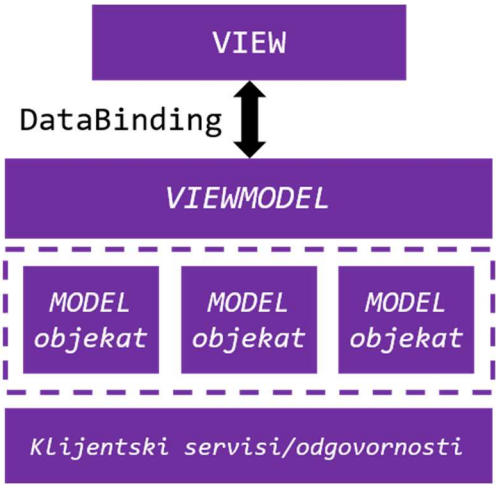
Napredni WPF: *Model View ViewModel Design Pattern* #1

Model View ViewModel predstavlja ***Design Pattern*** (projektni obrazac). Projektni obrasci predstavljaju višekratno rešenje za probleme koji se često dešavaju prilikom dizajna softverskih rešenja. ***Model View ViewModel (MVVM)*** nalazi primenu u **WPF** programiranju sa ciljem da omogući kreiranje održivih, testabilnih i proširivih aplikacija. Čine ga tri osnovne komponente:

- **Model:** uglavnom klase koje formiraju osnovne entitete sa kojima se radi
- **ViewModel:** veza između **Model**-a i **View**-a, funkcionalnost i izlaganje podataka ka interfejsu
- **View:** formatirani podaci (*DataBinding*)

Ovde se javlja koncept *Razdvojene prezentacije* – razdvajanje koda vezanog za izgled (dizajn interfejsa), logike aplikacije i podataka (klasa), što za posledicu ima da je UI klasa mnogo jednostavnija (*.xaml.cs fajlovi ne sadrže kod).

MVVM obrazac predstavlja modernu strukturu projektnog obrasca ***Model View Controller (MVC)***, sa istim ciljem – jasno razdvajanje logike domena i prezentacionog sloja (prikazano na **Slici 1**).



Slika 1. Izgled strukture MVVM aplikacije – Razdvojena prezentacija

Model je najjednostavniji deo obrasca za razumevanje. Kako mu ime kaže, predstavlja model podataka sa klijentske strane. On se ogleda u objektima sa property-jima i ponekim promenljivima koje čuvaju podatke u memoriji. Pozivaju događaj *PropertyChanged*, kojim se notifikuje ostatak aplikacije da se desila promena vrednosti određenog property-ja.

Iz tog razloga, klasa koja modeluje entitete mora da implementira interfejs **INotifyPropertyChanged**, koji sa sobom nosi događaj *PropertyChanged,* čija je implementacija navedena u listingu koda ispod.

```
public event PropertyChangedEventHandler PropertyChanged;

private void RaisePropertyChanged(string property)
{
    if(PropertyChanged!=null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(property));
    }
}
```

Listing 1. Implementacija interfejsa INotifyPropertyChanged

ViewModel je najvažniji deo MVVM aplikacije i on omogućava pristup i interakciju sa podacima koji će biti prikazani na **View**-u. Pored toga, omogućavaju navigacionu logiku, odnosno promenu aktivnog **View**-a. Implementira se kao C# klasa, koja sadrži podatke koji se prikazuju na interfejsu.

Podaci se dodeljuju u dinamičku kolekciju tipa **ObservableCollection**, koja omogućava notifikacije kada se elementi dodaju, brišu ili se sama kolekcija izmeni. Ako je potrebno, kolekcija se može u startu popuniti ako će se podaci povezati preko *DataBinding*-a. Primer ovako neke klase je dat u listingu ispod.

```
public ObservableCollection<Student> Students { get; set; }

public StudentViewModel()
{
     LoadStudents();
}

public void LoadStudents()
{
     ObservableCollection<Student> students = new ObservableCollection<Student>();

     students.Add(new Student { FirstName = "Petar", LastName = "Petrovic" });
     students.Add(new Student { FirstName = "Marko", LastName = "Markovic" });
     students.Add(new Student { FirstName = "Jovan", LastName = "Jovanovic" });

     Students = students;
}
```

Listing 2. Implementacija jedne ViewModel klase koja sadrži samo podatke za prikaz.

