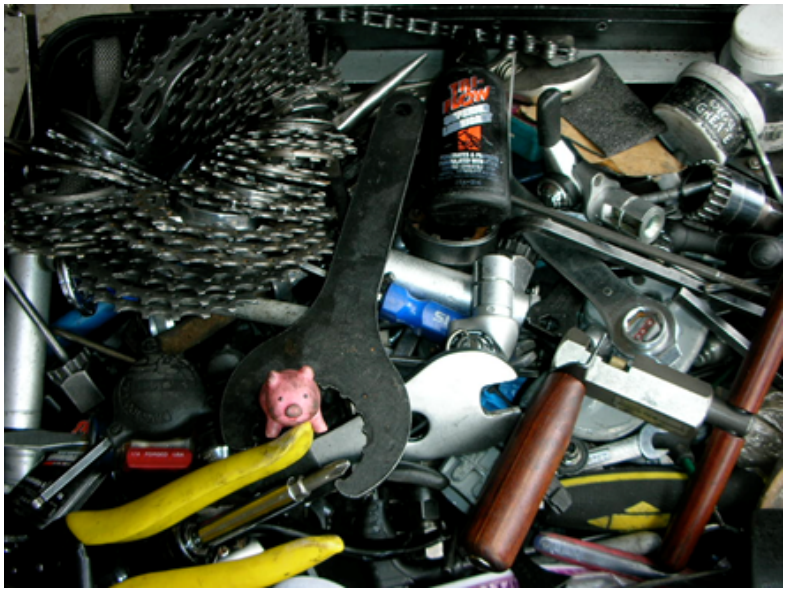
WPF alkalmazások architektúrája

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

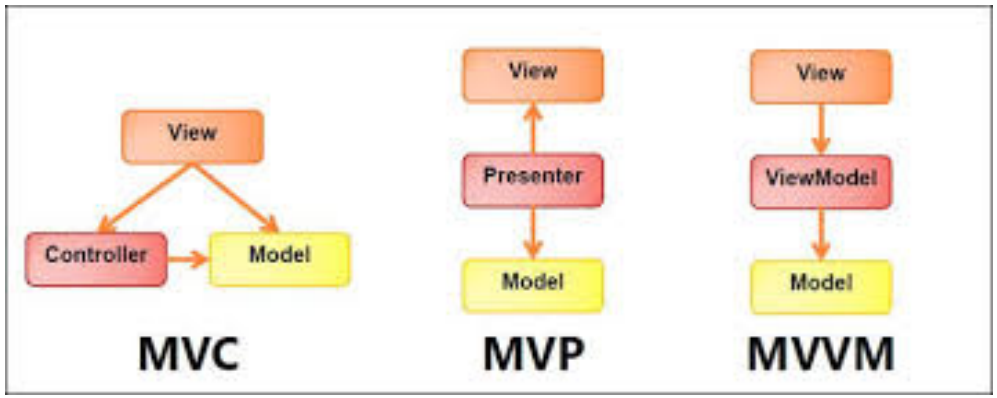
Rend a lelke mindennek

- strukturálatlanság, logikátlan elrendezés, stb. –> spagetti kód



- megoldás: MVC, MVP, **MVVM**

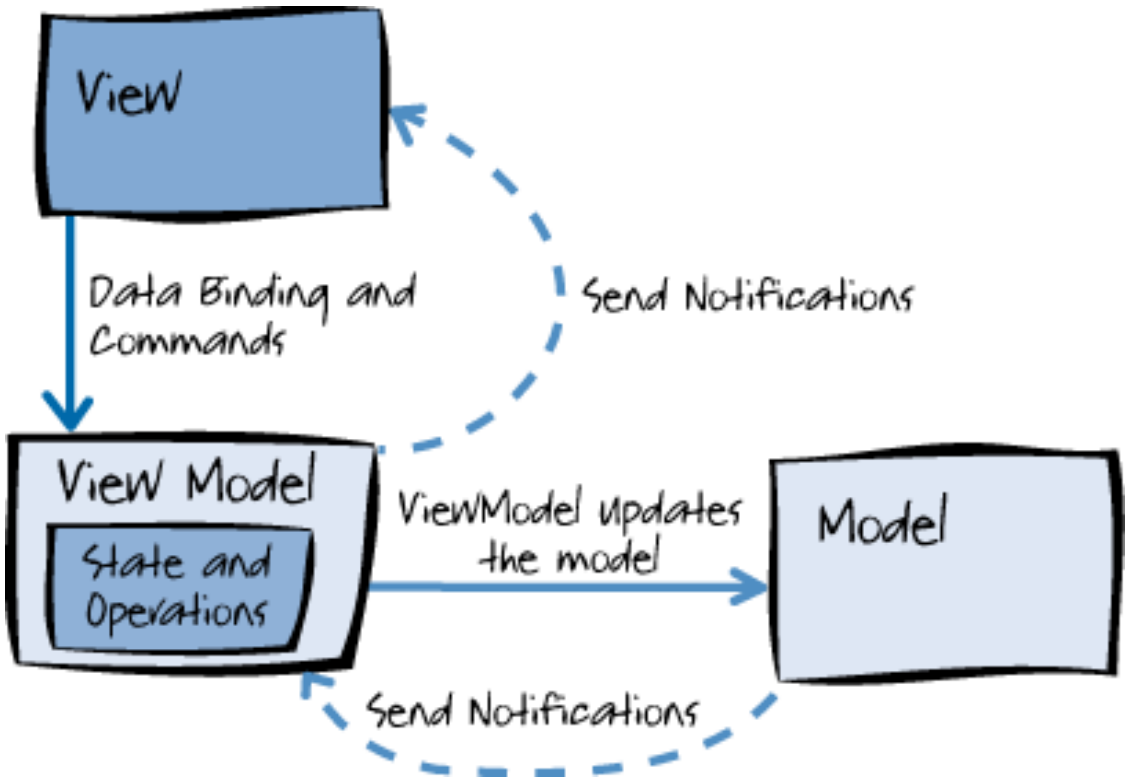
MVC vs. MVP vs. MVVM



MVC	**MVVM**
a Controller az alkalmazás belépési pontja	a View az alkalmazás belépési pontja
a Controller kötelező	a ViewModell opcionális
a View látja a Modellt	a View NEM látja a Modellt
a Viewnek van egy Modell példánya	
Smalltalk, ASP.NET MVC	Silverlight, WPF, HTML5 Knockout, AngularJS-el

MVVM - Modell/Nézet/NézetModell (Modell/View/ViewModel)

