STRUCTURA MVVM A APLICATIILOR WPF

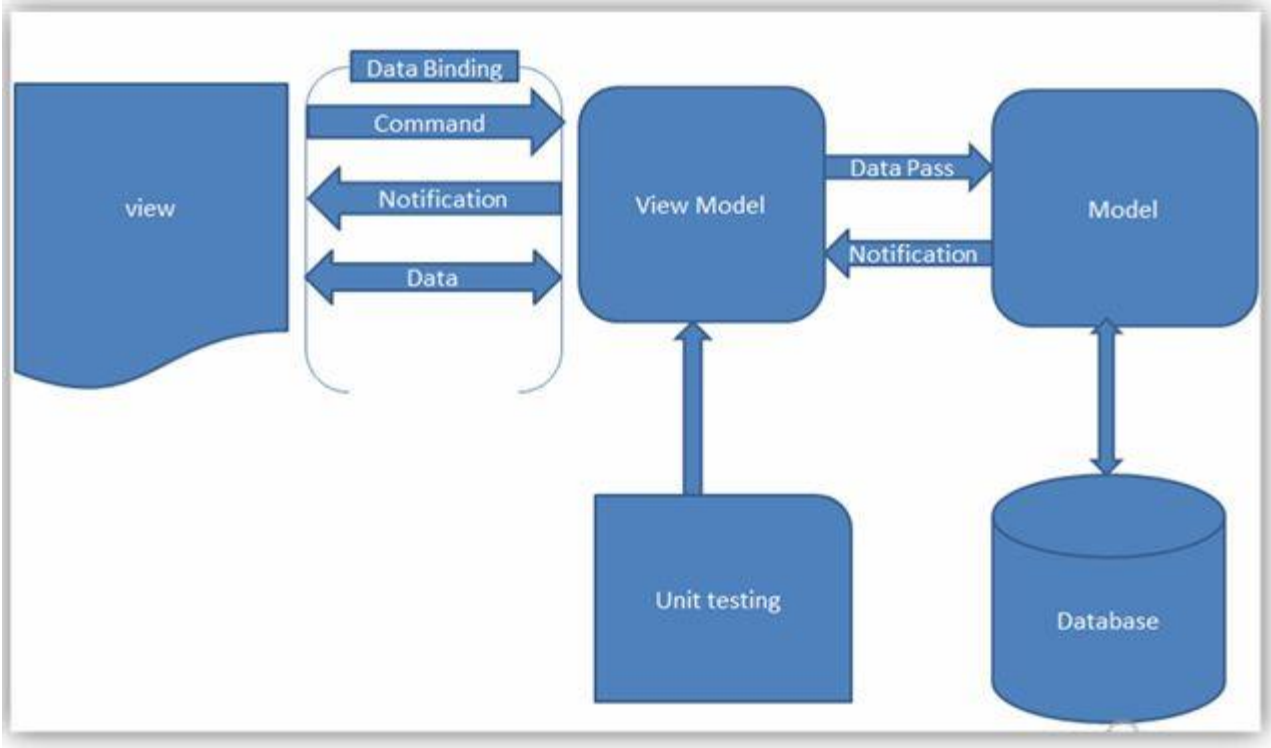
MVVM este un “Design Pattern” bazat pe interfata grafica. Un design pattern (sablon de proiectare / sablon arhitectural) reprezinta de fapt o serie de instructiuni orientative care ne ghideaza spre o realizare mai eficienta a unei aplicatii. Obiectivul principal al MVVM este acela de a oferi o interfata grafica bogata, capacitate sporita de testare, cod reutilizabil in foarte mare masura si mecanisme complexe de asociere a datelor (data binding). Acesta ajuta la o mai buna separare a nivelului de logica a aplicatiei de nivelul de prezentare, fara a avea comunicare directa intre ele. Astfel, designerul si dezvoltatorul pot lucra simultan, fara a se intrerupe sau incurca unul pe celalalt. Programatorul nu mai trebuie sa stie cum functioneaza designul, iar designerul nu mai trebuie sa stie cum functioneaza codul. De altfel, principiul separararii atributiilor (separation of concern principle) este unul valabil in cazul tuturor sabloanelor de proiectare a aplicatiilor.

