Hans Karlsen

MDriven The book

Doing effective Business – by taking control of Information

Hans Karlsen, Stockholm Sweden
[Datum]

The ViewModel

It is a common problem that User Interface (UI) code gets filled with business logic that does not belong there. The logic is often left there as a quick fix and the developer has every intention to someday return and clean it all up by refactoring the code so that business rules are handle by the model and the UI only handles the user interaction. This actually never happens. I have never seen a team that has enough room in their schedule to go back and redo work that to the client is already done, delivered and paid for.

Every developer knows that the degrees of freedom rapidly decrease once you fill your UI with business logic:

1. You cannot easily reuse the logic placed in a UI so you copy it and increase the maintenance (BAD!)
2. You forget about rules placed in UI so your system gets a life of its own (BAD!)
3. Once you have logic in the UI you cannot be expected to know it all so you get afraid to make changes to your system because something will or may break (BAD!)
4. You dare not give the UI to that brilliant designer because the designer will break logic for sure (BAD!)

Still I see business logic in the UI everywhere I go. When I ask about it I always get answers like: “Well this is not finished yet”, or “Well this is really special stuff, only used in one place”, or “Yea I know it sucks, I will fix that later”. But “Later” never comes, does it? It will always be just “Later”.

I am no superman for sure. I see business logic in UI code I have written myself too.

If everyone or at least most of us is getting into these situations could it be that we are doing things the wrong way? Could it be that doing things the right way is just a tad bit too complicated?

These are some strategies to make developers do the correct thing and actually follow the coding guidelines:

1. Automated review tools like FXCop
2. Adhere very strictly to ModelView patterns as MVVC or MVC – but still you need to verify that you follow the pattern
3. Peer review (that usually will be done “later”)
4. Some other strategy that will force violators (you and me) to mend our ways – even if the correct way is really complicated (maniac team leader with rabies foam in the face)
5. Make it easier and faster to follow the “coding guidelines” then to be quick and dirty.

To no surprise I am in favor of making things easier. To be able to make things easier we need to understand what cause things to go wrong in the first place: Why is the UI filling up with business

logic? I think there are a couple of reasons, depending on how far you are from being model driven some of these reasons will apply:

1. The UI will need to transform data (could be business logic) in order to display it according to specification.
2. Actions performed in UI act on selections from UI-components, that has data in the transformed presentation format, and need to be transformed back (could be business logic) to model-scope in order to let the model work on the data (business logic). You also need to check that the parameters sent to the model method are valid (could be business logic).
3. The existing business logic may not match 100% what your action should do, you may want to call two, three of more methods to get the job done (new business logic) and you want to do some small checks based on the results of each method (could be business logic).
4. Validation – your UI will do a lot of small checks to see that the data fulfills the overall rules for your application (business logic)

How do we make these reasons go away?

1. We do it by offering a good and easy way to transform model-elements (or data if you will) into data elements suitable for render to match the specification.
2. We do it by making it real easy to add business logic where it belongs (in the model), and make it easy to call it.
3. We offer a clean and easy way to add validation logic.

By setting the model in focus, and making it dirt simple to add methods, derived attributes and derived associations you can do everything you need for a specific UI in the model. This is an improvement even if not the solution.

The problem is that if you have a 100 UI's your model is filled with 100 times x derived attributes and derived links. That does not sound like a good thing to me. It will eventually get hard to see the trees for the forest in a model like that.

Further more when a UI is dropped and deleted for some reason, will we remember to drop the derived associations and derived attributes we added to the model to accommodate it? Probably not.

A transformation layer between the UI and the Model can fix this. The transformation layer is allowed to have business logic, as long as it is unique for the context it works on. If the logic is not unique, it should go in the model ready to be re-used. We call this transformation Layer for a ViewModel. The name View-Model is from this being a particular view or perspective in how to look at the model. The perspective often correlates to UI-views.

A ViewModel transforms a piece of the model for a specific purpose. The purpose is often, but not always, to prepare the model information for interaction with a user for a specific use case – a UI. I said often but not always; it can also be for creating data transfer objects that are suitable for exposure for another application or system, or for some reporting engine or the like.

