CHAPTER

Presentation Layer

1.- SITUATION IN N-LAYER ARCHITECTURE

This section describes approaches to architect the User Interface/presentation layer. We will begin by introducing patterns which could be applied to the presentation layer and then drill down on applying these different patterns using .NET technologies.

The following diagram shows how the Presentation layer typically fits within our Domain Oriented N-layer Architecture:

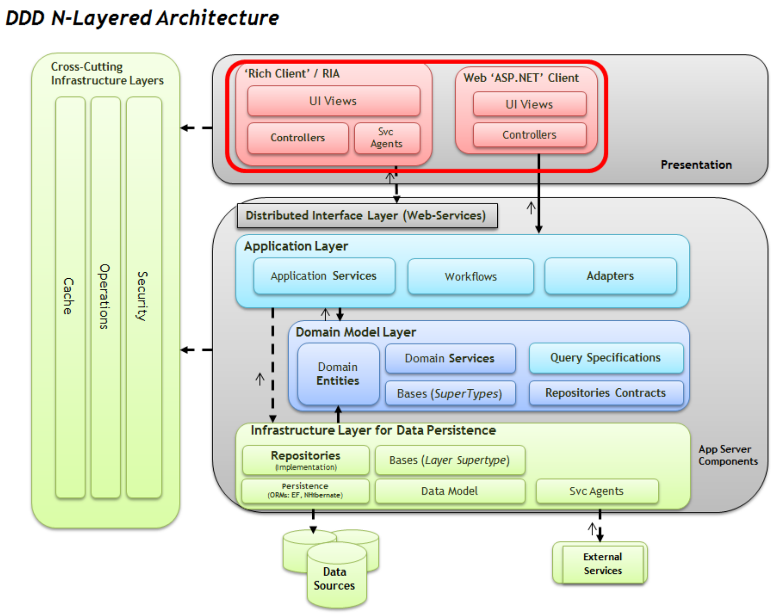


Figure 1.- Domain-Oriented N-layered Architecture

The Presentation Layer includes different elements, such as Views, Controllers, Models, etc. However, before reviewing aspects of the Architecture, we will introduce important aspects regarding the nature of this layer. This layer´s function is to introduce the user to business concepts through a User Interface (UI), facilitate the operation of these processes, report on their condition and implement the validation rules of that interface. In the end, from the end user's point of view, this layer is the "application" itself. While its important to ensure that good architectural practices are followed, for this layer ensuring a great user experience is of paramount importance.

One of the peculiarities of the user interface is that it requires skills beyond the developer's scope, such as artistic design skills, knowledge of accessibility, usability, and control of the location of applications. Therefore, it is highly recommended that professionals in this area, such as a graphic designer and usability experts, work together with developers to achieve a high-quality user interface. It is important to follow the right approaches and tools to enable co-operation between roles while building this layer.The developer will program in the chosen object-oriented language, creating the presentation layer logic, and the designer will use other tools, such as Microsoft Expression or SketchFlow, and technologies, such as HTML or XAML, to create the GUI and make it interactive, among other things.

A very important consideration is that this layer must be tested as the lower layers. It will be use patterns to make easier the separation of concerns for testability, that means decouple the view from underlying layers to promote automated testing of lower layers, to be able to develop them using TDD and have unit tests for those layers. Therefore, there should be a mechanism to automate such testing and to include it in the process of automated integration, whether it is continuous or not.

2.- REQUIREMENT TO INVEST IN USER INTERFACE

The investment of companies in applications with intuitive user interfaces (simple and/or tactile) has been possible thanks to the increase in consumers in devices through which they can interact using a visual interface using traditional methods such as using pointing devices and using more contemporary methods such as touch and multi-touch.

A picture speaks a thousand words and the advantages of having a visual interface which is appealing results in productivity increases, reduces expenses and sales growth. Many studies have been conducted on this subject and all of them favor the argument of having intuitive user interfaces.

Large companies such as Microsoft, Apple, Google, Yahoo or Amazon invest a lot on user experience. An efficient design for a user interface enables users to solve tasks more quickly and in the best possible way, thereby providing a great impact on user productivity. Please note that a well-designed and optimized user interface involves reducing the chances of users making mistakes and leads to an improvement in productivity. It is not only easy to use and intuitive, but fast. Users will be able to do more in less time; again increasing productivity. Psychology also plays a role: if a good user interface does not create problems and failures, its users will feel more comfortable when they are working and, therefore, become more productive.

Nowadays, expense reduction is one of the most important things for companies to accomplish. If we think we have intuitive, user-friendly tools, the investment in training or documentation of the tool may be reduced. Another reduction we will see is that when users feel comfortable using the tools they require less support. Another very important factor is using the proper UI technology to reduce complexity and therefore the cost in the deployment of the application.

The most important issue is market differentiation and the ability to gain competitive advantage by providing users with better user experience than what is offered in the current market. It is said that first impressions are what sell-- a bad image can turn a very powerful product into a terrible failure.

3.- THE NEED FOR ARCHITECTURE IN THE PRESENTATION LAYER

Designing the presentation layerand architecting the presentation layer during early stages considerably reduces development time and cost associated with subsequent design changes in the presentation layer and lower layers.

3.1.- Decoupling Between Layers

Investing time in the separation between the presentation logic code and the interface reduces the expenses incurred by future changes, but it also favors cooperation between designers, developers and project managers, which helps to avoid downtime due to poor communication. If the architecture is done in such a way that the designer's work does not interfere with the developer's work, it will reduce the project's completion time.

The main problem we encounter is coupling between components and layers of the user interface. When each layer knows how the other layer does its job, the application is considered to be coupled. For example, in an application typical coupling occurs when the search query is defined and the logic of this search is implemented in the controller of the "Search" button that is in the code behind. Since the requirements of the application change and we have to modify it as we go along, we will have to update the code behind the search screen. If, for example, there is a change in the data model, we will have to make these changes in the presentation layer and confirm that everything is working correctly again. When everything is highly coupled, any change in one part of the application can cause changes in the rest of the code and this is an issue of complexity and quality of the code.

When we create a simple application, such as a movie player, this type of problem is not common because the size of the application is small and the coupling is acceptable and manageable. However, in the case of a Line of Business Application

(LOB), with hundreds of screens or pages, this becomes a critical issue. As the size of a project is increased, it becomes more complex and we have more user interfaces to maintain and understand. Therefore, the code behind approach is considered an anti-pattern in business line applications.

3.2.- Performance Trade-Off

On the other hand, we have performance requirements. Optimizations frequently consist of complicated tricks that make it difficult to understand the code. These tricks are not easily understood, which makes application maintenance more difficult. This might not always be the case, but it is true in most cases. Therefore, with regard to business applications, it is better to have good maintainability than to have the greatest graphic performance, so we must concentrate on optimizing only the parts which are critical to the system to get the enough responsiveness to avoid a system unusable, because, in any case, responsiveness must be always good, at least.

3.3.- Unit testing

Another part of the problem is testing the code. When an application is coupled, the functional part is the part that can be tested, which is the user interface. Again, this is not a problem with a small project, but as a project grows in size and complexity, being able to check the application layers separately is crucial. Imagine that something changes in the data layer, how do you confirm it has affected the presentation layer? You would have to perform functional tests on the entire application. The performance of functional tests is very expensive because they are not usually automated, and they require people to test all the functionalities one by one. It is true that there are tools that are suitable for recording the users' movements and automating those tests, but labor and cost incurred by this well suggests that we should avoid them.

If we are also capable of providing the product with a set of automated tests, we will improve quality of the after sale support, since we will be able to reduce time in locating and solving the failures. This will increase our product quality, customer satisfaction and we will reinforce his loyalty.

For these reasons, the presentation layer should be architected to decouple what the user sees (views) with the interaction logic and communication with lower layers to promote testability.

4.- ARCHITECTURE PATTERNS IN THE PRESENTATION LAYER

There are some well-known architecture patterns that can be used for the design of the presentation layer architecture, such as MVC, MVP, MVVM and others. Right now, these patterns are used to separate the concepts between the user interface and the presentation logic but, at the beginning, were created to separate the presentation layer from the business layer, since presentation technologies at that time did not separate themselves from the business logic.

4.1.- MVC pattern (Model-View-Controller)

The MVC pattern first appeared in the eighties, with its first implementation in the popular SmallTalk programming language. This pattern has been one of the most important and influential in the history of programming and is a fundamental pattern even today. This pattern belongs to a set of patterns grouped in separated presentation architecture styles. The purpose when using this pattern is to separate the code responsible for representing data on screen from the code responsible for executing logic. To do so, the pattern divides the presentation layer into three types of basic objects: models, views and controllers. The use of the pattern is to describe communication flows between these three types of objects, as shown in the diagram below:

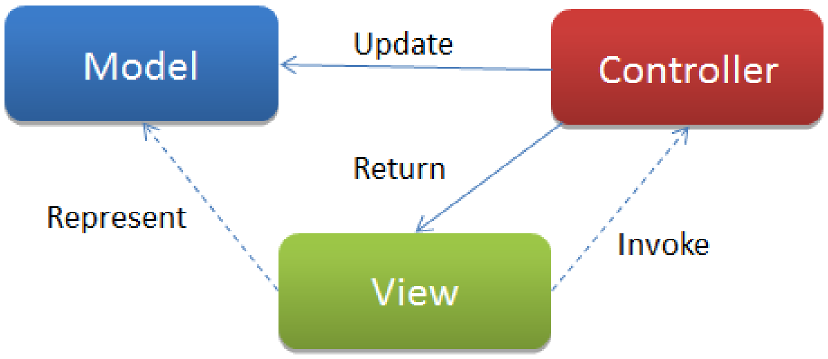


Figure 2.- MVC Architecture - Presentation Layer

In a classic scenario, this view represents what the user sees about the data, and invokes actions of a controller in response to the user's actions. The controller monitors actions of the user, interacts with the model and returns a specific view as an answer.

In most MVC implementations, the three components can directly relate from one to another, but in some implementations, the controller is responsible for determining which view to show. This is the Front Controller pattern which is is an evolution/variant of the MVC pattern (**http://msdn.microsoft.com/en-us/library/ms978723.aspx**).

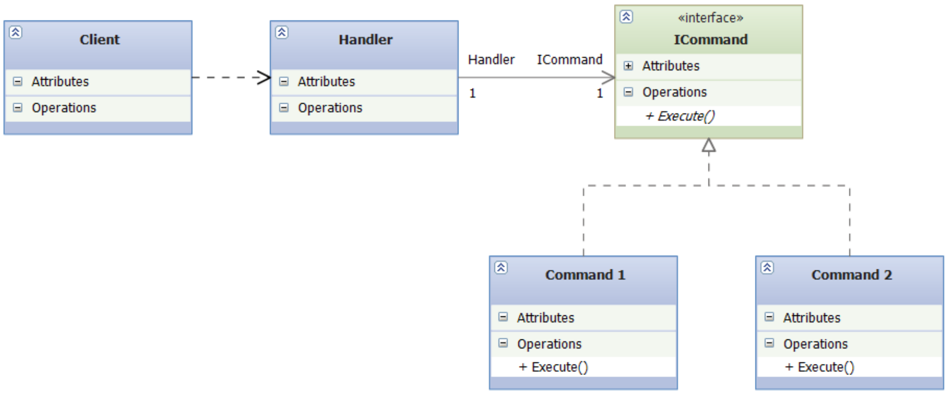


Figure 3.- 'Front Controller' Pattern Evolution

In Front-Controller, the controller is divided into two parts, one is the handler that collects relevant information from the requests that come from the presentation and directs them to a particular command, which executes the action performed in the presentation.