Acest sablon presupune de fapt **separarea claselor non vizuale** - ***cele care incapsuleaza partea de logica a aplicatiei (Business Logic Layer), partea de acces la date (Data Layer), precum si relatiile dintre acestea*** - **de clasele vizuale,** ***cele care afiseaza elemente grafice pe ecran.*** Aceasta separare face codul mult mai usor de intretinut. In primul rand, este prevenita tendinta de a face clasele de interfata grafica (de UI) sa fie **monolitice** - adica sa contina si controale de interfata grafica si logica a aplicatiei si relatii logice amestecate in aceeasi clasa. In al doilea rand, aceasta separare permite automatizarea testarii (contine module de testare automata) in loc sa se faca testarea manual, aceasta din urma fiind costisitoare si nesigura. Deasemenea, folosind aceasta structura, partea de View (interfata grafica) poate fi modificata fara a modifica codul C#, cel putin cat timp datele si operatiile raman aceleasi.

Astfel, separarea codului, testarea usoara, dar si mentenanta usoara sunt caracteristici principale ale sablonului de proiectare MVVM. Acesta este pretabil aplicatiilor WPF, Silverlight si Windows Phone.

O abordare din punctul de vedere al MVVM presupune **existenta a cat mai putin cod in spatele ferestrelor grafice** (fisierele xaml.cs cu un minimum de cod). Deasemenea se furnizeaza mecanisme puternice de Data Binding, de realizare a comenzilor, de validare a datelor etc.

Diagrama arhitecturii MVVM:



Model - responsabil cu reprezentarea datelor pentru a fi intelese usor de catre WPF. Acesta trebuie sa implementeze in mod convenabil proprietatile INotifyPropertyChanged si / sau INotifyCollectionChanged. Acesta reprezinta nivelul de Data Layer si / sau Business Logic Layer

ViewModel - reprezinta maniera in care dorim sa reprezentam modelul catre utilizator. Este o abstractizare a View-lui (mai exact, View-ul afiseaza proprietatile si comenzile publice ale View Model-ului) Acesta este un convertor de valori (value converter), ceea ce inseamna ca este responsabil cu reprezentarea (conversia) ***datelor modelului*** intr-o maniera usor de gestionat si reprezentat de catre ***view***. In acest sens, View Modelul este mai mult Model decat View si se ocupa de aproape toata logica afisarii datelor intr-un View. La acest nivel, arhitectura MVVM foloseste un "**binder**" care mediaza comunicarea dintre view si sursa de date. View Model-ul a fost descris si ca o reprezentare a starii datelor din Model.

View - reprezinta interfata cu utilizatorul (interfata grafica). Acest nivel contine toate elementele grafice ale ferestrei: controale grafice, controale utilizator, fisiere, stiluri si teme; defineste structura, aspectul si modul de aparitie a ceea ce utilizatorul vede pe ecran.

TRANSFORMAREA EVENIMENTELOR IN COMENZI IN WPF

Am mai vorbit si in primul capitol despre implementarea comenzilor in WPF prin adaugarea in ViewModel a unor proprietati de tip ICommand pe care le vom asocia via "Binding" proprietatatii Command a unui control grafic.

Comenzile sunt alternativa MVVM a lucrului cu evenimente din aplicatiile clasice de tip Windows Forms. Cu toate ca se gasesc multe similitudini intre cele doua, ele reprezinta totusi lucruri cu totul diferite.

Comanda reprezinta actiunea care se executa. Sursa comenzii este obiectul care cheama comanda. Tinta comenzii este obiectul asupra caruia se executa comanda. Asocierea comenzii este obiectul care mapeaza comanda cu partea sa logica (cu codul).

Avantajele folosirii comenzilor:

1. Sunt mult mai usor de manevrat decat evenimentele. Comenzile pot fi asociate prin Binding - care este unul dintre cele mai puternice mecanisme ale MVVM - la proprietati de tip Command, iar handler-ele de evenimente nu pot fi asociate.

