Proiectarea Sistemelor Software Complexe

Curs 11 –Modele Arhitecturale Care Asigură Separarea

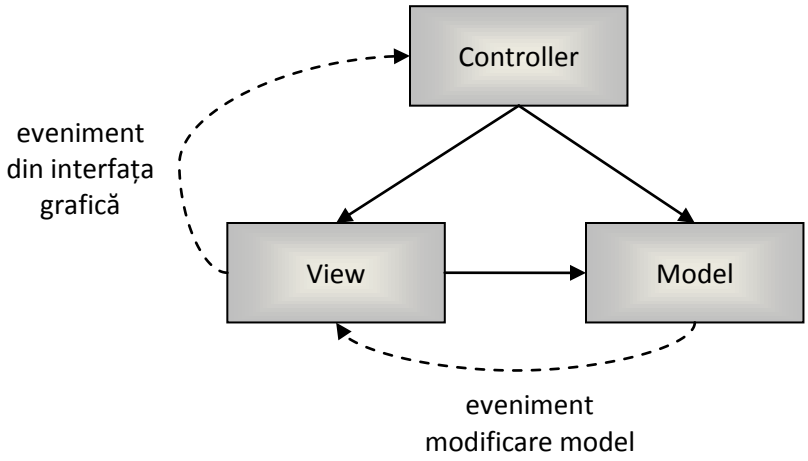
11.1 Model-View-Controller

Model-View-Controller (MVC) este un model arhitectural care separă funcționalitatea specifică domeniului pentru care este dezvoltat sistemul software de interfața grafică a aplicației, permițând dezvoltarea, întreținerea și testare separată a celor două părți. În Figură1 este descris grafic modelul arhitectural MVC. Acest model împarte un sistem software în trei părți, și anume: controller, view și model.

Modelul gestionează datele sistemului software, răspunde la interogări referitoare la stare (de obicei solicitate de View) și realizează operații de modificare a datelor (de obicei invocate de controller). În cazul sistemelor bazate pe evenimente, modelul notifică observatorii (de obicei view-urile), atunci când informația se modifică pentru ca aceștia să poată reacționa la aceste modificări.

View-ul redă modelul într-o formă care permite interacțiunea cu utilizatorul, de obicei prin intermediul elementelor de interfață cu utilizatorul. Pentru un singur model pot exista mai multe view-uri pentru a deservi diferite scopuri.