Why is it having the rules in a ViewModel is much better than having them in the UI? There are a lot of reasons:

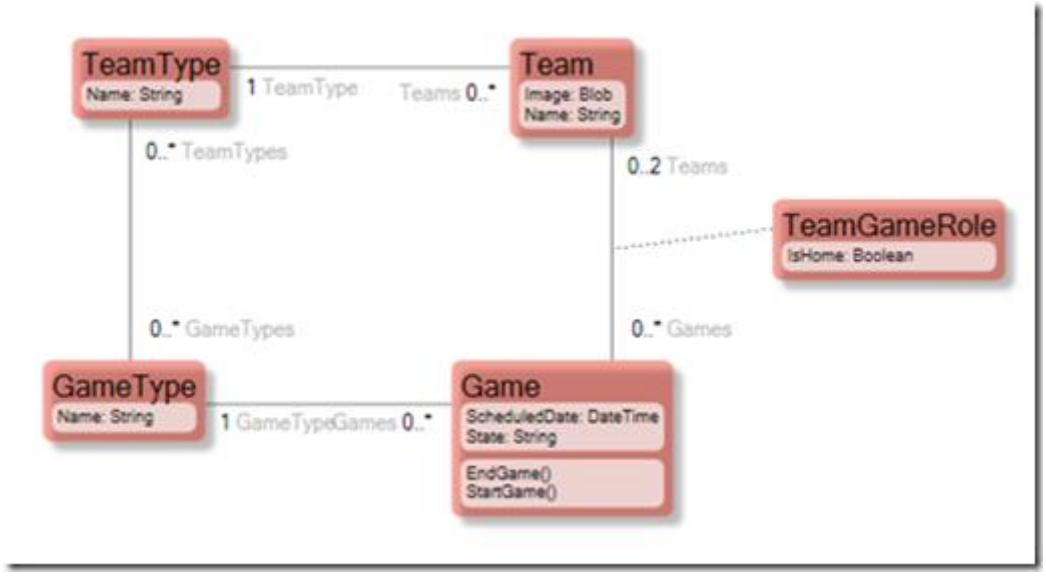
1. Testing; it is a good thing to be able to test the logic separated from the UI, because it is awkward and error prone to test and read the results back from the UI.
2. ViewModel re-use ; you may have different UI's for the exact same use case (beginner/advanced, Web-api/Rich client etc).
3. Design time type checking; most UI-binding strategies rely on using strings that can only be checked at runtime (true for winforms and ASP.NET – MVC with Razor is type checked and also WPF), whereas a good ViewModel is type checked at design or compile time.
4. When important logic is in the ViewModel the designers can work on the UI without risking any damage to the logic.
5. If we have dedicated designers we will not want to wait for them to release a UI file in order to fix business logic within.
6. The UI may be on the other side of a network (another physical tier) so the UI cannot have access to the domain layer tier
7. UI and logic have very different motivators and hence will often change for different reasons (looking good versus working correctly), mixing them up adds confusion regarding the distinction between these important aspects.
8. Security, designer that get access to the ViewModel cannot go beyond the ViewModel and unintentionally expose information that should not get exposed in the use case at hand.

The thing is that you do not have to use a ViewModel pattern to create a great application, it is just that is a good idea to use the ViewModel pattern when building applications that are going to be worked on for a long time, released in several versions over several iterations, and will most likely see different developers and UI-designer, and may very well be radically changed with regards to presentation framework during its lifespan. In short – it is always a good idea for successful applications to use a ViewModel pattern.

The declarative ViewModel

Presented with a model that I got from colleague, that incidentally helps out as a Hockey Referee when he is not coding, I wanted to create a ViewModel for the use-case "Set up new Hockey game".

