Model-View-ViewModel (MVVM) Design Pattern using Windows Presentation Foundation (WPF) Technology

ERIK SØRENSEN[1], PhD.
MARIUS IULIAN MIHAILESC2[2], Eng.

[1]Department of Computer Engineering
University of Southern Denmark, The Maersk Mc-Kinney Moller Institute
Campusvej 55, DK-5230 Odense M, Denmark

[2]Department of Computer Science,
Titu Maiorescu University, Faculty Science and Information Technology,
Str. Dambovnicului, nr. 22, sector 4, 040441 Bucuresti, Romania, E-Mail: mihailescu.marius@yahoo.com

Abstract: Information society has and will involve electronic information and dependence computer networks. The program includes a substantial proportion of resources which are allocated to information and communication activities. Globalization digital consequently, the priority of the Internet, is a phenomenon inevitably, specific information for society. In this whirlwind are attracted education systems which are beginning to adapt and to transform the electronic versions, marking transition from traditional to digital age in which earning is approached as a process that takes place during the life cycle through interactive learning environments. In the present communication, explores how use of computers beyond the traditional form of IAC (most frequently encountered form of drill & practice).

Keywords: *Model-View-ViewModel, MVVM*

1. INTRODUCTION

Since people began to create user interfaces for software were popular design patterns to help make it easier. For example, the Model-View-Presenter pattern (MVP), has gained popularity in different programming platforms IU. MVP is a variant of the Model-View-Controller pattern (MVP), which has been around for decades. In case you have never used before MVP model, here is a simplified explanation. What you see on screen is the view, which displays the data is a model and presenter joins the two together. View relies on a presenter to populate the data model, react to user input, ensure input validation (perhaps by delegating to the model), and other such tasks.
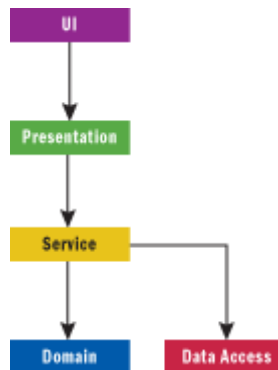

Figure 1, Application Architecture

Back in 2004, Martin Fowler published an article about a pattern called Presentation Model (PM). AM pattern is similar to MVP, in that it separates a view of behavior and its status. The interesting pattern of PM is that a view is created by abstraction, called PM. One view, then, becomes one of PM's rendering. In Fowler's explanation, he shows that a PM's frequently updates its view, so that the two are in synchronization with each other. This synchronous logic code exists as Presentation Model classes.

In 2005, John Gossman, one of the architects of WPF and Microsoft Silverlight on his blog reveals pattern Model-View-ViewModel (MVVM). MVVM presentation is the same model (PM) presented by Fowler, is that both models have that feature an abstraction of a View, which contains state and behavior of View. AM Fowler brought a meaning to create a platform independent graphical interface View abstractizand a while Gossman MVVM introduced as a standardized way to inluenta basic features of WPF thus simplifying the creation of interfaces. In this sense, I think MVVM pattern as a specialization of more general AM pattern in order for WPF and Silverlight platforms.

In the article's of Glenn Block's "Prism: Patterns for Building Composite Applications with WPF", he explains the guidelines for Microsoft Composite Application for WPF. ViewModel only term used. But word of Presentation Mode is used to describe an abstract view. In this project, I mean I MVVM pattern and abstraction of the view that a ViewModel. I find this terminology more common in books about WPF and the Silverlight community.

In contrast to the Presenter in MVP, a ViewModel not need a reference to a view. A view of ViewModel assigned properties, which, in turn, expose data models and objects contained in other state-specific view. ViewModel award of view and are easy to build as one object is assigned as a DataContext ViewModel for that view. If properties in ViewModel values change, those new values will propagate the view is automatically your data via the award. When the user clicks on a button in the View, a command is executed ViewModel action necessary to achieve that. ViewModel, never View, realizaeaza all changes made to the data model.

Classes view, have no idea that such classes exist model, and model as long as ViewModel are unconscious of View. In fact, the model is completely clear on the fact that there ViewModel and view. This is a very delicate design, which pays, dividends in many ways as we will see you soon.

2. WHY MVVM ?

Once a developer is comfortable with WPF and MVVM may be difficult to differentiate the two. MVVM is the lingua franca of WPF developers, the platform that is well adapted to WPF, and WPF was designed to make it easier to build applications using MVVM model (among others). In fact, Microsoft has used internally MVVM to develop WPF applications such as Microsoft Expression Blend, while WPF was the basic core design. Many aspects of WPF, such as the look of control and data model templates, use powerful separation of state and behavior display MVVM promoted.

Two other features of the WPF technology that make these models are patterns of data and system resources. Templates for data visualization applied to objects ViewModel presented user interface. Being able to declare templates in XAML and leaving reurse system automatically locate acetic sis patterns for you to apply on time.

If children were not only WPF support orders, MVVM model would be much less powerful. In this project we show how one can expose VIewModel commands a view to allowing the use of its functionality.

In addition to the features of WPF (and Silverlight) MVVM forming a natural way to structure an application, the model is also popular because VIewModel classes are easy to test drive. When an interaction logic applications, a set of classes traieste"intr ViewModel, you can easily write code to test. In a sense, View and test sites are joined two other different types of consumers ViewModels. Having a test suite for an application based on free and ViewModel providing a rapid regression tests, helps reduce application maintenance costs over time.