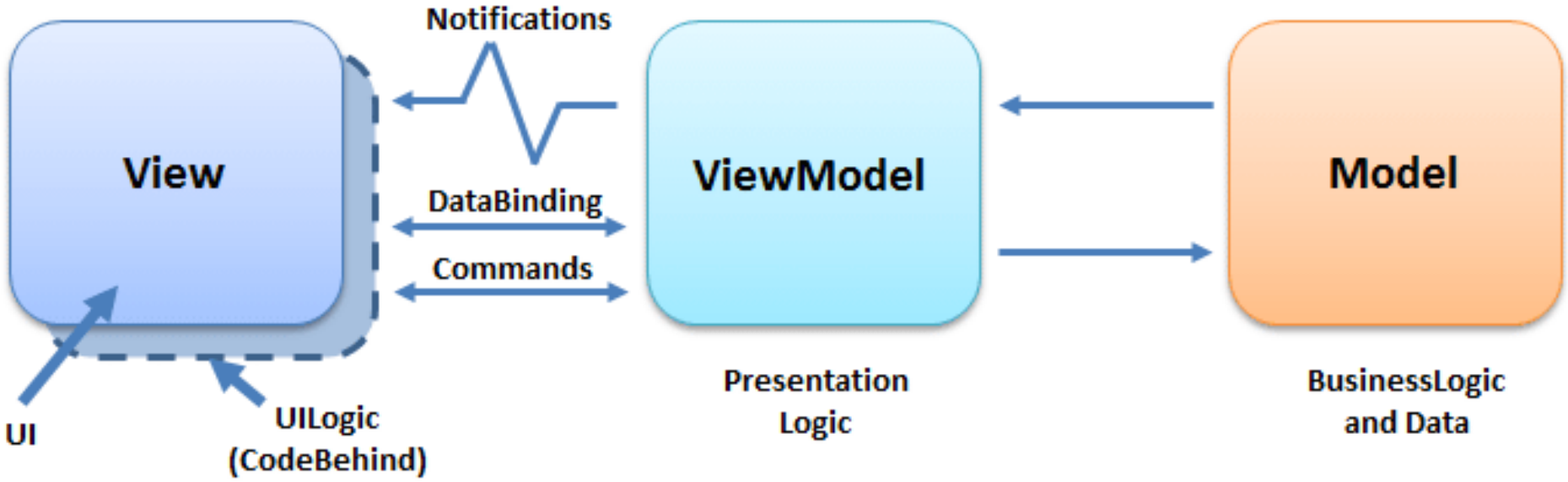
- az **MVVM** olyan tervezési minta, amelynek a célja, hogy teljes egészében elválassza a megjelenítést és a mögötte lévő tevékenységeket
  - egy közvetítő réteget (NézetModell-t) használ, amely a háttérkód feladatát veszi át



https://msdn.microsoft.com/en-us/library/hh848246.aspx

MVVM architektúra

- a **modell** tartalmazza az alkalmazás logikáját (algoritmusok, adatelérés), önálló, újrafelhasználható
- a **nézet** tartalmazza a felület vezérlőit (ablakok, vezérlők, …) és az erőforrásokat (animációk, stílusok, …)
- a **nézetmodell** lehetőséget ad a modell változásainak követésére és tevékenységek végrehajtására



