**MVVM Pattern Example**

MVVM Pattern Made Simple

**Important Note**

Friends, I would very much appreciate if you leave me a line or two in the comments section stating how you think this article can be improved and what other topics on MVVM you want me to cover. Thanks.

**Introduction**

As many companies are switching from WinfForms to WPF/Silverlight, several project managers recently asked me almost identical questions about MVVM:

* What is MVVM Pattern?
* What are the advantages of the MVVM approach?

The purpose of this article is to answer these questions and to explain the MVVM pattern in the simplest possible way.

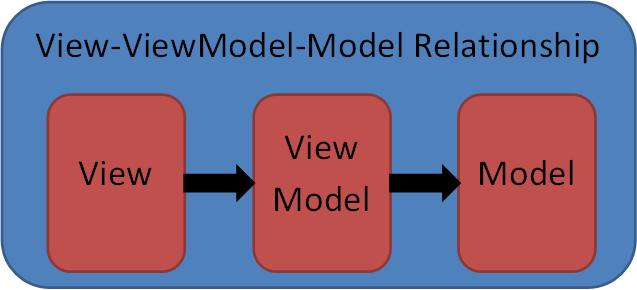
I assume that the article readers have not had any previous exposure to the MVVM pattern but have some knowledge of WPF or Silverlight.

Time permitting, I plan to write more MVVM articles which will include comparison of different MVVM frameworks and introduce a new MVVM framework.

**MVVM Overview**

Model-View-ViewModel (MVVM) pattern splits the User Interface code into 3 conceptual parts - Model, View and ViewModel out of which the concept of the ViewModel is the new and the most exciting.

* Model is a set of classes representing the data coming from the services or the database.
* View is the code corresponding to the visual representation of the data the way it is seen and interacted with by the user.
* ViewModel serves as the glue between the View and the Model. It wraps the data from the Model and makes it friendly for being presented and modified by the view. ViewModel also controls the View's interactions with the rest of the application (including any other Views).



ViewModel code can (and should) refer to the Model, but the Model classes (if separate from the ViewModel) should not be aware of the ViewModel. View should be aware of the ViewModel, but ViewModel should be not aware of the View. In the diagram above, the arrows are showing which MVVM part is aware of which.

Similar to the models in the older patterns, the Model within MVVM pattern is simply data coming from the service or the database. Quite often, the Model can be built to be part of the ViewModel. Because of that, in our samples, we'll skip the model and concentrate primarily on the ViewModel, the View and interactions between them.

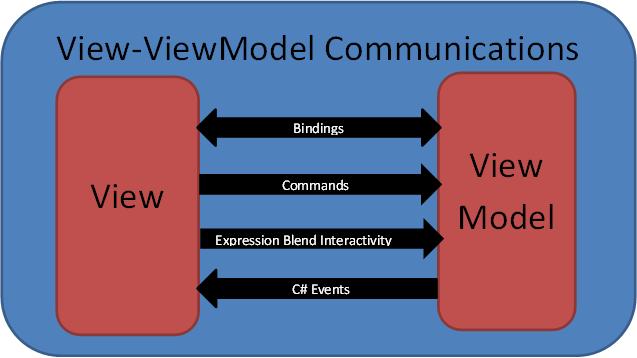
**WPF Concepts Related to the MVVM Pattern, Communications between the View and the ViewModel**

MVVM pattern became possible because of the following new concepts that came into being as part of the WPF (and later Silverlight).

* WPF Bindings - Each binding connects two properties (possibly on two different objects) so that if one of them changes, the other changes too.
* WPF Data Templates, which convert non-visual data (ViewModel) into its visual representation (View).
* WPF Commands (or Microsoft Expression Blend SDK interactivity behaviors) serve to pass the events from the Views to the ViewModels.

WPF Bindings, WPF Commands and MS Expression Blend SDK interactivity functionality provide the necessary plumbing for communications between the View and the ViewModel. Another method can be employed for such communications - C# events can be used to trigger a change within a View when something happens within its ViewModel. This method should be avoided whenever possible because it implies code behind within the view (to register an event handler). Instead, one should use the bindings (perhaps combined with the triggers) to pass information about changes within the ViewModel to the View.