In addition to promoting creearii automated regression tests, classes testabilitatea ViewModel properly can help in designing user interfaces. When you design an application, you can often decide whether something should be in terms of site ViewModel by imagining that you write a unit test without creating other object ViewModel IU.

In the end, for developers who work with visual designs using MVVM is much easier to create a sleek designer / developer workflow for. The intention is only consumed by a ViewModel arbitrary, it is easy to 'break"a way to set a new visuality sis make ViewModel vision. This simple step allows for rapid prototyping and evaluation of user interfaces made by designers.

Development team can focus on creating robust ViewModel classes, and design team can be concentrated to make user-friendly views. Connecting two outputs of teams that may involve little more than ensuring that proper linkages exist in view of how a XAML file.

3. PRACTICAL MODEL-VIEW-VIEWMODEL DESIGN PATTERN ON APPPLICATION USING WINDOWS PRESENATATION FOUNDATION (WPF)

Application is presented within this paper uses MVVM in several ways. It provides a fertile source of examples in order to put the concepts into an understandable context. I created this application for a company using Visual Studio 2008 SP1, Microsoft. NET Framework 3.5 SP1.

Application can contain any number of 'work spaces"utiliatorul can open by clicking the blue link (see Figure 2) in the left navigation section. All work spaces are in the TabControl's main window. The user can close the workspace by clicking the Close button in the tab workspace. The application has two workspaces available: "All Customers" and "New Customer." After running the application and opening workspaces, windows (IU) as shown in Figure 2.
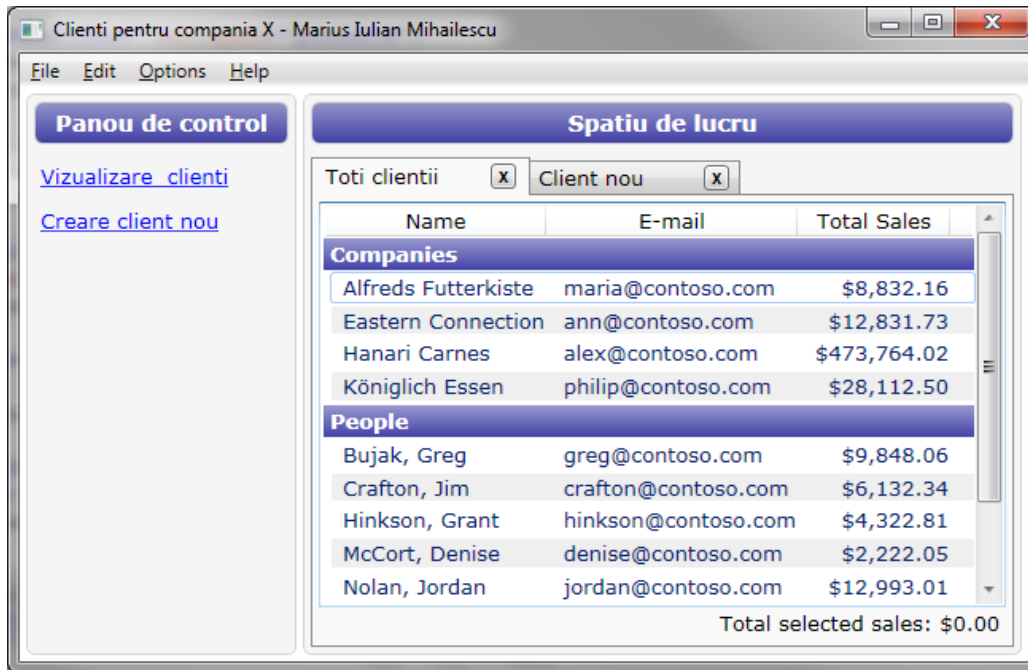
Figure 2. Workspaces

Only one instance of the workspace, all clients can be opened in a single moment, but any number of workspaces New user can be opened again. After completing the registration form data with valid values and clicking the Save button, new customer name will appear on the View tab customer. The application has support for deleting or editing an existing customer, but this functionality as well as many other similar features, are easy to create, based on top of existing application architecture. Now we have a high level of understanding of what application can do, let invetigam way it was designed and implemented.

3.1. Relaying Command Logic

Each application has a view on an empty file called codebehind, except for standard templates and code InitializeComponent manufacturer in its class. In fact, those lines could be removed from the file. Despite the lack of methods that allow event handling in some ways, when the user clicks the button, the application reacts and meets user demands. It works because the connection was established Command property of the hyperlink button, buttons that depict MENUITEM like displayed in the UI controls. These links ensure that, when the user clicks on controls, ICommand objects exposed by ViewModel run. We can think the command object as an adapter that makes it easy to make a feature ViewModel to eat from the stated purpose in XAML.

