MVVM Architecture

# Basic Architecture

Recently a new arrangement of components within the GUI has entered the stage. The MVVM takes a fresh look at structuring the responsibilities in the Display Layer and achieves high flexibility while streamlining the dependencies between components.

## Model

The **Model** is a collection of classes describing the underlying domain. Being domain-specific they lack view-specific properties and are allowed to block for longer periods of processing.

Pros:

* The Model can be re-used in various scenarios.
* Unit-testing the Model pays off across the many users of the library.

Cons:

* The Model may carry data that is not needed in a particular use-case, leading to bloated runtimes.

## ViewModel

The **ViewModel** wraps, combines and enhances the Model for with display logic. The interface presented by the ViewModel is a superset of the functionality needed by the intended View. This includes data transformations, latency hiding, progress indicators, storage for keeping selected items and more. Most of the functionality formerly implemented in a Controller or Presenter would go here. The major difference is the presentation of this functionality as easily bindable Properties and Commands.

Pros:

* Unit-tests of the ViewModel do not need a message loop or similar GUI constructs, thus reducing test-complexity.
* The complete functionality of the UI can be seen in the ViewModel.

## View

The **View**’s only responsibility is to faithfully show the ViewModel’s surface to the user and translate his or her input back into modified properties or called commands. Since GUI surfaces are inherently hard to unit-test and debug, this reduced scope greatly lowers the amount of work necessary to assure the proper working of the UI. With advanced toolkits, like WPF, which have databinding as first-class constructs, the UI-oriented structure of the ViewModel pays off twice by enabling a straight forward consumption.

# Putting MVVM in Context

The generated Kistl.Objects.Client form the Model. The client generator already creates INotifyPropertyChanged and INotifyCollectionChanged implementations to facilitate easier ViewModel construction.

Currently, the ViewModel is manually constructed and resides in Kistl.­Client/­Kistl.­Client.­Presentables. At the root of all ViewModel classes is the PresentableModel, which stores the GUI and Data contexts and carries a State flag to signal when the ViewModel finished loading and/or processing data. The names Presentable and ViewModel will be used interchangeably.

The View is toolkit specific and is located in provider assemblies. Currently there is an ASP.NET (Kistl.­Client.­ASPNET) and a WPF (Kistl.­Client.­WPF) client. A skeletal WinForms client is prototyped in Kistl.­Client.­Forms, but needs much more work.

The default connection between Models and ViewModels is made by the **ObjectClass.­DefaultModel** reference, which points to a TypeRef describing an appropriate PresentableModel for instances of this class. Each Presentable can be displayed in one or more **Layout**s. The Layout carries configuration information like whether or not a string value should be displayed in one or more lines. View Providers can register controls for Layouts. The infrastructure then chooses a control from the current provider which is registered for the most specific applicable layout.

## Connecting the Pieces

The first thing a client does is instantiating the first WorkspaceModel. The workspace is a representation of an IKistlContext and thus a user’s transaction. The default layout for a WorkspaceModel is the WorkspaceLayout. To allow basic navigation, the WorkspaceModel exports a (lazily loaded) list of ModuleModels, which in turn provide access to the contained DataTypeModels and so on.

To actually display the various ViewModels, the Kistl.Client.Presentables.ModelFactory creates DefaultViews via the default layout of the ViewModel.

Since the View is in the ultimate control of the screen, it may choose to use various non-default Views of the models to facilitate the intended user experience. Depending on the situation, it may do so directly or by calling back into the ModelFactory with a non-default Layout.

# Layout vs. Formatting

The appearance of a given Presentable can be influenced on two levels. The first is its **Format**. The format determines the top-level or “outer” style. For example, a DataObjectModel can be displayed with the formats “FullDisplay” (all properties available), “Summary” (only the most important properties) or “LineItem” (one-line summary).

The second level is the **Layout**. This specifies the details or “inner” style. The layout of a “Summary” would detail which values of the displayed Presentable go where on the screen while the layout of a text field might specify how tall a multi-line input field should be.

While the Format influences the choice of View, the Layout will be interpreted by the chosen View.

**Detailed example:** a data entry form for editing a Person. To be able to edit all properties of the object, the format for the whole form would be FullView with a ReadOnly property set to false to allow editing. The layout of this view could specify that the portrait of the person should be displayed in the top right corner of the form while the rest of the properties should be arranged as a list of editable fields. For the picture the format is PictureChooser with a “Picture, Path and Browse Button” layout. For most the editable fields the format is SingleLineTextBox, while the CV property has the MultiLineTextBox format, with a “200 lines” layout to provide enough space to edit. See the “Format vs Layout Example” for a graphical representation.