



# Recommendations and best practices for implementing MVVM and XAML/.NET applications

January 30, 2015, (updated on July 18, 2015), [17 comments](#), [Software Development](#)

In this article I'll describe the rules and practices I'm following for XAML and MVVM application development. I'm using the described techniques since multiple years and they have proven themselves for me. Most of the described rules apply for all types of XAML projects - Windows Store, Windows Phone, Silverlight and WPF projects.

**Sample code:** The source code of the application [Visual JSON Editor](#) contains sample implementations for most of the discussed recommendations. Some sample code snippets in this article use the MVVM classes from the [MyToolkit](#) library which is developed by myself.

## Project file structure

All MVVM/XAML applications should have a similar directory structure. The following tree shows a possible structure of a XAML application project:

- App.xaml
- **Controls:** Reusable UI controls (application independent views) without view models.
- **Localization:** Classes and resources for application localization
  - Strings.resx/.resw
- **Models:** Model and domain classes
- **ViewModels:** View models classes
  - MainWindowModel.cs

- MyViewModel.cs
- **Dialogs**
  - SelectItemDialogModel.cs
- **Views:** Contains the views
  - MainWindow.xaml
  - MyView.xaml
  - **Dialogs**
    - SelectItemDialog.xaml

As shown in the list, the name of a view should end with its type:

- **\*Window:** A non-modal window
- **\*Dialog:** A (modal) dialog window
- **\*Page:** A page view (mostly used in Windows Phone and Windows Store apps)
- **\*View:** A view which is used as subview in another view, page, window or dialog

The name of a view model should be composed of the corresponding view's name and the word "Model". The view models are stored in the same location in the **ViewModels** directory as their corresponding views in the **Views** directory.

For bigger projects or projects where the view models are used for multiple platforms, the **ViewModels** as well as the **Localization** directory (the UI independent components) can be moved to a separate ([PCL](#)) project which can be referenced by multiple application projects.

## Rules for views and view models

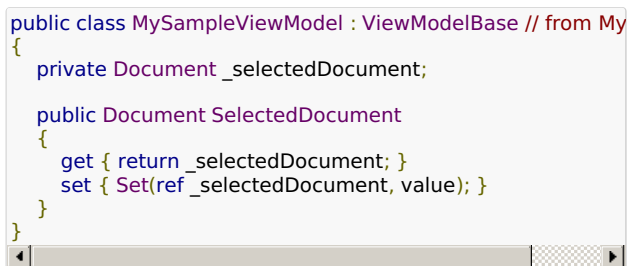
All views should follow the [MVVM pattern](#) to separate the user interface from the application logic. There is always a one-to-one relationship between the view and view model classes and instances. In other words, a view model class or instance belongs to exactly one view and a view has exactly one view model. Only the "parent" view has a reference to the view model, other views are not allowed to access another view's view model (except unit test classes).

It is not allowed to directly access the view object from the view model (also not through an interface). To send commands from the

view model to the view or call methods on the view, use a messenger (described later) or other IoC mechanisms (e.g. services).

If a view becomes too large, it is recommended to split it up into multiple views.

You should use a view model base class as well as an [INotifyPropertyChanged](#) base class to simplify observable property implementations like this:

A screenshot of a code editor showing the implementation of a C# class named MySampleViewModel. The class inherits from ViewModelBase. It has a private field \_selectedDocument of type Document. There is a public property SelectedDocument of type Document with a getter that returns \_selectedDocument and a setter that calls Set(ref \_selectedDocument, value).

```
public class MySampleViewModel : ViewModelBase // from My
{
    private Document _selectedDocument;

    public Document SelectedDocument
    {
        get { return _selectedDocument; }
        set { Set(ref _selectedDocument, value); }
    }
}
```

**Tip:** The page [ViewModelBase](#) from my library shows a sample implementation of a view model as well as the code of a view model base class. Also check out the page [ObservableObject](#) for a [INotifyPropertyChanged](#) base implementation and its usage.

## User controls vs templated controls

When a new user interface element is needed, you have to decide whether to implement a user control (view with view model) or a templated control (UI control).

My general rule for choosing between these two types is:

- Create a **user control** when implementing an application specific view with a view model and few dependency properties which have to be synchronized with properties of the view model. These views should be put into the **Views** directory.
- Create a **templated control** when implementing a reusable UI control without view model and/or lots of dependency properties which are directly bound to attributes in XAML. These controls should be put into the **Controls** directory.