In this application, class RelayCommand solve this problem. RelayCommand allow you to 'inject' control logic via the constructor or the last delegation. RelayCommand DelegateCommand is a simplified version of what is found in Microsoft Application Composite library. RelayCommand class is shown in your code snippet below:

Listing 1. RelayCommand Class

```csharp
public class RelayCommand : ICommand
{
        #region Fields

        readonly Action<object> _execute;
        readonly Predicate<object> _canExecute;

        #endregion // Campuri

        #region Constructors

        /// <summary>
        /// Crearea unei comeni (command) care se poate executa oricand.
        /// </summary>
        /// <param name="execute">Executia logica (vezi documentatie)param>
        public RelayCommand(Action<object> execute)
            : this(execute, null)
        {
        }

        /// <summary>
        /// Crearea unei noi comenzi (command)
        /// </summary>
        public RelayCommand(Action<object> execute, Predicate<object> canExecute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");

            _execute = execute;
            _canExecute = canExecute;
        }

        #endregion // Constructorii

        #region ICommand Members

        [DebuggerStepThrough]
        public bool CanExecute(object parameter)
        {
            return _canExecute == null ? true : _canExecute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public void Execute(object parameter)
        {
```

```
            _execute(parameter);
        }
        #endregion // Membrii interfetei ICommand
}
```

CanExecuteChanged event, which is part of implementing ICommand interface, has some interesting features. He creates a delegate to CommandManager.RequerySuggested. This infrastructure ensures that all objects RelayCmmand solitcita WPF control, in which case they can execute commands whenever require built-in type. Class CustomerViewModel following code, which we will examine later, shows how to configure a RelayCommand using lambda expressions:

Listing 2. Configuration of RelayCommand with the help of lambda expressions

```
RelayCommand _saveCommand;
public ICommand SaveCommand
        {
            get
            {
                if (_saveCommand == null)
                {
                    _saveCommand = new RelayCommand(
                        param => this.Save(),
                        param => this.CanSave
                        );
                }
                return _saveCommand;
            }
        }
```

3.2.   *ViewModel* Class Ierarchy

ViewModel Most classes require the same characteristics. They often need to implement the INotifyPropertyChanged interface usually require a user friendly display, and if work space, they need the ability to close (which you can be removed from IU). This problem lends itself naturally to a class of basic attributes on creation ViewModel or two, so that new classes can inherit all the functionality ViewModel a common base class. ViewModel classes inheritance hierarchy as seen in Figure 2
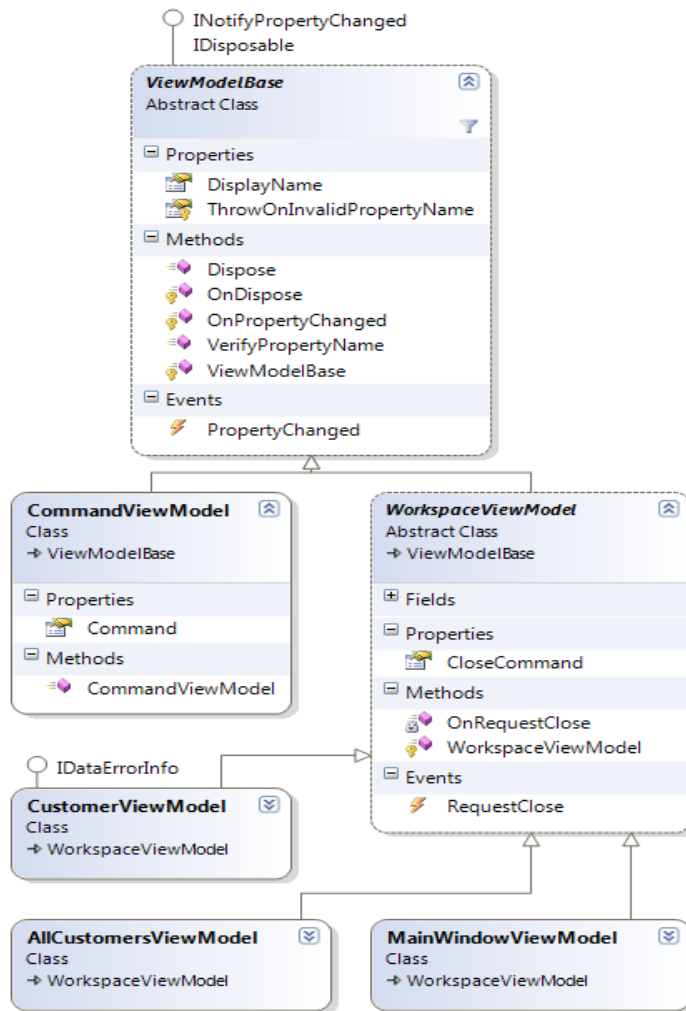
Figure 3. Inheritance Ierarchy

Having a base class for ViewModels not a necessity. If you prefer to obtain positions of your classes through the formation of several smaller classes together, instead of using the principle of inheritance, there is no problem. Like any other model design, MVVM is a set of guidelines, not rules.

3.3. ViewModelBase Class

ViewModelBase root in the class hierarchy, so that implements INotifyPropertyChanged interface typically used is proprietary and DisplayName. INotifyPropertyChanged interface includes an event called PropertyChanged. Whenever a proprietary ViewModel an object has a new value, may raise a PropertyChanged event to notify the WPF binding system of the new value. Upon receiving notification, mandatory system of queries on proprietary, and related property of the UI elements that receives the new value.

