Article/blog/learning materials on WPF MVVM

Rough outline:

Motivation

* Difficulty of finding good quality information online in 2015
* Prevalence of poor implementations AND documentation promoting it
* Sheer number of competing toolkits and frameworks

MVVM – What it is, what it is not

MVVM in WPF – the bare minimum implementation

MVVM toolkits/frameworks – a comparison of implementations; pros and cons, likes and dislikes

From bindings, implementing support for two-way updates between views and viewmodels, via INPC.

In order to meet the requirements of implementing INPC, we must raise our “property changed” event any time any of our properties is actually changed. As part of the WPF infrastructure, those views finding themselves bound to a data context that supports INotifyPropertyChanged will auto-wire themselves to the PropertyChanged event. That’s straightforward enough; we raise the event thus:

this.PropertyChanged(this, <property changed event detail>

What’s a little surprising is to find that when supplying the details for the property-changed-event-detail in the prescribed manner, the property in question is referenced by a string as opposed to something more explicit and type-safe. We must use the PropertyChangedEventArgs constructor, taking a string, and that’s an end to it. WPF is reusing old classes, invented much earlier in the development of .NET (the same event is used in WinForms for example.)

This temporary lapse in type-safety is often mitigated against by the toolkits out there, by using an overload of the helper method (typically called OnPropertyChange) that can take a compile-time-checked type *somehow* – both MVVM Light and the framework of the guy presenting at NYC DevReady use a lambda expression – this overload then calls a helper implementation taking a string parameter, having extracted it from whatever expression it takes.

These days, and as seen in Prism, decorating the property-change helper with [CallerMemberName] allows you to avoid the overload and the noise of an additional helper using a lambda. Given the following helper:

void OnPropertyChanged([CallerMemberName] string name = null)

{

var args = new PropertyChangedEventArgs(name);

PropertyChanged(this, args);

}

then we can simply call OnPropertyChanged() from within the ‘set’ accessor of our properties, and rely upon the compiler to inject the string for us. OnPropertyChanged() looks rather cleaner than OnPropertyChanged( () => Name ), but more importantly, there’s nothing stopping you passing a compile-time-checked property, valid in the context of the ViewModel, that happens to be the wrong one! For that reason, I much prefer the [CallerMemberName] solution; you can’t go wrong.

One thing in the DevReady talks I’d point out as questionable design, or at least slightly blurring the issue: He’s clearly making use of the Façade pattern in his SampleViewModel. I’d separate the façade out first, then reuse the façade as the property on the ViewModel. In this case, we’re only talking about “flattening out” one or two internal child properties to make them visible as parent properties, and so the façade is barely noticeable. But that’s what it is, and - especially if it might be useful elsewhere - I’d separate the two patterns and have an intermediate SuchAndSuchFacade class wrapping the model class - then the ViewModel exposing the façade class as a property.

Another thing that came out is some of the stuff in viewmodels migrating towards the business layer – the use of an ObservableObject base class, used by both models and views, and implementing INPC.

View First approach

In this approach, a view instantiates the correct view model. There are various ways this is achieved – either instantiating a viewmodel in the view’s XAML (design-time instantiation), requiring a default view model constructor, *or* explicitly creating one in the view’ constructor, after InitializeComponent().

In real-world applications,

V

Commanding

As explained in the last part of session 2, controls like Button support WPF commanding. Commands are defined by the ICommand interface; using your own commands will require implementing that interface. The problem is, if you define your own class to implement the command behaviour, you’ll typically lose the context to the view model – and thus, any properties you might need.

Several frameworks/toolkits provide a solution - RelayCommand aka DelegateCommand. These generic command wrappers allow you to pass in the CanExecute and Execute methods, pointing to methods in the ViewModel, which then allows the command methods (in the view model) to grab view model properties.

ADVANCED TOPICS (session 3)

* **MVVM “In The Box” – Requires VS2010, not ported to other platforms**

Begins with worked example of commanding.

**Custom behaviours in WPF – STILL TO DO!**