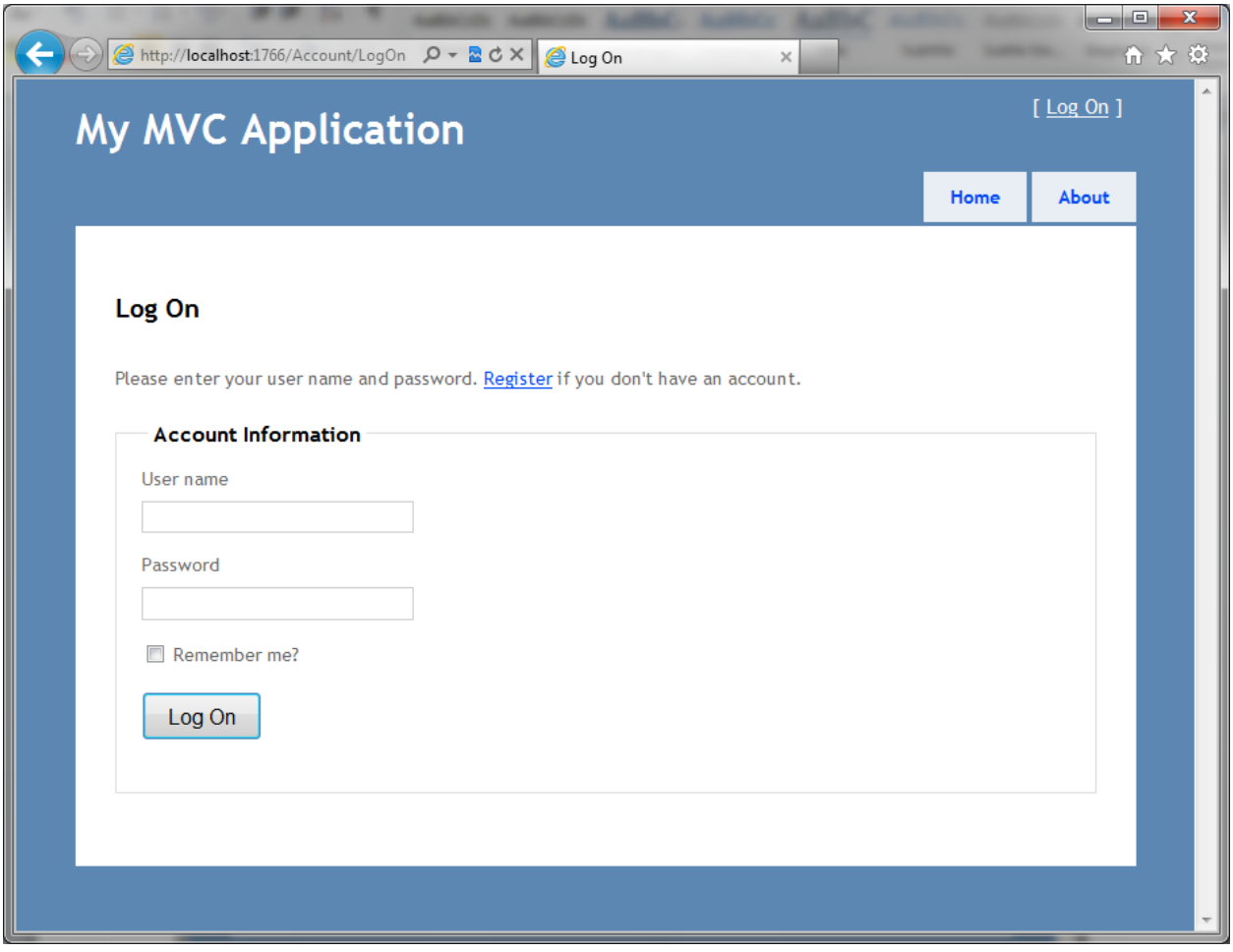
Controller-ul recepționează acțiunile utilizatorului și răspunde interogând modelul.



Figură1: Model-View-Controller

Un sistem software care implementează modelul arhitectural MVC poate fi văzut ca și o colecție de triplete (model, view, controller). Fiecare triplet este responsabil de un anumit element al interfeței grafice. Acest model arhitectural este întâlnit atât în sisteme software de tip aplicație desktop cât și în cele de tip aplicație web. Sistemul Swing de interfețe grafice este o tehnologie Java care modelează aproape toate componentele interfeței grafice ca și sisteme MVC individuale. Ca exemplu de tehnologie

web care folosește modelul arhitectural MVC se poate considera ASP.NET MVC. În cazul ASP.NET MVC modelul poate fi implementat prin clase .Net care modelează structurile de date (de ex.: se poate folosi Microsoft Entity Framework), controller-ul trebuie derivat dintr-o clasă specială (Controller), iar view-urile sunt implementate sub forma unor fișiere care conțin un amestec de cod C# și HTML (Razor view engine). În continuare este prezentat un exemplu de pagină web implementată în ASP.NET MVC (Figură2). Controller pentru această pagină este prezentat în Cod 1. Controller-ul conține metoda LogOn supraîncărcată: varianta fără parametrii este invocată atunci când un client face o cerere de tip GET (utilizatorul a scris în bara de adrese adresa pentru pagina de LogOn), iar varianta cu doi parametrii (model, returnUrl) este invocată atunci când un client trimite o cerere de tip POST (utilizatorul a dat click pe butonul de „Log On"). Testarea controller-ului se poate face foarte ușor creând un proiect de test și invocând metodele din controller. Implementarea view-ului, utilizând engin-ul Razor, este prezentat în Cod 2. View-ul nu face alt ceva decât să permită vizualizarea și editarea informației din model. Model din spatele paginii este prezentat în Cod 3, acesta conține proprietăți pentru fiecare câmp de pe ecran, în plus proprietățile sunt adnotate cu atribute care descriu validările ce trebuie să se execute asupra acelor câmpuri.