View definiše strukturu onoga što će korisnik videti na ekranu (dizajn interfejsa). Statičku strukturu predstavlja **XAML** hijerarhija kontrola i njihov raspored, dok se dinamička struktura ogleda u postojanju animacija i promene stanja.

U **WPF**-u, **View** se formira pomoću klasa **UserControl**. One se koriste za grupisanje **XAML** i C# koda u višestruko upotrebljivi kontejner, sa idejom da se isti dizajn, odnosno funkcionalnost mogu koristiti na više mesta u istoj ili u više različitih aplikacija. Ponaša se identično kao i svaki prozor, ima svoj par **.xaml* i **.xaml.cs* datoteka koje je čine. Ovako se omogućava da se jednostavije funkcionalnosti razdvoje i u vizuelnom rasporedu sa mogućnošću podele dizajna prozora prema manjim funkcionalnostima. **UserControl**-e će ovde imati ulogu dodatnih prozora, čiji će se sadržaj smenjivati na površini glavnog prozora, je po standardu **MVVM** aplikacija, u njima ne smeju da postoje dodatni prozori. Primer praznog objekta klase **UserControl** u **XAML** kodu je prikazan u listingu ispod.

```
<UserControl
x:Class="UserControl1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="300">
   <Grid>

   </Grid>
</UserControl>
```

Listing 3. Izgled praznog objekta klase UserControl u XAML kodu

Unutar **UserControl**-e koja bi služila za prikaz više objekata iz neke kolekcije može da se definiše neka kontrola koja se pomoću *DataBinding*-a povezuje sa kolekcijom objekata klase iz **Model**-a aplikacije. Osnovna kontrola koja ovo omogućava je **ItemsControl**, kojoj se jedino definiše izvor podataka, pomoću *DataBinding*-a. Definisanje kako će izgledati elementi koje prikazuje ova kontrola se postiže pomoću njenog svojstva ***ItemTemplate***. Njegov sadržaj je XAML definicija izgleda elementa. Kada je potrebno definisati vizuelnu strukturu podataka, koristi se klasa **DataTemplate**.

Kao njen sadržaj definišu kontrole potrebne za prikaz podataka (U slučaju klase **Student**, to će biti dva **TextBox**-a i jedan **TextBlock** - instance **TextBox** kontrole služe da se podaci mogu menjati, dok se **TextBlock** kontrola koristi za prikaz rezultata izmene). Svaka od ovih kontrola se preko *DataBidning*-a povezuje na vrednost property-ja klase za koje je namenjena.

Ako ima potrebe da se unutar datih kontrola menja vrednost koja je povezana, ***Mode*** *DataBinding*-a se postavlja na vrednost *TwoWay*. *DataBinding*-u se takođe može definisati svojstvo ***UpdateSourceTrigger*** koje utiče na to kada će se izvorna vrednost ažurirati. Tekstualnim podacima je to inicijalno kada se ukloni fokus sa datog polja (*LostFocus*), a može da se promeni na vredost *PropertyChanged*, što znači da će se vrednost ažurirati kako samom property-ju bude menjana vrednost. Ako se vrednost samo prikazuje, ***Mode*** će kao vrednost dobiti *OneWay*. Implementacija opisanog rešenja je data u listingu koda ispod.

```
<ItemsControl ItemsSource=”{Binding Path=Students}”>
   <ItemsControl.ItemTemplate>
      <DataTemplate>
         <StackPanel Orientation=”Horizontal”>
            <TextBox Text=”{Binding Path=FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}” Width=”100”/>
            <TextBox Text=”{Binding Path=LastName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}” Width=”100”/>
            <TextBlock Text=”{Binding Path=FullName, Mode=OneWay}”/>
          </StackPanel>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Listing 4. Definicija prikaza objekta klase Student kroz DataTemplate

Pošto se **View UserControl**-a i **ViewModel** klasa nalaze u različitim imenskim prostorima (*namespace*), a potrebno je da se povežu preko *DataBinding*-a, treba omogućiti njihovo spajanje na taj način. Pošto će se **View** generisati prvi (*ViewFirstConstruction*), postoje dva načina da se **XAML** i C# implementacija povežu (**ViewModel** se dodeljuje **View**-u kao ***DataContext***):

- Iz **XAML** implementacije, kako je prikazano u **Listingu 5**. Potrebno je u okviru **XAML** koda referencirati imenski prostor u kojem se nalazi **ViewModel** klasa, kako bi se ona mogla tu definisati.

