Contents

View/ViewModel data binding isn't always the best way to trigger a behavior in the *View*. When the binding logic becomes too convoluted and difficult to follow, it's better to switch to a different strategy and let the *ViewModel* talk directly to the view.

We'll revive the *View* code-behind and teach the *ViewModel* to call into the *View* via an interface.

The 06-HappyHour tutorial folder holds the state of this solution at the end of the lesson.

Reset the view after each new drink order

Notice that, as you added new drink orders, the old value remains in the *TextBox*. Keep pressing the button and you place a new order for the same drink. That works for me; once I've found my drink, I tend to stay with it all night.
It may not work for the business. The specification says the *View* should prepare for a new choice after each order. We should clear the *TextBox* and set the focus so it's easy to make a new choice.

Should the *ViewModel* clear the *TextBox* and set focus? We don't think so. The *ViewModel* shouldn't know about *TextBoxes* and input focus. These are the details of interaction design decisions that are the proper concern of the *View*, not the *ViewModel*. The *ViewModel* should signal its intention to the *View* – it wants the *View* to "reset itself", whatever that may mean. The *View* should respond to that signal in the visual and functional way that the *View* designer deemed appropriate. This is a concrete example of the principle of "*separation of concerns*."

Supervising Controller

There is no easy way in XAML to set the *TextBox* focus at the *ViewModel*'s request.

I'm not saying you can't do it. I'm saying it's not easy. A simple thing like this should be easy.

An obvious approach would be to put the *TextBox* focus logic in a reset method in the *View*'s code-behind and let the *ViewModel* call that method as appropriate.
This approach violates a fundamental tenet of MVVM: the *ViewModel* should never have a reference to the *View*. We're going to do it anyway.

We'll do it in a way that preserves separation of concerns. We won't call the method "*ClearTextBoxAndGiveItFocus*"; the *ViewModel* doesn't need to know those details. We'll call it "*ReadyForNewDrink*" because that's the intent.
We'll define the *ReadyForNewDrink* method in an interface that insulates the *ViewModel* from any particular concrete *View*.

Finally, we'll write the *ViewModel* code such that it only calls *ReadyForNewDrink* if the *View* supports the interface; if the *View* doesn't, no big deal.

When our *ViewModel* maintains a reference to the *View* and tells the *View* what to do through an interface, we're using the Supervising Controller pattern. This is another in the family of UI Architectures that, like MVVM, help us maintain separation between the easily tested, non-visual presentation logic and the hard-to-test UI widgets on screen.

Define the interface

1. Select the **ViewModels** folder.

2. **Add | Class | "IMainPage.cs".**

3. Replace entire contents with this:

```
namespace HappyHour.ViewModels
{
  public interface IMainPage
   {
     void ReadyForNewDrink();
```

```
    }
}
```

4. Open ***MainPageViewModel***.

MainPageViewModel can't call *ReadyForNewDrink* until it "becomes aware" of a *View* that implements it. Fortunately, *MainPageViewModel* inherits from *Screen* …

```
[Export]
public class MainPageViewModel : Screen
```

… and *Screen* implements the *IViewAware* interface as described below. What you need to know is that *Screen* calls its *OnViewAttached* method when the *View* becomes available. *MainPageViewModel* can override that method to gain access to the *View* … as shown here.

```
protected override void OnViewAttached(object view, object context)
{
    _view = view as IMainPage;
}
private IMainPage _view;
```

Casting the *View* as *IMainPage* makes its *ReadyForNewDrink* method accessible.

5. Revise the ***AddDrinkOrder*** method to ready the view after adding a new drink order to the collection.

```
public void AddDrinkOrder()
{
    var drink = new DrinkOrder {DrinkName = DrinkName};
    DrinkOrders.Add(drink);
    SelectedDrinkOrder = drink;
    ReadyForNewDrink();
}
private void ReadyForNewDrink()
{
   if (null != _view) _view.ReadyForNewDrink();
}
```

Notice that our *ViewModel* remains testable despite its dependence on a View. That dependence is limited to the *IMainPage* interface which is trivial to fake.

Of course we'll have to remember to pass a fake view into the *MainPageViewModel* when we test it. We can do that by casting the tested *MainPageViewModel* to *IViewAware* and calling *AttachView* with the fake view during setup of *AddDrinkOrder* tests.

Teach MainPage to get ready … in code-behind

Now that the *MainPageViewModel* is calling *IMainPage.ReadyForNewDrink*, we have to make *MainPage* implement that method … and it can only do so in code-behind.

There is no code-behind for the *MainPage* at the moment. We deleted the *MainPage.xaml.cs* file a few lessons ago because we didn't need it. It is our strong preference to eliminate code and files that aren't contributing. No problem; we can restore it now that we need it.

1. **Project | Add | Class | "MainPage.xaml.cs"**

2. Replace entire contents with this:

```
namespace HappyHour.Views
{
   public partial class MainPage : IMainPage
    {
       void IMainPage.ReadyForNewDrink()
        {
            DrinkName.Text = string.Empty;
            DrinkName.Focus();
        }
    }
}
```

Our implementation resets the view for the next new drink order by clearing the *TextBox* and giving it focus. You could restore the constructor that calls *InitializeComponent* if you wish. We decline to do so; we'll let Punch call *InitializeComponent* for us.

3. **Rebuild and run** [F5]

Enter a drink name and click the button. Immediately after, you should see that the *TextBox* is clear, the button disabled, and focus is in the *TextBox* where it is ready for your next drink.