Here is a diagram depicting different ways of communicating between the View and the ViewModel:



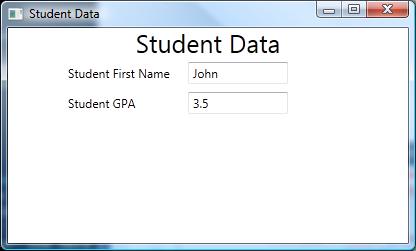
When looking at this diagram, one might ask why the Binding and C# Events arrows are pointing from the ViewModel to the View even though, as stated above, the ViewModel is not aware of the View. The answer is that the arrows on this diagram are pointing from an MVVM part that causes a change to the part which receives the change. The Bindings are set within the View and even though the ViewModel is not aware of the View, the changes to the ViewModel still propagate to the View via the Bindings. In case of C# events, the event handlers on ViewModel's events should also be set within the View's code behind, so that the View functionality will trigger a change when changes occur within the ViewModel.

**WPF Bindings**

Below is a short (and by no means comprehensive) overview of the WPF bindings needed for understanding the MVVM pattern. For more information on the bindings, try e.g. [Data Binding Overview](http://msdn.microsoft.com/en-us/library/ms752347.aspx), or [Data Binding in WPF](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx) or plenty of other resources available on the internet.

WPF binding connects two properties (called binding target property and binding source property), so that if one changes, the other changes too. The target property of a binding should always be a WPF dependency property (for the definition of a dependency property - check WPF tutorials). The source property of a binding should be either a dependency property or fire PropertyChanged event on the property change.

A simple binding sample is located under "BindingSample" solution. It displays student data (student first name and student grade point average) within its window:



The student data is represented by an object of class StudentData. One can see that StudentData has two properties: StudentFirstName and StudentGradePointAverage:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

public class StudentData : INotifyPropertyChanged

{

...

string \_firstName = null;

public string StudentFirstName

{

get

{

return \_firstName;

}

set

{

\_firstName = value;

OnPropertyChanged("StudentFirstName");

}

}

double \_gradePointAverage;

public double StudentGradePointAverage

{

get

{

return \_gradePointAverage;

}

set

{

\_gradePointAverage = value;

OnPropertyChanged("StudentGradePointAverage");

}

}

}

These properties serve as the source properties. When changed, both these properties fire PropertyChanged event that carries the corresponding property name in its event argument. This allows the binding to update its target properties to the new values. StudentData plays the role of a ViewModel in this simple example.

The View is located within the *MainWindow.xaml* file. It simply displays the student data within a Grid panel. The important parts are the WPF bindings connecting the ViewModel's StudentData properties to properties of the TextBoxes within the View:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

<TextBox **Text="{Binding Path=StudentFirstName}"**

Grid.Row="1"

Grid.Column="2"

VerticalAlignment="Center" />

...

<TextBox **Text="{Binding Path=StudentGradePointAverage}"**

Grid.Row="2"

Grid.Column="2"

VerticalAlignment="Center" />

The TextBoxes' Text properties serve as the binding's targets.

Looking at the XAML code above, one might wonder how the Text property knows which StudentData object it should connect to. There are different ways to specify the source object within the binding, e.g., by setting Source,ElementName or RelativeSource properties of the binding. However, when those properties are not set, as in our sample, the Binding's source object is assumed to be specified by the DataContext property of the corresponding visual element. DataContext property has a feature that it propagates down the Visual Tree from ancestor to its descendants, unless changed explicitly or unless it hits a ContentControl or an ItemsControl on its path (to be explained later). So once the DataContext is set to the root element of the Visual Tree, e.g. the Window, most elements under that tree will have the same DataContext. MVVM pattern uses this frequently by setting the View'sDataContext property to contain the ViewModel for that View.

In our case, the DataContext is set within MainWindow class constructor inside *MainWindow.cs* file:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

this.DataContext = \_studentData;

Correspondingly, the two TextBox elements within *MainWindow.xaml* file will have the same DataContext as theirMainWindow ancestor and the bindings will connect the StudentFirstName and StudentGradePointAverageproperties from \_studentData object to the corresponding TextBox'es properties.