```
xmlns:viewModel="clr-namespace:MVVM1.ViewModel"
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="300">
<UserControl.DataContext>
   <viewModel:StudentViewModel/>
</UserControl.DataContext>
```

Listing 5. Dodeljivanje ViewModel-a DataContext-u View-a u okviru XAML koda

- Iz C# pozadinskog koda **View**-a (**.xaml.cs*, jedini izuzetak, iz razloga ako **ViewModel** koji se povezuje sa konkretnim **View**-om treba da primi jedan ili više parametara u svom konstruktoru), prikazano u **Listingu 6**.

```
public StudentView()
{
    InitializeComponent();
    this.DataContext = new MVVM1.ViewModel.StudentViewModel();
}
```

Listing 6. Dodeljivanje ViewModel-a DataContext-u View-a unutar C# klase

Još jedan način spajanja **View**-a i **ViewModel**-a koji se može univerzalno koristiti nezavisno od toga koja je klasa u pitanju je *meta pattern* **ViewModelLocator**. Njegova implementacija zavisi od konvencije imenovanja klasa. Njegova svrha jeste da identifikuje **View** koji se prikazuje, potom pronađe **ViewModel** klasu koja odgovara tom **View**-u, potom da konstruiše sam **ViewModel** i na kraju da kao ***DataContext*** aktuelnom **View**-u dodeli **ViewModel** koji mu odgovara. Ova implementacija je prikazana u listingu koda ispod.

```
private static void AutoHookedUpViewModelChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
{
     if(DesignerProperties.GetIsInDesignMode(d))
     {
         return;
     }

     var viewType = d.GetType();
     string str = viewType.FullName;
     str = str.Replace(".Views", ".ViewModel");
     var viewTypeName = str;

     var viewModelTypeName = viewTypeName + "Model";
     var viewModelType = Type.GetType(viewModelTypeName);
     var viewModel = Activator.CreateInstance(viewModelType);

     ((FrameworkElement)d).DataContext = viewModel;
}
```

Listing 7. Implementacija spajanja View i ViewModel klasa kroz ViewModelLocator

Implementacija **ViewModelLocator** klase je realizovana tako da postoji jedan **DependencyProperty** čijom promenom vrednosti će se aktivirati spajanje **View**-a i **ViewModel**-a. Pošto se i **ViewModelLocator** razdvaja od ostatka klasa u okviru svog ličnog imenskog prostora, i njega će biti potrebno referencirati iz njegovog imenskog prostora u **XAML** kodu i izvršiti promenu datog property-ja kako bi se realizovala implementacija. Ovo je prikazano u listingu koda ispod.

```
xmlns:vml="clr-namespace:MVVM1.VML"
vml:ViewModelLocator.AutoHookedUpViewModel="True"
```

Listing 8. Instanciranje ViewModelLocator klase među imenskim prostorima View-a

Na kraju, pošto je **View** dizajn, odnosno **XAML** hijerarhija koja stoji sama za sebe, on se nigde neće prikazati, te je potrebno dodati ga na površinu glavnog prozora, nakon što se prvo referencira i njegov imenski prostor, kako bi na osnovu njega, moglo da se pristupi samoj View klasi i postaviti njenu instancu unutar glavnog prozora.

```
<views:StudentView x:Name="StudentViewControl"/>
```

Listing 9. Definicija View-a u okviru XAML koda glavnog prozora

Sve ove funkcionalnosti su iskorišćene da se kreira primer za sedmu nedelju vežbi koji sarži izlistane objekte klase Student koji su predstavljeni kao kombinacija kontrola TextBox, TextBox, TextBlock sa podacima o studentima, koji se mogu menjati u okviru instanci TextBox klase tako da se u sadržaju TextBlock kontrole vide promene, a sve je realizovano primenom MVVM projektnog obrasca.