MVVM - pattern

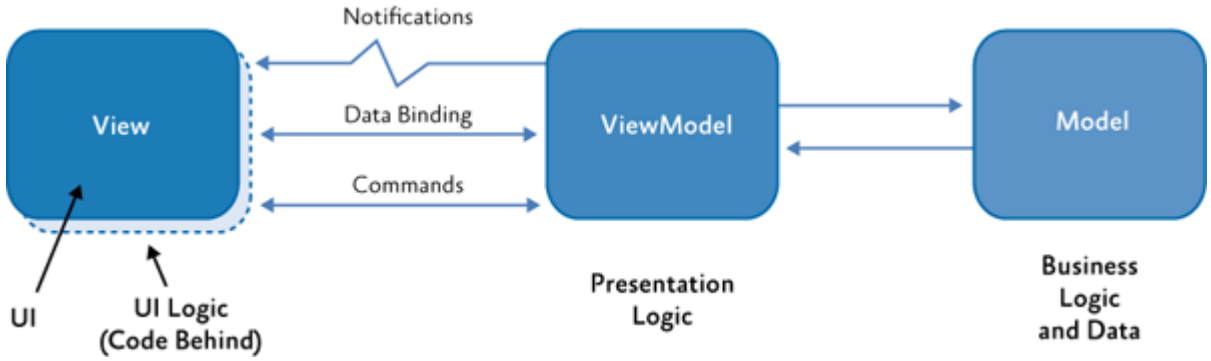
https://msdn.microsoft.com/en-us/library/gg405484

Pattern-ul MVVM este o varianta a pattern-ului MP (Presentation Model), optimizat pentru WPF. MVVM foloseste databinding, data templates, comenzi si comportari din WPF.

(V) - View – incapsuleaza UI si logica din UI.
(VM) - View Model – incapsuleaza starea si prezentarea logica.
(M) - Model – contine date si logica necesara acestora.

View interactioneaza cu **ViewModel** folosind data binding, comenzi si evenimente de notificare. **VM** coordoneaza actualizarile modelului, validand, convertind si agregand datele necesare pentru a fi afisate in **View**.

Clasele si interactiunile dintre acestea in cadrul pattern-ului MVVM sunt prezentate in figura urmatoare.



Problema principala cu acest pattern este aceea de a plasa corect codul in clasele necesare si de a intelege interactiunea dintre aceste clase.

V - Clasa View

In WPF, expresiile de asociere a datelor din view sunt evaluate folosind contextul pentru date. Contextul pentru date este setat in VM.

VM implementeaza proprietati si comenzi la care View se poate lega si notifica View asupra schimbarilor starilor folosind evenimente de notificare. In mod normal este o legatura de *unu-la-unu* intre View si VM.

View contine clase derivate din **Cotrol** sau **UserControl**. In alte situatii View poate fi reprezentat de de un template de date, ce specifica ce elemente UI trebuiesc folosite pentru a reprezenta un obiect cand acesta este afisat.

Data templates poate fi gandit ca fiind vizualizari ce nu au code-behind. Acestea sunt proiectate pentru a se lega la un tip specific de VM atunci cand li se cre sa afiseze date in UI. La run time, View definit ca data template, va fi instantiat in mod automat iar contextul de date va fi setat la VM corespunzator (vezi DataContext de la Data Binding).

In WPF, putem asocia un *data template* cu un tip de VM la nivel de aplicatie. WPF va aplica in mod automat *data template* la orice obiecte VM de tipul specificat cand acestea sunt afisate in UI, lucrul cunoscut sub numele de "*implicit data templating*". Data template poate fi definit in-line cu controlul ce il foloseste sau intr-un dictionar de resurse in afara parintelui View si in mod declarativ in dictionarul de resurse al View.

In concluzie, View are urmatoarele caracteristici:

- ✓ View este un element vizual (window, page, user control sau data template). View defineste controalele pe care le contine, aparenta vizuala si stilurile.
- ✓ View referentiaza VM prin proprietatea **DataContext**. Controalele din View au datele asociate (legate) la proprietati si comenzi expuse de VM.
- ✓ View poate utiliza conversii ale datelor pentru a le prezenta sub formatul cerut de UI.
- ✓ View defineste si manipuleaza comportarea vizuala a UI (animatii sau tranzitii de la o stare la alta in VM sau interactiunea utilizatorului cu UI).
- ✓ Code-behind din View poate defini logica UI pentru a implementa comportarea vizuala a controalelor, lucru ce e dificil de facut in XAML.