2. Daca dorim sa creem o actiune care sa se execute si la click pe un buton, dar si la click pe un MenuItem, atunci, in cazul comenzilor, vom crea o singura metoda care face actiunea si o singura proprietate de tip ICommand in ViewModel care sa apeleze acea metoda. In interfata grafica nu avem decat sa facem asocierea prin Binding la acea comanda. Ceea ce inseamna ca o comanda nu este dependenta de un control grafic. Daca lucram cu evenimente, va trebui sa creem cate un eveniment pentru fiecare control in parte, si in acel handler de evenimente vom apela metoda care face actiunea.

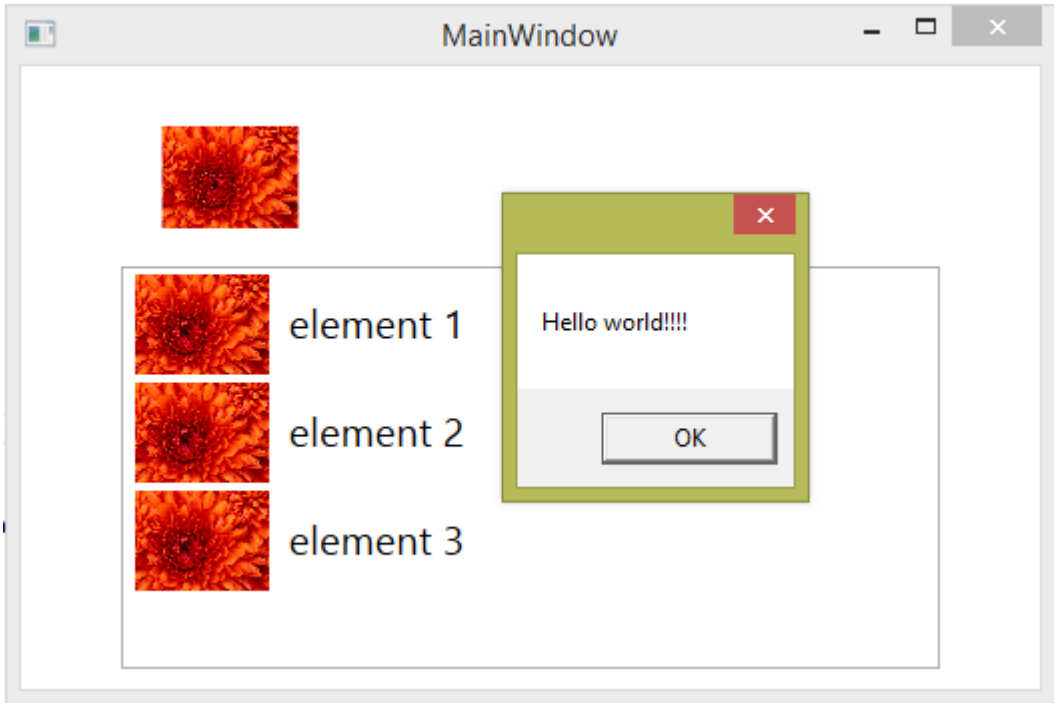
3. Comenzile ajuta la separarea codului. Actiunea care se executa va fi parte a Business Logic Layer-ului, proprietatea de tip ICommand se va gasi in ViewModel, iar Binding-ul cu proprietatea se face din XAML-ul ferestrei. Daca folosim evenimente, nu avem cum sa evitam scrierea de cod in spatele ferestrei in aceste cazuri deoarece handler-ul de evenimente se trateaza exclusiv acolo, caz in care vom muta in interfata grafica o parte din Business Logic Layer.

4. Comenzile mai vin si cu ceva beneficii pe care evenimentele nu le ofera. Datorita metodei bool CanExecute, un anumit control poate fi activat sau dezactivat daca se permite sau nu executia comenzii. Daca s-ar folosi evenimente, o astfel de facilitate ar trebui implementata de catre programator.

5. Daca evenimentul este cel care se declanseaza in momentul in care se petrece ceva cu interfata grafica (se apasa un buton, se selecteaza un item dintr-un ComboBox, se bifeaza un CheckBox etc.), comanda reprezinta modul de manevrare al evenimentului respectiv de catre modelul aplicatiei.