Figură2: Pagină ASP.NET implementată folosind ASP.NET MVC

```
publicclassAccountController : Controller
{

// Method invoked when a client performs a GET operation on
// the url http://<site>/Account/LogOn
```

```
publicActionResult LogOn()
        {
return View();//returning view will redirect to the LogOn page, action performed automatically
                        //by the MVC system
        }

//Method Invoked when user dose a POST back on the LogOn page, e.g., hits „Log On” button
// POST: /Account/LogOn
        [HttpPost]
publicActionResult LogOn(LogOnModel model, string returnUrl)
        {
if (ModelState.IsValid)
            {
if (Membership.ValidateUser(model.UserName, model.Password))
                {
FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 && returnUrl.StartsWith("/")
&& !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\"))
                    {
                        //redirect to the specified URL
return Redirect(returnUrl);
                    }
else
                    {
                        //go to home page
return RedirectToAction("Index", "Home");
                    }
                }
else
                {
                 //report login error
                    ModelState.AddModelError("", "The user name or password provided is incorrect.");
                }
            }

// If we got this far, something failed, redisplay form
return View(model);
        }
}
```

Cod 1: Exemplu de Controller ASP.NET MVC (sursa: Microsoft Visual Studio Template)

```
@model SampleMVC.Models.LogOnModel

@{
    ViewBag.Title = "Log On";
}

<h2>Log On</h2>
<p>
    Please enter your user name and password. @Html.ActionLink("Register", "Register") if you don't have
an account.
</p>

<scriptsrc="@Url.Content("~/Scripts/jquery.validate.min.js")"type="text/javascript"></script>
<scriptsrc="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"type="text/javascript"></script>

@Html.ValidationSummary(true, "Login was unsuccessful. Please correct the errors and try again.")

@using (Html.BeginForm()) {
<div>
<fieldset>
<legend>Account Information</legend>

<divclass="editor-label">
@Html.LabelFor(m => m.UserName)
```

```
</div>
<divclass="editor-field">
@Html.TextBoxFor(m => m.UserName)
@Html.ValidationMessageFor(m => m.UserName)
</div>

<divclass="editor-label">
@Html.LabelFor(m => m.Password)
</div>
<divclass="editor-field">
@Html.PasswordFor(m => m.Password)
@Html.ValidationMessageFor(m => m.Password)
</div>

<divclass="editor-label">
@Html.CheckBoxFor(m => m.RememberMe)
@Html.LabelFor(m => m.RememberMe)
</div>

<p>
<inputtype="submit"value="Log On"/>
</p>
</fieldset>
</div>
}
```

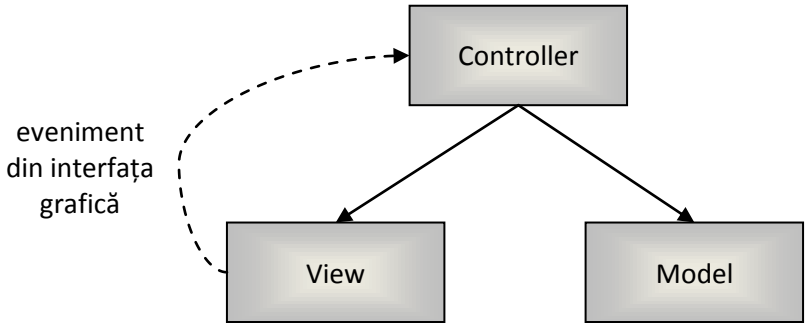
Cod 2: Exemplu de view ASP.NET MVC (sursa: Microsoft Visual Studio Template)

```
publicclassLogOnModel
    {
        [Required]
        [Display(Name = "User name")]
publicstring UserName { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
publicstring Password { get; set; }

        [Display(Name = "Remember me?")]
publicbool RememberMe { get; set; }
    }
```

Cod 3: Exemplu de model ASP.NET MVC (sursa: Microsoft Visual Studio Template)