VM - Clasa ViewModel

VM in pattern-ul MVVM incapsuleaza data si prezentarea logica pentru View.
VM nu are o referinta directa la View sau alte informatii despre tipurile din View sau implementari specifice din View.
VM implementeaza proprietati si comenzi la care View poate asocia date si notifica View asupra modificarilor intervenite in starea datelor.
Proprietatile si comenzile pe care le prezinta VM, definesc functionalitatea ce va fi oferita de UI, dar View determina cum va fi redata in UI aceasta functionalitate.
VM este responsabil pentru coordonarea interactiunii View cu orice clasa din Model.
In mod obisnuit exista o relatie de unu-la-unu intre VM si Model.
VM poate alege sa expuna clasele din model direct la View facand o asociere directa. In acest caz, clasele din **Model** trebuiesc proiectate astfel incat sa suporte data binding si evenimente de notificare.
VM poate defini proprietati aditionale pentru a suporta View. Aceste proprietati nu fac parte din Model.
VM poate implementa logica de validare a datelor pentru a asigura consistenta datelor.
VM poate defini starile logice pe care View le poate folosi pentru a furniza modificarile vizuale in UI.
VM va defini comenzi sau actiuni ce pot fi reprezentate in UI si pe care utilizatorul le poate invoca.

In concluzie, VM are urmatoarele caracteristici:

- ✓ VM este o casa non-vizuala si deci nu este derivata din calase de baza din WPF. VM este testabila in mod independent de View si Model.
- ✓ VM nu referentiaza in mod direct View. VM implementeaza *proprietati* si *comenzi* la care View se poate lega. VM notifica View asupra modificarilor aparute asupra datelor, implementand interfetele **INotifyPropertyChanged** si **INotifyCollectionChanged**.

- ✓ VM coordoneaza interactiunea View cu Model. VM poate implementa interfetele **IDataErrorInfo** si **INotifyDataErrorInfo** in vederea realizarii procesului de validare al datelor.
- ✓ VM defineste starile logice pe care View le poate reprezenta in mod vizual utilizatorului final.

Observatie
Orice este legat de prezentarea vizuala a UI pe ecran si care poate fi restilizat mai tarziu trebuie sa fie in View. Orice este important pentru comportarea logica a aplicatiei ar trebui sa fie in VM.

M – Clasa Model

Caracteristici ale calselor din Model

- ✓ M incapsuleaza logica aplicatiei si datele.
- ✓ M poate defini structuri de date bazate pe modelul aplicatiei, validari logice.
- ✓ M poate include cod pentru acces la date, caching. Poate fi folosit EF, servicii WEB, servicii WCF.
- ✓ M suporta notificari prin intermediul implementarii interfetelor **INotifyPropertyChanged** si **INotifyCollectionChanged**. Clasele din model ce reprezinta colectii de obiecte sunt derivate in mod obisnuit din clasa **ObervableCollection<T>,** ce furnizeaza o implementare a interfetei **INotifyCollectionChanged**.
- ✓ M suporta validarea datelor si raportarea erorilor prin implementarea interfetlor **IDataErrorInfo** si **INotifyDataErrorInfo**.