Modell réteg

- a Modell tartalmazza:
  - adat modellt és business logikát
  - modell objektumok közötti kapcsolatokat
  - kiszámolt tulajdonságokat
  - validálást - INotifyDataErrorInfo
  - **NB**: a Modellnek referenciaként NEM kell adni semmit!!!

View

- **CSAK** a megjelenítéssel foglalkozzon
- feladata **NEM** az, hogy adatot fogadjon, küldjön, transzformáljon, hanem, hogy a kinézettel foglalkozzon (ablakok, stílusok, pozícionálás stb.)
- CÉL: semmi/kevés háttérkód a View.xaml.cs-ben
- a View **NEM** látja a Modellt, a View nem tud semmit a VM-ről(és fordítva sem), az adatokat a ViewModeltől kapja, ők **DataBindiggal** kommunikálnak
- egy Viewnek lehet saját ViewModel-je vagy örökölheti az őse ViewModeljét
- a View - **Command**-al küldhet eseményt a ViewModelnek
- futás közben, a nézet akkor változik meg, amikor egy UI vezérlő válaszol a ViewModel egy tulajdonságának a módosulásának eseményére
- a View az alkalmazás belépési pontja –> először a View hívódik meg, ez meghívja a ViewModelt és a Modelt
  - **NB**: ViewModel is beállítható belépési pontnak!
- **NB**: a Viewnek **NEM** kell referenciaként adni a ViewModelt!

ViewModel

- bevezetésének egyik fő oka a tesztelhetőség (egységbezárás)
  - a VM külön UnitTest-elhető
- a View és a Model összekötését oldja meg
- a háttérkód feladatát veszi át
- feladata a View számára biztosítsa azon információkat, amelyek a megjelenéssel kapcsolatosak
- a ViewModel és a Model szoros kapcsolatban van
- a ViewModel kapcsolatba lép a Modellel (meghívja a metódusait stb), a modelltől kapott adatokat olyan formára hozza, hogy a View könnyen tudja használni azokat
- a ViewModel implementálja a **Command**-okat, amelyeket az alkalmazás felhasználói kezdeményeznek a View-n (pl. button click)
- a ViewModelnek referenciaként kell adni a Modelt, a Common-t

ViewModel -2

- itt kerülnek példányosításra az objektumok, itt határozzuk meg a gyűjteményeket, melyeket valamilyen formában szeretnénk a felületen megjeleníteni
- a ViewModelben létre kell hozni változókat, amelyek a UI változóinak az értékeit fogják felvenni, transzformálni stb.!!!
- változások értesítésének kiváltását – INotifyPropertyChanged
- ViewModel és View összekapcsolása:
  - a View hivatkozik ViewModelre a **DataContext** tulajdonsággal
  - **DataContext** tulajdonság segítségével tudjuk megadni a vezérlők adatforrását, vagyis adatkötéskor innen kérjük le az adatokat

Az MVVM előnyei

- könnyebb ellenőrizni (tesztelni), karbantartani és fejleszteni
- a grafikus és a programozó tevékenysége elhatárolódik
- újrafelhasználható
- a View, illetve a ViewModel könnyen cserélhető, módosítható anélkül, hogy a másikat befolyásolná
- a fejlesztők készíthetnek **unit teszt**-eket, amivel tesztelhetik a Model és ViewModel működését, anélkül, hogy használnák a Viewt

- egyszerű alkalmazásoknál nem célszerű, mivel hosszabb tervezést és körülményesebb implementációt igényel

Megvalósítás

- megvalósításához több eszközt kell használnunk:
  - felület és nézetmodell közötti adattársítás (**Binding**)
  - az adatokban történt változások nyomon követése a nézetmodellben (**INotifyPropertyChanged**)
  - tevékenységek végrehajtása eseménykezelők használata nélkül, parancsok formájában (**ICommand**) a nézetmodellben
- az architektúra tovább bővíthető a **perzisztencia** (adatelérés) réteg bevezetésével, így 4 rétegű architektúrát kapunk

Adatkötés

- Adatkötés (Data Binding) (Előadás 6)
- Adatkötés változáskövetéssel (INotifyPropertyChanged) (Előadás 6)

