# Well Defined Model: Model+ViewViewModel

The formal ModelViewViewModel part defines only 3 parts. The View and ViewModel are clearly defined; the rest we simply call a Model. Is this enough? Well from my day-to-day experience I have to say that there is a lot more to worry about than simply separating the visuals from code!

What is very important when building a composite application is to well define the responsibilities of each part and then stick with the rules you define. I call the pattern “Model+” since in this case the Model has being well defined.

The following design addresses many scenarios in a composite application. You can find the source code with a sample application under: <https://bakopanos.googlecode.com/svn/trunk/prism>   
The sample is a modified version of the original sample that ships with the Prism quickstarts.



## Controller

The controller is responsible to create views and position the views on the appropriate placeholders. For example if you are using Prism (Composite WPF) these placeholders are called Regions and there is a RegionManager and a RegionRegistry to deal with regions. Another concern that the controller have is to activate views. For example activate the tab page in a tabstrip that contains the view you would like to show. But who is driving the user interface process (UIP)? It’s important to make sure that you decouple UIP from the model. This is the reason that an eventaggregator comes very handy here. The controller can subscribe to strongly types events and get notified about changes in the state. Typically in most cases this will be notification about selected item in a workflow and dirty items.

## View

In order to implement correctly the Model-View-ViewModel pattern (Fowler: PresentationModel) you really need to restrict the View to only the visuals. The Passive View doesn’t require any unit testing and the visual designer can work decoupled from the developer.

## ViewModel

When I think about ViewModels I really see them as facades. I am aware that most people see workflows as the place where all logic goes; the problem is that if you put too much in a ViewModel then better consider a ModelViewController approach. What I see sometimes is ViewModels being sub classed in order to allow reuse. To my understanding this is by definition wrong. A ViewModel should simply aggregate all that the view needs to display and provide them to the view in a simple structure. You can also have logic there that has to do with the current view, but nothing more.

## WorkItem

The work item is the key to a composite application. Why? Because the workitem is where you store your state. Let’s consider an example: Your application should change to the current customer. In other words after you select the customer in a list box, anything that you do in any screen should consider that specific customer. So if you get a bug assigned to you that somewhere in the application the customer is not displayed properly, where is the place where you first look? What part of your architecture remembers that? To my understanding this should be the WorkItem. The ViewModel binds the selected item to the workitem, and when the selected item changes the workitem publishes an event. The interested controllers register for that event and drive in turn any UIP.

## Services

BEWARE: services should be stateless. This is very important. If you would like your application to be easily changed and tested then model anything that has to do with state in a WorkItem.

There are two categories of services:

* Infrastructure Services, basically cross-domain functionality. Example ILogService
* Services, that interact with the business/service layer in order to support the application with domain specific functionality. Example IProjectService