**Further reading:** Read [this article](#) for more information about the differences of the two control types.

## View model instantiation and assignment

My recommendation is that the view model should be instantiated in XAML in the `Resources` element of the root element. Then it is bound to the `DataContext` property of the child element of the view's root element. The following sample XAML shows the described implementation:

```
<UserControl x:Class="My.Namespace.MySampleView" ...>
  <UserControl.Resources>
    <viewModel:MySampleViewModel x:Key="ViewModel" />
  </UserControl.Resources>

  <Grid DataContext="{StaticResource ViewModel}">
    ...
  </Grid>
</UserControl>
```

To access the view model in the code-behind of the view, simply add the following property to your view's code-behind class (for example in **MySampleView.xaml.cs**):

```
protected MySampleViewModel Model
{
    get { return (MySampleViewModel)Resources["ViewModel"]
    }
}
```

The benefit of instantiating the view models as a resource in XAML are:

- The Visual Studio's XAML editor "knows" the type of the view model and provides code completion and IntelliSense; also the XAML code is correctly updated when renaming bound properties.
- The view model can always be accessed even if the current data context is different – for example when binding to view model properties inside an item template.

Access the view model when current binding context is different:

```
<Grid DataContext="{StaticResource ViewModel}">
  <ItemsControl ItemsSource="{Binding Items}">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <TextBlock FontSize="{Binding SelectedFontSize, ...}"
          Text="{Binding}" />
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</Grid>
```

## Bind root element properties to view model properties

Because the view model object is only accessible after the `Resources` element, it cannot directly be used in the root element – for example to set the `Title` attribute of `Window`. To solve this problem, define the property using the tag syntax after the resource declaration:

```
<Window ...>
  <Window.Resources>
    <viewModels:MyWindowModel x:Key="ViewModel" />
  </Window.Resources>
  <Window.Title>
    <Binding Source="{StaticResource ViewModel}" Path="..." />
  </Window.Title>
</Window>
```

## Command implementation

All button, context menu or other actions are implemented as commands which are implemented either directly in the view model class itself or encapsulated in an own `ICommand` implementation.

All command property names have the postfix “Command”. The following sample code shows a command property in a view model:

```
public ICommand SaveProjectCommand { get; private set; }
```

The commands are instantiated in the view model constructor like this:

```
public MyViewModel()
{
  SaveDocumentCommand = new RelayCommand<Document>
  SaveDocumentAsCommand = new SaveDocumentAsCommand
}

public void SaveDocument(Document document)
{
  ...
}
```

The next code snippet shows the implementation of a command using the “original” [command pattern](#):

```
public class SaveDocumentAsCommand : CommandBase<Do
{
    ...
}
```

In XAML, a control can be bound to a command in the view model in the following way:

```
<Button Command="{Binding SaveDocumentCommand}"
        CommandParameter="{Binding SelectedDocument}" />
```

If the view model is not directly accessible in the current context, for example when inside a context menu, you can access the view model using the `StaticResource` keyword:

```
<Button Command="{Binding MyButtonCommand, Source={StaticResource MyViewModel}}" />
```

This trick can also be used when accessing other properties of the view model in a location where the view model is not directly accessible, i.e. the current data context is not the view model.

## Use dependency properties to pass parameters to a view

**Important rule:** It is not allowed to set the `DataContext` property in the XAML code where a view is instantiated. For example this is NOT allowed:

```
<MySubView DataContext="{Binding MySubViewModel}" />
```

As described in the previous chapter, only the view itself is allowed to set its `DataContext` property. To pass parameters to a view, always use dependency properties which then are propagated to the view's view model. The following code shows how to do this:

```

public class MySubView
{
    public static readonly DependencyProperty ProjectProperty
        = DependencyProperty.Register(
            "Project", typeof(Project), typeof(MySubView),
            new PropertyMetadata(default(Project), OnProjectChanged));

    public Project Project
    {
        get { return (Project)GetValue(ProjectProperty); }
        set { SetValue(ProjectProperty, value); }
    }

    private static void OnProjectChanged(DependencyObject o,
        DependencyPropertyChangedEventArgs args)
    {
        ((MySubView)obj).Model.Project = (Project)args.NewValue;
    }

    public ProjectPropertiesViewModel Model
    {
        get { return (ProjectPropertiesViewModel)Resources["View"]; }
    }
}

```

Now, a `Project` object can be passed “into” the `MySubView` instance:

```
<MySubView Project="{Binding SelectedProject}" />
```

This rule enforces the development of more encapsulated and decoupled views.

