Handling events in an MVVM WPF application

In a WPF application that uses the MVVM (Model-View-ViewModel) design pattern, the view model is the component that is responsible for handling the application's presentation logic and state. This means that the view's code-behind file should contain no code to handle events that are raised from any user interface (UI) element such as a Button or a ComboBox nor should it contain any domain specific logic.

Ideally, the code-behind of a view – typically a *Window* or a *UserControl* – contains only a constructor that calls the InitializeComponent method and perhaps some additional code to control or interact with the view layer that is difficult or inefficient to express in XAML, e.g. complex animations.

In other words, an MVVM application should not have any code like this where a button's click event is handled in the code-behind of the view:

```
<Button Content="Click here!" Click="btn_Click"/>

protected void btn_Click(object sender, RoutedEventArgs e)

{

  /* This is not MVVM! */

}
```

Commands

Instead, in addition to providing properties to expose data to be displayed or edited in the view, the view model defines actions that can be performed by the user and typically expose these as **commands**. A command is an object that implements the *System.Windows.Input.ICommand* interface and encapsulates the code for the action to be performed. It can be data bound to a UI control in the view to be invoked as a result of a mouse click, key press or any other input event. As well as the command being invoked as the user interacts with the UI, a UI control can be automatically enabled or disabled based on the command.
The Execute method of the *ICommand* interface encapsulates the operation itself while the CanExecute method indicates whether the command can be invoked at a particular time or not. The interface also defines a CanExecuteChanged event that is raised on the UI thread to cause every invoking control to requery to check if the command can execute.
WPF provides two implementations of the *ICommand* interface; the *System.Windows.Input.RoutedCommand* and *System.Windows.Input.RoutedUICommand* where the latter is a subclass of the former that simply adds a Text property that describes the command. However, neither of these implementations are especially suited to be used in a view model as they search the visual tree from the focused element and up for an element that has a matching *System.Windows.Input.CommandBinding* object in its CommandBindings collection and then executes the Execute delegate for this particular *CommandBinding*. Since the command logic should reside in the view model, you don't want to setup a *CommandBinding* in the view in order to connect the command to a visual element. Instead, you can create your own command by creating a class that implements the *ICommand*. The below implementation is a common one that invokes

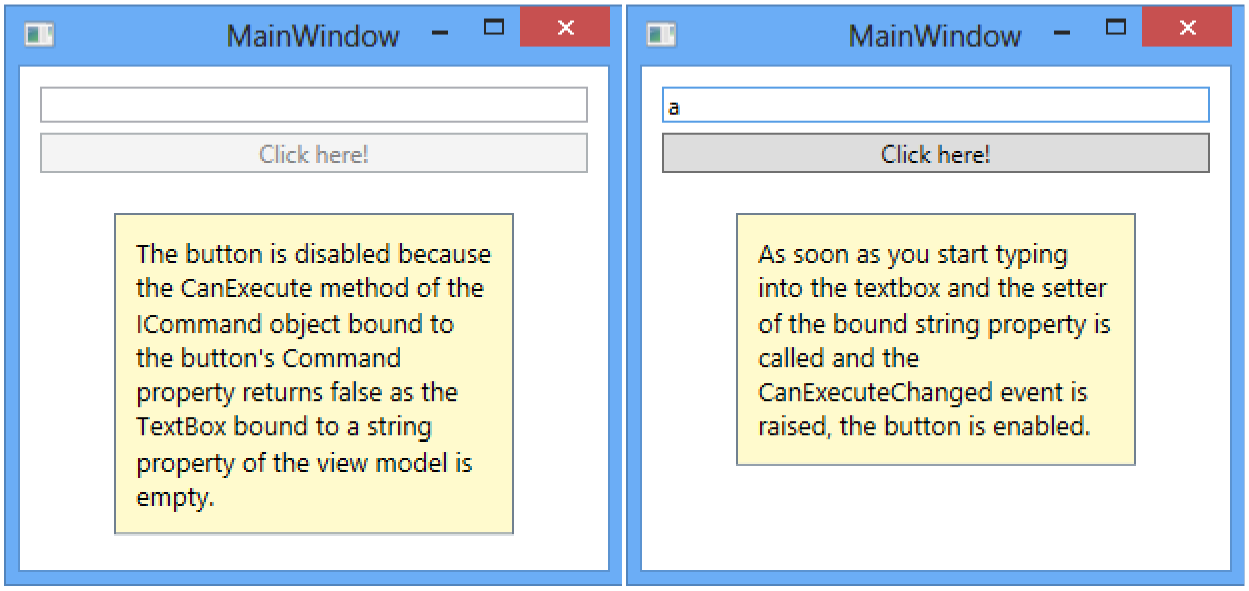
delegates for the Execute and CanExecute methods. If you are using Prism, the framework for building composite WPF and Silverlight applications from the Microsoft Patterns and Practices Team, there is a similar *Microsoft.Practices.Prism.Commands.DelegateCommand* class available.

```
public class DelegateCommand<T> : System.Windows.Input.ICommand

{

    private readonly Predicate<T> _canExecute;

    private readonly Action<T> _execute;


    public DelegateCommand(Action<T> execute)

        : this(execute, null)

    {

    }


    public DelegateCommand(Action<T> execute, Predicate<T> canExecute)

    {

        _execute = execute;

        _canExecute = canExecute;

    }


    public bool CanExecute(object parameter)

    {

        if (_canExecute == null)

            return true;


        return _canExecute((parameter == null) ? default(T) : (T)Convert.ChangeType(paramet
typeof(T)));

    }


    public void Execute(object parameter)

    {

        _execute((parameter == null) ? default(T) : (T)Convert.ChangeType(parameter, typeo

    }
    public event EventHandler CanExecuteChanged;

    public void RaiseCanExecuteChanged()

    {

        if (CanExecuteChanged != null)

            CanExecuteChanged(this, EventArgs.Empty);

    }

}
```