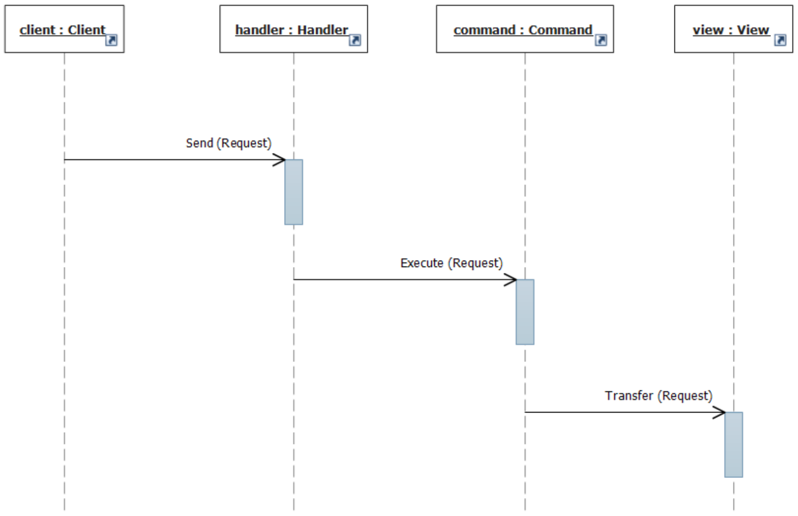


Figure 4.- Front-Controller Sequence Diagram

We will describe the responsibilities of each component in more detail below.

The Model

The model is the set of classes responsible for representing the information the user works with. It is important to understand that these classes can be domain classes, DTOs or Specific Models for the View. The option we choose will depend on the situation:

- **Domain classes**: the model corresponds to a domain class and we are not using DTOs in the application.

- **DTOs:** the model corresponds to the data of the DTO we are using in our application.

- **Specific Models for the View**: the data needed by the view does not correspond directly to the DTOs or to the domain classes of our application. These Specific Models store these pieces of additional information and probably includes small query methods that simplify the description of the view. For example, a paginated list of a set of data, where the specific model is composed of the page's data and properties such as page number, size, the total number of pages, etc.

The views

The views are responsible for graphically representing the model and offering controller actions for the user to be able to interact with the model. Views should not invoke any method that causes a change of state in the model or call upon any method requiring parameters. In other words, it should only access simple properties and object query methods that do not have any parameters.

The controller

The controller implements interactions between views and the model. It receives the requests from the user, interacts with the model making queries and modifications to it, decides which view to show in response and provides the model with the required data for its rendering, or delegates the answer to another action of another controller.

4.2.- MVP Pattern (Model View Presenter)

Before explaining the MVP pattern itself, let’s look at the history.

In the 90´s the Forms and Controllers model was the trend, driven by development environments and programming language such as Visual Basic or Borland Delphi. These development environments enabled the developer to define the screen layout with a graphic editor that allowed dragging and dropping controls within a form. These environments increased the number of applications because it enabled rapid application development and applications with better user interfaces to be created.

The form (design surface in development environments) has two main responsibilities:

- Screen layout: definition of positions of the screen controls, with the hierarchical structure of some controls with regard to others.

- Form logic: behavior of controls that usually respond to events launched by controls set by the screen.

The proposed architecture for Forms & Controller became the best approach in the world of presentation architectures. This model provided a very understandable design and had reasonably good separation between reusable components and the specific code of form logic. But Forms & Controller does not have something provided by MVC, which is the ‘Separated Presentation pattern’, and the facility to integrate to a omain Model, both of which are essential.

In 1996, Mike Potel published the document “MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java” where took a step towards merging these streams (MVC, and Form & Controller and Front Controller) by trying to take the best characteristics from each of them.

The MVP pattern separates the domain model, presentation and actions based on interaction with the user into three separate classes. The view delegates the responsibility for handling user events to its presenter. The Presenter is responsible for updating the model when an event is raised in the view, but it is also responsible for updating the view when the model indicates that a change has occurred. The model does not know about the existence of the Presenter. Therefore, if the model changes due to the action of any component other than the Presenter, it must trigger an event to make the Presenter aware of it. The presenter and model interact using events.

When implementing this pattern, the following components are identified:

- IView: The interface used by the Presenter to communicate with the view.

- View: The view concrete class implementing the IView interface, responsible for handling the visual aspects. It always refers to its Presenter and delegates the responsibility of handling the events to it.

- Presenter: Contains the logic to provide an answer to the events and handles the view state through a reference to the IView interface. The Presenter uses the model to know how to provide an answer to the events. The presenter is responsible for setting and managing the view state.

- Model: Composed of the objects that know and manage data within the application. For example, they can be the classes that compose the business model (business entities).

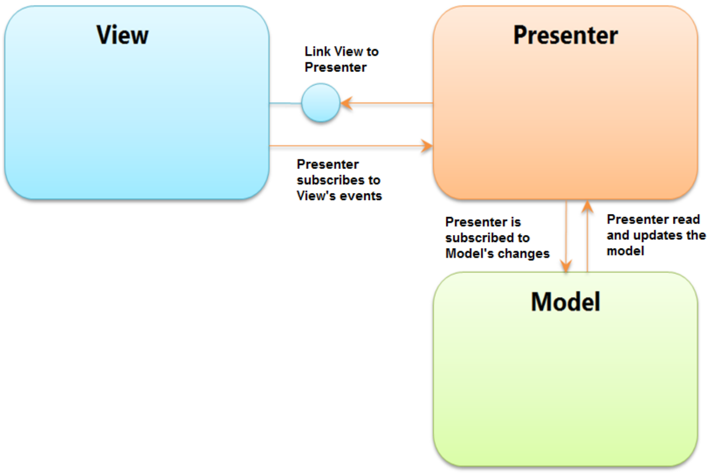

View
Link View to Presenter
Presenter
Presenter subscribes to View's events
Presenter is subscribed to Model's changes
Presenter read and updates the model
Model

Figure 5.- MVP Pattern (Model View Presenter)

In July 2006, the renowned author Martin Fowler published a note suggesting the removal of the Presentation Model pattern (as Fowler called the MVP) from the patterns catalogue. He said that after a long study and analysis he decided to divide the MVP pattern into two patterns, depending on the responsibility of the view level:

- **Supervising Controller:** The view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention. The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required. Examples of complex UI logic might include changing the color of a control or dynamically hiding/showing controls.

- **Passive View:** the presenter updates the view to reflect changes in the model. The interaction with the model is handled exclusively by the presenter; the view is not aware of changes in the model..

The decision to use Passive View or Supervising Controller primarily depends on how testable you want your application to be. If testability is a primary concern in your application, Passive View might be more suitable because you can test all the UI logic by testing the presenter. On the other hand, if you prefer code simplicity over full testability, Supervising Controller might be a better option because, for simple UI changes, you do not have to include code in the presenter that updates the view. When choosing between Passive View and Supervising Controller

4.3.- MVVM pattern (Model-View-ViewModel)

The Model View ViewModel (MVVM) pattern is also derived from MVP and this, in turn, from MVC. It emerged as a specific implementation of these patterns when using certain very powerful capabilities of newly available user interface technologies. Specifically, this is due to certain Windows Presentation Foundation (WPF) and Silverlight capabilities.

This pattern was adopted and used by the Microsoft Expression product team since the first version of Expression Blend was developed in WPF. Actually, without the specific aspects brought by WPF and Silverlight, the MVVM (Model-View-ViewModel) pattern is very similar to the Passive-View pattern defined by Martin Fowler, but because of the capabilities of current user interface technologies, we can see it as a generic pattern of presentation layer architecture.

The fundamental concept of the MVVM is to separate the Model and the View by introducing an abstract layer between them, a “ViewModel”. Views maintain a reference to a ViewModel and ViewModels expose commands and entities that are “observable” or bindable to those where the View can be bind. The user’s interactions with the View will trigger commands against the ViewModel and in the same way, updates in the ViewModel will be propagated to the View automatically through a data bind. To maximize reusabilty, the ViewModel should be unit testable and be able to function without any UI at all and should nor reference any UI elements.

.

The following diagram shows this at a high level, without specifying details of technological implementation:

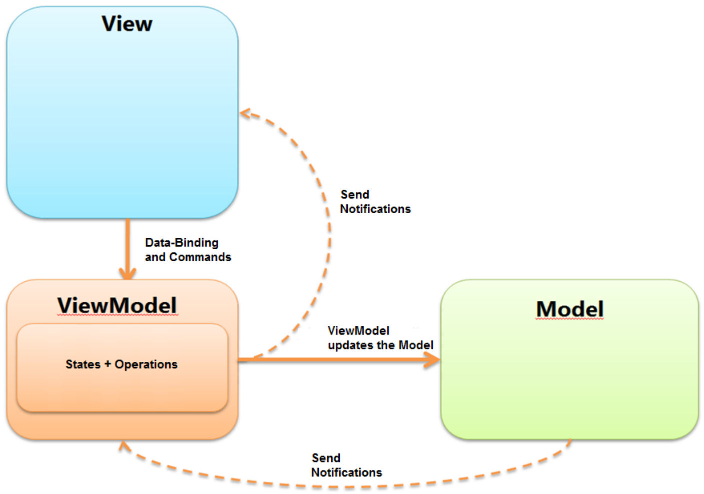


Figure 6.- MVVM pattern (Model-View-ViewModel)

The objective of the MVVM is to present and manage data transferred to the view in the easiest possible way. ViewModels expose the data from the model so, in this sense, ViewModel are models and controllers rather than views. However, ViewModel also manages the viewing logic of the view so, from this perspective, the ViewModel manages the data which is presented by the View and the interaction between the View and Model. It also bridges any gaps in data representation if the data in the Model is in a format which is not easily presentable i.e it transform any data to make it easily presentable.

4.4.- Global vision of MVVM in domain-oriented architecture

Within the domain oriented N-layered architecture, the MVVM presentation layer sub-architecture is not positioned in a straight line as you might think. The diagram below shows a view on how the MVVM layers can communicate with the lower layers.

Please note how the Model may or may obtain its data (DTOs or Entities) from the rest of the architecture. The Model defines the entities or DTOs to be used in the presentation layer when this model is a specific models for the view not the same as the entity model. If we have entities that are the same in the domain layer and in the presentation layer (domain classes or DTOs), we will not have to repeat our work. This diagram shows that the ViewModel does not have to communicate uniquely and exclusively with the Model, but it can call directly to the rest of the domain oriented architecture and it can even return domain entities instead of the Model. . In complex scenarios, ViewModel and Model can use an Adapter Layer to decouple the communication semantics to not dealing with WCF proxies directly.

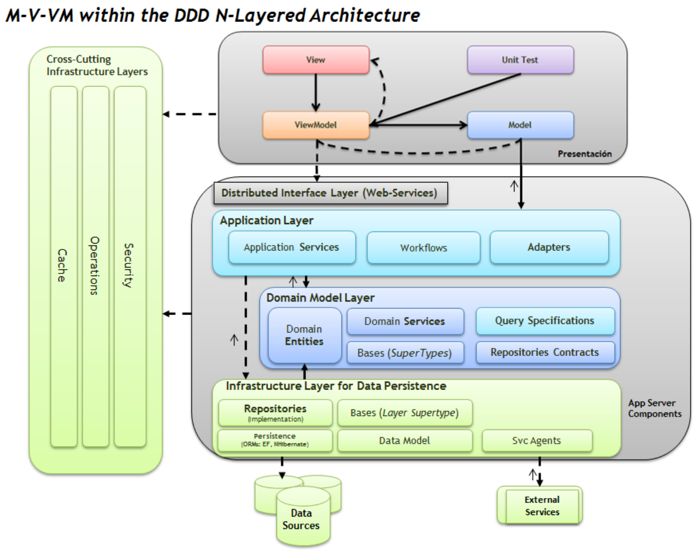


Figure 7.- Global vision of MVVM within the DDD architecture

5.- IMPLEMENTING THE PRESENTATION LAYER

The objective of this chapter is to show the different options we have to implement the Presentation Layer, depending on the nature of each application and the chosen design patterns.