Vom considera in continuare un exemplu de aplicatie care utilizeaza comenzi. Atasarea unei comenzi pe un buton am vazut in capitolul anterior. Dar exista destul de putine controale grafice care sa detina o proprietate Command, iar aceasta proprietate se utilizeaza de obicei pentru evenimentul de click. Daca dorim sa se execute o actiune cand trecem cu mouse-ul peste buton sau cand dam click pe o imagine, atunci va trebui sa avem o abordare diferita. Exista mai multe modalitati de a rezolva aceasta situatie. Modalitatea prezentata se bazeaza pe EventTrigger.

Vom avea astfel o pagina cu o imagine si cu o lista cu imagini si text. La click pe oricare dintre imaginile din pagina dorim sa apara o fereastra de dialog cu mesajul “Hello World”, ca in imaginea de mai jos.



Vom dafini o clasa ViewModel care va reprezenta sursa de date a ferestrei. Clasa arata astfel:

```
class ViewModel
{
    public List<string> MyList { get; set; }
    public ViewModel()
    {
        MyList = new List<string>() { "element 1", "element 2", "element 3" };
    }

    private void DoSomething(object parameter)
    {
        MessageBox.Show("Hello world!!!!");
    }
    private ICommand myCommand;
    public ICommand MyCommand
    {
        get
        {
            if (myCommand == null)
                myCommand = new RelayCommand(DoSomething);
            return myCommand;
        }
    }
 }
```

Clasa RelayCommand pe care am folosit-o mai sus arata astfel:

```
class RelayCommand : ICommand
{
    private Action<object> commandTask;

    public RelayCommand(Action<object> workToDo)
    {
        commandTask = workToDo;
    }

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

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        commandTask(parameter);
    }
}
```

Cum obiectul grafic <Image> nu are o proprietate Command, am folosit un EventTrigger. Codul pentru imaginea de deasupra listei este urmatorul:

```
<Image Source="/WpfApplication1;component/Chrysanthemum.jpg" Margin="50, 30, 350, 230">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseDown">
            <i:InvokeCommandAction Command="{Binding MyCommand}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Image>
```

Am transformat astfel evenimentul MouseDown intr-o comanda. Deoarece obiectul de tip ViewModel reprezinta DataContext-ul ferestrei si acest obiect are proprietatea MyCommand, la proprietatea Command a lui **InvokeCommandAction** am asociat acea proprietate de tip ICommand, numita **MyCommand**.

Pentru a putea folosi **Interaction.Trigger** va trebui adaugat la referintele aplicatiei (din fereastra SolutionExplorer a lui VisualStudio) System.Windows.Interactivity, iar la proprietatile lui <Window>, in XAML, se va adauga namespace-ul urmator:

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.
Interactivity"
```

Acum vom observa ca in urma executiei aplicatiei, la click pe imagine va aparea fereastra de dialog.

Codul XAML pentru lista cu imagini si text este urmatorul:

```
<ListBox Name="ItemsList" ItemsSource="{Binding MyList}" Margin="50,100,50,10">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <WrapPanel>
            <Image Source="/WpfApplication1;component/Chrysanthemum.jpg" Height="50">
                <i:Interaction.Triggers>
                        <i:EventTrigger EventName="MouseDown">
                        <i:InvokeCommandAction Command="{Binding Source={StaticResource theResource},
                                                                 Path=MyCommand}"/>
                    </i:EventTrigger>
                </i:Interaction.Triggers>
            </Image>
                <TextBlock FontSize="20" Text="{Binding}" Margin="10"/>
            </WrapPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

ListBox-ului i-am definit ca si ItemTemplate un WrapPanel cu un Image si un TextBlock. Am folosit tot EventTrigger pentru imagine, doar ca nu am mai scris Command={Binding Path=MyCommand}. Imaginea face parte din template-ul pentru elementele din ListBox.