Parancsok - Command

- mivel az eseménykezelők összekötnék a felületet a modellel, nem használhatóak az MVVM architektúrában
  - pl. egy button Click eseményét azért nem használhatom, mert az eventjében nem látom a ViewModel-t
- az eseménykezelők helyettesítésére a ViewModelben parancsokat (**ICommand**) használunk
  - adattársítással kapcsolható vezérlőhöz, annak **Command** tulajdonságán keresztül
  - megadják a végrehajtás tevékenységét (**Execute**), valamint a végrehajthatóság engedélyezettségét (**CanExecute**)
  - a végrehajthatóság változását is jelzi (**CanExecuteChanged**)
- a parancsnak adható végrehajtási paraméter is (a vezérlő **CommandParameter** tulajdonságával)

Parancsok - Még miért használjuk?

- pl. egy mentést szeretnénk végezni
- többféle lehetőség van
  - SaveButton Click, CTRL S, jobb click Save
  - 3 különböző tevékenység ugyanazért az eredményért
  - mindenikre külön eseménykezelőt???
- Commandokkal egyszerűbbé tehetjük
- társítjuk ezeket a tevékenységeket egyetlen paranccsal

```
<Button Content="Add user" Command="{Binding AddCommand}" Grid.Row="2"
    />
```

- a parancshoz kapcsolunk egyetlen eseménykezelőt, amely tartalmazza a logikát

Command példa

- ahhoz, hogy létrehozzunk egy parancsot kell implementálni az ICommand interfészt

```
public class MyCommand : ICommand
{
    public void Execute(object parameter)
    {
        // tevékenység végrehajtása (paraméterrel)
        Console.WriteLine(parameter);
    }
    public Boolean CanExecute(object parameter) //mindig végrehajtódik
    {
        // tevékenység végrehajthatósága
         return parameter != null;
         // return true mindig végrehajtható, return false nem hajtható végre
    }
    public event EventHandler CanExecuteChanged;
    // kiválthatóság változásának eseménye
}
```

- a **CanExecute** minden Clickre, Focus Last-ra meghívódik, ezért vigyázni kell, hogy mit írunk bele, AB. műveletet semmiképp

Command példa -2

```
// ViewModel
public class MyViewModel : INotifyPropertyChanged
{
    // parancs elhelyezése a ViewModelben
    public MyCommand ClickCommand{ get; set; }
   ...
   //ctor
   //ClickCommand = new MyCommand(this); // parancs létrehozása

   ClickCommand = new MyCommand(this.ExecuteClickCommand);
    ...
    private void ExecuteClickCommand(object parameter)
    {
      if (parameter is string)
      {
        string message = parameter as string;
        if (!string.IsNullOrEmpty(message))
        {
          MessageBox.Show(message.Trim());
        }
      }
    }
...
}
```

Parancsok a nézetmodellben

- egy alkalmazásban sok parancsra lehet szükség, nem célszerű mindegyik számára külön osztályt készíteni
  - a parancsoknak egy tevékenységet kell végrehajtania, amely **Action** típusú λ-kifejezéssel is megadható, míg a feltétel egy **Func** típusúval
  - a tényleges tevékenységet végrehajtó művelet elhelyezhető a ViewModel osztályban, így nem kell külön osztályokba helyezni a kódot
  - elég csupán egy parancs osztályt létrehoznunk (legyen ez **DelegateCommand**) a tevékenység végrehajtásához, és a tényleges tevékenységet a parancs példányosításakor λ-kifejezés formájában adjuk meg