In this Architecture diagram we highlight the location of the Presentation Layer:

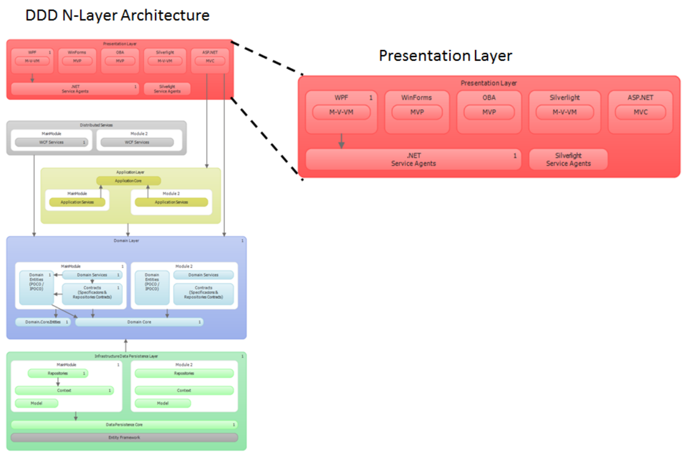


Figure 11.- Presentation Layer Location in Layer Diagram using VS.2010

As can be seen in detail in the Presentation Layer, current applications have various technologies and user interfaces that enable the use of different functionalities in different ways.

5.1.- Associated Archetypes, UX technologies and Design Patterns

Currently, there are a series of archetype applications distinguished by the nature of their visual technology, their User Experience (UX) and the technology with which they are implemented. These Archetypes are also explained one by one in this Architecture guide . However, after analyzing the detailed study of the Presentation Layer patterns and technologies, we will only review some of them.

Depending on each archetype, there are one or more technologies to implement the applications. The design and implementation of one or another architecture pattern of the Presentation Layer is recommended depending on each technology. These possibilities are exposed in the matrix shown below:

Table 1.- Presentation Layer Architecture Archetypes and Patterns

Archetype	Technology	Architecture Pattern – Presentation layer
Rich applications (Desktop applications / Windows)	• WPF/Silverlight	→ MVVM
	• WinForms	→ MVP
Web applications (HTML dynamic applications)	• ASP.NET MVC	→ MVC
	• ASP.NET Forms	→ MVP
RIA applications	• Silverlight	→ MVVM
	• HTML 5 and JavaScript	→ MVVM
Mobile applications	• Silverlight Mobile	→ MVVM
OBA Applications (Office)	• .NET VSTO	→ MVP

In this edition of this architecture guide, we will specifically emphasize the archetypes and patterns related to Silverlight and ASP.NET MVC.

5.2.- Implementing MVVM Pattern with Silverlight 4

The MVVM pattern is a specialization of the Presentation Model pattern that leverages some of the specific capabilities of Silverlight, such as data binding, commands, and behaviors. MVVM is similar to many other patterns that separate the responsibility for the appearance and layout of the UI from the responsibility for the presentation logic.

The View in MVVM is responsible for defining the structure, layout, and appearance of what the user sees on the screen. Ideally, the view is defined purely with XAML, with no code-behind other than the standard call to the InitializeComponent method in the constructor, although this is not always possible.

The Model in MVVM is an implementation of the application's domain model that can include a data model along with business and validation logic.

The ViewModel in MVVM acts as an intermediary between the View and the Model, and is responsible for handling the view logic. The ViewModel provides data in a form that the View can easily use. The ViewModel retrieves data from the Model and then makes that data available to the View, and may reformat the data in some way that makes it simpler for the View to handle. The ViewModel also provides implementations of commands that a user of the application initiates in the View. For example, when a user clicks a button in the UI, that action can trigger a command in the ViewModel. The ViewModel may also be responsible for defining logical state changes that affect some aspect of the display in the View, such as an indication that some operation is pending.

The following diagram represents this pattern:

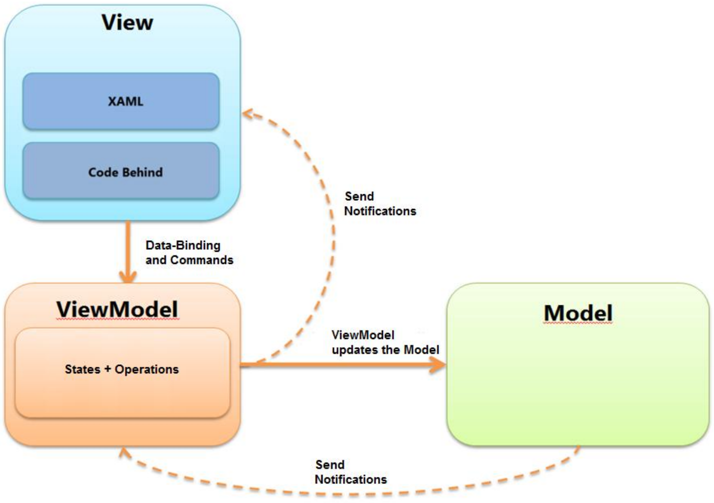


Figure 13.- MVVM using WPF

In some scenarios, the ViewModel may call a web service directly instead of using a Model class that itself makes a call to the web service. For example, if a web service retrieves a collection of Person objects that you can deserialize and use directly in the ViewModel, there's no need to define another Person class in the model.

In addition to understanding the responsibilities of the three components, it's also important to understand how the components interact with each other. At the highest level, the view "knows about" the view model, and the view model "knows about" the model, but the model is unaware of the view model, and the view model is unaware of the view.

MVVM relies on the data binding capabilities in Silverlight to manage the link between the View and ViewModel. The following are important features of an MVVM application that uses these data binding capabilities:

- The View can display richly formatted data to the user by binding to properties of the ViewModel instance. If the View subscribes to events in the ViewModel, the ViewModel can also notify the View of any changes to its property values using these events.
- The View can initiate operations by using commands to invoke methods in the ViewModel.
- If the View subscribes to events defined in the ViewModel, the ViewModel can notify the View of any validation errors using these events.

Typically, the ViewModel interacts with the Model by invoking methods in the model classes. The Model may also need to be able to report errors back to the ViewModel by using a standard set of events that the ViewModel subscribes to (remember that the Model is unaware of the ViewModel). In some scenarios, the Model may need to be able to report changes in the underlying data back to the ViewModel; again, the Model should do this using events.

5.3.- Justification of MVVM

Application Design without UI Patterns

Let think about a typical UI, for example, Microsoft Office. We can change the text format from the menu bar or from the secondary menu associated with the right button of the mouse.

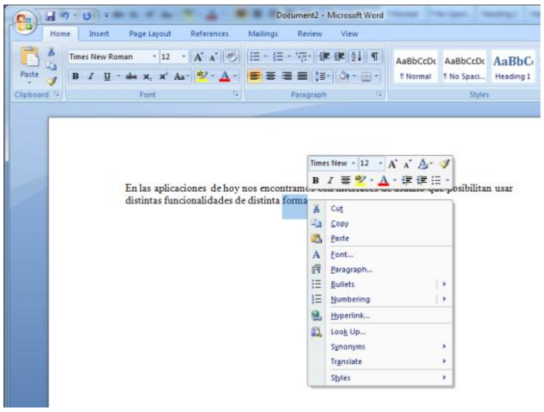


Figure 12.- Microsoft Office Word GUI as a Presentation Layer

If we simplify the problem and we want to implement an interface with repeated functionalities from multiple sites, we see that in XAML we can define an event handler such as the following:

```
    <Button Name="UnoBtn" Click="OneBtnClick" Width="100"
            Height="25">Button</Button>
```

And also...

```
    <Label KeyDown="LabelKeyDown">One<Label>
```

We can then implement this in the code behind:

```
       private void OneBtnClick (object sender, RoutedEventArgs e)
       {
            Functionality();
       }
       private void LabelKeyDown (object sender, KeyEventArgs e)
       {
            Functionality();
       }
```

We should note one detail. On one hand, we are defining the event handler from the XAML itself, and we are mixing two roles: developer and designer. The designer that generates XAML visually with Blend does not have to know anything about the handler. For example, he knows that the button he is designing is for saving and will be associated to the action “Save.”

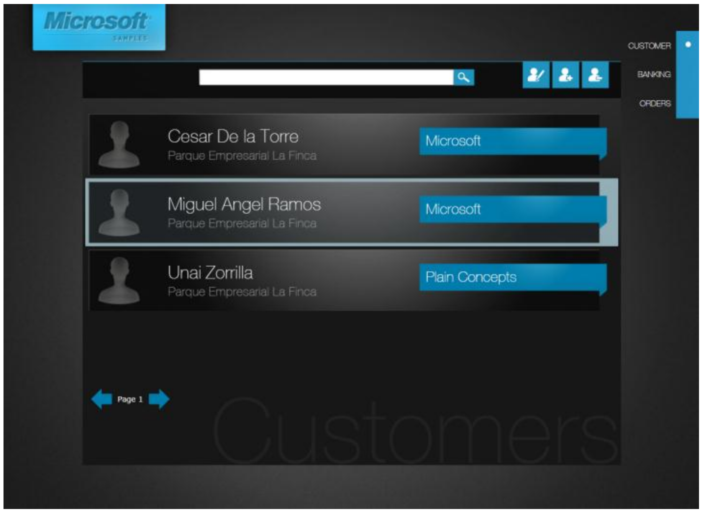
On the other hand, we are creating a strong dependency on XAML with the functionality. That’s means, we are coupling the XAML (the view) with the code behind. If we want to test to see if something is being saved, we will have to test the code behind. If we were to reuse this functionality in other XAMLs, we would have to bring it to a different class. Then again, we still require the programmer to see the XAML and link the event handlers to each XAML. In this case, we do not have separation of concerns.

In the example above, since the action is performed by two different XAML controls, we must use two event handlers if we are unlucky and these handlers have different delegations. If we put in 10 repeated functionalities from 3 different components, we put in 30 event handlers. It seems unmanageable, doesn’t it? Therefore, we recommend using architecture that helps solve these problems as MVVM.

Simplistic design of Application

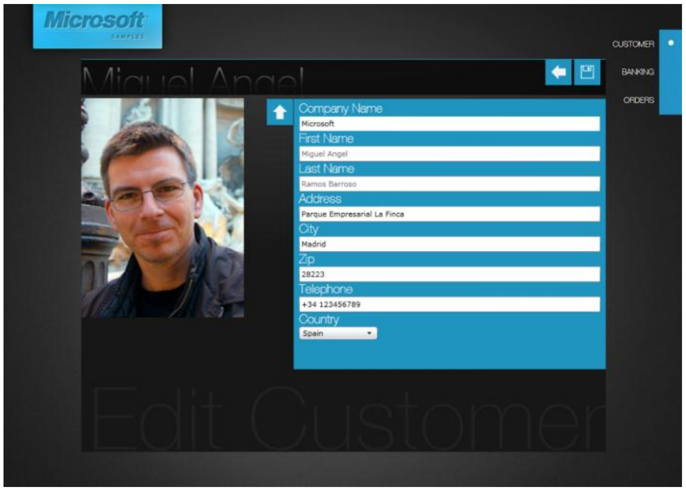
To prove the benefits of using an architecture that separates the presentation from the model, we will start with a simple scenario of an application where we use the Separated Presentation pattern.

Let's assume a scenario where we have two Views. One View with a list where we can see all the customers of the application and another view showing the details of simply one customer. We would have a User Interface similar to the following:



Cesar De la Torre	Parque Empresarial La Finca	Microsoft
Miguel Angel Ramos	Parque Empresarial La Finca	Microsoft
Unai Zorrilla	Parque Empresarial La Finca	Plain Concepts

Figure 14.- Customer list view



Company Name	Microsoft
First Name	Miguel Angel
Last Name	Ramos Barroso
Address	Parque Empresarial La Finca
City	Madrid
Zip	28223
Telephone	+34 123456789
Country	Spain

Figure 15.- Customer detail view

To distinguish between Model and View for this particular scenario, the Model will be the Customer's data and the Views will be the two Views shown.