As a WPF application on a proprietary know what changed VIewModel object class exposes a proprietary PropertyChangedEventArgs string propertyName. But care must be taken when it comes to transmission propietatii correct name, otherwise property of WPF will end badly for querying a new value.

An interesting aspect is that the ViewModelBase furnizeaa ability to check with a proprietary name given to an object exist ViewModel. This is useful when it comes to refactoring, because changing the name of properties via the Visual Studio 2008 refactoring feature you update strings in the source code containing the name propietatii. Invoking the names of proprietary event PropertyChange wrong event argument can lead to bugs that are hard is pursued in finding this little

feature can be a huge saving of time. ViewModelBase code that adds this support helpful is presented in the following listing:

Listing 3. Suport for Code Refactoring

```
public event PropertyChangedEventHandler PropertyChanged;
protected virtual void OnPropertyChanged(string propertyName)
{
    this.VerifyPropertyName(propertyName);
    PropertyChangedEventHandler handler = this.PropertyChanged;
    if (handler != null)
    {
        var e = new PropertyChangedEventArgs(propertyName);
        handler(this, e);
    }
}
[Conditional("DEBUG")]
[DebuggerStepThrough]
public void VerifyPropertyName(string propertyName)
{
    if (TypeDescriptor.GetProperties(this)[propertyName] == null)
    {
        string msg = "Invalid property name: " + propertyName;
        if (this.ThrowOnInvalidPropertyName)
            throw new Exception(msg);
        else
        Debug.Fail(msg);
      }
}
```

3.4.   CommandViewModel Class

Simple and concrete subclass, ViewModelBase is CommandViewModel. Exhibit a type ICommand Command called proprietary. MainWindowViewModel exhibit a collection of objects through propietatii Commands. Left navigation area of the main window, a link to feicare CommandViewModel afiseaa exposed by MainWindowViewModel, and 'View"and customers, "Creating customer." When the user clicks the link, run one of these commands, a working space opens in the main window TabControl control. CommandViewModel class definition is shown in the following listing:

Listing 4. Definition of CommandViewModel Class

```
public class CommandViewModel : ViewModelBase
    {
        public CommandViewModel(string displayName, ICommand command)
        {
            if (command == null)
                throw new ArgumentNullException("command");

            base.DisplayName = displayName;
            this.Command = command;
        }

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

In the file there is a DataTemplate MainWindowResources.xaml identified key 'CommandsTemplate." MainWindow uses this template to render CommandViewModels collection mentioned earlier. Simple template CommandViewModel renders each object as a link in an ItemsControl. Each proprietary HyperlinkCommand is assigned to the Command property of a CommandViewModel. The following XAML code shown in the following listing:

Listing 5. Render list of Commands

```xml
<DataTemplate x:Key="CommandsTemplate">
  <ItemsControl IsTabStop="False" ItemsSource="{Binding}" Margin="6,2">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <TextBlock Margin="2,6">
          <Hyperlink Command="{Binding Path=Command}">
            <TextBlock Text="{Binding Path=DisplayName}" />
          </Hyperlink>
        </TextBlock>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</DataTemplate>
```

3.5.   Clasa *MainWindowViewModel*

As we noted in the class diagram, derived from ViewModelBase WorkspaceViewModel class and add the ability to close. By 'closing"means that the space delete some user interface work in run time mode. Three classes are derived from WorkspaceViewModel: MainWindowViewModel, AllCustomersViewModel and CustomerViewModel.
MainWindowViewModel is a request to close the deal by the App class that creates the MainWindow and ViewModel as it shows the following listing:

Listing 6. Creation of ViewModel

```csharp
protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            MainWindow window = new MainWindow();

            // Create the ViewModel to which
            // the main window binds.
            string path = "Data/customers.xml";
            var viewModel = new MainWindowViewModel(path);

            // When the ViewModel asks to be closed,
            // close the window.
            EventHandler handler = null;
            handler = delegate
            {
                viewModel.RequestClose -= handler;
                window.Close();
            };
            viewModel.RequestClose += handler;

            // Allow all controls in the window to
            // bind to the ViewModel by setting the
            // DataContext, which propagates down
            // the element tree.
```

```
            window.DataContext = viewModel;
            window.Show();
        }
```

MainWindow menu contains an item whose type is assigned propietatii proprietary Command CloseCommand of MainWindowViewModel. Can the user clicks that menu item, App class responds by calling Close window method, as follows:

```
  <Menu KeyboardNavigation.TabNavigation="Cycle">
      <MenuItem Header="_File">
        <MenuItem Header="E_xit" Command="{Binding Path=CloseCommand}" />
      </MenuItem>
      <MenuItem Header="_Edit" />
      <MenuItem Header="_Options" />
      <MenuItem Header="_Help" />
  </Menu>
```

MainWindowViewModel contains a collection of objects observable WorkspaceViewModel called Workspaces. The main window contains a TabControl whose proprietary ItemSource is assigned to that collection. Each tab has a button closure whose proprietary Command is assigned to CloseCommand WorkspaceViewModel court correspondent. A version of the template that configures each tab is shown in the following. Code can be found in MainWindowResources.xaml file and template rendering an item explains how to type a tab Close button:

```
  <DataTemplate x:Key="WorkspacesTemplate">
    <TabControl
      IsSynchronizedWithCurrentItem="True"
      ItemsSource="{Binding}"
      ItemTemplate="{StaticResource ClosableTabItemTemplate}"
      Margin="4"/>
  </DataTemplate>
```

When a user clicks the Close button in an item-type tab, then WorkspaceViewModel CloseCommand run, causing the launch event or RequestClose. MainWindowViewModel RequestClose event monitor desktop and delete the workspace Workspaces collection until the next request. Since when has the property of ItemsSource MainWindowTabControl attributed to observable WorkspaceViewModels collection, removing an item from the collection causes the workspace to be deleted from TabControl correspondent. This logic is shown in the following listing MainWindowViewModel:

Listing 7. Deleting an Workspace din UI

```
// In MainWindowViewModel.cs
ObservableCollection<WorkspaceViewModel> _workspaces;
public ObservableCollection<WorkspaceViewModel> Workspaces
        {
            get
            {
                if (_workspaces == null)
                {
                    _workspaces = new ObservableCollection<WorkspaceViewModel>();
                    _workspaces.CollectionChanged += this.OnWorkspacesChanged;
                }
                return _workspaces;
            }
        }

        void OnWorkspacesChanged(object sender, NotifyCollectionChangedEventArgs e)
```

```
        {
            if (e.NewItems != null && e.NewItems.Count != 0)
                foreach (WorkspaceViewModel workspace in e.NewItems)
                    workspace.RequestClose += this.OnWorkspaceRequestClose;

            if (e.OldItems != null && e.OldItems.Count != 0)
                foreach (WorkspaceViewModel workspace in e.OldItems)
                    workspace.RequestClose -= this.OnWorkspaceRequestClose;
        }

        void OnWorkspaceRequestClose(object sender, EventArgs e)
        {
            WorkspaceViewModel workspace = sender as WorkspaceViewModel;
            workspace.Dispose();
            this.Workspaces.Remove(workspace);
        }
```

The project UnitTest, MainWindowViewModelTests.cs file contains a test method that check that this function works correctly. The ease with which you can create unit tests for classes ViewModel is a strength for MVVM pattern because it allows for simple testing of application functionality without writing code that reaches"IU. This method is shown in the following listing:

Listing 8. Test Method

```
// In MainWindowViewModelTests.cs
[TestMethod]
public void TestCloseAllCustomersWorkspace()
{
            // Create the MainWindowViewModel, but not the MainWindow.
            MainWindowViewModel target =
                new MainWindowViewModel(Constants.CUSTOMER_DATA_FILE);

            Assert.AreEqual(0, target.Workspaces.Count, "Workspaces isn't empty.");

            // Find the command that opens the "All Customers" workspace.
            CommandViewModel commandVM =
                target.Commands.First(cvm => cvm.DisplayName ==
Strings.MainWindowViewModel_Command_ViewAllCustomers);

            // Open the "All Customers" workspace.
            commandVM.Command.Execute(null);
     Assert.AreEqual(1, target.Workspaces.Count, "Did not create viewmodel.");

            // Ensure the correct type of workspace was created.
            var allCustomersVM = target.Workspaces[0] as AllCustomersViewModel;
            Assert.IsNotNull(allCustomersVM, "Wrong viewmodel type created.");

            // Tell the "All Customers" workspace to close.
            allCustomersVM.CloseCommand.Execute(null);
            Assert.AreEqual(0, target.Workspaces.Count, "Did not close
viewmodel.");
        }
```

3.6. Apply an View to ViewModel

MainWindowViewModel indirectly add and delete items to TabControl WorkspaceViewModel the main window. Based on data binding, the Content property of a derived object ViewModelBase Tabita receives a display. ViewModelBase is not an IU, the conclusion has no support for rendering to legacy itself. By default, a non-visual object is rendered by WPF afiarea results of a call to the ToString method in a TextBlock. In conclusion, this is not what we want, only if users have a burning desire to see the type of classes ViewModel name.

In a very simple way to say how WPF's rendering an object using DataTemplates ViewModel sites. A DataTemplate has a value x: Key assigned to him. If WPF is trying to rendering one ViewModel objects, it will check if the system resources in order to have a DataTemplate type in which datatype is the same object type ViewModel. If you find one, he will use the template to render the object referenced by ViewModel propietatii Content tab.

It has a ResourceDictionary MainWindowResources.xaml. This dictionary is added to the hierarchy of resources to the main window, which means that the resources they contain are in order resource box. When a content of a tab is set to an object ViewModel a DataTemplate of this dictionary provides a view (which is a user control) was used for rendering, as the following listing:

Listing 9. View Supply

```xml
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:vm="clr-namespace:DemoApp.ViewModel"
  xmlns:vw="clr-namespace:DemoApp.View"
  >

  <!--
  This template applies an AllCustomersView to an instance
  of the AllCustomersViewModel class shown in the main window.
  -->
  <DataTemplate DataType="{x:Type vm:AllCustomersViewModel}">
    <vw:AllCustomersView />
  </DataTemplate>