When source property and target property depend on each other, but are not equal (they can even be of different types) WPF Binding can employ the ValueConverters which transform the source property value into the target property value and vice versa.

DataTemplates

The above binding sample shows how to bind properties on a ViewModel to the corresponding properties within the View. However if we want to present multiple data objects of the same structure, it is better to factor out the presentation code as a DataTemplate.

SimpleDataTemplateTest solution contains a DataTemplate sample. The ViewModel is exactly the same as in the previous sample - represented by StudentData and it is attached to the View's DataContext within MainWindowclass'es constructor. The View itself, however, is defined by the DataTemplate "StudentView" defined within the resources of MainWindow inside *MainWindow.xaml* file:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

<DataTemplate x:Key="StudentView"

DataType="this:StudentData">

...

</DataTemplate>

To display the StudentData object using the DataTemplate, a ContentControl element is employed:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

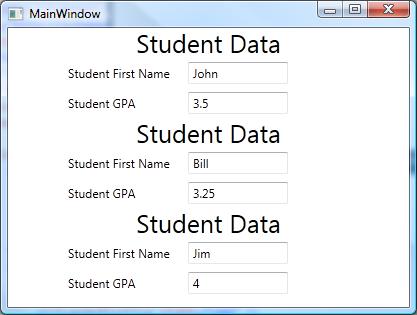
<ContentControl Content="{Binding}"

ContentTemplate="{StaticResource StudentView}" />

Its Content property is bound directly to its DataContext (since the binding path is not specified), making itsContent to be set to the StudentData object. It's ContentTemplate property is set to the "StudentView" data template. The ContentControl "marries" the data and the DataTemplate producing the visual representation of the data.

Note that the Visual children/descendents of a ContentControl will have the ContentControl's Content property as their DataContext. Also note that DataContext does not automatically propagate to the Content property of the ContentControl. In order for the Content to be set to the ContentControl's DataContext, it has to be explicitly bound to it.

ContentTemplate is ideal to display a signed data item in a View, but if one has a collection of data, ItemsControlclass can be employed instead to display the whole collection. For ItemsControl sample, see*ItemsControlDataTemplateTest.sln* solution:



In this solution, the ViewModel is represented by StudentListViewModel class which contains a collection ofStudentData objects within its TheStudents property. ItemsControl element is used to display the collection of students:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

<ItemsControl ItemsSource="{Binding TheStudents}"

ItemTemplate="{StaticResource StudentView}" />

ItemsControl's ItemsSource property is bound to the collection of students and ItemTemplate property provides the DataTemplate for each element within that collection.

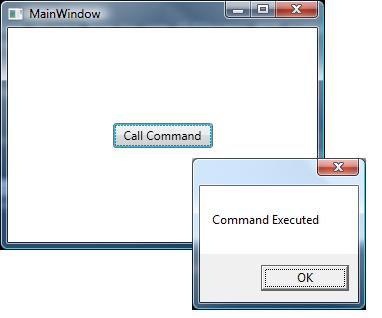
Note that the ItemsControl's children cells will have the corresponding element from the ItemsSource collection as their DataContext.

Using Commands for Passing the Events from the View to the ViewModel

Now, suppose that we have a button as part of the View. We want some action to be taken by the ViewModel when the button is pressed. This can be achieved by employing the WPF Commands or Microsoft Expression SDK interactivity functionality. I prefer the 2nd method, but using the commands is more wide-spread so, we'll discuss it here first.

Command objects are derived from ICommand interface which has two important methods: CanExecute method controls whether the corresponding control (e.g. Button or MenuItem) that is bound to this command is enabled or disabled, and Execute method specifies the action to be taken once the Button or MenuItem is clicked. Here, we are using the DelegateCommand which simply allows us to set arbitrary delegates for these two methods. These delegates are passed within the constructor of the DelegateCommand.