Following our simple model, the design would look like this:

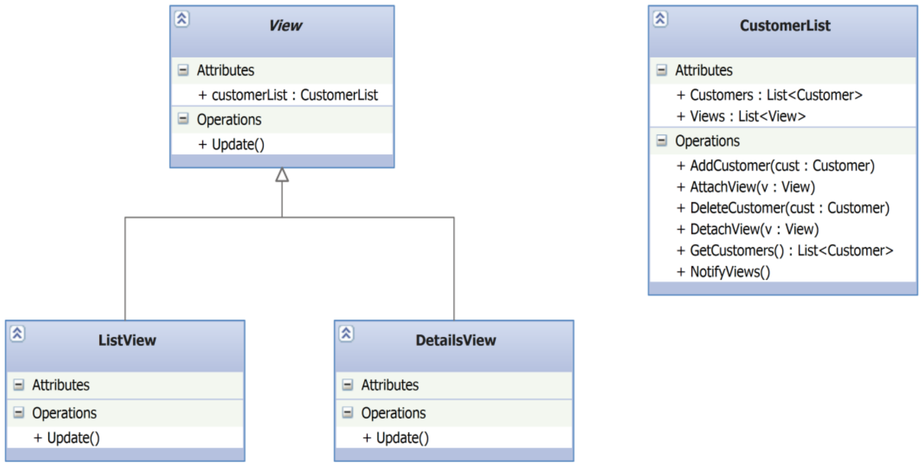


Figure 16.- Simplistic Design – The views are directly associated with the Model

As can be seen, `CustomerList` defines a View list whith can be added or removed with `AttachView()` and `DetachView()`. When a Customer changes, `CustomerList` will notify all the Views by calling the `Update()` method and all the views would be updated by calling the `GetCustomers()` method of the Model. Once instantiated, the ListView and DetailsView will have a reference to the CustomerList, defined as a field member whith is actually defined in the abstract base class 'View'. As you may have noticed, the 'observer' pattern has been applied.

Note:
Note that the two Views in this case are **strongly coupled** with the CustomerList class. If we add more client business logic, it will only increase the coupling level.

Simplistic design of Application using Silverligh

Before continuing the analysis of the logical design, we will turn it into a Silverlight/Wpf-friendly design. Keep in mind that Silverlight/WPF provides two very powerful aspects that we can use directly:

- **DataBinding**: The capacity to bind GUI elements to properties.

- **Commands**: Provide the capability to notify the underlying data, that there were changes made in the GUI.

According to these capabilities, the design would look like this:

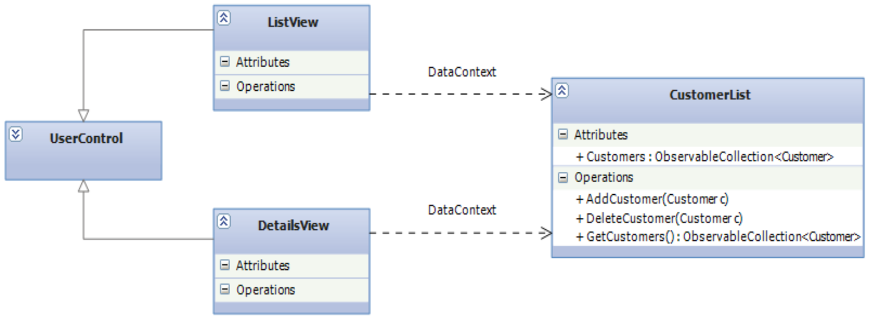

ListView
Attributes
Operations
DataContext
CustomerList
Attributes
+ Customers : ObservableCollection<Customer>
Operations
+ AddCustomer(Customer c)
+ DeleteCustomer(Customer c)
+ GetCustomers() : ObservableCollection<Customer>
UserControl
DataContext
DetailsView
Attributes
Operations

Figure 17.- Friendly and simple design of the Silverlight application

Views derive from the `UserControl` whith is a base class of the Silverlight/WPF and `CustomerList` does not have to maintain a Views list any longer, since it does not have to know anything about the views. Instead, ListView and DetailsViews aim at `CustomerList` as their DataContext and they use DataBinding in order to bind to the Customer list.

Also note that we have replaced `List<Customer>` by `ObservableCollection<Customer>`. The advantage of using ObservableCollection is to facilitate binding with changes to underlying collection to be reflected automatically on UI and reverse (depending on binding mode used).

Problems of the Silverlight Simplistic design

The problem is that things get complicated when we introduce the following functionality:

1.- Propagation of the selected element, so that we update the `DetailsView` as long as the selection of an element in `ListView` changes, and vice versa.

2.- Enabling or disabling parts of the User Interface of `DetailsView` or `ListView` based on any rule (e.g., highlighting an entry with an incorret ZIP code).

Point “1” can be implemented by adding a `CurrentEntry` property directly in `CustomerList`. However, this solution has some problems if we have more than one instance of UI connected to the same `CustomerList`.

Point “2” can be implemented in `ListView` and `DetailsView`, but if we want to change the rule, then we will need to change both views. Changes will start to affect multiple sites.

In short, it seems convenient to gradually add a third sub-layer in our application. We need a sub-layer between the Views and the CustomerList model that saves the states shared by the views. In fact, we need a ViewModel concept.

5.4.- Design with the Model View ViewModel (MVVM) pattern

A ViewModel provides an abstraction that acts as a meta-view (a view's model), saving states and policies shared by one or a group of Views.

When introducing the ViewModel concept, our design will be able look like this:

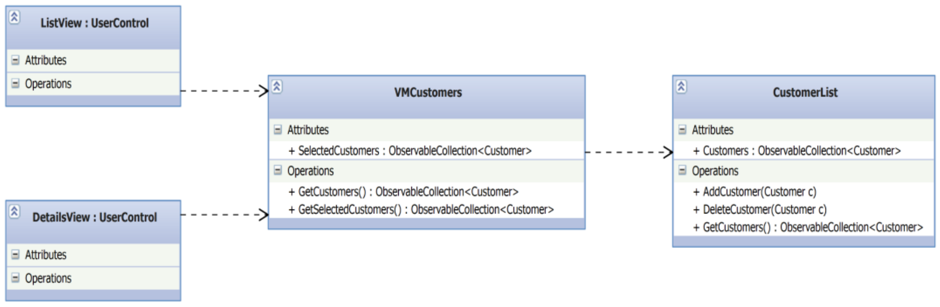

ListView : UserControl
Attributes
Operations
DetailsView : UserControl
Attributes
Operations
VMCustomers
Attributes
+ SelectedCustomers : ObservableCollection<Customer>
Operations
+ GetCustomers() : ObservableCollection<Customer>
+ GetSelectedCustomers() : ObservableCollection<Customer>
CustomerList
Attributes
+ Customers : ObservableCollection<Customer>
Operations
+ AddCustomer(Customer c)
+ DeleteCustomer(Customer c)
+ GetCustomers() : ObservableCollection<Customer>

Figure 18.- Application design using MVVM

In this design, the Views know the ViewModel and are binded to its data, so they can reflect any change that occurs. The ViewModel does not have any reference to the Views, only a reference to the Model, in this case `CustomerList`.

For Views, the ViewModel acts as a Facade Design Pattern (which provides a simple interface and controls access to a series of complicated interfaces and or sub system) of the Model but it also acts as a way to share states among views (`SelectedCustomers` in the example). In addition, the ViewModel usually exposes user actions which results in View events.

5.4.1.- Data Binding

Data binding plays a very important role in the MVVM pattern. WPF and Silverlight both provide powerful data binding capabilities. Your ViewModel and your Model classes should be designed to support data binding so that they can take advantage of these capabilities.

This means that they must implement the correct interfaces. If it defines properties that can be data bound, it should implement the **INotifyPropertyChanged** interface. If the view model represents a collection, it should implement the INotifyCollectionChanged interface or derive from the ObservableCollection<T> class that provides an implementation of this interface. Both of these interfaces define an event that is raised whenever the underlying data is changed. Any data bound controls will be automatically updated when these events are raised.

Silverlight and WPF data binding supports multiple data binding modes. The most important are:

- **One-way data binding**: UI controls can be bound to a view model so that they reflect the value of the underlying data when the display is rendered.

- **Two-way data binding:** Automatically update the underlying data when the user modifies it in the UI.

In many cases, a ViewModel will define properties that return objects. WPF and Silverlight data binding supports binding to nested properties via the Path property.

```
    <DataGrid ItemsSource="{Binding Path=LineItems}" />
```

5.4.2.- INotifyPropertyChanged in MVVM applications

In Silverlight/WPF there is an interface called INotifyPropertyChanged, which can be implemented to notify the user interface that an object’s property has been modified and, therefore, the interface must update its data. This entire subscription mechanism is done by the Silverlight DataBind bound automatically. As explained above, ObservableCollection is a class which Silverlight/WPF provides and has the ability to keep the View and ViewModel data in sync built in i.e it supports automatic change notification during data binding. For other classes like for example a Customer class which has indiviual properties which are not collections for keeping the View and ViewModel in sync the ViewModel should implement INotifyPropertyChanged. When we want to return a set of objects we use the observable collection (ObservableCollection). However, when we have to go from the Model to the View, passing only one object through the ViewModel, we will have to use this Interface.

This interface defines a single event, called PropertyChanged, that must be launched to inform us of the property change. Each model’s class is responsible for raising the event whenever applicable:

```
        public class SimpleClass : INotifyPropertyChanged
        {
            private string _name;

            // Event defined by the interface
            public event PropertyChangedEventHandler PropertyChanged;

            // Launching the "PropertyChanged" event
            private void NotifyPropertyChanged(string info)
            {
                var handler = this.PropertyChanged;
                if (handler != null)
                {
                    handler(this, new PropertyChangedEventArgs(info));
                }
            }

            // Property that informs changes
            public string Name
            {
                get { return _name; }
                set
                {
                    if (_name != value)
                    {
                        _name = value;
                        NotifyPropertyChanged("Name");
                    }
                }
            }
        }
```

This code is too long to be performed in classes with many properties, and it is easy to make mistakes. Thus, it is important to create a small class that prevents us from having to repeat the same code over and over again. Therefore, we recommend doing something like this:

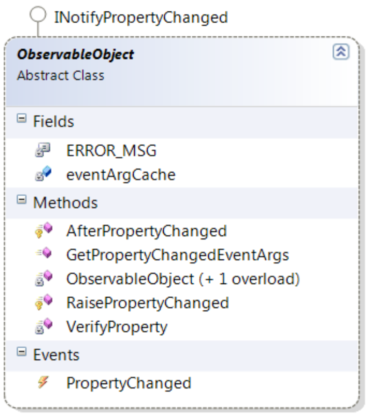


Figure 23.- Reusable implementation of INotifyPropertyChanged interface

5.4.3.- Implementing ICollectionView

The preceding code example shows how to implement a simple ViewModel property that returns a collection of items that can be displayed via data bound controls in the View. Because the ObservableCollection<T> class implements the INotifyCollectionChanged interface, the controls in the View will be automatically updated to reflect the current list of items in the collection as items are added or removed.

However, you will often need to more finely control how the collection of items is displayed in the View, or track the user's interaction with the displayed collection of items, from within the ViewModel itself. For example, you may need to allow the collection of items to be filtered or sorted according to presentation logic implemented in the ViewModel, or you may need to keep track of the currently selected item in the View so that commands implemented in the ViewModel can act on the currently selected item.

WPF and Silverlight support these scenarios by providing various classes that implement the **ICollectionView** interface. This interface provides properties and methods to allow a collection to be filtered, sorted, or grouped, and allow the currently selected item to be tracked or changed. Silverlight and provide implementations of this interface called PagedCollectionView class.