Part 3 – ViewModels the declarative way

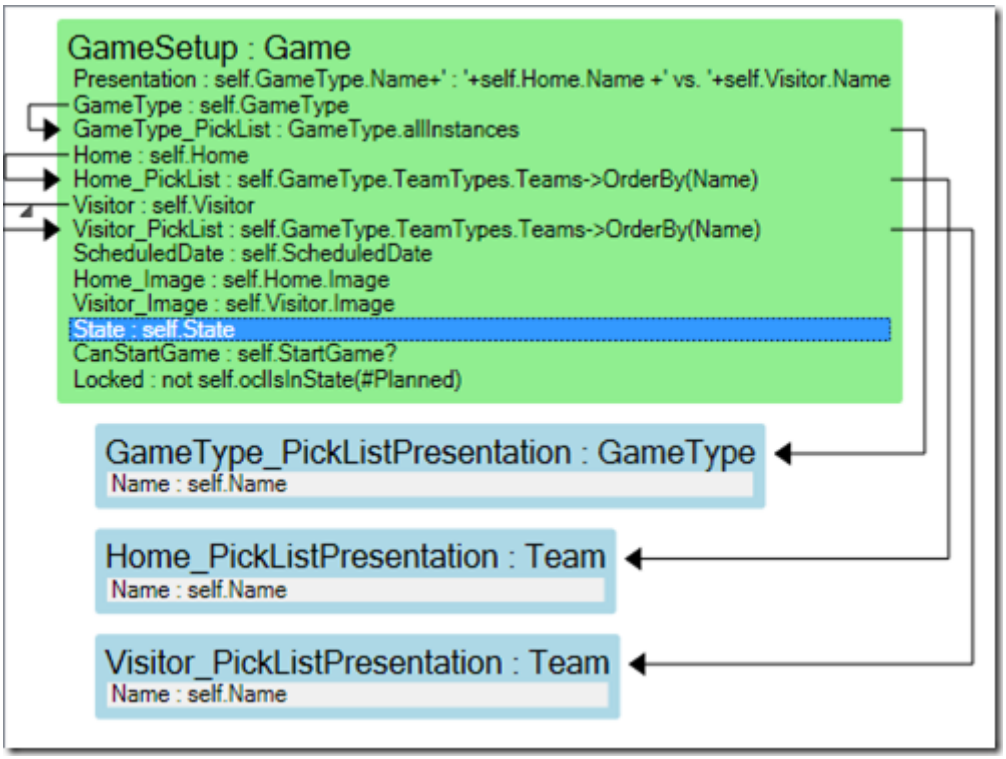


The Game class has a state machine:

I took a piece of paper and draw the UI I wanted to achieve:

Now I know what the requirements are on the ViewModel since I can see from the drawing what data the ViewModel needs to hold.

And then created this ViewModel That I named GameSetup:



Notice that it is just a series of named ocl expressions. Some expressions are nested to other list definitions like Home_PickList that states that if the Game has a picked GameType, then we know what teams that can be picked – namely those teams that are associated to that GameType

I created some test data so that UI can show something. My first attempt was to manually code a WPF UI and bind the values to the ViewModel

The UI looks like this:

This has minimal amount of fashion acceptable styling. The good thing is that you can hand it to any WPF savvy designer in the world – the data and the rules are safe in the ViewModel.

It already shows some of the good effects of separating UI from logic.

1. The PickLists for Home and Visitor are filtered based on Type of Game
2. The Picklist for Home team filters away the Visitor team if set (and vice versa)
3. Start game is enabled only after both home and visitor are set
4. The End game button is disabled until the Game is started