11.2 Pasive View

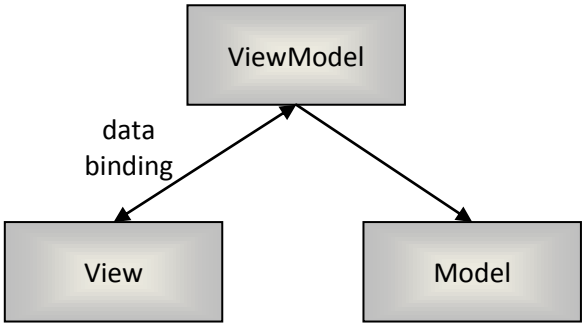
Modelul arhitectural Pasive-View (Figură3) este derivat din modelul Model-View-Controller, diferența majoră constă în faptul că view-ul este folosit doar pentru prezentarea informației, în rest el fiind pasiv, iar toate cererile utilizatorului sunt preluate de către controller. În plus view-ul nu se mai actualizează automat direct din model, astfel nu mai există o dependență între model și view. Principalul motiv pentru care se folosește modelul Pasive-View este posibilitatea de a implementa unit testing. Astfel având în vedere că view-ul este pasiv este suficient să se testeze doar controler-ul fără însă a fi nevoie de interacțiunea cu interfața grafică, ceea ce înseamnă că testele pot fi automatizate cu ușurință.



Figură3: Pasive-View

11.3 Model-View-ViewModel

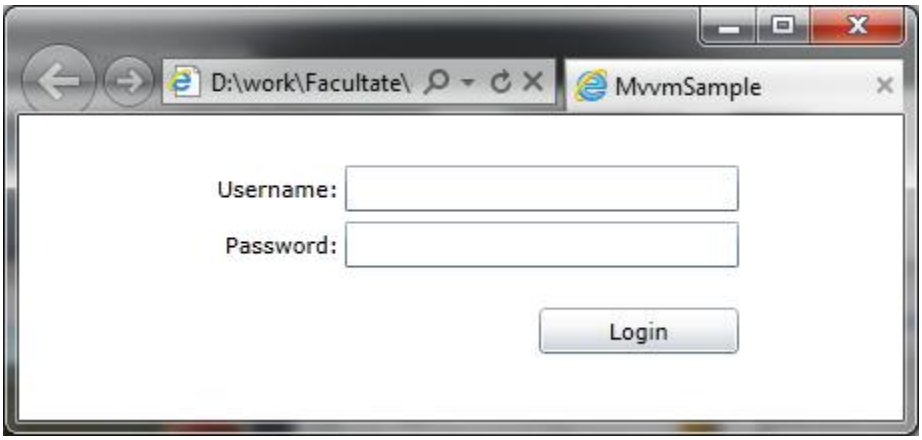
Model-View-ViewModel (MVVM) este un alt model arhitectural derivat din MVC. La fel ca și în cazul modelului Pasive-View nu există o dependență între view și model, dar spre deosebire de acesta view-ul nu este pasiv ci poate actualiza controller-ul care în acest caz este reprezentat de ViewModel. Cu toate acestea din punct de vedere al testării automate doar ViewModel-ul trebuie testat, întrucât comunicarea între View și ViewModel se face prin data binding, care cel puțin teoretic nu poate conține erori de logică. Modelul arhitectural Model-View-ViewModel este un model specific platformelor .Net: Windows Presentation Foundation și Silverlight. Pentru a ușura dezvoltarea aplicațiilor MVVM au fost dezvoltate o serie de toolkit-uri care facilitează implementarea acestui model arhitectural. Unul din cele mai cunoscute astfel de toolkit-uri este MVVM Light Toolkit [2].



Figură4: Model-View-ViewModel

În Figură5 este prezentată o pagină Silverlight care a fost implementată utilizând modelul arhitectural MVVM; pentru implementarea modelului MVVM s-a folosit toolkit-ul Mvvmlight. ViewModel-ul este prezentat în Cod 4. ViewModel-ul este reprezentat de o clasă derivată din clasa ViewModelBase. Pentru fiecare control din View, ViewModel-ul conține câte o proprietate, pe care se face binding în View. Astfel pentru exemplul discutat există trei proprietăți: UserName, Passowrd și LoginCommand. Proprietățile UserName și Password sunt de tipul text, în plus se poate observa că setarea unei valori în aceste proprietăți se face prin intermediul funcției Set care pe lângă setarea valori va genera un eveniment care va notifica View-ul că proprietatea s-a modificat. Proprietatea LoginCommand este mai specială pentru că ea corespunde unui buton (butonul Login), astfel această proprietate este de tipul RelayCommand și

