Implementing MVVM Pattern

When implementing the MVVM pattern, we have a separation between the view (the XAML code coupled with its code-behind), the ViewModel and the model. **Typically, we try to develop the ViewModel so that it doesn't know anything about the view that it drives.** This has multiple advantages: the developer team can work in an independent manner from the UI team; the ViewModel can be unit-tested easily, simply by calling some commands and methods and asserting the value of properties; changes can be made to the view without having to worry about affecting the ViewModel and the model.

In a typical XAML application, the developer uses the powerful data-binding system to declaratively synchronize a property from a XAML UI element with a property of another object in the application. This synchronization can go in one direction only (for example, when a TextBlock's Text property changes to reflect the value of an object's property), or in two ways (for example, when a TextBox's Text property updates the String value of an object's property, or when a CheckBox's IsChecked property updates a Boolean value). Data binding is very comfortable and convenient, especially when it's used in a visual designer like Blend or the Visual Studio Designer. However, it has limitations. For instance, a simple data binding cannot trigger an animation on the UI or cause a dialog to be shown to the user. Even basic actions such as navigation to a different page cannot be caused by simple data binding.

Figure 1 shows two-way data binding between a XAML view and its ViewModel (point 1). In addition, other possible interactions are represented.

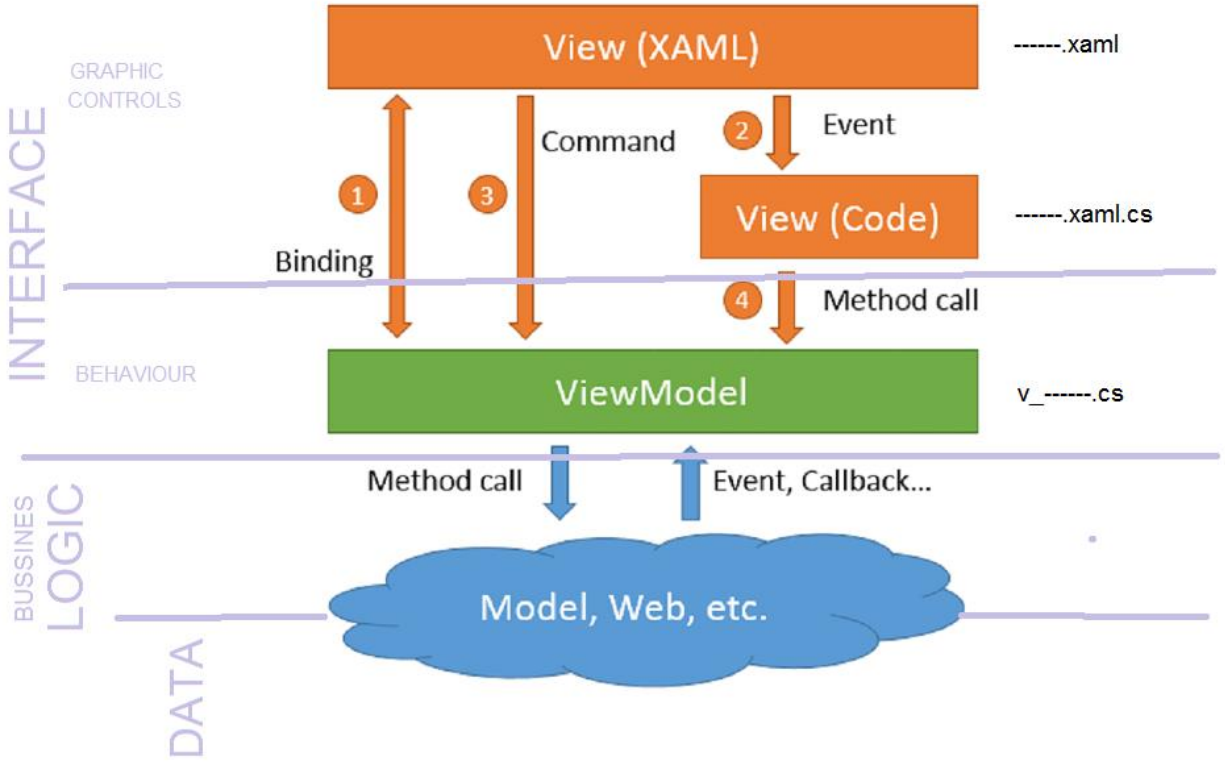

Figure 1. Dependencies between Layers in MVVM

Point 2 shows the normal event flow. This is probably the best-known way for the XAML to communicate with the attached code-behind. It is very familiar to developers coming from more traditional environments, such as Windows Forms or even HTML/JavaScript. While events can be useful, they also cause issues when the XAML needs

to be decoupled from the code. This can be the case when a XAML DataTemplate needs to be moved to a ResourceDictionary, for example. It can also be an issue when the code that handles the event needs to be moved from the code-behind to another object, such as the ViewModel. Events create a tight coupling between the XAML and the code-behind and limit the amount of refactoring that can be done easily.

Point 3 shows an alternative way for the XAML to trigger an action in the code. Typically, commands (which are an implementation of the ICommand interface) are exposed as properties of a ViewModel and are attached to a UI element by data binding. For instance, a Button control supports a Command property. The bound command will be invoked when the Button control is clicked. Commands also have limitations, notably the fact that only a few UI elements expose a Command property and only for one event (usually the Click event). If another event needs to be handled, the default commanding implementation is not sufficient. In another installment in this series, I'll describe ways to work around command limitations, notably with the use of MVVM Light's RelayCommand component, the EventToCommand behavior and (if needed) custom attached behavior implementations.

Point 4 shows the view's code-behind calling a method on the ViewModel directly. This may sound like a violation of the principle that a view should be decoupled from its ViewModel. Actually, it's not a problem for the view to know its ViewModel because the view will rarely need to be abstracted. A view's code-behind is rarely unit tested; while possible, it is not very easy to trigger event handlers programmatically to test their action. So, while the ViewModel should be kept ignorant of the view, the contrary is not absolutely necessary. In fact, the code shown in Figure 2 is very frequently found in MVVM applications. While a principle of MVVM states that developers should keep the code-behind thin, it is sometimes easier to have a small snippet of code-behind handling a special situation than to look for complicated workarounds.

Figure 2. Getting the ViewModel in the View's Code-Behind

```
public MainViewModel Vm
{  get
  {
    return (MainViewModel)DataContext;
  }
}
```

..//..

https://msdn.microsoft.com/es-es/magazine/jj694937(en-us).aspx