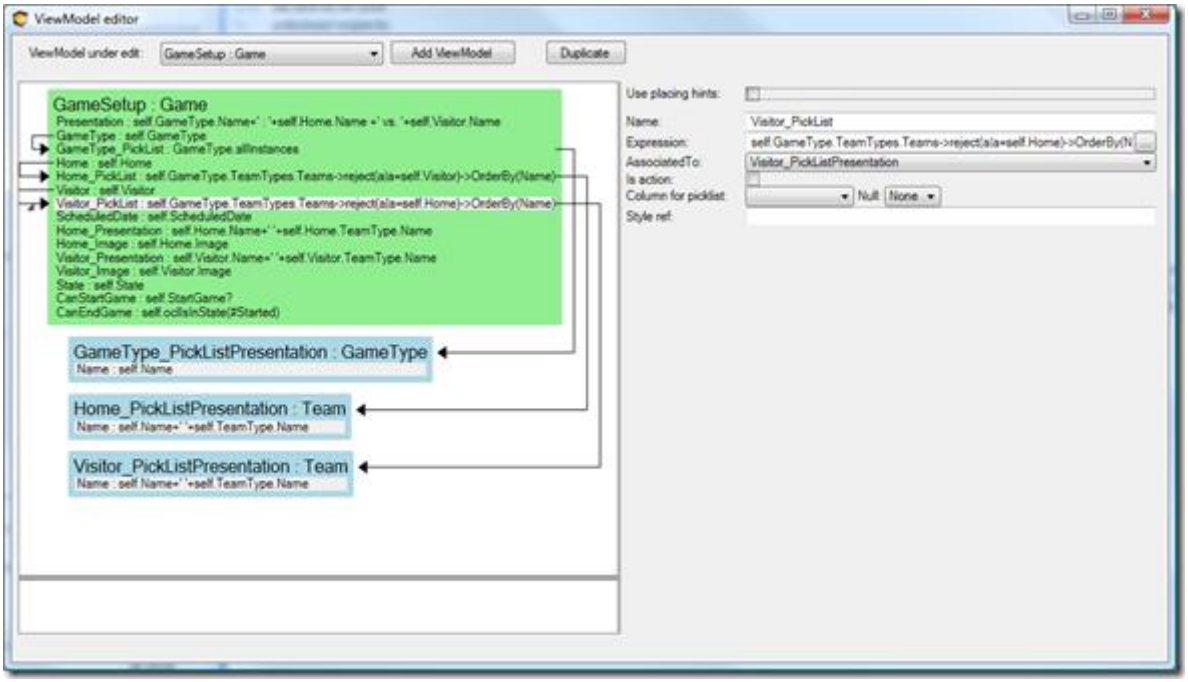
These are some examples of business logic that would have easily ended up in the UI if we did not have a good place to define it.

Taking it further still

If the cost of creating and maintaining a ViewModel is high fewer ViewModels will be created. So our mission is to reduce the cost of creating and maintaining them.

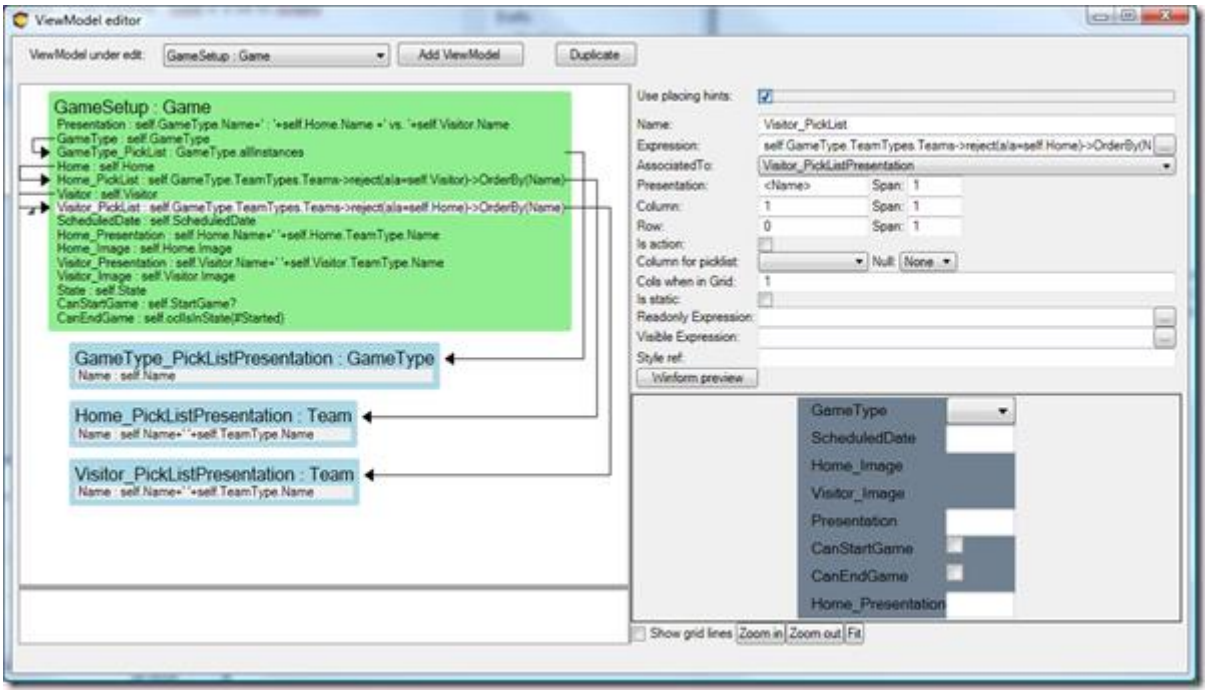
WPF is a declarative way to describe the UI. This mean that the same basic lookless components like TextBlock, TextBox, CheckBox, Combobox and Image etc will be used again and again and they will be given a look by an external style or template.