The view model will then expose properties of this type for the view to bind to. Below is a sample implementation of a view model that exposes a ButtonClickCommand that will disable any UI control that is bound to it if the private string field _input, which is in turn exposed through a string property called Input, is empty. If you only pass a single delegate to the constructor of the *DelegateCommand<T>* class, it will assume that the command should always by available and the CanExecute method will always return true. Also note that when the value of the string property changes, a RaiseCanExecuteChanged method is called to raise the CanExecuteChanged event in order to update the status of any control in the view that is bound to the command.

```
public class ViewModel

{

    private readonly DelegateCommand<string> _clickCommand;


    public ViewModel()

    {

        _clickCommand = new DelegateCommand<string>(

            (s) => { /* perform some action */ }, //Execute

            (s) => { return !string.IsNullOrEmpty(_input); } //CanExecute

            );

    }


    public DelegateCommand<string> ButtonClickCommand

    {

        get { return _clickCommand; }

    }


    private string _input;
    public string Input
    {
        get { return _input; }
        set
        {
            _input = value;
            _clickCommand.RaiseCanExecuteChanged();
        }
    }
}
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = new ViewModel();
    }
}
<StackPanel Margin="10">
    <TextBox Text="{Binding Input, UpdateSourceTrigger=PropertyChanged}"/>
    <Button Content="Click here!" Command="{Binding ButtonClickCommand}"
            Margin="0 5 0 0"/>
</StackPanel>
```

With this approach you have now moved the presentation logic from the view to the view model. Instead of hooking up the button's click handler, its Command property is now bound to the command defined in the view model and when the user clicks on the button the command's Execute method will be invoked.



The button is disabled because the CanExecute method of the ICommand object bound to the button's Command property returns false as the TextBox bound to a string property of the view model is empty.

As soon as you start typing into the textbox and the setter of the bound string property is called and the CanExecuteChanged event is raised, the button is enabled.

EventTriggers

There is an alternative way of associating a control in the view with a command object exposed by the view model. Only some controls can actually bind to a command through the Command property, notably those derived from *System.Windows.Controls.Primitives.ButtonBase* or *System.Windows.Controls.MenuItem*. **If you want to attach a command to some other control or when you want to invoke a command on an event other than the click event for a button, you can use Expression Blend interaction triggers and the *System.Windows.Interactivity.InvokeCommandAction* class.** Below is an example on how you would execute a command object called MouseEnterCommand in the view model when the user moves the mouse pointer over a Rectangle element in the view. You specify the event for which the command will be executed in the EventName property of the *EventTrigger*. Remember to add a reference to System.Windows.Interactivity.dll for this to compile.
Also note that using the *InvokeCommandAction* doesn't automatically enable or disable the control based on the command's CanExecute method, unlike controls that have a Command property and can be bound directly to a command.

```
<Window x:Class="Mm.HandlingEventsMVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <Rectangle Fill="Yellow" Stroke="Black" Width="100" Height="100">
            <i:Interaction.Triggers>
                <i:EventTrigger EventName="MouseEnter" >
                    <i:InvokeCommandAction Command="{Binding MouseEnterCommand}" />
                </i:EventTrigger>
            </i:Interaction.Triggers>
        </Rectangle>
    </StackPanel>
</Window>
```

CommandParameters

If you wish to pass a parameter to a command from the view you do so by using the CommandParameter property. The type argument of the generic *DelegateCommand<T>* class specifies the type of the command parameter that gets passed to the Execute and CanExecute methods. The CommandParameter property exists in both the *ButtonBase* and *MenuItem* derived controls as well as in the *InvokeCommandAction* class:

```
<Button Content="Click here!" Command="{Binding ButtonClickCommand}"
                CommandParameter="some string to be passed..."
                Margin="0 5 0 0"/>
<i:InvokeCommandAction Command="{Binding MouseEnterCommand}"
                       CommandParameter="some string to be passed..."/>
```

CallMethodAction

Besides the *InvokeCommandAction* class, there is also another class named *CallMethodAction* that can be used to invoke a method in the view model from the view without using commands. It has a MethodName property for specifying the name of the method to call and a TargetObject property that needs to be bound to an instance of the class containing the method, i.e. the view model:

```
<Window x:Class="Mm.HandlingEventsMVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Intera
        xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <Rectangle Fill="Yellow" Stroke="Black" Width="100" Height="100">
            <i:Interaction.Triggers>
                <i:EventTrigger EventName="MouseEnter" >
                    <!-- Execute a method called 'SomeMethod' defined in the view model --
                    <ei:CallMethodAction TargetObject="{Binding}" MethodName="SomeMethod"/>
                </i:EventTrigger>
            </i:Interaction.Triggers>
        </Rectangle>
    </StackPanel>
</Window>
public void SomeMethod()
{
    /* do something ... */
}
```

Note that the *CallMethodAction* class is defined in another assembly and namespace and you will need to add a reference to Microsoft.Expressions.Interactions.dll to be able to use it. Also note that it doesn’t support parameters.