  <!--
  This template applies a CustomerView to an instance
  of the CustomerViewModel class shown in the main window.
  -->
  <DataTemplate DataType="{x:Type vm:CustomerViewModel}">
    <vw:CustomerView />
  </DataTemplate>
```

No need to write any code to determine which view should be shown for one object ViewModel. WPF system's resources do all the heavy stuff for you so you can concentrated on more important things. In more complex scenarios, it is possible to programmatically select your view, but in most situations it is necessary only.

3.7. DataModel si Repository

I saw ViewModel objects are launched, displayed and closed by the application. Now after everything is in place, we may review the details of implementing a more specific application domain. Before us deep into the second workspace sites of application, ie, all customers and'"Client"new, first time to examine the model and data access classes. The designs of these classes have almost nothing to do with pattern MVVM because ViewModel can create a class to adapt almost any object on something friendly to WPF.

Single class model within this application is the Customer. This class has a set of properties that is helpful information on a company's customer as name, address and email. It also provides message validation by implementing IDataErrorInfo interface that existed years before WPF to hit the market. Customer class has nothing in it that shows that architecture is used in WPF application. Class can also be as good a legacy of a business library.

Data should come and go somewhere. In this application, an instance of class load and keeps all objects CustomerRepository Customer. It happens to upload customer data (customer) from an XML file, but the type of external data sources is irrelevant. Data can also come from a BAA data, web service, a file on disk. As long as there is an object. NET with some data in it, considering the place whence MVVM pattern can display data on screen.

CustomerRepository class provides several methods that allow to take over all the objects Customer, Customer to add a repository, and check if a customer already exists in the repository. Since the application does not allow the user to delete a client repository does not allow to delete a customer. CustomerAdded event is launched when a new client enters CustomerRepository through AddCustomer method.

A clearer definition of the application data model is very small compared with actual business application requirements, but this is important. What is important to understand is how the classes Customer and CustomerRepository ViewModel use. Note that is wrapped around the object CustomerViewModel Customer. It exposes a client state, and other states used to control CustomerView through a set of properties. CustomerViewModel not duplicate a client state, is simply exposed via delegation as well:

```
public string FirstName
{
    get { return _customer.FirstName; }
    set
    {
        if (value == _customer.FirstName)
            return;

        _customer.FirstName = value;

        base.OnPropertyChanged("FirstName");
    }
}
```
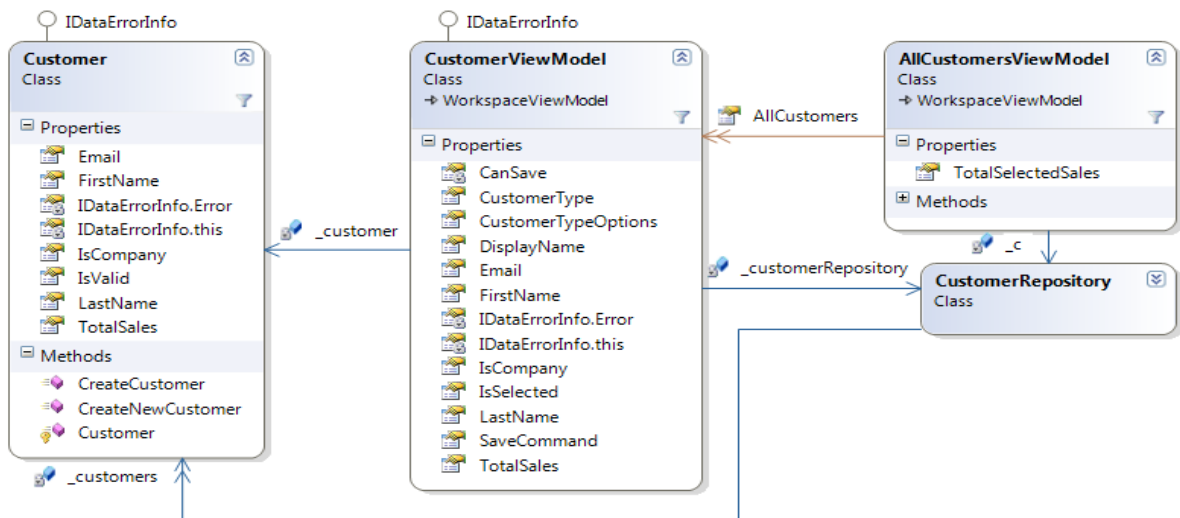

Figure 4. *Customer* Relation

13

When a user creates a new customer and click the Save button in the control CustomerView, CustomerViewModel associated with that view it will add new Client object CustomerRepository. This causes the event to be launched CustomerAdded that AllCustomersViewModel let know that you should add a new collection CustomerViewModel to AllCustomers. In a sense, acting as a mechanism CustomerRepository synchronized between various models of type ViewModels dealing with Customer objects. At a mere appearance he presents as Mediator pattern. I go back on it and how they work on these sections to come, but now we resume the following diagram for a high-level understanding of how all the parts are integrated.

3.8. New Customer Form

When the user clicks the link, create new client"adds a new CustomerViewModel MainWindowViewModel to its list of workspaces, control and display CustomerView. After the user enters valid values in the data fields, enter the Save button Enabled state so that the user can persist the new customer information.

Customer class has built-in support for validation, available through IDataErrorInfo interface implementation. This validation ensures that a customer has a name, e-mail address correct, and if the customer is a person has a name. If IsCompany property of the Customer class returns true, LastName property of a value can not have (the idea being that a company has no first name). This validation logic can make sense of objects of type Customer perspectives, but it does not meet the necessary requirements of the user interface. The user interface requires the user to decide if new customer is a person or company. Combo boxes which allow selection of the client initially has the value "(not specified). What if a user says IU is not specified and if the client type property of a Customer IsCompany values may allow for true or false?

Assuming control over the whole system was completed application, you can change the property of type Nullable IsCompany be <bool>, which will allow for values "unselected". Somehow, the real world is not always so simple. Assuming you can not change the Customer class as it comes from a legacy of libraries with different teams as a proprietary company. What happens if other applications already use the Customer class and based on what may be the property of the normal Boolean value?. Once again, ViewModel comes to help.

Test method in Figure 6 shows how this feature works in CustomerViewModel. CustomerViewModel exposes a proprietary CustomerTypeOptions so that Customer Type box is displayed three things. He also exposes the property of CustomerType, which keeps the string value of the checkbox selected. When CustomerType is set, it maps to a Boolean string value for the property of Customer obiectelui evidentiierea IsCompany. Figure 7 provides two properties.

Listing 10. Test Method