permite asocierea unei funcții care să fie invocată atunci când butonul Login este apăsat. ViewModel-ul conține și o referință la model (_dataService) prin intermediul căreia poate interoga și modifica modelul (atunci când este apăsat butonul Login se invocă metoda Authenticate din model). Întrucât ViewModel-ul nu conține o referință la View pentru a notifica View-ul să navigheze la altă pagină sau să afișeze un mesaj se folosește sistemul de mesaje disponibil în Mvvmlight. În exemplul din Cod 4 ViewModel-ul trimite un mesaj View-ului atunci când are loc autentificare (`MessengerInstance.Send(autResult)`).



Figură5: Aplicație Silverlight pentru ilustrarea modelului arhitectural MVVM

```
///<summary>Class that implements view model for the login view.</summary>
publicclassLoginViewModel : ViewModelBase
{
publicconststring UserNamePropertyName = "UserName";
publicconststring PasswordPropertyName = "Password";

privatestring _userName = "";
privatestring _password = "";
privateRelayCommand _loginCommand;
///<summary> Reference to the service that can be used to access the model. </summary>
privateIDataService _dataService;

///<summary>
/// Sets and gets the UserName property.
/// Changes to that property's value raise the PropertyChanged event.
///</summary>
publicstring UserName
    {
get{return _userName;}
set{Set(UserNamePropertyName, ref _userName, value);}
    }

///<summary>
/// Sets and gets the Password property.
/// Changes to that property's value raise the PropertyChanged event.
///</summary>
publicstring Password
    {
get{return _password;}
set{Set(PasswordPropertyName, ref _password, value);}
    }


///<summary>
/// Gets the LoginCommand. This command is invoked when user clicks login.
///</summary>
publicRelayCommand LoginCommand
    {
```

```
get
        {
return _loginCommand
                ?? (_loginCommand = newRelayCommand(
                () =>
                {
                    _dataService.Authenticate(UserName, Password, (autResult, ex) => {
                        MessengerInstance.Send(autResult);
                    });
                }));
        }
    }

///<summary>Initializes a new instance of the LoginModel class.</summary>
public LoginViewModel(IDataService dataService)
    {
        _dataService = dataService;
    }
}
```

Cod 4: Exemplu de ViewModel pentru o aplicație Silverlight.

Pentru a facilita accesul la ViewModel-uri în View-uri se folosește o clasă denumită locator (Cod 5), care va conține referințe statice la toate modelele din aplicație, în plus tot această clasă este responsabilă cu instanțierea modelului și a ViewModel-urilor.

```
///<summary>
/// This class contains static references to all the view models in the
/// application and provides an entry point for the bindings.
///<para>
/// Use the <strong>mvvmlocatorproperty</strong> snippet to add ViewModels
/// to this locator.
///</para>
///<para>
/// See http://www.galasoft.ch/mvvm/getstarted
///</para>
///</summary>
publicclassViewModelLocator
{
static ViewModelLocator()
    {
ServiceLocator.SetLocatorProvider(() =>SimpleIoc.Default);

if (ViewModelBase.IsInDesignModeStatic)
        {
SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
        }
else
        {
SimpleIoc.Default.Register<IDataService, DataService>();
        }

SimpleIoc.Default.Register<LoginViewModel>();
    }

///<summary>
/// Gets the Login property.
///</summary>
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Performance",
"CA1822:MarkMembersAsStatic",
        Justification = "This non-static member is needed for data binding purposes.")]
publicLoginViewModel Login
    {
get
        {
returnServiceLocator.Current.GetInstance<LoginViewModel>();
        }
```

```
    }
}
```