Command sample is located under *CommandSample.sln* solution. After starting the solution, press the button in the middle and you'll see the message with the popup:



CommandSampleViewModel represents the ViewModel for the sample. It contains a DelegateCommand as itsTheCommand property. Its "CanExecute" delegate always returns "true" thus keeping the corresponding button always enabled. Its "Execute" delegate is implemented by the "OnExecute" function which simply shows a Message Box with "Command Executed" text. Please note, that in real life coding, you should not put any visual code within the ViewModels, even the calls to MessageBox.Show. Here I did it just for brevity and clarity sake so that the OnExecutefunction would perform something that the users can see.

The Button within the view file *MainWindow.xaml* binds its Command property to the TheCommand property of the view model...

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

<Button Content="Call Command"

Width="100"

Height="25"

HorizontalAlignment="Center"

VerticalAlignment="Center"

**Command="{Binding TheCommand}"**/>

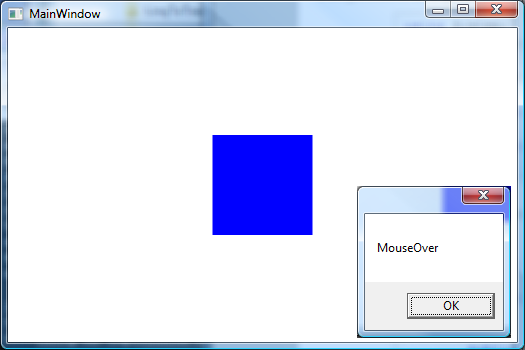
...which ensures that the command fires on the button click.

Using Microsoft Expression Blend SDK Interactivity Functionality for Passing the Events from the View to the ViewModel

The WPF commands have a shortcoming that they can only be attached to a few visual controls that have Commandproperty, e.g. Button or MenuItem and they only fire on Click events. Microsoft Expression Blend SDK Interactivity functionality is much more generic in a sense that it allows to trigger corresponding methods on the ViewModel for almost any event that occurred on almost any visual element within the View.

Microsoft Expression SDK sample can be found in *MSExpressionSDKInteractivitySample.sln* solution. To make the Microsoft Expression Blend SDK interactivity functionality available to the project, we need to add two DLL files to the project's references: *Microsoft.Expression.Interactions.dll* and *System.Windows.Interactivity.dll* files. These files are part of Microsoft Expression Blend SDK but can be used separately from the rest of the SDK. They are located under*MSExpressionBlendSDKDlls* directory.

The sample shows how to trigger a ViewModel's method when the MouseEnter event is fired on a Rectangle object. After starting the sample, move the mouse pointer over the blue rectangle in the middle, and you'll see a MessageBoxpopping up.



The ViewModel is represented by MSExpressionBlendSDKSampleViewModel class. It has just one methodMethodToCallOnMouseEnterEvent which pops up a message box (remember that you should not put any visual code including the MessageBox related code into the real life ViewModels. I put it here simply for the sake of clarity). The View is calling this method via Microsoft Expression Blend SDK's EventTrigger and CallMethodAction:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/278901/MVVM-Pattern-Made-Simple)

<Rectangle x:Name="TheRectangle"

Fill="Blue"

Width="100"

Height="100"

VerticalAlignment="Center"

HorizontalAlignment="Center">

**<i:Interaction.Triggers>**

**<i:EventTrigger EventName="MouseEnter"**

**SourceObject="{Binding ElementName=TheRectangle}">**

**<se:CallMethodAction MethodName="MethodToCallOnMouseEnterEvent"**

**TargetObject="{Binding Path=DataContext, ElementName=TheRectangle}"/>**

**</i:EventTrigger>**

**</i:Interaction.Triggers>**

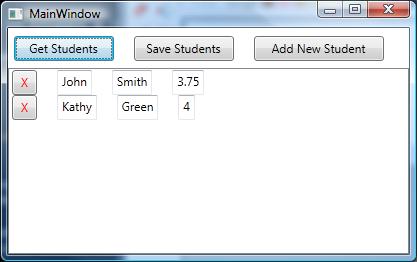
</Rectangle>

Unlike the commands, Microsoft Expression Blend SDK functionality does not provide a way to disable a visual element on which they "listen" for events. It makes perfect sense, because they can be used with the visual elements that cannot be disabled, e.g., Rectangle in the sample above. If you want to provide an ability to control whether an element is enabled or not via the ViewModel, you can simply bind that element's IsEnabled property to some boolean property within the ViewModel.