## Handling the view model life cycle

Sometimes, a view model needs life cycle methods to initialize and cleanup itself. The following code samples show how to implement them in a save manner. All life cycle methods in the view models should be implemented in the same way.

In the XAML code-behind:

```

public MyView()
{
    InitializeComponent();
    Model.Initialize();
    Loaded += delegate { Model.OnLoaded(); };
    Unloaded += delegate { Model.OnUnloaded(); };
}

```

And the view model should implement these two methods this way:

```

public class MyViewModel
{
    private bool _isLoading = false;

    public MyViewModel()
    {
        // TODO: Add your constructor code here
        // The ctor is always called, initialize view model so that
    }

    public void Initialize()
    {
        // TODO: Add your initialization code here
        // This method is only called when the application is running
    }

    public void OnLoaded()
    {
        if (!_isLoading)
        {
            // TODO: Add your loaded code here
            _isLoading = true;
        }
    }

    public void OnUnloaded()
    {
        if (_isLoading)
        {
            // TODO: Add your cleanup/unloaded code here
            _isLoading = false;
        }
    }
}

```

Now, you may ask why we need a separate `Initialize` method when we can put all code into the view model's constructor. The problem here is, that the constructor gets called from the Visual Studio designer but your code may only run when the application is running. To avoid this problem, it is recommended to put all logic which needs a running application (e.g. access to the service locator) into the `Initialize` method and the rest in the constructor (e.g. instantiation of command objects). This way, the Visual Studio designer only calls the view model constructor but not the view's constructor with the `Initialize` call. This ensures – when applied correctly – that the XAML designer is working correctly. Another benefit of the `Initialize` method is that it allows to inject objects into the view model – this is described in the next section.

**Note:** The [MyToolkit](#) library provides [view model base classes](#) which simplify the implementation of these life cycle methods.

## View model constructors and dependency injection

This chapter describes how to inject services into the view model using the [service locator](#) pattern or another IoC mechanism. I usually



use one of the following two ways of injecting services into the view models: Using an initialization method or using constructor overloads.

### Using an initialization method

If you want to avoid dependencies between the view model and the IoC framework, you should implement an `Initialize` method on the view model with the injected objects as parameters and call this method in the view's constructor:

```
public class MySampleView
{
    public void MySampleView()
    {
        InitializeComponent();
        Model.Initialize(ServiceLocator.Default.Resolve<IMyService>());
    }

    protected MySampleViewModel Model
    {
        get { return (MySampleViewModel)Resources["ViewModel"]; }
    }
}

public void MySampleViewModel
{
    private IMyService _myService;

    public void Initialize(IMyService myService)
    {
        _myService = myService;
    }
}
```

**Note:** The `ServiceLocator` class can be found in the [MyToolkit library](#) or in many other IoC frameworks.

In this implementation the view models have no dependencies to the IoC framework and are well suited for unit tests. However the client has to call the `Initialize` method which may not be obvious.

### Using constructor overloads

The other variant for implementing dependency injection is by providing two constructors: A default constructor with no parameters (for instantiation in XAML) and a constructor with all required service objects as parameters. The second constructor is used in unit tests with constructor dependency injection.

```
public MySampleViewModel()
: this(ServiceLocator.Default.Resolve<IMyService>()) { }

public MySampleViewModel(IMyService myService) { }
```

As you can see in the sample code, the default constructor always calls the other constructor with resolved service objects. The default constructor must not contain any logic. This implementation has the disadvantage that the view model has a dependency to the IoC framework. This is why I would prefer the first solution where a `Initialize` method is implemented.

## View model decoupling and messaging

It is recommended to use the messenger pattern (also known as [publish-subscribe pattern](#)) which essentially is a message bus to send messages from any location to unknown receivers. Every component – for example a view or a service – may subscribe to receive messages of a particular type.

In the MVVM context, this facility is used to send messages from the view model to its view or other UI components – for example to show a dialog or navigate to another page. It is not required to use the messenger pattern – you can also use an IoC container and use service objects instead: For example, you can use a dialog service instead of sending messages to show message boxes.

The following libraries provide a messenger implementation:

- [MyToolkit](#) (used in the samples below)
- [MVVM Light](#)

To use the messenger, register a given message type (in the sample code `ShowFileDialog` is the method which is called when the message has been sent):

```
Messenger.Default.Register<ShowFileDialogMessage>(t
```

To send a message use the following code:

```
Messenger.Default.Send(new ShowFileDialogMessage())
```

If a message provides a result, for example the

result of a “YesNo”-dialog, the result should be “returned” using a callback. This is because messages should be immutable and the result may be returned asynchronously.