The following code example shows the use of the **PagedCollectionView** in Silverlight to keep track of the currently selected customer.

```
public class MyViewModel : INotifyPropertyChanged
{
    public ICollectionView Customers { get; private set; }

    public MyViewModel( ObservableCollection<Customer> customers )
    {
        // Initialize the CollectionView for the underlying model
        // and track the current selection.
        Customers = new PagedCollectionView( customers );
        Customers.CurrentChanged +=
                       new EventHandler( SelectedItemChanged );
    }

    private void SelectedItemChanged( object sender, EventArgs e )
    {
        Customer current = Customers.CurrentItem as Customer;
        ...
    }
}
```

In the View, you can then bind an ItemsControl, such as a ListBox, to the Customers property on the ViewModel via its ItemsSource property, as shown here.

```
    <ListBox ItemsSource="{Binding Path=Customers}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding Path=Name}"/>
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
```

When the user selects a customer in the UI, the ViewModel will be informed so that it can apply the commands that relate to the currently selected customer. The ViewModel can also programmatically change the current selection in the UI by calling methods on the collection View object, as shown in the following code example.

```
    Customers.MoveCurrentToNext();
```

When the selection changes in the collection View, the UI automatically updates to visually represent the selected state of the item. The implementation is similar for WPF, though the **PagedCollectionView** in the preceding example will typically be replaced with a **ListCollectionView** or **BindingListCollectionView** class, as shown here.

```
    Customers = new ListCollectionView( _model );
    Customers.CurrentChanged+=new EventHandler( SelectedItemChanged );
```

5.4.4.- Using Commands in MVVM Silverlight applications

Silverlight/WPF implements the Commands design pattern as an event driven programming mechanism. These commands allow decoupling between the origin and the action managed with the window, and has the advantage that multiple sources (view controls) can invoke the same command. In addition, the same command can be managed in different ways, depending on the target.

From the user's point of view, a command is simply a user interface element property launched by the logical command after invoking an event handler directly (remember the problem we addressed at the beginning of this section). We can have multiple components of that user interface bind to the same command. In addition, we can hook the command up in the XAML, so a designer only sees an action and a programmer will have the command's functionality in some part of its code (it is not recommended to be in the codebehind).

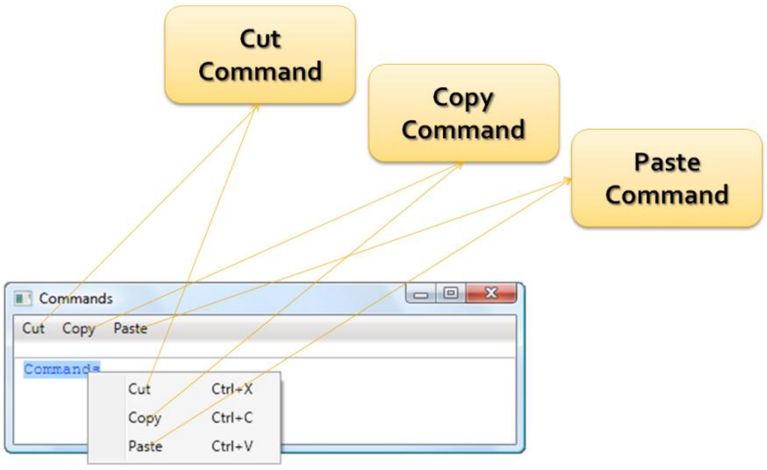


Figure 19.- Using a Command

There are also more functionalities that could be requested to the Commands such as requesting that if an action is not available, it cannot be launched. For example, if we have not modified anything in a document, why would we activate the Undo action? The XAML components of our application that launch the Undo action should be disabled.

All the Commands are implemented in the **ICommand** interface.

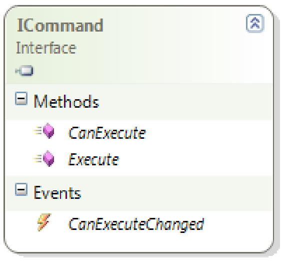


Figure 20.- ICommand Interface

This interface consists of two methods and one event.

- **Execute**: Contains the logic of the action that must be implemented by the command in the application.
- **CanExecute**: Returns to the command state and tells us whether it is enabled or disabled.

- **CanExecuteChanged**: Each time the CanExecute value changes, an event will be launched informing us of this.

This would be an example of a command implementation:

```
public class SaveCommand : ICommand
{
   private CustomerViewModel _view;

   public SaveCommand(CustomerViewModel view)
   {
     _view = view;
   }

   public bool CanExecute(object parameter)
   {
     return true;
   }

   public event EventHandler CanExecuteChanged;

   public void Execute(object parameter)
   {
     //to do something, like saving a customer
   }
}
```

Implementing the ICommand interface is simple. However, there are a number of implementations of this interface that you can use in your application. For example, you can use the ActionCommand class from the Expression Blend SDK or the **DelegateCommand** class provided by Prism.

The DelegateCommand class encapsulates two delegates that each reference a method implemented within your ViewModel class. It inherits from the DelegateCommandBase class, which implements the ICommand interface's Execute and CanExecute methods by invoking these delegates. You specify the delegates to your ViewModel methods in the DelegateCommand class constructor.

Considering linkable commands, the design applying MVVM would look like this:

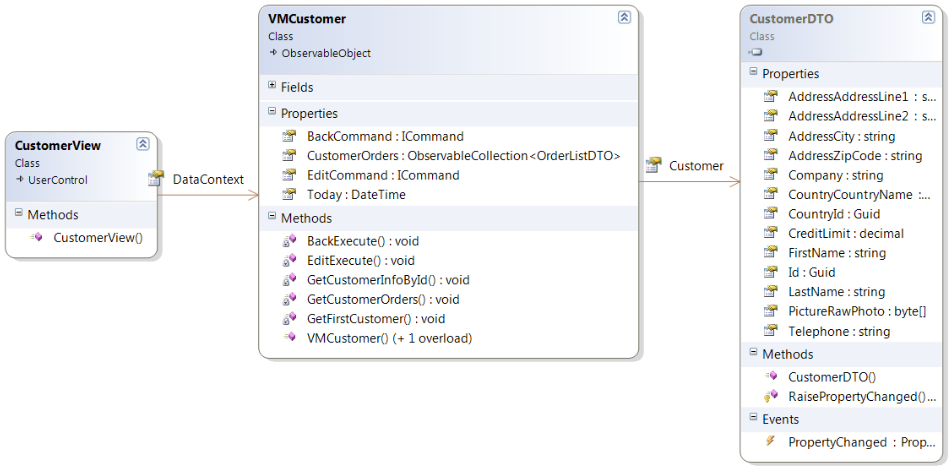


Figure 22.- Application design using ViewModels and exposing linkable Commands-

A DelegateCommand allows views to be linked to the ViewModel. Although the diagram above does not show this explicitly, all the ICommand exposed by the ViewModel are DelegateCommands. For example, the code for EditCommand in the ViewModel would be something like this:

```
public class VMCustomer : ObservableObject
{


   private ICommand _editCommand;

   public ICommand EditCommand
   {
      if (_editCommand == null)
      {
         _editCommand = new DelegateCommand<Customer>(EditExecute,
                                                    CanEditExecute);
      }
        return _editCommand;
   }

   private void EditExecute(...) { ... }
   private bool CanEditExecute() { ... }
}
```

And the XAML code that uses this Command, to make it simple, would be something like this:

```
<UserControl>
...
<StackPanel>
...
<Button  Content="Button"  Command="{Binding  EditCommand,  Mode=OneWay}"
CommandParameter="{Binding SelectedItem, ElementName=listBox}"/>
...
</StackPanel>
...
</UserControl>
```

5.4.5.- Behaviors in not Command-enabled controls

Silverlight/WPF controls that support commands allow you to declaratively hook up a control to a Command. These controls will invoke the specified Command when the user interacts with the control in a specific way. For example, for a Button control, the Command will be invoked when the user clicks the button. This event associated with the Command is fixed and cannot be changed.

Behaviors also allow you to hook up a control to a Command in a declarative fashion. However, behaviors can be associated with a range of events raised by the control, and they can be used to conditionally invoke an associated Command object or a Command method on the ViewModel. In other words, behaviors can address many of the same scenarios as Command-enabled controls, and they may provide a greater degree of flexibility and control.

You will need to choose when to use Command-enabled controls and when to use behaviors, as well as which kind of behavior to use. If you prefer to use a single mechanism to associate controls in the view with functionality in the ViewModel or for consistency, you should consider using behaviors, even for controls that inherently support commands.

If you only need to use Command-enabled controls to invoke commands on the view model, and if you are happy with the default events to invoke the command, behaviors may not be required. Similarly, if your developers or UI designers will not be using Expression Blend, you may favor command-enabled controls (or custom attached behaviors) because of the additional syntax required for Expression Blend behaviors.

An example of Expression Blend interaction triggers and InvokeCommandAction behavior is:

```
<Button Content="Submit" IsEnabled="{Binding CanSubmit}">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Click">
            <i:InvokeCommandAction Command="{Binding SubmitCommand}"/>
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

Remenber, this approach can be used for any control to which you can attach an interaction trigger. It is especially useful if you want to attach a command to a control that does not derive from ButtonBase, or when you want to invoke the command on an event other than the click event. If you need to supply parameters for your command, you can use the CommandParameter property.

5.4.6.-Asynchronous Programming Model

One of the main advantages of using the user interface architecture created from the Model-View-ViewModel Pattern is that we can leverage its asynchronous model to create interfaces that respond even to heavy tasks. Avoiding delay times between requests and response due to the network overload or tasks requiring very complicated calculations is one of the great benefits of using a Async Programming support.

- The ViewModel could execute methods in the model asynchronously executed in a different thread using **DispatcherThread**, so no one action started through a ViewModel object will affect the screen drawing process or the visual transitions carried out by that screen.

- Silverlight, in turn, forces us to use an asynchronous proxy model to use services referred to by our application.

These two characteristics provide an application model that leads us, unequivocally, to obtaining an application with a high degree of response to end user interactions.

Let's look at an example of how our Silverlight application's client proxy object utilization methods are created (automatic generation):

```
public void GetFirstCustomer()
{
    try
    {
        MainModuleServiceClient client = new MainModuleServiceClient();

        client.GetPagedCustomerAsync(
            new PagedCriteria(){ PageIndex = 0, PageCount = 10 });

        client.GetPagedCustomerCompleted +=
            delegate(object sender,
                     GetPagedCustomerCompletedEventArgs e)
        {
            Customer[] listCustomers = e.Result;
            if (listCustomers != null && listCustomers.Length > 0)
            {
                Customer = listCustomers[0];
```

```
                 GetCustomerOrders();
             }
         };
     }
     catch (Exception excep)
     {
         Debug.WriteLine("GetFirstCustomer: Error at Service:"
                              + excep.ToString());
     }
 }
```

5.4.7.-Validation Model

Your ViewModel or Model will often be required to perform data validation and to signal any data validation errors to the View so that the user can act to correct them.

Silverlight and WPF provide support for managing data validation errors that occur when changing individual properties that are bound to controls in the View. For single properties that are data-bound to a control, the ViewModel or Model can signal a data validation error within the property setter by rejecting an incoming bad value and throwing an exception. If the **ValidatesOnExceptions** property on the data binding is true, the data binding engine in WPF and Silverlight will handle the exception and display a visual cue to the user that there is a data validation error.

However, throwing exceptions with properties in this way should be avoided where possible. An alternative approach is to implement the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces on your view model or model classes. These interfaces allow your view model or model to perform data validation for one or more property values and to return an error message to the view so that the user can be notified of the error.

The **IDataErrorInfo** interface provides basic support for property data validation and error reporting. It defines two read-only properties: an indexer property, with the property name as the indexer argument, and an Error property. Both properties return a string value.

The indexer property allows the view model or model class to provide an error message specific to the named property. An empty string or null return value indicates to the view that the changed property value is valid. The **Error** property allows the view model or model class to provide an error message for the entire object. Note, however, that this property is not currently called by the Silverlight or WPF data binding engine.
The **IDataErrorInfo** indexer property is accessed when a data-bound property is first displayed, and whenever it is subsequently changed. Because the indexer property

is called for all properties that change, you should be careful to ensure that data validation is as fast and as efficient as possible.

When binding controls in the view to properties you want to validate through the IDataErrorInfo interface, set the ValidatesOnDataErrors property on the data binding to true. This will ensure that the data binding engine will request error information for the data-bound property.