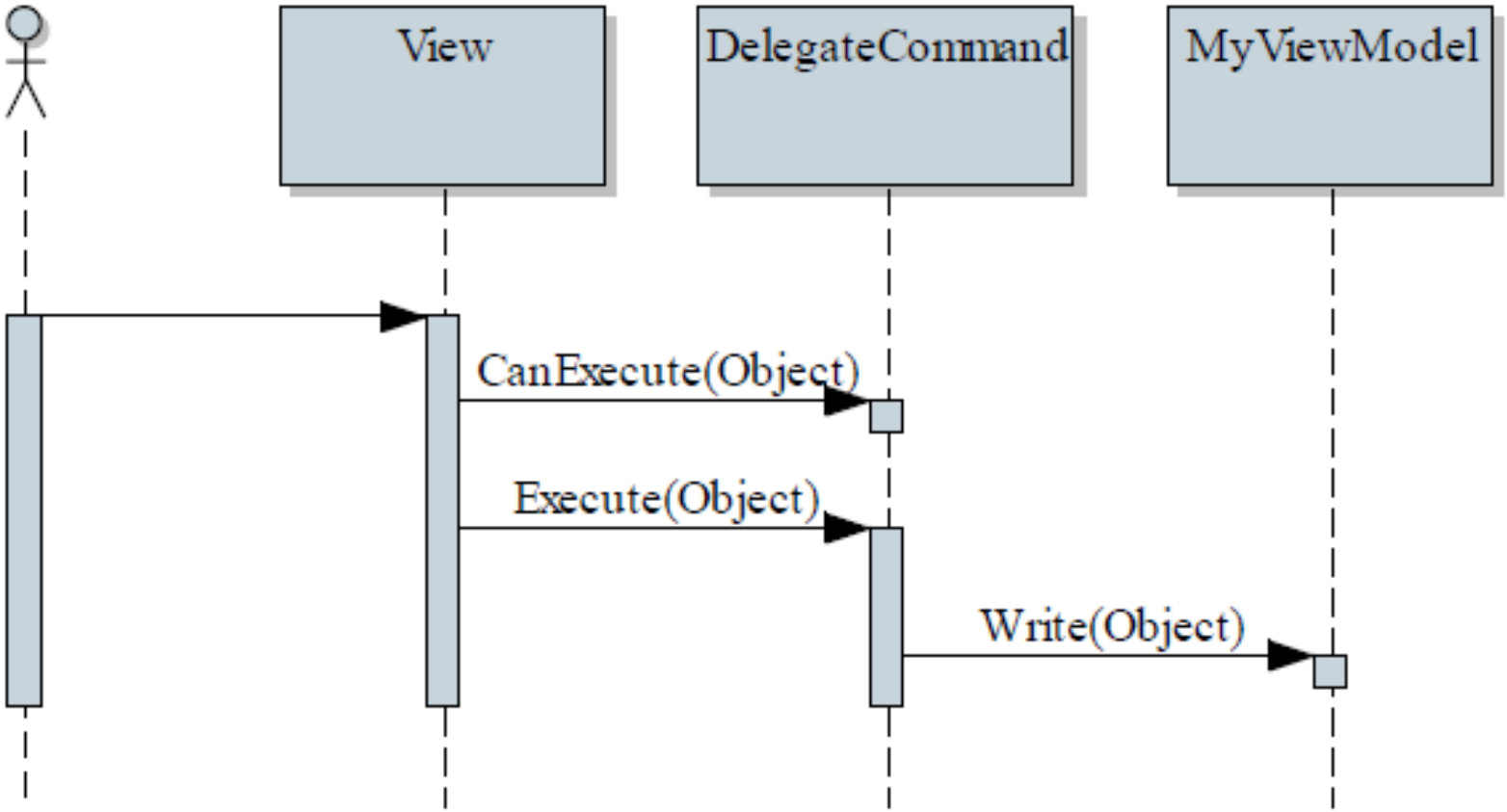
Parancsok a nézetmodellben -2

```
public class DelegateCommand: ICommand
{
    private Action<Object> _execute;
    private Func<Object, Boolean> _canExecute;
    // tevékenység és feltétel eltárolása
    ...
    public DelegateCommand(Action<Object> execute)
    {
        _execute= execute; // tevékenység rögzítése
    }
    public void Execute(Objectparameter)
    {
        _execute(parameter);
        // tevékenység végrehajtása
    }
}
```

Példa

```
// nézetmodell
public class MyViewModel: INotifyPropertyChanged
{
    // parancs elhelyezése a nézetmodellben
    public DelegateCommand MyCommand { get; set; };
    public void Write(Object parameter)
        {
          Console.WriteLine(parameter);
          // tevékenység
        }
...
    MyCommand = new DelegateCommand(x => Write(x));
    // tevékenység tényleges megadása
...
```

Példa


View
DelegateCommand
MyViewModel
CanExecute(Object)
Execute(Object)
Write(Object)

Parancsok végrehajthatósága

- A parancs bármikor jelezheti, hogy állapota megváltozott a **CanExecuteChanged** eseménnyel
  - amennyiben nem végrehajtható, a vezérlő kikapcsolt állapotba kerül
  - az eseményt alapesetben a parancsnak kell kiváltania, de általános parancsok esetén ez nem végezhető el
  - megoldást nyújt a **CommandManager** osztály, amelynek **RequerySuggested** statikus eseménye jelzi, ha újra kell vizsgálni az állapotot
    - automatikusan meghívja a rendszer, amikor beavatkozás szükségességét érzi (pl. ha valamilyen tevékenység fut a felületen)