ItemsSource pentru ListBox este MyList, care este un List<string>, definit in clasa ViewModel. Astfel, sursa de date pentru un element al ListBox-ului este un string din MyList. Aceasta inseamna ca atunci cand scriem {Binding} in interiorul lui ItemTemplate ne vom referi la un string si nu la un obiect de tip ViewModel. Deoarece proprietatea MyCommand este a lui ViewModel, atunci nu vom putea scrie Command={Binding Path=MyCommand}. Pentru a reusi totusi sa avem acces la aceasta proprietate, trebuie sa ajungem inapoi la ViewModel din <Image>. Am definit astfel pentru fereastra curenta o resursa numita **theResource**, in care am pastrat un obiect de tip ViewModel, pe care il vom invoca in momentul in care se defineste comanda pentru imagine. Prin scrierea **Command = {Binding Source = {StaticResource theResource}, Path = MyCommand}**, vom schimba sursa de date dintr-un string, in cazul de fata, intr-un obiect de tipul ViewModel, prin apelarea resursei **theResource**. Astfel vom avea acces si la proprietatea MyCommand din ViewModel.

Resursa am definit-o astfel:

```
<Window.Resources>

    <ctx:ViewModel x:Key="theResource" />

</Window.Resources>
```

Acum vom observa ca in urma executiei codului, la click pe imaginile din lista, va aparea fereastra de dialog.

FOLOSIREA VALIDATORILOR IN WPF

Orice aplicatie care detine o interfata grafica trebuie sa aiba implementat un sistem de validare a datelor de intrare. In WPF validarea sa poate face in mai multe moduri: prin implementarea propriilor "**ValidationRules**", implementarea interfetei **IDataErrorInfo**, implementarea interfetei **INotifyErrorDataError** (care a fost introdusa in .NetFramework 4.5), prin adaugarea de "**data annotations**".

Daca un utilizator introduce intr-un TextBox spre exemplu date care nu pot fi convertite automat de catre CLR in tipul de data al proprietatii care este asociata textului TextBox-ului, atunci in mod implicit vom vedea ca eroarea este semnalata printr-un chenar rosu.

Astfel, presupunem ca avem o aplicatie care doreste sa calculeze suma a doua numere intregi, care are doua TextBox-uri pe interfata grafica, iar in ViewModel are 2 proprietati de tip int numite

Nr1 si Nr2. Textul celor doua TextBox-uri va fi asociat prin Binding celor doua proprietati: Nr1 si Nr2. Daca utilizatorul introduce in TextBox litere, atunci va aparea un border rosu care va semnala eroarea (deoarece literele introduse nu pot fi convertite automat la int).

Mesajul care descrie eroarea este stocat de catre proprietatea **ErrorContent** a obiectului System.Windows.Controls.**ValidationError** care este adaugat in momentul rularii in colectia Validation.Errors a TextBox-ului in care a aparut eroarea. Atunci cand proptietatea Validation.Errors are obiecte de tip ValidationError in ea, o alta proprietate numita Validation.HasErrors va primi valoarea “true”.

Crearea propriului sablon de eroare

Pentru a putea vedea mesajele de eroare pe interfata grafica va trebui sa inlocuim sablonul implicit de reprezentare al erorii (acela care seteaza border rosu elementului) si sa definim propriul sablon prin setarea proprietatii **Validation.ErrorTemplate**.

```
<TextBox FontSize="20" Width="150" Text="{Binding Nr1,
                    UpdateSourceTrigger=PropertyChanged}">
    <Validation.ErrorTemplate>
        <ControlTemplate>
            <StackPanel>
                <AdornedElementPlaceholder Name="TextBox"/>
                <TextBlock Text="Eroarea mea" Foreground="Red"/>
            </StackPanel>
        </ControlTemplate>
    </Validation.ErrorTemplate>
</TextBox>
```

Proprietatea Validation.ErrorTemplate se va gasi in stratul (nivelul) care vine in completarea TextBox-ului, mesajul fiind in mod implicit exact peste controlul grafic. Acest sablon de afisare a erorii este de fapt un decorator al TextBox-ului. Daca dorim sa introducem un spatiu si sa afisam mesajul sub controlul grafic, atunci trebuie sa introducem un element “**AdornedElementPlaceholder**”, asa ca in exemplul de mai sus. Mesajul de eroare din TextBlock

va aparea astfel exact sub TextBox, dar avand in vedere ca modul de vizualizare este deasupra (OnTop), daca se gaseste un element grafic imediat sub TextBox (fara spatiu intre ele), atunci mesajul de eroare va aparea peste acel element grafic.