```
<TextBox
Text="{Binding Path=CurrentEmployee.Name, Mode=TwoWay,
ValidatesOnDataErrors=True, NotifyOnValidationError=True }"
/>
```

The **INotifyDataErrorInfo** interface is more flexible than the IDataErrorInfo interface. It supports multiple errors for a property, asynchronous data validation, and the ability to notify the view if the error state changes for an object.

To use **INotifyDataErrorInfo,** we simply have to implement this interface. We have implemented it in the **ObservableObject** class, which is the base of all the view models. Implementation is very easy. We have created a series of help methods to add the properties with errors to a dictionary and to keep track of them.

```
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
    private readonly Dictionary<string, List<string>> _currentErrors;

    public IEnumerable GetErrors(string propertyName)
    {
       if (string.IsNullOrEmpty(propertyName))
              return (_currentErrors.Values);

       MakeOrCreatePropertyErrorList(propertyName);
              return _currentErrors[propertyName];
    }

    public bool HasErrors
    {
        get
        {
          return (_currentErrors.Where(c => c.Value.Count > 0).Count() > 0);
        }
    }

    void FireErrorsChanged(string property)
    {
        if (ErrorsChanged != null)
                ErrorsChanged(this, new
    DataErrorsChangedEventArgs(property));
       }

    public void ClearErrorFromProperty(string property)
    {
        MakeOrCreatePropertyErrorList(property);
        _currentErrors[property].Clear();
        FireErrorsChanged(property);
    }

    public void AddErrorForProperty(string property, string error)
    {
        MakeOrCreatePropertyErrorList(property);
        _currentErrors[property].Add(error);
        FireErrorsChanged(property);
    }

    void MakeOrCreatePropertyErrorList(string propertyName)
    {
        if (!_currentErrors.ContainsKey(propertyName))
                 _currentErrors[propertyName] = new List<string>();
    }
```

Validation rule handlers will be created in the next example. In this version, however, we will only have one required rule implemented in the same **ObservableObject** class.

The rule is simple, check that the property has a value and if not, add this property to the error dictionary and display the error.

```
    public void CheckRequiredValidationRule(string propertyName, string
    value)
    {
       ClearErrorFromProperty(propertyName);
       if (string.IsNullOrEmpty(value) ||
                                string.IsNullOrWhiteSpace(propertyName))
       {
          AddErrorForProperty(propertyName, "This field is required");
       }

       RaisePropertyChanged(propertyName);
    }
```

Finally, the only thing left is to display the validation rule in the property we want to validate, as follows:

```
       public string CompanyName
       {
           get { return _currentCustomer.CompanyName; }
           set
           {
               _currentCustomer.CompanyName = value;
               _currentCustomer.CustomerCode = (value.ToString().Length > 3)
                                 ? value.ToString().Substring(0, 3) : null;
               CheckRequiredValidationRule("CompanyName", value);
           }
       }
```

Then, perform the binding in the **Text** property of the text box, but without forgetting to set the **ValidatesOnNotifyDataErrors** binding property as true.

```
        {Binding CompanyName, Mode=TwoWay, ValidatesOnNotifyDataErrors=True}
```

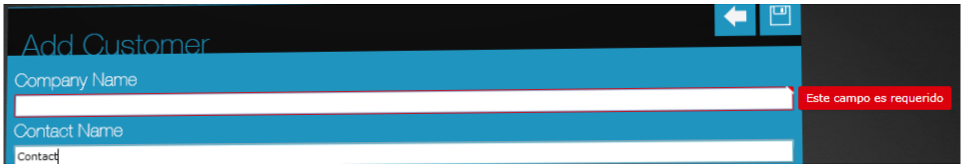


Figure 24.- Validation in Silverlight

5.4.8.- Instantiation of the View and ViewModel

The next step is to consider how the view, view model, and model classes are instantiated and associated with each other at run time. Typically, there is a one-to-one relationship between a View and its ViewModel. The View and ViewModel are loosely coupled via the View's data context property; this allows visual elements and behaviors in the View to be data bound to properties, Commands, and methods on the ViewModel. You will need to decide how to manage the instantiation of the View and ViewModel classes and their association via the DataContext property at run time.

Care must also be taken when constructing and connecting the View and ViewModel to ensure that loose coupling is maintained. As noted in the previous section, the ViewModel should ideally not depend on any specific implementation of a View. Similarly, the View should ideally not depend on any specific implementation of a ViewModel.

However, it should be noted that the View will implicitly depend on specific properties, Commands, and methods on the ViewModel because of the data bindings it defines. If the ViewModel does not implement the required property, command, or method, a run-time exception will be generated by the data binding engine, which will be displayed in the Visual Studio output window during debugging.

There are multiple ways the View and the ViewModel can be constructed and associated at run time. The most appropriate approach for your application will largely depend on whether you create the View or the ViewModel first, and whether you do this programmatically or declaratively.

Creating the View Model Using XAML

Perhaps the simplest approach is for the view to declaratively instantiate its corresponding ViewModel in XAML. When the View is constructed, the corresponding ViewModel object will also be constructed. You can also specify in XAML that the ViewModel be set as the View's data context.

```
        <UserControl.DataContext>
             <Silverlight_Client_ViewModels:VMMenu/>
        </UserControl.DataContext>
```

When the MenuControl is created, an instance of the VMMenu is automatically constructed and set as the view's data context. This approach requires your view model to have a default (parameter-less) constructor.

The declarative construction and assignment of the ViewModel by the View has the advantage that it is simple and works well in design-time tools such as Microsoft Expression Blend or Microsoft Visual Studio. The disadvantage of this approach is that the View has knowledge of the corresponding ViewModel type.

Creating the View Model Programmatically

An approach is for the View to instantiate its corresponding ViewModel instance programmatically in its constructor. It can then set it as its data context, as shown in the following code example.

```
public CustomerListView()
{
    InitializeComponent();
    this.DataContext = new CustomerListViewModel();
}
```

The programmatic construction and assignment of the ViewModel within the View's code-behind has the advantage that it is simple and works well in design-time tools like Expression Blend or Visual Studio. The disadvantage of this approach is that the View needs to have knowledge of the corresponding ViewModel type and that it requires code in the View's code-behind. Using a dependency injection container, such as Unity, can help to maintain loose coupling between the View and ViewModel.

Creating a View Defined as a Data Template

A View can be defined as a data template and associated with a ViewModel type. Data templates can be defined as resources, or they can be defined inline within the control that will display the ViewModel. The "content" of the control is the ViewModel instance, and the data template is used to visually represent it. WPF and Silverlight will automatically instantiate the data template and set its data context to the ViewModel instance at run time. This technique is an example of a situation in which the ViewModel is instantiated first, followed by the creation of the View.

Data templates are flexible and lightweight. The UI designer can use them to easily define the visual representation of a ViewModel without requiring any complex code. Data templates are restricted to Views that do not require any UI logic (code-behind). Microsoft Expression Blend can be used to visually design and edit data templates.

You can define a data template as a resource. The following example shows the data template defined a resource and applied to a content control via the StaticResource markup extension.

```
<UserControl ...>
   <UserControl.Resources>
       <DataTemplate x:Key="CustomerViewTemplate">
              <local:CustomerContactView />
        </DataTemplate>
</UserControl.Resources>
<Grid>
   <ContentControl Content="{Binding Customer}"
      ContentTemplate="{StaticResource CustomerViewTemplate}" />
</Grid>
```

Here, the data template wraps a concrete View type. This allows the View to define code-behind behavior. In this way, the data template mechanism can be used to externally provide the association between the View and the ViewModel. Although the preceding example shows the template in the UserControl resources, it would often be placed in application's resources for reuse.

Creating the View and View Model Using Unity

Choosing an appropriate strategy to manage this step is especially important if you are using a dependency injection container in your application. The Unity Application Block (Unity) provide the ability to specify dependencies between the View, ViewModel, and Model classes and to have them fulfilled by the container at run time.

You have to define the ViewModel as a dependency of the View, so that when the View is constructed (using the container) it automatically instantiates the required ViewModel. In turn, any components or services that the ViewModel depends on will also be instantiated by the container. After the ViewModel is successfully instantiated, the View then sets it as its data context.

Typically, you define an interface on the ViewModel so the ViewModel's specific concrete type can be decoupled from the View. For example, the View can define its dependency on the ViewModel via a constructor argument, as shown here.

```
public CustomerView()
{
    InitializeComponent();
}

public CustomerView(CustomerListViewModel viewModel)
: this()
{
    this.DataContext = viewModel;
}
```

The default parameter-less constructor is necessary to allow the View to work in design-time tools, such as Visual Studio and Expression Blend.

Alternatively, you can define a write-only ViewModel property on the View, as shown here. Unity will instantiate the required ViewModel and call the property setter after the View is instantiated.

```
public CustomerView()
{
    InitializeComponent();
}

[Dependency]
public CustomerListViewModel ViewModel
{
    set { this.DataContext = value; }
}
```

The ViewModel type is registered with the Unity container, as shown here.

```
IUnityContainer container;
...
container.RegisterType<CustomerListViewModel>();
```

The View can then be instantiated through the container, as shown here.

```
IUnityContainer container;
...
var view = container.Resolve<CustomerView>();
```

5.4.9.- Communicating Between Loosely Coupled Components

When building large complex applications, a common approach is to divide the functionality into discrete module assemblies. It is also desirable to minimize the use of static references between these modules. This allows the modules to be independently developed, tested, deployed, and updated, and it forces loosely coupled communication.

For communication across ViewModels when there is not a direct action-reaction expectation, you can use a **messaging service**. The idea with this pattern is to have a component which is referenced by any part of the application that wants to listen or to send messages. This component is commonly implemented using the **mediator pattern** and allows us to create an architecture in which senders and listeners do not know each other’s in order to obtain a better modularity.

If you don't want to implement your own messaging service, the Patterns and Practices Prism Project has the **Event Aggregation Service**. This mechanism allows publishers and subscribers to communicate through events and still do not have a direct reference to each other. The EventAggregator provides multicast publish/subscribe functionality. This means there can be multiple publishers that raise the same event and there can be multiple subscribers listening to the same event. Consider using the EventAggregator to publish an event across modules and when sending a message between ViewModels.

5.5.- Testing MVVM Applications

Silverlight Unit Test projects are not default in Visual Studio 2010; it is part of the Silverlight Toolkit. After you install the toolkit, you'll have a new project type called Silverlight Unit Test Application. The Project Template will add a new page (or web project) for the unit tests.

Testing models and ViewModels from MVVM applications is the same as testing any other classes, and the same tools and techniques can be used. However, there are some testing patterns that are typical to model and ViewModel classes and can benefit from standard testing techniques and test helper classes.

5.5.1.- Testing INotifyPropertyChanged Implementations

Implementing the **INotifyPropertyChanged** interface allows Views to react to changes originated in models and ViewModels. These changes are not limited to domain data shown in controls; they are also used to control the View, such as ViewModel states that cause animations to be started or controls to be disabled.

Properties that can be updated directly by the test code can be tested by attaching an event handler to the **PropertyChanged** event and checking whether the event is raised after setting a new value for the property. Helper classes, such as the **ChangeTracker** class used in the MVVM sample projects, can be used to attach a handler and collect the results; this avoids repetitive tasks when writing tests.