Parancsok végrehajthatósága -2

- az egyik eseményt elfedhetjük a másikkal, ehhez az esemény feliratkozását/leiratkozását kell megváltoztatnunk
- pl.:

```
public event EventHandler CanExecuteChanged
{
    add
    {
        // feliratkozás módja
        CommandManager.RequerySuggested += value;
        // a RequerySuggested kiváltása hatására
        // kiváltódik a CanExecuteChanged is, így frissül az állapot
    }
    remove
    {
        // leiratkozás módja
        CommandManager.RequerySuggested -= value;
    }
}
```

Speciális parancskötések

- A speciális egér és billentyű utasításokhoz a vezérlő **InputBindings** tulajdonságát használjuk, amibe helyezhetünk
  - billentyűzetkötést (**KeyBinding**), megadva a billentyűt (**Key**), vagy billentyűkombinációt (**Gesture**)
  - egérkötést (**MouseBinding**), megadva a gombot (**MouseAction**), vagy a kombinációt (**Gesture**)

```
<Window.InputBindings> <!--bemeneti kötések -->
    <KeyBinding Command=”{Binding MyCommand}” Gesture=”CTRL+R” />
        <!--Ctrl+R billentyukombináció kötése -->
</Window.InputBindings>
```

ObservableCollection vs List?

- Mi a különbség?
- ha felvesszük az elemeket, elvégezzük az adatkötést, látszólag semmi különbséget nem veszünk észre. Az INotifyPropertyChanged interfész által implementált metódus jelzést ad le a felület számára, hogy az adott adattaghoz tartozó adatkötés frissítésre szorul, mivel változott az érték.
- hasonló módon egy ObservableCollection-ben ha új elemet veszünk fel, vagy éppen távolítunk el, a változás a felületen is reflektálva lesz.
- **NB:** Ez nem jelenti azt, hogy az objektumok összes adattagja automatikusan jelezni fogja a változásokat, azt nekünk ugyanúgy az osztályon belül meg kell valósítanunk az INotifyPropertyChanged interfész segítségével.

Egyedi vezérlők - UserControl

- **User Control**-ok lehetőséget bitosítanak, hogy különböző vezérlőket egyesítsünk és újrahasználjuk XAML-ben
  - a WPF applikációkhoz hasonlóan a User Control tartalmazhat más vezérkőket, erőforrásokat, animációkat, stb.
  - a különbség, hogy Window vagy Page helyett, a gyökér elem egy UserControl
  - a saját vezérlőket névtér hivatkozáson keresztül érjük el, pl.:

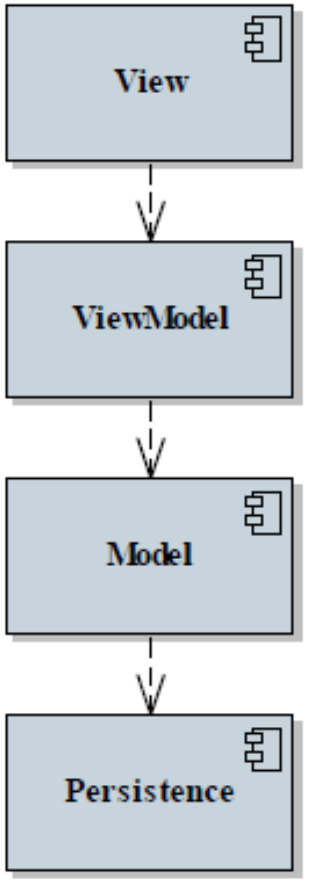
```
<Window
...
<!--megadjuk a névteret -->
    xmlns:view=”clr-namespace:MyApp.View” ... >

   <!--példányosítjuk az egyedi vezérlot -->
    <view:MyControl ...>
</Window>
```

- egyedi vezérlők és a ViewModel-ek összekötése:

```
<UserControl.DataContext>
     <Project_ViewModels:ClientsViewModel/>
</UserControl.DataContext>
```