What if we use this fact and provide some basic rendering/placing hints for the ViewModel columns? We could then use those clues to spill out the correct lookless control in the intended relative position so we would not need to mess about with xaml every 5 minutes. This is what the ViewModel-Editor looks like without rendering hints:

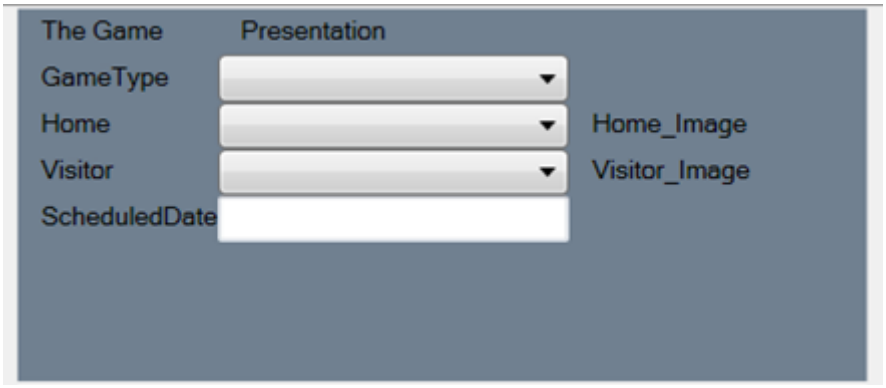


And this is the way it looks when I have checked the “Use Placing Hints” checkbox:

Part 3 – ViewModels the declarative way



Given the extra fields for “Presentation”, “Column”,”Row”,”Span” etc I can work the ViewModel – preview to look like this:



Now I really need to stress this so that I do not get misunderstood: We are not designing the presentation here at all. We are describing what data is available, which values are valid, possible selection lists of data, and, if the UI designer wishes to take notice of it, *hints* as to how to arrange the controls in relation to each other which happens to give the option of generating the user-interface automatically by whatever front end is currently in fashion. This is all natural information we have in mind while designing the ViewModel.

Having a ViewModel with placing hints, you can add a ViewModelWPFUserControl to your form with just one row:

```
<ecoVM:ViewModelWPFUserControl Grid.Row=”2” x:Name=”VMU1”
    EcoSpaceType=”{x:Type ecospace:WPFBindingEcoSpace}”
        ViewModelName=”GameSetup” ></ecoVM:ViewModelWPFUserControl>
```

And the result is:

And remember that these auto layout controls also adheres to external set styles.

Having the ability to get simple UI automatically derived from the ViewModel placing hints lowers the effort to produce and maintain. Experience has shown that a lot of the administrative parts of your application is left as automated so that more time can be spent on the signature screens that are most important for your users.