**MVVM Sample**

While the samples above showed how to use different WPF components to connect the View and the ViewModel, this example puts everything together and shows the MVVM pattern in action.

The MVVM sample can be found under *MVVMSample.sln* solution. It allows to get a list of students from a mock server, to add or remove entries corresponding to the individual students, to modify individual student's properties and to save the list to the mock server:



The ViewModel for the whole application is represented by StudentListViewModel class. This class contains a collection of ViewModel objects of type StudentViewModel class. StudentViewModel class defines the ViewModel for the individual students.

When the sample is started "Save Students" and "Add New Student" buttons are disabled to force the user to load the student data from the mock server. Once the student data is loaded, those buttons get enabled. These button's enabled/disabled state is controlled by IsSaveStudentsActionEnabled and IsAddStudentsActionEnabledproperties of StudentListViewModel class. The actions that need to be performed once any of these buttons are clicked are defined by the ViewModel's methods GetStudentsAction, SaveStudentsAction andAddStudentAction and are mapped to the button's click events using Microsoft Expression Blend Interactivity functionality.

ViewModel representing the individual students is defined by objects of StudentViewModel class, which has three properties FirstName, LastName and GradePointAverage. It also defines a method DeleteStudentAction used to delete the object from its parent collection. This method calls DeleteStudentEvent event which triggers the removal.

The corresponding Views are defined as the DataTemplates within *StudentsViewResources.xaml* resource dictionary located under *XAMLResources* folder. StudentView is the DataTemplate for a single student defined byStudentViewModel object and StudentListView is the DataTemplate for the whole application.

The MainWindow class contains only a ContentControl that converts the StudentListViewModel object to display as StudentListView.

**MVVM Discussions**

Different MVVM Flavors

All the MVVM samples presented above have their Views represented by the DataTemplates. I thinkDataTemplates are the best for representing the views because their usage provides the clearest separation between the visual and non-visual functionality. In some projects, however, people prefer to use UserControls as views. In my experience, UserControls or CustomControls should be avoided as much as possible within the MVVM pattern, since they are more heavy weight than the templates and have some code behind, which can easily degenerate into containing some business logic.

When to Use UserControls

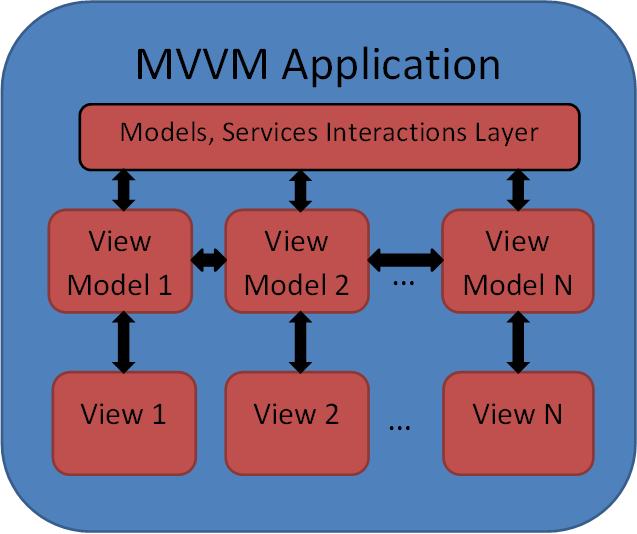
There are, however, two important cases when building a UserControl or a CustomControl is required:

* When you are building a brand new control with brand new capabilities. For example, WPF does not contain built-in charting functionality, so, when you build a charting control, you are forced to build a UserControl or a CustomControl that takes some data input and draws a chart based on it.
* When you are forced to use the functionality of an existing control and that existing control does not have the required capabilities, or does not provide ways to bind to all the data, you will have to extend such control to provide the required capabilities and to fit it in as MVVM View's building block. Example of such a control would be a DevExpress GridControl which has great features and great performance but was not built with MVVM in mind and therefore requires a lot of adaptations.