Cod 5: Clasa locator folosită pentru a face legăturaîntre ViewModel și View

View-ul este prezentat în Cod 6 și Cod 7. Legătura dintre ViewModel și View se face exclusiv prin data binding. Astfel proprietatea DataContext este legată la proprietatea Login din clasa locator, care reprezintă o instanță a ViewModel-ului. Fiecare control este legat de câte o proprietate din ViewModel: TextBox-ul pentru introducerea numelui utilizator este legat de proprietatea UserName, TextBox-ul pentru introducerea parolei este legat de proprietatea Password, iar butonul Login este legat de comanda LoginCommand. În clasa din spatele View-ului există foarte puțin cod, doar se interceptează mesajele primite din ViewModel și se afișează mesaje pentru utilizator.

```
<UserControl x:Class="MvvmSample.LoginView"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 mc:Ignorable="d"
 DataContext="{Binding Login, Source={StaticResource Locator}}">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="25"></RowDefinition>
<RowDefinition Height="23"></RowDefinition>
<RowDefinition Height="5"></RowDefinition>
<RowDefinition Height="23"></RowDefinition>
<RowDefinition Height="20"></RowDefinition>
<RowDefinition Height="23"></RowDefinition>
<RowDefinition Height="*"></RowDefinition>
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
<ColumnDefinition Width="10"></ColumnDefinition>
<ColumnDefinition Width="150"></ColumnDefinition>
<ColumnDefinition Width="200"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>

<TextBlock Grid.Column="1" Grid.Row="1"
 HorizontalAlignment="Right"
 VerticalAlignment="Center" >
            Username:
</TextBlock>
<TextBox Grid.Column="2" Grid.Row="1" Margin="3,0,0,0"
 Text="{Binding Path=UserName, Mode=TwoWay}"
 />
<TextBlock Grid.Column="1" Grid.Row="3"
 HorizontalAlignment="Right"
 VerticalAlignment="Center" >
            Password:
</TextBlock>
<PasswordBox Grid.Column="2" Grid.Row="3" Margin="3,0,0,0"
 Password="{Binding Path=Password, Mode=TwoWay}"
 />
<Button Content="Login" Grid.Column="2" Grid.Row="5"
 Width="100" HorizontalAlignment="Right"
 Command="{Binding Path=LoginCommand}"/>

</Grid>
</UserControl>
```

Cod 6: Exemplu de View pentru o aplicație Silverlight

```
///<summary>
/// Description for LoginView.
///</summary>
```

```
publicpartialclassLoginView : UserControl
{
///<summary>
/// Initializes a new instance of the LoginView class.
///</summary>
public LoginView()
    {
        InitializeComponent();

Messenger.Default.Register<LoginDataItem>(this, (autResult) => {
if (autResult.LoginSuccessul) MessageBox.Show("User successfully authenticated.");
elseMessageBox.Show("User authentication faild.");
        });
    }
}
```

Cod 7: Exemplu de code-behind pentru o aplicație Silverlight

Pentru reprezentarea modelului (Cod 8) s-a definit mai întâi o interfață (IDataService) pentru a se putea comuta cu ușurință între diferite tipuri de modele. Fiind vorba de o aplicație Silverlight operațiile de interogate și modificare a modelului de date sunt operații asincrone pentru a evita blocarea aplicației atunci când se așteaptă un răspuns de la serviciul de date.

```
publicinterfaceIDataService
{
void Authenticate(string userName, string password, Action<LoginDataItem, Exception> callback);
}

publicclassDataService : IDataService
{
publicvoid Authenticate(string userName, string password, Action<LoginDataItem, Exception> callback)
    {
if (userName.ToLower().Equals("admin") && password.Equals("admin"))
        {
            callback(newLoginDataItem(true), null);
        }
else
        {
            callback(newLoginDataItem(false), null);
        }
    }
}

publicclassLoginDataItem
{
public LoginDataItem(bool loginSuccessful)
    {
        LoginSuccessul = loginSuccessful;
    }
publicbool LoginSuccessul { get; set; }
}
```

Cod 8: Exemplu de model pentru o aplicație Silverlight

Bibliografie

[1] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.
[2]http://mvvmlight.codeplex.com/
[3] http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
[4] http://msdn.microsoft.com/en-us/library/ff647543.aspx
[5] http://codebetter.com/jeremymiller/2007/05/31/build-your-own-cab-part-4-the-passive-view/
[6] http://www.asp.net/mvc

[7] http://msdn.microsoft.com/en-us/library/dd381412(VS.98).aspx
[8] http://weblogs.asp.net/scottgu/archive/2010/07/02/introducing-razor.aspx