Last call

In this lesson, we found ourselves wanting a user experience that is difficult to accomplish by MVVM-style data binding. The easier approach is to write the desired behavior in the View code-behind and invoke it from the *ViewModel*.

We wrote a *View* interface (*IMainPage*) to abstract the View from the *ViewModel*. We made sure that interface member names expressed the intention ("*ReadyForNewDrink*") rather than dictating the user interaction ("*ClearTextBoxAndGiveItFocus*").

We saw that a *ViewModel* derived from the Screen class gains access to the *View* through the *OnViewAttached* method and thereby becomes capable of calling *View* members through the *View* interface.

This business of a *ViewModel* calling a *View* through an interface is an example of the Supervising Controller pattern which is an alternative to the more common **ViewModel** pattern.

We recommend that you stick with the **ViewModel** pattern when you can; it generally yields code that is cleaner and easier to maintain (no need for view interfaces).

But don't be dogmatic about it. Use **Supervising Controller** when the *ViewModel* initiates a change in the View that isn't easy to communicate through data binding.

Ask the Mixologist

This lesson is finished. Feel free to move on directly to the next one. This "Ask the Mixologist" section is an optional digression from the lesson's main course to related points of interest.

Is *View* code-behind evil?

We are not opposed to code in the code-behind. We're wary of it because code-behind is difficult to test and developers have a bad habit of hiding business logic there. But we don't mind if the code is simple (no conditional logic) and confined to purely local matters of design and usability. Try not to make a habit of it ... please.

Shouldn't the *MainPage* code-behind call *InitializeComponent*?

Certainly *something* has to call *InitializeComponent* before the view can appear on screen. That *something* is the Caliburn Micro view composition process which is why we were able to delete the code-behind in the first place.

If you feel more comfortable with a more traditional looking code-behind that has a constructor that calls *InitializeComponent*, go ahead and add it. There is no harm in calling it twice.

Why *IMainPage* belongs to the *ViewModels*

You might think *IMainPage* belongs in the *Views* folder with a *HappyHour.Views* namespace because it prescribes functionality to be implemented by a *View*. In fact it belongs with the *ViewModels*.

Someday we might have a third assembly of interfaces. It could make sense to move it there. But that's complexity we don't need today. Right now we need to determine to which folder and namespace this interface belongs.

The unshakeable principle is this: ***Views* may depend upon *ViewModels* but *ViewModels* may not depend upon *Views***. To understand why, let's indulge a couple of thought experiments.

Suppose that *IMainPage* were defined in the *Views* namespace and folder. Then suppose we later discover a good reason to breakout the *ViewModels* into their own assembly.

Because *MainPageViewModel* depends upon *IMainPage*, the *ViewModels* assembly would have to depend upon the *Views* assembly. But we know that *MainPage* must implement *IMainPage* which means the *Views* assembly would have to depend upon the *ViewModels* assembly. We'd be trapped in a circular dependency – *Views* depends upon *ViewModels* which depends upon *Views* – and circular dependencies are disallowed in .NET.

Suppose we want to reuse our *ViewModels* to support multiple clients written in different XAML technologies such as Silverlight, WPF, Windows Phone and Windows 8. We couldn't do that; the *MainPageViewModel* dependence on *IMainPage* in the Silverlight *HappyHour Views* assembly would pin the entire *ViewModels* assembly to the Silverlight client.

These problems disappear when *IMainPage* is defined in *ViewModels*. The *Views* assembly must depend upon the *ViewModels* assembly. But there is no reciprocal dependency from *ViewModels* to *Views*. The *Views* in Silverlight, WPF, Windows Phone and Windows 8 could each depend on the same *ViewModels* assembly (assuming other aspects of the technologies permitted such dependence).

Define *View* interfaces of the Supervising Controller pattern in the *ViewModels* namespace, not the *Views* namespace.

What is *IViewAware*?

The *Screen* class implements *IViewAware*, an interface that makes a *ViewModel* "aware" of the *View* to which it is attached. That "awareness" is well short of deep familiarity; such familiarity would break the separation we seek between the visual manifestation of the *View* and the non-visual view support that is the proper role of the *ViewModel*.

But there is no harm (and plenty of benefit) in knowing a few things about a *View* when it is abstracted behind this interface.

```
public object GetView(object context = null) { }
public void AttachView(object view, object context = null) { }
public event EventHandler<ViewAttachedEventArgs> ViewAttached;
```

1. ***GetView*** is called first, giving the *ViewModel* an opportunity to provide its companion *View* instance directly rather than rely on the framework to create it. A *ViewModel* could return a concrete cached *View*. If it returns *null*, that means the framework should create the *View*. In *HappyHour*, where the *MainPageViewModel* is only displayed once, the base class implementation returns null and a new *View* is created.

2. **AttachView** is called next with the concrete *View* to which this *ViewModel* is bound (either the one returned by *GetView* or the one created by the framework). *Screen* implements this method privately. *MainPageViewModel* is not involved but can learn about the view by overriding *OnViewAttached* ... as we did in this lesson.

3. **ViewAttached** is the event raised after *AttachView* completes. It informs other interested components that this *ViewModel* has been bound to its *View* and is ready for business.
MainPageViewModel doesn't bother listening for this event because it gets the same information from *OnViewAttached*.
OnViewAttached is called before handlers of the *ViewAttached* event.