Aplicatie demo

Codul este scris de Josh Smith si preluat din MSDN Code Gallery

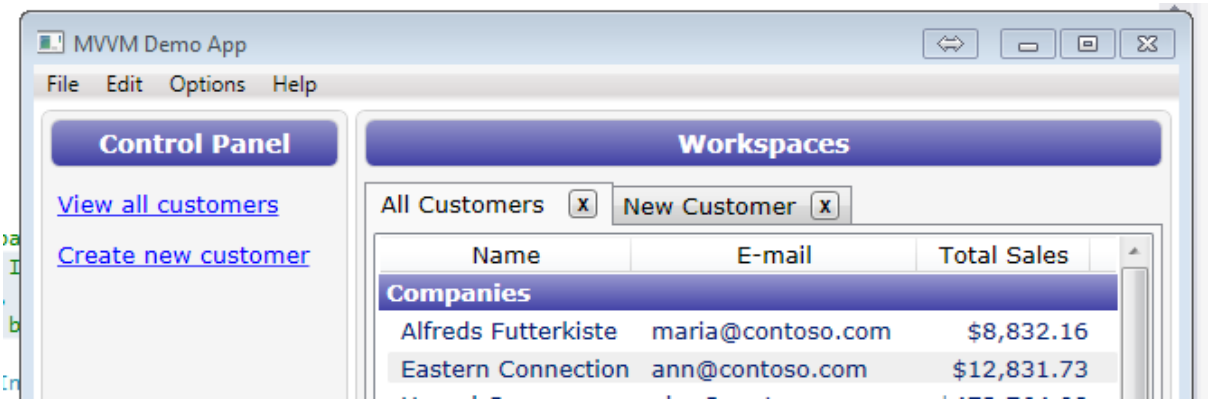
Modelul este dat de clasa **Customer** definita astfel:

```
namespace DemoApp.Model
{
    public class Customer : IDataErrorInfo
    {
        #region Creation
       // cod lipsa
        #endregion // Creation
        #region State Properties

        public string Email { get; set; }
        public string FirstName { get; set; }
        public bool IsCompany { get; set; }
        public string LastName { get; set; }
        public double TotalSales { get; private set; }

        #endregion // State Properties

       // cod lipsa
    }
}
```

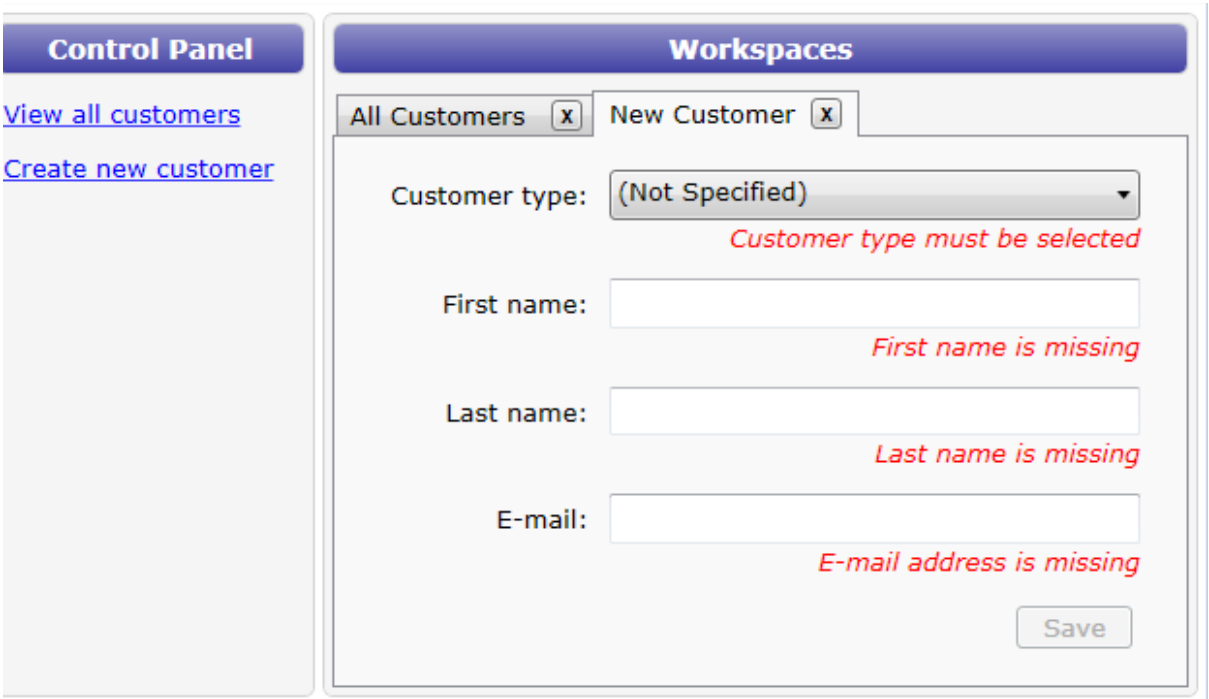
Aceasta clasa are preconstruita logica de validare. Clasa *CustomerViewModel* este un wrapper pentru aceasta clasa si permite sa fie afisata si editata de UI - WPF.