```
[TestMethod]
    public void TestCustomerType()
    {
        Customer cust = Customer.CreateNewCustomer();
        CustomerRepository repos = new
CustomerRepository(Constants.CUSTOMER_DATA_FILE);
        CustomerViewModel target = new CustomerViewModel(cust, repos);

        target.CustomerType =
Strings.CustomerViewModel_CustomerTypeOption_Company;
        Assert.IsTrue(cust.IsCompany, "Should be a company");

        target.CustomerType =
Strings.CustomerViewModel_CustomerTypeOption_Person;
        Assert.IsFalse(cust.IsCompany, "Should be a person");

        target.CustomerType =
Strings.CustomerViewModel_CustomerTypeOption_NotSpecified;
        string error = (target as IDataErrorInfo)["CustomerType"];
```

```
        Assert.IsFalse(String.IsNullOrEmpty(error), "Error message should be
returned");
      }
```

Listing 11. CustomerType Properties

```csharp
public string[] CustomerTypeOptions
    {
        get
        {
            if (_customerTypeOptions == null)
            {
                _customerTypeOptions = new string[]
                {
                    Strings.CustomerViewModel_CustomerTypeOption_NotSpecified,
                    Strings.CustomerViewModel_CustomerTypeOption_Person,
                    Strings.CustomerViewModel_CustomerTypeOption_Company
                };
            }
            return _customerTypeOptions;
        }
    }

public string CustomerType
    {
        get { return _customerType; }
        set
        {
            if (value == _customerType || String.IsNullOrEmpty(value))
                return;

            _customerType = value;

            if (_customerType ==
Strings.CustomerViewModel_CustomerTypeOption_Company)
            {
                _customer.IsCompany = true;
            }
            else if (_customerType ==
Strings.CustomerViewModel_CustomerTypeOption_Person)
            {
                _customer.IsCompany = false;
            }

            base.OnPropertyChanged("CustomerType");
            base.OnPropertyChanged("LastName");
        }
```

CustomerView ComboBox control contains a box which is attributed to these properties, as in:

```xml
<ComboBox
    x:Name="customerTypeCmb"
    Grid.Row="0" Grid.Column="2"
    ItemsSource="{Binding Path=CustomerTypeOptions, Mode=OneTime}"
```

```
            SelectedItem="{Binding Path=CustomerType, ValidatesOnDataErrors=True}"
            Validation.ErrorTemplate="{x:Null}"/>
```

When an item is selected in the ComboBox list, sources IDataErrorInfo data is interrogated to see if new values are valid. This occurs because the SelectedItem property of a set of mapped is placed on ValidatesOnDataErrors true. Since the data source is an object CustomerViewModel, mapping system, and ask for a"CustomerViewModel property of CustomerType validation errors. In most of the time, all require validation errors CustomerViewModel delegation to a Customer object it contains. Somehow, decand Customer has no notion of having the property of unselected state IsCompany class CustomerViewModel must handle new items in box ComboBox control. This code can be seen in the following listing:

Listing 12. Validarea unui obiect de clasa CustomerViewModel

```
string IDataErrorInfo.this[string propertyName]
        {
            get
            {
                string error = null;

                if (propertyName == "CustomerType")
                {
                    // The IsCompany property of the Customer class
                    // is Boolean, so it has no concept of being in
                    // an "unselected" state.  The CustomerViewModel
                    // class handles this mapping and validation.
                    error = this.ValidateCustomerType();
                }
                else
                {
                    error = (_customer as IDataErrorInfo)[propertyName];
                }

                // Dirty the commands registered with CommandManager,
                // such as our Save command, so that they are queried
                // to see if they can execute now.
                CommandManager.InvalidateRequerySuggested();

                return error;
            }
        }
```

The key aspect of this code is that implementation of IDataErrorInfo CustomerViewModel class can handle specific requirements for property for ViewModel and delegating other requests by the Customer object. This allows the use of validation logic and model classes with additional validation for the properties that make sense for ViewModel classes.

Ability to save a CustomerViewModel is available through the property of CustomerViewModel view to decide if it can save itself and what to do when it receives the request for relief of his condition. In this application, saving a new customer means of adding a CustomerRepository. Customer must be asked whether the object is valid or not, and must decide if CustomerViewModel valid. This two-part decision is necessary because the specific properties of ViewModel and examination prior validation. Logical saving for CustomerViewModel is shown in Figure 6.

Listing 13. Logica pentru Salvare pentru CustomerViewModel