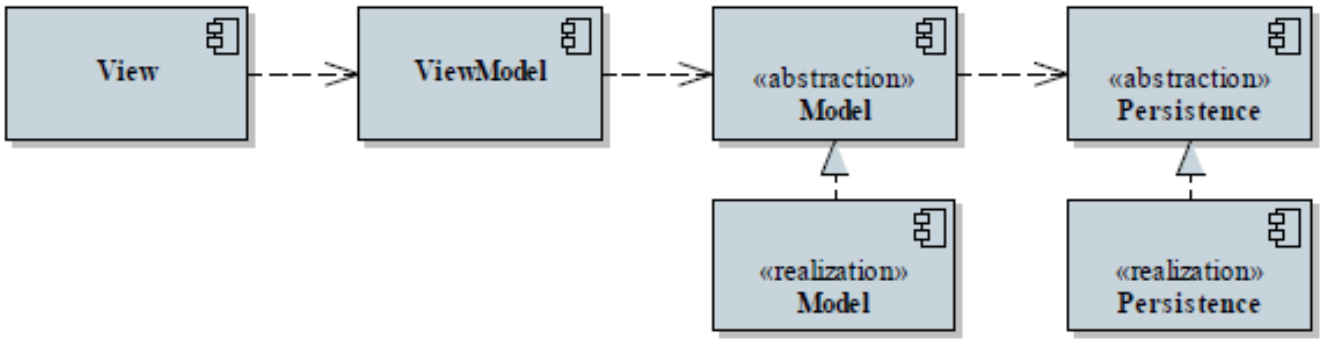
Az MVVM architektúra

- Az MVVM architektúrában
  - a **nézet** tartalmazza a grafikus felületet és annak erőforrásait
  - a **nézetmodell** egy közvetítő réteg, lehetőséget ad a modell változásainak követésére és tevékenységek végrehajtására
  - a **modell** tartalmazza az alkalmazás logikáját
  - a **perzisztencia** a hosszútávú adattárolást és adatelérést biztosítja



Függőség kezelés MVVM architektúrában

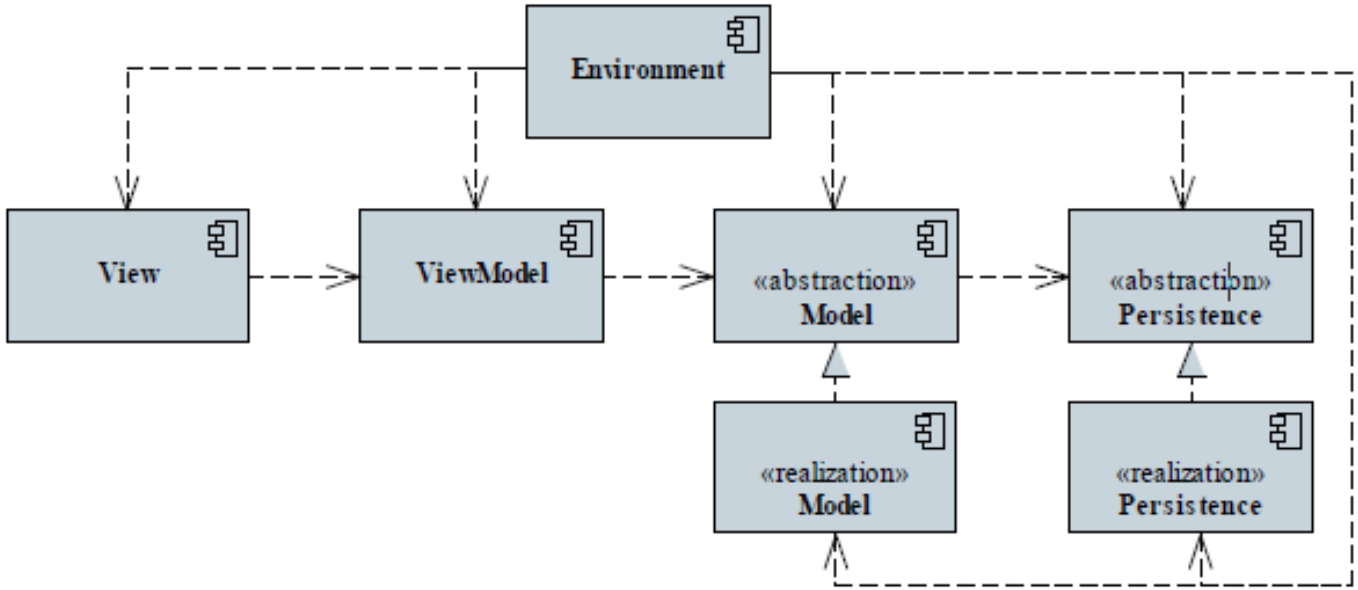
- az architektúra akkor megfelelő, ha az egyes rétegek között minél kisebb a függőség (**loose coupling**)
  - egyik réteg sem függhet a másik konkrét megvalósításától, és nem avatkozhat be a másik működésébe
  - ennek eléréséhez függőség befecskendezést (**dependency injection**) használunk



Függőség kezelés MVVM architektúrában

-
  - a nézetmodellt a nézetbe egy tulajdonságon keresztül fecskendezzük be (**setter injection**)
  - a modellt a nézetmodellbe, a perzisztenciát a modellbe konstruktoron keresztül helyezhetjük (**constructor injection**)
- a programegységek példányosítását és befecskendezését az alkalmazás környezete (**application environment**) végzi
  - ismeri és kezeli az alkalmazás összes programegységét (absztrakciót és megvalósítást is)
  - nem az adott komponens, hanem a környezet dönti el, hogy a függőségek mely megvalósításai kerülnek alkalmazásra (**Inversion of Control, IoC**)