The messenger implementation of the **MyToolkit** library provides the class `CallbackMessageBase` which can be used with the messenger’s `SendAsync` method. The `CallbackMessageBase` class provides two callbacks: A success and failure callback. The receiver of the message performs the desired action and must call either the `ProcessSuccessCallback` or `ProcessFailCallback` method.

The following code shows how to send a callback message (MyToolkit only):

```
var msg = new TextMessage(..., MessageBoxButton.YesNo);
msg.SuccessCallback = result =>
{
    if (result == MessageBoxResult.Yes)
    {
        // TODO: Add your code
    }
};
Messenger.Default.Send(msg);
```

The same using async/await:

```
var result = await Messenger.Default.SendAsync(msg);
if (result.Result == MessageBoxResult.Yes)
{
    // TODO: Add your code
}
```

The message receiver, which actually shows the dialog box, would look like this:

```
Messenger.Default.Register<TextMessage>(this, ShowMessa
...
private void ShowMessageBox(TextMessage msg)
{
    var result = MessageBox.Show(
        msg.Content, msg.Title,
        msg.Button, msg.Icon);

    msg.ProcessSuccessCallback(result);
}
```

**Note:** Message object should be immutable and results should be passed back by using callbacks as shown in the samples before.

### Location for message handler registrations

Global message handler registrations like showing dialogs, navigating to pages, etc. should be registered in the **App.xaml.cs** class on application startup. View specific messages

should be registered in the `Loaded` event of a view and deregistered in the `Unloaded` event to prevent memory leaks and problems multiple callback registrations.

## Always use typed XAML bindings

**Important rule:** Every binding in the XAML code must be typed.

This rule is needed to safely refactor view model and model classes and to find usages of bound properties. Usually this is no problem, because the view model is instantiated in the view's resources and automatically provides a type. If the type is not automatically provided, you should manually set the type as shown in the following XAML code:

```
<Window xmlns:MyNamespace="clr-namespace:MyNamespa
  xmlns:mc="http://schemas.openxmlformats.org/markup
  xmlns:d="http://schemas.microsoft.com/expression/bler
  mc:Ignorable="d"
  ...>
  ...
  <StackPanel d:DataContext="{d:DesignInstance Type=My
  ...
  </StackPanel>
```

## Localization

Localized strings – provided by a **.resx** or **.resw** file – should be loaded from a `Strings` property of the view model. The following code shows how to implement the base view model. In the sample, the resource file is called **Strings.resx** and automatically generates the `Strings` class which is instantiated in the property:

```
public class ViewModelBase
{
    private Strings _strings = new Strings();
    public Strings Strings
    {
        get { return _strings; }
    }
    ...
}
```

The XAML binding to load a localized string:

```
<TextBlock Text="{Binding Strings.LabelPerson}" />
```

Strings in dialog messages and other modal dialogs, can directly be loaded using the `Strings` class.

**Tip:** For **.resw** files (used in Windows Store

apps) this resource access class is not automatically generated. [This article](#) shows a way on how to solve this problem.

### Key prefixes for localized strings

All localized strings should be located in the same `Strings.resx` or `.resw` resource file. If you have bigger projects, you can distribute the strings into multiple resource files. To have better structure, the string keys should start with a prefix depending on their usage.

The following prefixes or postfixes are recommended:

- **Label\***: Labels which belong to an input control
- **GridLabel\***: Data grid headers
- **Message\*Title**: Titles for message boxes
- **Message\*Content**: Content messages for message boxes
- **Log\***: Log messages
- **Button\***: Button content texts
- **Header\***: Header texts to use in window titles, tab headers and similar locations
- **Menu\***: Strings to use in menu items (e.g. context menu)
- **Task\***: Texts which are shown in the task manager (e.g. "Loading Document")
- **RibbonButton\***: Headers for ribbon buttons
- **RibbonMenu\***: Labels used in the ribbon menu

### Final words

This article showed my way of doing MVVM. Do you have any suggestions or improvements? Are you doing it in a completely different way? Is something unclear or missing?