Scenariul de baza al aplicatiei este de a incarca informatiile (obiecte Customer) dintr-o sursa de date, sa le afiseze si sa creeze altele noi. Nu exista Update sau Delete.

Aplicatia poate contine orice numar de “workspaces”. Un workspace este deschis cand se face clic pe un buton din panel-ul de navigare.



Cand se face clic pe “Create new customer” (este un link) se creaza un articol in TabControl si care arata astfel:



Se afiseaza o serie de controale pentru a prelua informatia necesara crearii unui nou *Customer* si apoi salvarea acestuia in baza de date.

Command

Fiecare view din app are un fisier code-behind care nu contine decat apelul metodei pemtru initializarea componentelor – **InitializeComponent**(). Nu exista cod care sa trateze evenimentul click al utilizatorului pe buton. Acest lucru este rezolvat de proprietatea **Command** a controalelor *HyperLink*, *Button* si *MenuItem*. Cand utilizatorul face clic pe controale, obiectele **ICommand** expuse de *ViewModel* se executa. Ne putem imagina un obiect comanda ca un adaptor ce ne ajuta sa folosim functionalitatea unui VM dintr-un V declarat in XAML.

Obiectul **Command** foloseste obiectul VM pentru a-si realiza executia. Asta inseamana ca fiecare clasa VM trebuie sa implementeze intr-un anume fel tipuri ce sunt derivate din **ICommand** pentru fiecare comanda expusa de VM.

Problema se rezolva folosind delegates.

In aplicatia demo, clasa ***RelayCommand*** rezolva aceasta problema. ***RelayCommand*** este o varianta simplificata pentru **DelegateCommand** din Microsoft Composite Application Library.

```
using System;
using System.Diagnostics;
using System.Windows.Input;

namespace DemoApp
{
    /// <summary>
    /// A command whose sole purpose is to relay its functionality to other
    /// objects by invoking delegates. The default return value for the CanExecute
    /// method is 'true'.
    /// </summary>
    public class RelayCommand : ICommand
    {
        #region Fields

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

        #endregion // Fields

        #region Constructors
        /// <summary>
        /// Creates a new command that can always execute.
        /// </summary>
        /// <param name="execute">The execution logic.</param>
        public RelayCommand(Action<object> execute)
            : this(execute, null)
        { }

        /// <summary>
        /// Creates a new command.
        /// </summary>
        /// <param name="execute">The execution logic.</param>
        /// <param name="canExecute">The execution status logic.</param>
        public RelayCommand(Action<object> execute, Predicate<object> canExecute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");
```

```
            _execute = execute;
            _canExecute = canExecute;
        }

        #endregion // Constructors

        #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 // ICommand Members
    }
}
```

Observatie
Evenimentul **CanExecuteChanged**, ce face parte din interfata **ICommand**, transfera subscrierea evenimentului catre evenimentul **CommandManager.RequerySuggested**. Acest lucru asigura ca infrastructura de comanda din WPF "intreaba" toate obiectele RelayCommand daca pot executa comanda in caz contrar se executa comanda preconstruita. Un exemplu de apel este urmatorul:
Cod din CustomerViewModel.cs. (Save si CanSave sunt in aceasta clasa).

```
        /// <summary>
        /// Returns a command that saves the customer.
        /// </summary>
        public ICommand SaveCommand
        {
              // _saveCommand este de tip RelayCommand
              // definit in clasa CustomerViewModel.
            get
            {
                if (_saveCommand == null)
                {
                    _saveCommand = new RelayCommand(
                        param => this.Save(),
                        param => this.CanSave
                        );
                }
                return _saveCommand;
            }
        }
```