```
  public ICommand SaveCommand
        {
            get
            {
                if (_saveCommand == null)
```

```
                {
                    _saveCommand = new RelayCommand(
                        param => this.Save(),
                        param => this.CanSave
                        );
                }
                return _saveCommand;
            }
        }
public void Save()
        {
            if (!_customer.IsValid)
            throw new InvalidOperationException(
            Strings.CustomerViewModel_Exception_CannotSave);

            if (this.IsNewCustomer)
                _customerRepository.AddCustomer(_customer);

            base.OnPropertyChanged("DisplayName");
        }
    bool IsNewCustomer
        {
            get { return !_customerRepository.ContainsCustomer(_customer); }
        }
bool CanSave
        {
            get { return String.IsNullOrEmpty(this.ValidateCustomerType())
            && _customer.IsValid; }
        }
```

ViewModel used here is much easier to create sites that can display view a Customer object and allowing for things like status "unselected" a Boolean properties. It also provides the ability to say in an easy way to save his client state. If your view is mapped directly to a Customer object, your view will require much more code to make the property of working in a prosperous way. MVVM a well-defined architecture, your codebehind to view most sites should be empty or contain code to handle checks and resources contained in that view. Sometimes you need to write code in your codbehind a view to interact with an object ViewModel as an event or call a method that can be very difficult altcumva invoked ViewModel himself.

3.9.   All Customers Form

Application of this object contains a workspace that displays all the clients in a ListView. Customers are grouped list agree with the category of part are individual companies. User can select one or more clients at the same time visualizing the total amount of sales in the lower right side.

The user interface is represented by AllCustomersView control. Each item is an object of the ListViewItem's collection CustomerViewModel AllCustomers AllCustomerViewModel exposed by the object. In the previous section, we saw a CustomerViewModel can yield such data in an entry form, and now the same object is rendered as an object CustomerViewModel a ListView. CustomerViewModel class has no idea what visual elements to display, and because of that it can reuse is possible.

AllCustomerView create groups that may be visible in ListView. He accomplishes this by mapping the ListView's ItemSource propietatii to CollectionViewSource configured as in the following listing:

Listing 14 CollectionViewSource

```xml
<!-- In AllCustomersView.xaml -->
<CollectionViewSource
  x:Key="CustomerGroups"
  Source="{Binding Path=AllCustomers}"
  >
  <CollectionViewSource.GroupDescriptions>
    <PropertyGroupDescription PropertyName="IsCompany" />
  </CollectionViewSource.GroupDescriptions>
  <CollectionViewSource.SortDescriptions>
    <!--
    Sort descending by IsCompany so that the ' True' values appear first,
    which means that companies will always be listed before people.
    -->
    <scm:SortDescription PropertyName="IsCompany" Direction="Descending" />
    <scm:SortDescription PropertyName="DisplayName" Direction="Ascending" />
  </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

And an association between a ListViewItem object is determined by CustomerViewModel ItemContainerStyle property of the ListView's. Assigned to this proprietary style is applied to each ListViewItem, which operates the property of a ListViewItem to be mapped to properties on CustomerViewModel. An important map in the style that creates a link between property of a ListViewItem and property of IsSelected IsSelected a CustomerViewModel, as in the following snippet:

```xml
<Style x:Key="CustomerItemStyle" TargetType="{x:Type ListViewItem}">
  <!--  Stretch the content of each cell so that we can
  right-align text in the Total Sales column.  -->
  <Setter Property="HorizontalContentAlignment" Value="Stretch" />
  <!--
  Bind the IsSelected property of a ListViewItem to the
  IsSelected property of a CustomerViewModel object.
  -->
  <Setter Property="IsSelected" Value="{Binding Path=IsSelected,
    Mode=TwoWay}" />
</Style>
```

The user interface is the property of TotalSelectedSales maps and apply the format for the local currency value. ViewModel subject may apply the current format, replace the view, by returning a string replace the value of property of type Double TotalSelectedSales. ContentStringFormat properties of ContentPresenter was added in. NET Framework 3.5 SP1, so if you need to use older versions of WPF, it takes money and apply formatting to the following code:

```xml
<!-- In AllCustomersView.xaml -->
<StackPanel Orientation="Horizontal">
  <TextBlock Text="Total selected sales: " />
  <ContentPresenter
    Content="{Binding Path=TotalSelectedSales}"
    ContentStringFormat="c"
  />
</StackPanel>
```

Bibliography

1. C# 3.0 Design Patterns, Judith Bishop, O'Reilly, 2007
2. Introduction to Design Patterns in C# and WPF, James W. Cooper, 2002, IBM T J Watson Research Center
3. WPF patterns: MVC, MVP or MVVM, http://www.orbifold.net/default/?p=550
4. MVVM, a WPF UI Design Pattern, http://channel9.msdn.com/shows/Continuum/MVVM/
5. Model View ViewModel, http://en.wikipedia.org/wiki/Model_View_ViewModel
6. MVVM Foundation, http://en.wikipedia.org/wiki/Model_View_ViewModel