Tags: [.NET](#), [Architecture](#), [C#](#), [MVVM](#), [XAML](#)

**17 responses to "Recommendations and best practices for implementing MVVM and XAML/.NET applications"**



**Alok Arya** Says August 18, 2016 at 12:38 am

```
public MyView()
{
    InitializeComponent();
    Model.Initialize();
    Loaded += delegate { Model.OnLoaded(); };
    Unloaded += delegate {
        Model.OnUnloaded(); };
}

public void Initialize()
{
    // TODO: Add your initialization code here
    // This method is only called when the
    application is running
}
```

I think you mean

```
Initialized += delegate { Model.Initialize();
};
```

instead of,

```
Model.Initialize();
```

Am I correct ?

Because otherwise initialize method will also be called by XAML

[Reply](#)



**Rico Suter says:** [August 27, 2016 at 11:44 pm](#)

It is correct as is, because the code-behind class constructor is not called in the designer – only the XAML code is loaded and rendered (i.e. the vm ctor is called).

[Reply](#)



**Petr says:** [July 21, 2016 at 6:19 pm](#)

Hi,

when you recommend to create viewmodel in view's resources, how do you pass parameters to the view?

I.e. I have a "Customers view", and there is a command to edit customer record. What information and how you would recommend to pass from "masterViewModel" to "detailViewModel"?

Thank you very much.

Petr

[Reply](#)

[July 28, 2016 at 6:08 pm](#)



**Rico Suter says:**

You should treat your detail view as an independent component and add all required properties to the view and pass them from the detail view to the detail view model (i.e. add a Customer property to the detail view and propagate changes to the VM as described in the article). This way the detail view is decoupled from the master view and can also be used in other application parts.

[Reply](#)

**MVVM and WPF - Literature** **Wolfgang Riedmann Blog says:**

[...] Rico Suter: Recommendations and best practices implementing MVVM WPF applications [...]

[Reply](#)



**Stefan says:** [April 14, 2016 at 5:27 pm](#)

Hi Rico,

thank you very much for this very good and helpful article! Anyway I have some open questions where I miss some explanation.

Why do you only suggest to put the view models into a separate assembly? I'm a beginner in WPF/MVVM but my thought was that it would be a good idea to put also the models into a separate assembly which the view layer is not allowed to reference. This will ensure decoupling of view and model. Or is it allowed/okay to use the model in the view directly?

I have a big problem understanding the "flow" of the model objects in your szenario? How do you put the model into the view model when the view model is instantiated in XAML? Imaging the following: there is a MainView which is bound to MainViewModel which has a MainModel with all the configuration data inside, for example a DatabaseConnectionModel. Now I want to edit the database connection. I can open the EditDatabaseView with dialog service or messaging. The EditDatabaseViewModel is created in XAML of EditDatabaseView. How will the DatabaseConnectionModel be put into the corresponding EditDatabaseViewModel? Which mechanism do you use to tell the (EditDatabase)ViewModel which model it has to change?

Thanks and best regards,  
Stefan

[Reply](#)



**Rico Suter** says, 2016 at 11:50 am

Hi Stefan,

### **Question 1: Separation into multiple assemblies**

You also have to put the models in a separate assembly, otherwise you would end up with circular references. In your scenario, you would ideally implement three assemblies: Views => ViewModels => Models.

### **Question 2: Directly use models**

In every MVVM project, you have to decide whether to directly use your model classes or wrap them with a view model class (e.g. directly use the Person class or implement a PersonModel class). I think it depends on the project, but I mostly take the pragmatic way and directly use the model classes. Also see: <http://blog.alner.net/archive/2010/02/09/mvvm-to-wrap-or-not-to-wrap.aspx>

### **Question 3: Pass data to other view**

If your EditDatabaseView is opened via message bus, you would pass the DatabaseConnectionModel as a parameter message. The message listener would create an instance of EditDatabaseView and assign the DatabaseConnectionModel via a dependency property on the EditDatabaseView view to the EditDatabaseViewModel view model.

Is now everything clear?  
Rico

[Reply](#)



**Stefan Jany** says, 2016 at 5:05 pm

Hi Rico,

thank you for your detailed answer, which helped me understanding how it's working. Anyway one more question. How can I share data between different view models? For example when I have a main view model which has the state "IsInProductionMode", how can I use this information in other view



models which have been created in XAML later. Normally I would do composition of view models but when I create the view models in XAML that's now working. Do I have to pass the main view model as parameter to the new view model like described in the answer of question 3 or is there another more elegant possibility to do this?