Clasele ViewModel

Aceste clase trebuie sa implementeze interfata **INotifyPropertyChanged**. In acest exemplu ierarhia de clase este data in figura ce urmeaza. Clasa de baza **ViewModelBase** implementeaza interfata **INotifyPropertyChanged** si de asemenea verifica ca proprietatea cu numele specificat exista.

```
/// <summary>
 /// Warns the developer if this object does not have
 /// a public property with the specified name. This
 /// method does not exist in a Release build.
 /// </summary>
 [Conditional("DEBUG")]
 [DebuggerStepThrough]
 public void VerifyPropertyName(string propertyName)
 {
     // Verify that the property name matches a real,
     // public, instance property on this object.
     if (TypeDescriptor.GetProperties(this)[propertyName] == null)
     {
         string msg = "Invalid property name: " + propertyName;

         if (this.ThrowOnInvalidPropertyName)
             throw new Exception(msg);
         else
             Debug.Fail(msg);
     }
 }
```

Metoda este apelata din

```
/// <summary>
 /// Raises this object's PropertyChanged event.
 /// </summary>
 /// <param name="propertyName">The property that has a new value.</param>
 protected virtual void OnPropertyChanged(string propertyName)
 {
     this.VerifyPropertyName(propertyName);

     PropertyChangedEventHandler handler = this.PropertyChanged;
     if (handler != null)
     {
         var e = new PropertyChangedEventArgs(propertyName);
         handler(this, e);
     }
 }
```

Clasa CommandViewModel

Aceasta clasa expune o proprietate numita ***Command*** de tip **ICommand**. *MainWindowViewModel* expune o colectie de aceste obiecte prin proprietatea ***Commands***. Zona de navigare din stanga ferestrei principale afiseaza un link pentru fiecare CommandViewModel expus de MainWindowsViewModel, cum ar fi “View all customers” si “Create new customer”. Definita clasei CommandViewModel este:

```
/// <summary>
/// Represents an actionable item displayed by a View.
/// </summary>
```

```
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 *MainWindowResources.xaml* exista un DataTemplate a carei cheie este “CommandsTemplate”. MainWindow foloseste acest template pentru a reda colectia de CommandViewModels. Proprietatea Command din HyperLink este legata de proprietatea Command din CommandViewModel. XAML est dat in continuare:

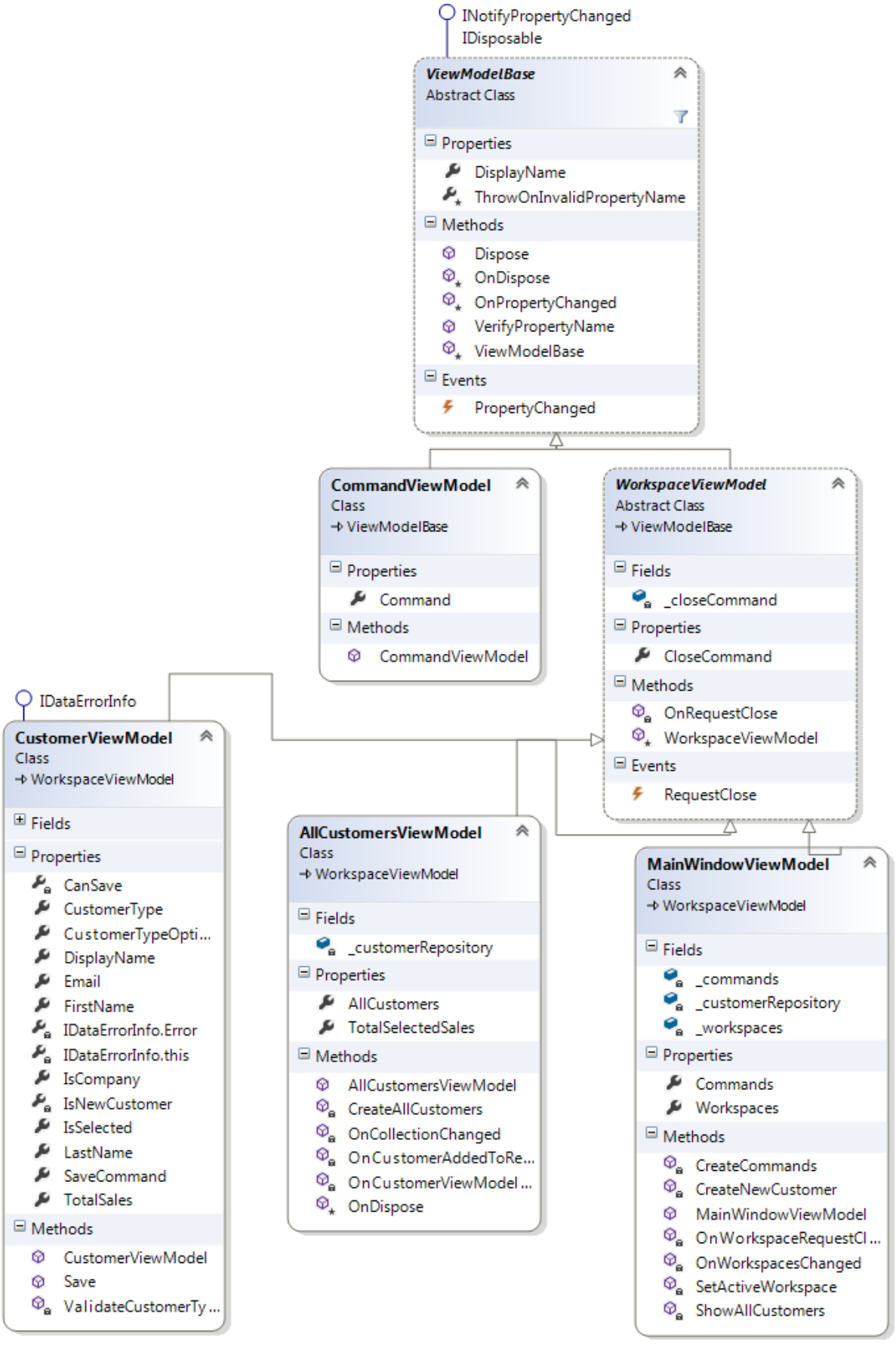
```
<!--
  This template explains how to render the list of commands on the left
  side in the main window (the 'Control Panel' area).
  -->
  <DataTemplate x:Key="CommandsTemplate">
    <ItemsControl IsTabStop="False" ItemsSource="{Binding}" Margin="6,2">
      <ItemsControl.ItemTemplate>
        <DataTemplate>
          <TextBlock Margin="2,6">
            <Hyperlink Command="{Binding Path=Command}">
              <TextBlock Text="{Binding Path=DisplayName}" />
            </Hyperlink>
          </TextBlock>
        </DataTemplate>
      </ItemsControl.ItemTemplate>
    </ItemsControl>
  </DataTemplate>
```

Clasa MainWindowViewModel

Trei clase sunt derivate din WorkspaceViewModel: MainWindowViewModel, AllCustomersViewModel si CustomerViewModel. Inchiderea unei ferestre paote fi facuta cu urmatorul cod:

```
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();
```


INotifyPropertyChanged
IDisposable
ViewModelBase
Abstract Class
Properties
DisplayName
ThrowOnInvalidPropertyName
Methods
Dispose
OnDispose
OnPropertyChanged
VerifyPropertyName
ViewModelBase
Events
PropertyChanged
CommandViewModel
Class
ViewModelBase
Properties
Command
Methods
CommandViewModel
WorkspaceViewModel
Abstract Class
ViewModelBase
Fields
_closeCommand
Properties
CloseCommand
Methods
OnRequestClose
WorkspaceViewModel
Events
RequestClose
IDataErrorInfo
CustomerViewModel
Class
WorkspaceViewModel
Fields
Properties
CanSave
CustomerType
CustomerTypeOpti...
DisplayName
Email
FirstName
IDataErrorInfo.Error
IDataErrorInfo.this
IsCompany
IsNewCustomer
IsSelected
LastName
SaveCommand
TotalSales
Methods
CustomerViewModel
Save
ValidateCustomerTy...
AllCustomersViewModel
Class
WorkspaceViewModel
Fields
_customerRepository
Properties
AllCustomers
TotalSelectedSales
Methods
AllCustomersViewModel
CreateAllCustomers
OnCollectionChanged
OnCustomerAddedToRe...
OnCustomerViewModel...
OnDispose
MainWindowViewModel
Class
WorkspaceViewModel
Fields
_commands
_customerRepository
_workspaces
Properties
Commands
Workspaces
Methods
CreateCommands
CreateNewCustomer
MainWindowViewModel
OnWorkspaceRequestCl...
OnWorkspacesChanged
SetActiveWorkspace
ShowAllCustomers