is tested, if it is also true then the mouse left button down event is triggered.

```
static void _Container_MouseLeftButtonDown(object sender,
                                    MouseEventArgs args)
    {
        Point HitPoint = args.GetPosition(_Container);
        HitTestResult result = VisualTreeHelper.HitTest(_Container,
              HitPoint);
        if (result!=null)
        {
            MessageBox.Show(result.VisualHit.GetType().ToString());
        }
    }
```

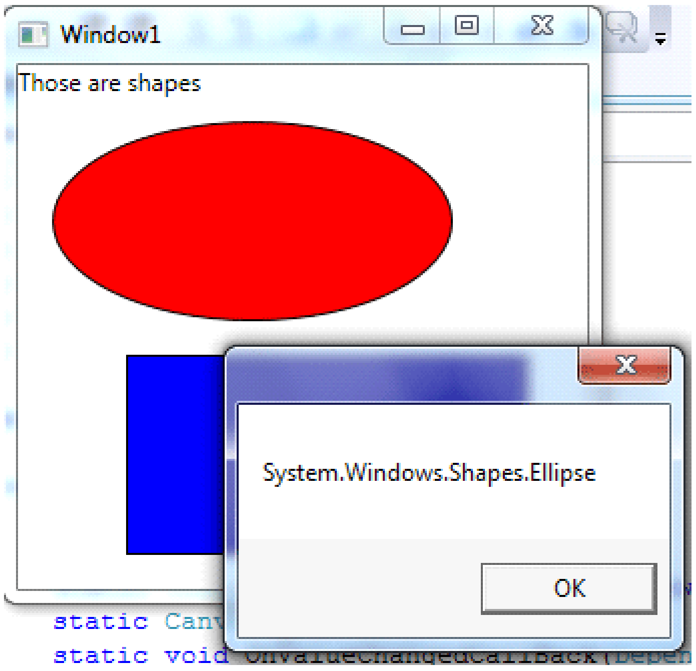
And then the business logic is applied to display the shape that the mouse hits or element type name, Of Corse, within the targeted container.
The way to achieve that is quite simple and consists of two steps, the first one is to map the namespace that holds the attached properties within the XAML scope, and then we apply the attached property to the targeted canvas, by setting its value to true

```
<Window x:Class="Graphics04.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:attached="clr-namespace:Graphics04.AttachedProperties"
```

```
    Title="Window1" Height="300" Width="300"
    <Canvas attached:HitTest.HitTestEnabled="true" >
       <TextBlock>Those are shapes</TextBlock>
       <Ellipse Canvas.Left="17" Canvas.Top="28"
               Height="100" Name="ellipse1" Stroke="Black"
               Fill="Red" Width="200" />
       <Rectangle Canvas.Left="54" Canvas.Top="145"
         Fill="Blue" Height="100" Name="rectangle1"
                     Stroke="Black" Width="200" />

    </Canvas>
</Window>
```

Finally, let's run the project and do some hit tests with the mouse, that's it.



```
System.Windows.Shapes.Ellipse
```

In this article, I've shown a different technique to do the same activity as the previous article but with a different manner, I mean using the attached property notion. In the next article, I will show how to move and resize shapes and geometries within the scene.

Good dotnetting!!!