Thanks,  
Stefan

[Reply](#)



**Rico Suter says** July 28, 2016 at 6:05 pm

I think you have three options:

1. Pass view model as parameter (not very elegant, high coupling between the two views)
2. Use a global event (i.e. messenger or message bus): Seems good for a global `IsInProductionMode` property
3. Think of the other views as independent components with properties for all required state (i.e. add a `IsInProductionMode` property to the sub view and bind the state from main view model)

As always, there's no right or wrong here, but I'd recommend option 2 or 3.

[Reply](#)



**Dan Corwin says** April 8, 2016 at 9:19 pm

Hi Rico,

You have a lot of good principles in this article which is very helpful to me since I am still learning MVVM. I have a best practice question for how to implement opening a dialog from the menu of a main window. For example opening a page setup dialog. I have found 2 ways to do this:

Define the Click property of the menu item to call a method on the main window (e.g. Click="OnPageSetup"). Then the View.MainWindow.OnPageSetup(object

sender, EventArgs e) can open the View.PageSetupDialog and associate it with the corresponding ViewModel object. Have an ICommand property on the MainWindowViewModel like PageSetupCommand that could be invoked to open the PageSetupDialog. Then in the App startup code after it constructs the MainWindowViewModel it could assign the PageSetupCommand with an appropriate implementation.

Notice that in both cases the ViewModel doesn't reference the View. Case 1 goes from View to View directly. Case 2 is routed via the ViewModel. I'm wondering if there is any benefit to case 2 or if there are any principle that are violated with case 1.

Any thoughts are welcome.

Best regards,  
Dan

[Reply](#)



**Rico Suter** says, April 14, 2016 at 1:53 pm

Hi Dan,

I'd recommend to implement an ICommand property on the view model whose action sends a new ShowPageSetupMessage message with all required parameters. The message handler is registered on app startup and shows the page setup dialog. You can also use a dialog service interface (injected via DI) in your VM which has a method to show the dialog and which can be mocked in your unit tests...

Is your question answered?

Best regards Rico

[Reply](#)



**Panos Roditis** says, April 16, 2016 at 6:33 pm

Hi,

Any suggestions or best practice for triggering a new view (ex a Dialog) to collect user input, from inside the ViewModel?

Thank you.

[Reply](#)



**Rico Suter** says, April 17, 2016 at 5:03 pm

I see two possibilities:

- Use the messenger pattern and a message with a callback providing the user input
- Inject a dialog service (interface) into your view model so that it can be easily mocked in your unit tests

[Reply](#)



**Mike says:** January 15, 2016 at 4:14 am

Hi Rico!

Question about "Project file structure":

I don't need to use your entire toolkit (although it's great), I need only part of ObservableObject.cs with implemented INotifyPropertyChanged interface + one Set method.

Where should ObservableObject class go in file structure? "Helpers", "Common"? Or should I name it "ViewModelBase" and put it in ViewModels folder?

Thanks for advice in advance

Best regards

Mike

[Reply](#)



**Rico Suter says:** January 15, 2016 at 9:36 am

Hi Mike,

There's no correct or wrong answer. Personally, I'd put it into a "Common" directory and use the name ObservableObject as it has no relation to view models but only provides the functionality to track object changes. If you need view model specific logic, you can create a ViewModelBase class and inherit from ObservableObject. But it is really up to you and your taste how to name the classes and where to store them...

Best regards

Rico

[Reply](#)



**Veux says:** May 18, 2015 at 3:28 pm

Hi,  
thank you for the great blog entry.  
I have some question when I try to follow your rules.

I often have the Situation that I put model classes or data transfer objects into the view models constructur. How do I do that if I follow the rule to instantiate the view model in the XAML code ?  
Or if I have to jump to another page e.g master-detail pages ?  
What you suggest to "insert" Parameters into the view model ?

Cheers Veux

[Reply](#)



**Rico Suter** May 18, 2015 at 3:38 pm

Hi Veux,

As described in the article, I'd recommend to define the parameters as dependency properties on the view and push them from the view to the view model. As shown in the chapter "Use dependency properties to pass parameters to a view" you can use simple property assignments and evaluate all loaded properties in the OnLoaded method. Does this answer your questions? Or can you describe your problem in more detail?

Best regards  
Rico

[Reply](#)

### Leave a Reply

Your email address will not be published.  
Required fields are marked \*

**B** *I* | | | ?

Comment

Name \*

Email \*

Website

☐ Notify me of followup comments via e-mail.

You can also [subscribe](#) without commenting.

Post Comment