In summary - do not use UserControls or CustomControls to put together the Views within MVVM patter - this is better achieved by the DataTemplates (in conjunction with the DataTemplateSelectors, ControlTemplates,Styles and StyleSelectors). Use User and Custom controls to create building blocks for the Views within MVVM pattern instead, when necessary.

Multiple Views and ViewModels within an Application

Based on the discussion above, here is the diagram of communications between different modules within a well-built MVVM application



Each one of the Views interacts only with its personal ViewModel. The ViewModels, however, interact with the rest of the functionality: other ViewModels, Models, Services, etc. Please note that the ViewModels still need to be as independent of each other as they can be, to adhere to the separation of concerns principle. The fact that some ViewModels need to communicate between each other does not imply that they have to know each other's exact types. They can communicate e.g. via common interfaces or an event aggregator (for an example of an event aggregator, see e.g. [Prism for Silverlight/MEF in Easy Samples. Part 3 - Communication between the Modules](http://www.codeproject.com/KB/silverlight/PrismTutorial_Part3.aspx)).

Which View Properties should be Controlled by the ViewModel

Potentially the ViewModel can contain any types of non-visual object including the objects used only within the Views, e.g. Brushes or text label string, etc. For example, we could add BackgroundBrush property to theStudentViewModel class described above and bind our View's Background property to it. On the other hand, there is no reason for StudentViewModel to contain information about the background colors. The background brush can be specified very well by the View. Putting it into the ViewModel will only confuse potential users of the class. The rule of thumb is that the ViewModel should only contain the data displayed within the view and methods and properties that can be used for interacting with other ViewModels. Different brushes, colors, animations, changes of colors and shapes that do not affect any other Views can very well be handled by the View itself with the help of the WPF provided tools like Triggers StyleSelectors and DataTemplateSelectors.

**Advantages of the MVVM Pattern**

The major advantage of the MVVM pattern is that it provides the best [separation of concerns](http://en.wikipedia.org/wiki/Separation_of_concerns): under MVVM, View is in charge of the visual representation while the non-visual ViewModel is in charge of all of the interactions with the rest of the software including the Model, the Services (usually via the Model) and the rest of the ViewModels. All the MVVM advantages derive from this feature:

* **Flexibility and Customization** - Different Views can be used with the same ViewModels allowing completely different visual representations of the same functionality depending on what different customers want.
* **Re-use** - Because of the separation of Visual and non-Visual functionality, both the Views and the ViewModels have higher potential for re-use than if the Visual and non-Visual functionality were mixed, since e.g. a non-visual functionality usually would not be able to make use of a functionality that contains a Visual part to it.
* **Separation of the UI design and development** - MVVM makes it possible to separate the work of the developer and the designer as clearly as possible: at first the developer can produce an application with the crudest possible GUI. Then the designer, using designer tools will be able to modify the GUI (the Views) without touching any non-visual code.
* **Testing** - Writing automated tests for a Visual application is not easy. The View - ViewModel separation allows the ViewModel to be unit tested without the view. Since the ViewModel is in charge of the interactions with the rest of the application, such unit tests would cover the most important functionality while the View tests will only have to contain testing of the visual features, e.g. colors, animations, etc.

**Some MVVM Pattern History**

Model-View-ViewModel (MVVM) pattern evolved together with WPF as a new way of designing and building Visual applications.

It came naturally out of new programming concepts developed as part of WPF such as Bindings and DataTemplates. MVVM pattern was first introduced by Martin Fowler in [Presentation Model](http://martinfowler.com/eaaDev/PresentationModel.html) and John Gossman in [Model-View-ViewModel (MVVM) pattern](http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx). Since then, a lot of other people wrote about this pattern - the most famous article is probably by Josh Smith in [WPF Apps With The Model-View-ViewModel Design Pattern](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx).

**Conclusion**

The purpose of this article is to present the MVVM pattern to people who have some knowledge of WPF or Silverlight but did not have any previous exposure to the MVVM. I'd love to hear from you about possible ways of improving clarity of the presentation.

Bottom of Form