```
public class PropertyChangeTracker
{
    private INotifyPropertyChanged _changer;
    private readonly System.Collections.Generic.List<string>
         notifications = new List<string>();

    public PropertyChangeTracker(INotifyPropertyChanged changer)
    {
        _changer = changer;
        changer.PropertyChanged += (o, e) =>
                                notifications.Add(e.PropertyName);
    }

    public void Reset()
```

```
        {
            notifications.Clear();
        }

        public string[] ChangedProperties
        {
            get { return notifications.ToArray(); }
        }
    }
```

An example of use:

```
var changeTracker = new PropertyChangeTracker(ViewModel);

ViewModel.CurrentState = "newState";

CollectionAssert.Contains(changeTracker.ChangedProperties,
                          "CurrentState");
```

Properties that are the result of a code-generation process that guarantees the implementation of the **INotifyPropertyChanged** interface, such as those in code generated by a model designer, typically do not need to be tested.

5.5.2.- Testing INotifyDataErrorInfo Implementations

There are two aspects to testing **INotifyDataErrorInfo** implementations: testing that the validation rules are correctly implemented and testing that the requirements for implementations of the interface, such as raising the **ErrorsChanged** event when the result for the GetErrors method would be different, are met.

Testing Validation Rules

Validation logic is usually simple to test, because it is typically a self-contained process where the output depends on the input. For each property with validation rules associated, there should be tests on the results of invoking the GetErrors method with the validated property name for valid values, invalid values, boundary values, and so on. If the validation logic is shared, like when expressing validation rules declaratively using the data annotation's validation attribute, the more exhaustive tests can be concentrated on the shared validation logic. On the other hand, custom validation rules must be thoroughly tested.

```
// Invalid case
var notifyErrorInfo = (INotifyDataErrorInfo)customer;

customer.Age = -18;

Assert.IsTrue(notifyErrorInfo.GetErrors("Response")
                 .Cast<ValidationResult>().Any());

    // Valid case
    var notifyErrorInfo = (INotifyDataErrorInfo)customer;

    customer.Age = 18;
    Assert.IsFalse(notifyErrorInfo.GetErrors("Response")
                  .Cast<ValidationResult>().Any());
```

Cross-property validation rules follow the same pattern, typically requiring more tests to accommodate the combination of values for the different properties.

Testing the Requirements for INotifyDataErrorInfo Implementations

Besides producing the right values for the GetErrors method, implementations of the **INotifyDataErrorInfo** interface must ensure the ErrorsChanged event is raised appropriately, such as when the result for GetErrors would be different. Additionally, the HasErrors property must reflect the overall error state of the object implementing the interface.

There is no mandatory approach for implementing the **INotifyDataErrorInfo** interface. However, implementations that rely on objects that accumulate validation errors and perform the necessary notifications are typically preferred because they are simpler to test. This is because it is not necessary to verify that the requirements for all the members of the INotifyDataErrorInfo interface are met for each validation rule on each validated property (as long, of course, as the error management object is properly tested).

Testing the interface requirements should involve at least the following verifications:

- The **HasErrors** property reflects the overall error state of the object. Setting a valid value for a previously invalid property does not result in a change for this property if other properties still have invalid values.

- The **ErrorsChanged** event is raised when the error state for a property changes, as reflected by a change in the result for the GetErrors method. The error state change could be going from a valid state (that is, no errors) to an invalid state and vice versa, or it can go from an invalid state to a different invalid state. The updated result for GetErrors is available for handlers of the ErrorsChanged event.

When testing implementations for the **INotifyPropertyChanged** interface, helper classes, such as the **NotifyDataErrorInfoTestHelper** class, usually make writing tests for implementations of the **INotifyDataErrorInfo** interface easier by handling repetitive housekeeping operations and standard checks.

```
    using System;
```

```
using System.ComponentModel;
using System.Linq.Expressions;
using System.Reflection;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MVVM.TestSupport
{
    public class NotifyDataErrorInfoTestHelper<T, TProperty>
        where T : INotifyDataErrorInfo
    {
        private readonly T _viewModel;
        private readonly string _propertyName;
        private bool _errorsChangedNotified;

        public NotifyDataErrorInfoTestHelper(
            T viewModel,
            Expression<Func<T, TProperty>> propertyExpression)
        {
            _viewModel = viewModel;
            _propertyName = ExtractPropertyName(propertyExpression);
        }

        public void ValidatePropertyChange(
            TProperty newValue,
            NotifyDataErrorInfoBehavior behavior)
        {
            _viewModel.ErrorsChanged += HandleViewModelErrorsChanged;

            SetPropertyValueOnViewModel(newValue);

            var messageError = string.Format("When setting value to {0} ",
                                   newValue);

            Assert.AreEqual(
                IsSet(NotifyDataErrorInfoBehavior.FiresErrorsChanged, behavior),
                _errorsChangedNotified,
                 messageError + "fired ErrorsChanged event");
            Assert.AreEqual(
                IsSet(NotifyDataErrorInfoBehavior.HasErrors, behavior),
                _viewModel.HasErrors, messageError + "HasErrors");
            Assert.AreEqual(
                IsSet(NotifyDataErrorInfoBehavior.HasErrorsForProperty, behavior),
                (_viewModel.GetErrors(_propertyName)
                   ?? new object[0]).GetEnumerator().MoveNext(),
                messageError + "HasErrorsForProperty");

            _viewModel.ErrorsChanged -= HandleViewModelErrorsChanged;
            _errorsChangedNotified = false;
        }

        public void ValidateAllPropertyChanges(
            TProperty validValue,
            TProperty anotherValidValue,
            TProperty invalidValue,
            TProperty anotherInvalidValue)
        {
            // Valid -> Valid
            SetPropertyValueOnViewModel(validValue);
            ValidatePropertyChange(anotherValidValue,
                NotifyDataErrorInfoBehavior.Nothing);

            // Valid -> Invalid
            SetPropertyValueOnViewModel(validValue);
            ValidatePropertyChange(invalidValue,
                NotifyDataErrorInfoBehavior.FiresErrorsChanged
                | NotifyDataErrorInfoBehavior.HasErrors
                | NotifyDataErrorInfoBehavior.HasErrorsForProperty);

            // Invalid -> Valid
```

```
                SetPropertyValueOnViewModel(invalidValue);
                ValidatePropertyChange(validValue,
                    NotifyDataErrorInfoBehavior.FiresErrorsChanged);

                // Invalid -> Invalid
                SetPropertyValueOnViewModel(invalidValue);
                ValidatePropertyChange(anotherInvalidValue,
                    NotifyDataErrorInfoBehavior.HasErrors
                    | NotifyDataErrorInfoBehavior.HasErrorsForProperty);
            }

            private void HandleViewModelErrorsChanged(
                object sender,
                DataErrorsChangedEventArgs e)
            {
                if (e.PropertyName == _propertyName)
                {
                    _errorsChangedNotified = true;
                }
            }

            private void SetPropertyValueOnViewModel(
                TProperty newPropertyValue)
            {
                _viewModel.GetType()
                    .InvokeMember(
                        _propertyName,
                        BindingFlags.SetProperty
                        | BindingFlags.Public
                        | BindingFlags.Instance,
                        null,
                        _viewModel,
                        new object[] { newPropertyValue });
            }

            private static bool IsSet(
                NotifyDataErrorInfoBehavior behavior,
                NotifyDataErrorInfoBehavior behaviors)
            {
                return (behaviors & behavior) == behavior;
            }

            private static string ExtractPropertyName(
                Expression<Func<T, TProperty>> propertyExpression)
            {
                return ((MemberExpression)propertyExpression.Body).Member.Name;
            }
        }

        [Flags]
        public enum NotifyDataErrorInfoBehavior
        {
            FiresErrorsChanged = 1,
            HasErrors = 2,
            HasErrorsForProperty = 4,
            Nothing = 8
        }
    }
```

They are particularly useful when the interface is implemented without relying on some kind of reusable errors manager. The following code example shows this type of helper class.

```
    var helper =
        new NotifyDataErrorInfoTestHelper<NumericQuestion, int?>(
            question,
            q => q.Response);

    helper.ValidatePropertyChange(
        6,
        NotifyDataErrorInfoBehavior.Nothing);
    helper.ValidatePropertyChange(
        20,
        NotifyDataErrorInfoBehavior.FiresErrorsChanged
        | NotifyDataErrorInfoBehavior.HasErrors
        | NotifyDataErrorInfoBehavior.HasErrorsForProperty);
    helper.ValidatePropertyChange(
        null,
        NotifyDataErrorInfoBehavior.FiresErrorsChanged
        | NotifyDataErrorInfoBehavior.HasErrors
        | NotifyDataErrorInfoBehavior.HasErrorsForProperty);
    helper.ValidatePropertyChange(
        2,
        NotifyDataErrorInfoBehavior.FiresErrorsChanged);
```

5.5.3.- Testing Asynchronous Service Calls

When implementing the MVVM pattern, ViewModels usually invoke asynchronously operations on services. Tests for code that invokes these operations typically use mocks or stubs as replacements for the actual services.

The standard patterns used to implement asynchronous operations provide different guarantees regarding the thread in which notifications about the status of an operation occur. Dealing with threading concerns requires more complicated, and, therefore, usually harder to test, code. It also usually requires the tests themselves to be asynchronous. When notifications are guaranteed to occur in the UI thread, either because the standard event-based asynchronous pattern is used or because ViewModels rely on a service access layer to marshal notifications to the appropriate thread, tests can be simplified and can essentially play the role of a "dispatcher for the UI thread."

The way services are mocked depends on the asynchronous event pattern used to implement their operations. If a method-based based pattern is used, mocks for the service interface created using a standard mocking framework are usually enough, but if the event-based pattern is used, mocks based on a custom class that implements the methods to add and remove handlers for the service events are usually preferred.

The following code example shows a test for the appropriate behavior on the successful completion of an asynchronous operation notified in the UI thread using mocks for services. In this example, the test code captures the callback supplied by the ViewModel when it makes the asynchronous service call. The test then simulates the completion of that call later in the test by invoking the callback. This approach allows testing of a component that uses an asynchronous service without the complexity of making your tests asynchronous.

```
questionnaireRepositoryMock
    .Setup(
        r =>
            r.SubmitQuestionnaireAsync(
                It.IsAny<Questionnaire>(),
                It.IsAny<Action<IOperationResult>>()))
    .Callback<Questionnaire, Action<IOperationResult>>(
        (q, a) => callback = a);

uiServiceMock
    .Setup(svc => svc.ShowView(ViewNames.QuestionnaireTemplatesList))