Reguli de validare folosind clasa ValidationRule

Erorile se pot personliza prin implementarea unei reguli de validare si asocierea acesteia prin "Binding". O regula de validare personalizata este o clasa care deriva din clasa abstracta System.Windows.Controls.ValidationRule si care implementeaza metoda Validate din aceasta clasa. ValidationRule are si o proprietate numita ValidationStep care controleaza momentul in care motorul de "Binding" executa metoda Validate. ValidationStep este de tip enumerare si are urmatoarele optiuni:

1. RawProposedValue - regula de validare se executa inainte de conversia valorii validate (aceasta este valoarea implicita)

2. ConvertedProposedValue - regula de validare se executa dupa conversia valorii validate, dar inainte de apelul seter-ului proprietatii sursa care se valideaza.

3. UpdatedValue - regula de validare se executa dupa ce a fost modificata proprietatea sursa care se valideaza

4. CommittedValue - regula de validare se executa dupa ce valoarea a fost actualizata lasursa.

Mai jos este prezentat un exemplu in care se implementeaza o regula personalizata de eroare care valideaza transformarea unei valori de tip string in int. Daca valoarea nu poate fi transformata, se va seta proprietatea ErrorContent a obiectului ValidationError in colectia Validation.Errors. Proprietatea ValidationStep trebuie sa aiba valoarea RawProposedValue pentru ca regula sa fie aplicata inainte de a se face conversia valorii.

In clasa ViewModel se adauga proprietatea Age

```
private int age;
public int Age
{
    get { return age; }
    set
    {
        age = value;
    }
```

```
    }
```

Codul din interfata grafica este urmatorul:

```
<Window x:Class="WpfApplication1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:abc="clr-namespace:WpfApplication1"
    Title="MainWindow" Height="350" Width="525">
 <Window.DataContext>
     <abc:ViewModel />
 </Window.DataContext>
   <StackPanel Margin="50">
      <TextBox>
          <TextBox.Text>
              <Binding Path="Age" UpdateSourceTrigger="PropertyChanged">
                  <Binding.ValidationRules>
                      <abc:StringToIntValidationRule
ValidationStep="RawProposedValue"/>
                  </Binding.ValidationRules>
              </Binding>
          </TextBox.Text>
      </TextBox>
    </StackPanel>
</Window>
```

Codul clasei care se ocupa de validare:

```
public class StringToIntValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
    {
        int i;
        if (int.TryParse(value.ToString(), out i))
            return new ValidationResult(true, null);
        return new ValidationResult(false, "Please enter a valid integer value.");
    }
}
```

In acest caz, daca sciem in TextBox litere sau alte caractere care nu pot fi transformate in int, atunci va aparea un chenar rosu pe TextBox (stilizarea implicita a erorii).

Procesul de validare

Daca metoda Validate din clasa ValidationRule returneaza un ValidationResult invalid, atunci procedura de validare se va opri, iar daca valoarea returnata de proprietatea IsValid a obiectului este true, atunci procedura va continua cu pasul urmator.

1. Metoda Validate a oricarui obiect de tipul ValidationRule care este asociat unui control grafic prin “Binding” si are proprietatea ValidationStep setata cu vloarea RawProposedValue se executa pana cand o regula de validare returneaza un ValidationResult invalid sau pana cand trec toate regulile de validare.

2. Daca este asociat un convertor la “Binding”, atunci se apeleaza metoda ConvertBack

3. Motorul de “Binding” incearca sa transforme valoarea returnata de metoda ConvertBack, presupunand ca exista un convertor asociat cu “Binding” sau cu proprietatea care se valideaza si care trebuie sa fie o proprietate de dependinta (DependencyProperty).

4. Se apeleaza seter-ul proprietatii care se valideaza

5. Metoda Validate a oricarui obiect de tipul ValidationRule care este asociat unui control grafic prin “Binding” si are proprietatea ValidationStep setata cu vloarea UpdatedValue este evaluata ca la pasul numarul 1

6. La fel si pentru ValidationStep setata cu vloarea CommittedValue

Inainte ca metoda Validate sa se execute la acest pas, se va sterge orice eroare care a fost adaugata proprietatii Validation.Errors de vre-o procedura de validare apelata anterior. Colectia Validation.Errors este golita si in cazul in care se obtine o valoare valida.