- a környezetet egyszerű esetben megadhatja az alkalmazás (App), de használhatunk külön komponenst is
- a környezet hatásköre kibővíthető a globális, teljes alkalmazást befolyásoló tevékenységekkel (pl. időzítés)



Architektúra programcsomagok

- az alapvető MVVM támogató konstrukciók a nyelvi könyvtárban nem elegendőek a hatékony, gyors fejlesztésre
  - interfészek vannak (pl. **INotifyPropertyChanged, ICommand**), de nincsenek ősosztályok, gyűjtőosztályok
- Több olyan programcsomag került forgalomba, amely az MVVM alapú fejlesztést megtámogatja, pl.:
  - **Microsoft Prism**: támogatja a modul alapú fejlesztést, az MVVM architektúrákat, nézet-dekompozíciótés cserét
  - **MVVM LightToolkit**: támogatja az MVVM architektúrát, a többrétegű modellt, komponensek közötti üzenetküldést, alkalmazás környezet kialakítását
  - **Chinch**
  - **Onyx**

Példa

- Készítsünk WPF alkalmazást (MVVM patternt használva), amellyel a felületen megjeleníthetjük, módosíthatjuk, beszúrhatjuk a felhasználók adatait.
  - a változást jelezzük INotifyPropertyChanged-el
  - a felhasználók gyűjteménye változásfigyelő ObservableCollection típusú legyen
  - a View és a ViewModel-t az App.xaml.cs-ben társítsuk

Megvalósítás (User.cs)

```
class User : INotifyPropertyChanged
{
    private string name;
    public string Name {
                get { return name; }
                set {
                    if(name != value)
                    {
                        name = value;
                        NotifyPropertyChanged(”Name”); // jelezzük a
                            változást
                    }}
                }
    public event PropertyChangedEventHandler PropertyChanged; // az esemény

    public void NotifyPropertyChanged(string propName)
    {
        if(this.PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(propName)
                ); // eseménykiváltás
    }
}}
```

Megvalósítás (App.xaml.cs):

```
...
private void App_Startup(...) {
    UserWindow window= new UserWindow(); // View

    UserViewModel viewModel = new UserViewModel(); // ViewModel

    window.DataContext= viewModel; // View és ViewModel társítása

    window.Show();
}
...
```

Megvalósítás (UserWindow.xaml)

- mindennek van DataContextje, ezért tudok Bindolni egy osztály vmelyik tagjára, megadom, hogy az ItemsSource = osztálynév,

```
<ItemsControl ItemsSource="{Binding Users}"> <!--az adatforrás -->
    <ItemsControl.ItemTemplate><DataTemplate>
    <StackPanel>
        <TextBox Text="{Binding Name}" Width="100"/> <!--adatkötés a
            tulajdonságokhoz -->
...
</ItemsControl/>
...
```

Megvalósítás (UserAddCommand.cs)

```
class UserAddCommand: ICommand // parancs objektum
{
    private UserViewModel viewModel;
    public void Execute(Object parameter)
    {
    viewModel.AddNewUser();// új felhasználó felvétele
    }
...
}
```

Megvalósítás (UserWindow.xaml)

```
<StackPanel DataContext="{Binding ujUser}" Grid.Row="1" >
...
    <TextBox Text="{Binding Name}" Width="100"/>
</StackPanel>
<Button Content="Add user" Command="{Binding AddCommand}" Grid.Row="2" />
     <!--parancs hozzákötése -->
```

Megvalósítás (UserViewModel.xaml.cs)

```
public class UserViewModel : INotifyPropertyChanged
   {
        public ObservableCollection<User> Users { get; private set; }

        public User ujUser { get; private set; }

        public UserAddCommand AddCommand { get; private set; }

        public event PropertyChangedEventHandler PropertyChanged; // Tulajdonságváltozás eseménye

        public UserViewModel()
        {
            Users = new ObservableCollection<User>();
            ujUser = new User();
            AddCommand = new UserAddCommand(this.AddNewUser); // parancs létrehozása

            Students.Add(new Student { Id = 1, Name = ”Demeter”});
            Students.Add(new Student { Id = 2, Name = ”Fulop” });
            Students.Add(new Student { Id = 3, Name = ”Janos” });
            Students.Add(new Student { Id = 4, Name = ”Pistike” });
        }
        public void AddNewUser()
        {
            Users.Add(ujUser);
            ujUser = new User();
            NotifyPropertyChanged(”ujUser”);
        }

        /// <param name=”property”>A tulajdonság neve.</param>
        private void  NotifyPropertyChanged(String property)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(property));
            }
        }
```