    .Callback<string>(ViewName => requestedViewName = ViewName);
submitResultMock
    .Setup(sr => sr.Error)
    .Returns<Exception>(null);
CompleteQuestionnaire(ViewModel);
ViewModel.Submit();
// Simulate callback posted to the UI thread.
callback(submitResultMock.Object);
// Check expected behavior – request to navigate to the list View.
Assert.AreEqual(ViewNames.QuestionnaireTemplatesList,
 requestedViewName);
```

Using this testing approach only exercises the functional capabilities of the objects under test; it does not test that the code is thread safe.

5.6.- Benefits of using MVVM

The use of the MVVM pattern provides several fundamental **benefits**:

- A ViewModel provides a unique state storage and a representation of the data to be presented in the view, which improves the reuse of the Model (decoupling it from the Views) and enables changes to be made to the view (by removing specific presentation policies of the Views).

- A MVVM design makes it easier to write and run Tests (Unit Testing, specifically) on the application. When separating the logic from the views and visual controls, we can easily create unit tests exclusively responsible for the Model and ViewModel (since the Views will be normally just XAML, without code-behind). In addition, MVVM allows testing the ViewModels by mocking underlying layers which the ViewModels depend on.

- The MVVM pattern provides a decoupled design. The Views only reference the ViewModel and the ViewModel references the Model or it can call directly to the rest of the domain oriented architecture where it can even return domain entities instead of the Model. The rest is performed by the DataBinding and the Commands of the Silverlight/WPF infrastructure.

6.- IMPLEMENTING MVC WITH ASP.NET MVC

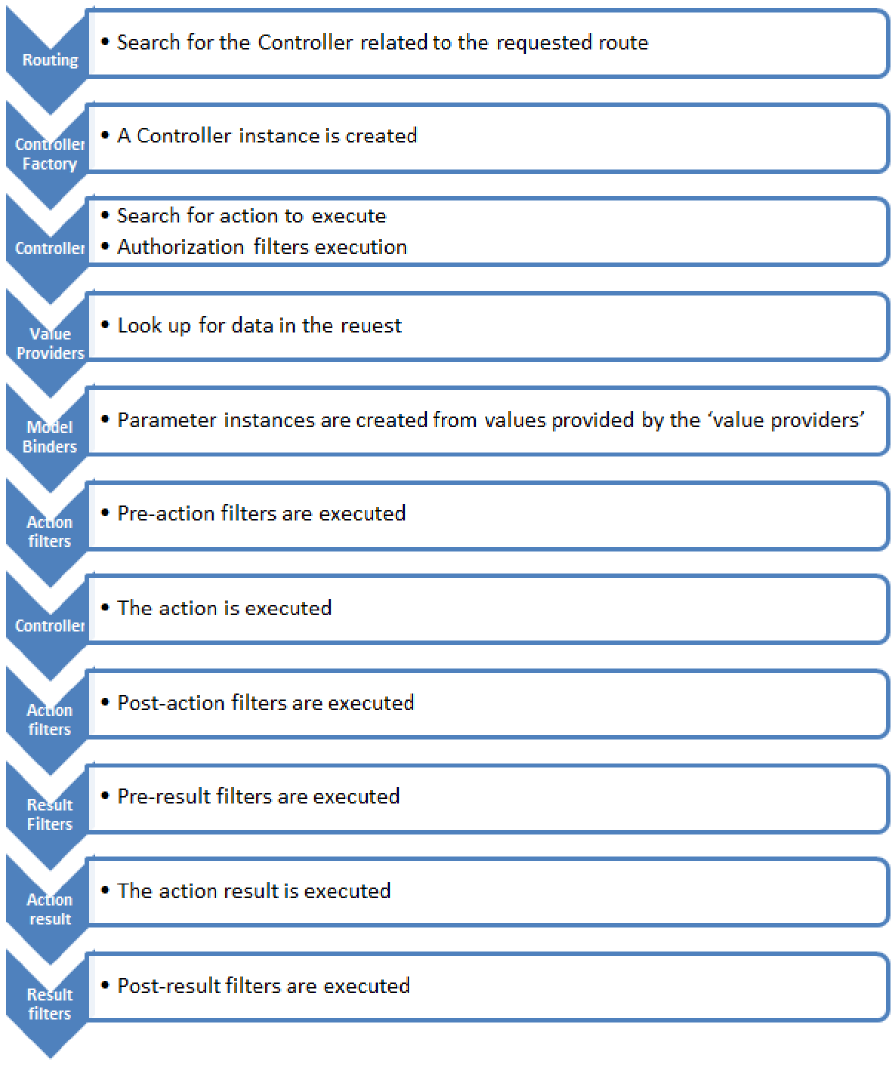
ASP.NET MVC is the implementation of the MVC pattern for the web. This implementation is based on the ASP.NET platform and therefore allows us to use many existing components such as authentication, authorization or cache.

6.1.- Basics of ASP.NET MVC

ASP.NET MVC organizes the user's interaction with the system of controllers that expose actions responsible for modifying or querying the system model. These actions return a result that, after being executed, returns an answer to the user. Let's look at the pipeline in more detail in order to understand it more easily.

6.2.- The ASP.NET MVC Pipeline

When an HTTP request appears, the Routing system tries to find an associated controller and action to answer the request based on the content and the same URL. After finding the proper controller, it requests an instance of it from the controller factory. Once it gets an instance from the controller, it searches within itself for the action to be invoked, maps the action parameters based on the content of the request (ModelBinding) and invokes the action with the parameters obtained. The controller action interacts with the system and returns a result. This result can be a file, a redirection to another controller, a view, etc. The most common case is the view (ViewResult). The results of an action are derived from ActionResult, and it is important to understand that they simply describe the result of the action and nothing happens until they are executed. In the case of a ViewResult, when executed, a ViewEngine is invoked, which is responsible for finding the requested view and rendering it to the HTML sent as a response.



Routing
• Search for the Controller related to the requested route
Controller Factory
• A Controller instance is created
Controller
• Search for action to execute
• Authorization filters execution
Value Providers
• Look up for data in the reuest
Model Binders
• Parameter instances are created from values provided by the 'value providers'
Action filters
• Pre-action filters are executed
Controller
• The action is executed
Action filters
• Post-action filters are executed
Result Filters
• Pre-result filters are executed
Action result
• The action result is executed
Result filters
• Post-result filters are executed

Figure 27.- ASP.NET MVC Pipeline

A controller communicates with a view through an object called ViewData, which is a dictionary in charge of storing the information necessary to render the view. Views can be strongly typed. In the case of being strongly typed, the view is a closed generic class (all the parameters of this type have been specified) and the ViewData exposes a Model property with the type of generic parameter of the view.

As a result, the intention of the view is better expressed, its contract with any controller is more explicit and clearer, and we have *Intellisense* to access the members of the model while coding the view.

6.3.- A Complete Example: Customer's Update

The first thing to do is delineate the scenario that we are implementing. Typically, in any MVC applications, updating an entity requires two actions: one to view a form with the entity we want to modify and another to send changes to the server. It is important to note that in this type of application we try to follow the REST protocol (Representational State Transfer) and this will be reflected in the controllers. Let's first look at what the two actions are like:

```
public ActionResult Edit(string customerCode)
{

   Customer customer =
            _CustomerService.FindCustomerByCode(customerCode);
   return View(customer);
}
public ActionResult Edit(Customer customer)
{
   if (ModelState.IsValid)
   {
     _CustomerService.ChangeCustomer(customer);
     return RedirectToAction("Details", new RouteValueDictionary()
                  { { "customerCode", customer.CustomerCode } });
   }
   else
   {
     return View();
   }
}
```

As we can see, we have a primary action to request the customer we want to edit and a secondary action to send the changes back. All the controller's work is divided into two responsibilities: to modify or query the domain, and to decide the screen to be displayed.

Let's look at the update view. If we are perceptive, we will realize that the first action is responsible for entering the current data in the form and the second action is responsible for processing that data.

If we take a look at the customer´s edition view, we see that it is divided into two parts; on one hand, the edition view corresponding to the editing action that is in charge of fitting the view within the page's global design, and linking the form to the corresponding action:

```
    <% using (Html.BeginForm("Edit","Customer",FormMethod.Post,
       new { enctype = "multipart/form-data", @class="editor-form"}))
       { %>
               <%: Html.EditorForModel() %>
               <%: Html.SerializedHidden(Model) %>
               <input type="submit" value="Guardar Cambios"
               class="text-button big-button" />
       <%} %>
```

On the other hand, the customer form view is responsible for showing the editable fields of a customer and performing as well as displaying their validation result:

```
       <div class="display-label">
       <%: Html.DisplayNameFor(x => x.CustomerCode) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.CustomerCode) %></div>
       <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.CustomerCode) %></div>
       <div class="display-label">
       <%: Html.DisplayNameFor(x => x.ContactName) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.ContactName) %></div>
       <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.ContactName) %></div>
       <div class="display-label">
       <%: Html.DisplayNameFor(x => x.ContactTitle) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.ContactTitle) %></div>
       <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.ContactTitle) %></div>
       <div class="display-label">
       <%: Html.DisplayNameFor(x => x.CompanyName) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.CompanyName) %></div>
       <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.CompanyName) %></div>
       <div class="display-label">
       <%: Html.DisplayNameFor(x => x.IsEnabled) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.IsEnabled,true) %></div>
       <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.IsEnabled) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.Country) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.Address) %></div>
       <div class="display-field">
       <%: Html.EditorFor(x => x.CustomerPicture) %></div>
```

The most important aspect we have to consider in this type of application is how we manage this state. When we update an entity we are modifying an existing state, so, to properly apply changes, we must have the original state. Both possible storage points of this original state are the server (database or distributed cache) or the client. We opted to store this original state in the client, so we sent a serialized copy of the original data to the page, using HtmlHelper, which we have designed for that purpose and we can see in the edition view:

```
<%: Html.SerializedHidden(Model) %>
```

We have finished the action responsible for displaying the form with data to be edited, but once we have made the changes, how are such changes applied to the entity? If we know a little about the HTTP protocol, we know that when we “post” a form, we are sending name=value pairs in the request. This is far different from the parameters of the action responsible for saving changes in the entity that directly receives a client object. The process of translation between the request elements and the action parameters is called ModelBinding. ASP.NET MVC provides a ModelBinder by default, in charge of performing this translation based on conventions.

However, since we are using self-tracking entities, we have created our own ModelBinder to simplify the update process. As can be expected, at some point a ModelBinder needs to create an instance of the class to which it is mapping the request parameters. Thanks to the extensibility of ASP.NET MVC, we precisely extend this point, and if we have serialized the original entity, we return it instead of creating a new copy. By mapping the parameters, the entity can keep track of what parameters have been changed, simplifying the update process:

```
   public class SelfTrackingEntityModelBinder<T> :
        DefaultModelBinder where T : class, IObjectWithChangeTracker
{
    protected override object CreateModel(ControllerContext controllerCo
ntext, ModelBindingContext bindingContext, Type modelType)
    {
        string steName = typeof(T).Name+"STE";
        if(bindingContext.ValueProvider.ContainsPrefix(steName)){
            var value = bindingContext.ValueProvider.GetValue(steName);
            return new SelfTrackingEntityBase64Converter<T>()
                                .ToEntity(value.AttemptedValue);
        }
        return base.CreateModel(controllerContext, bindingContext,
   modelType);
    }
   }
```

6.4.- Other Aspects of the Application

An MVC application is an ASP.NET application sub-type, so we have a global.asax file that is the access point for our application. Typically, all the necessary initializations are made in this file. In the case of ASP.NET MVC these initializations are often the following:

- Registration of URL mapping routes / (Controller, Action).
- Registration of Dependencies in the container.
- Registration of specialized controller factories.
- Registration of alternative view engines.
- Registration of specific model binders.
- Registration of alternative value providers.
- Registration of areas.

An MVC application can be divided into areas in order to improve its organization. Each area should be responsible for making all the indicated records, as necessary. This code should first be in the global.asax file (within the Application_Start method) but we can assume that in a large application this method can reach a large number of code lines. Therefore, although we have not created areas, we have delegated the responsibility of each "area" (we consider the application as the only defined area) in a BootStrapper responsible for initializing each area (Registering dependencies in the container, Model Binders, etc.)

To be more specific, each area is responsible for registering its dependencies in the dependency container (mainly the controllers). In the framework extensions, a factory is defined that uses the dependency container to create the controllers and automatically inject its dependencies.

REFERENCES:

- *Presentation Model - Martin Fowler, July 2004.*

  http://martinfowler.com/eaaDev/PresentationModel.html

- *Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2)*

- *Introduction to Model/View/ViewModel pattern for building WPF apps - John Gossman, October 2005.*

  http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx

- *Separated Presentation - Martin Fowler, June 2006.*

  http://www.martinfowler.com/eaaDev/SeparatedPresentation.html

- *WPF Patterns - Bryan Likes, September 2006*

  http://blogs.sqlxml.org/bryantlikes/archive/2006/09/27/WPF-Patterns.aspx.

- *WPF patterns: MVC, MVP or MVVM or…? - the Orbifold, December 2006.*

  http://www.orbifold.net/default/?p=550

- *Model-see, Model-do, and the Poo is Optional - Mike Hillberg, May 2008.*

  http://blogs.msdn.com/mikehillberg/archive/2008/05/21/Model-see_2C00_-model-do.aspx

- *PRISM: Patterns for Building Composite Applications with WPF - Glenn Block, September 2008.*

  http://msdn.microsoft.com/en-us/magazine/cc785479.aspx

- *The ViewModel Pattern - David Hill, January 2009.*

  http://blogs.msdn.com/dphill/archive/2009/01/31/the-viewmodel-pattern.aspx

- *WPF Apps with the Model-View-ViewModel Design Pattern* - *Josh Smith, February 2009.*

  http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

- *Model View Presenter – Jean-Paul Boodhoo, August 2006.*

  http://msdn.microsoft.com/en-us/magazine/cc188690.aspx