



Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások

7. előadás

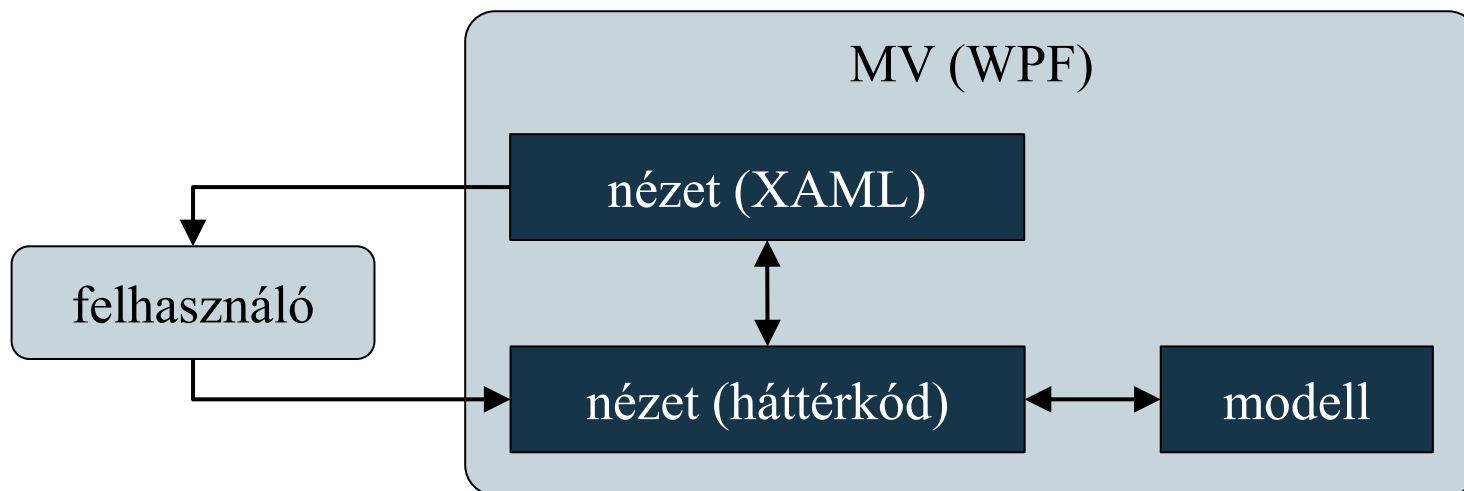
WPF alkalmazások architektúrája

Dr. Cserép Máté
mcserep@inf.elte.hu
<https://mcserep.web.elte.hu>

WPF alkalmazások architektúrája

A nézet rétegződése

- Grafikus alkalmazásoknál alapvető tervezési kérdés a felületi megjelenés, valamint a tevékenységek szétválasztása, vagyis a *modell/nézet* (*Model/View*, *MV*) architektúra használata
 - WPF alkalmazásoknál igazából a nézet is két részre bontható, felületi (XAML) kódra és háttérkódra



WPF alkalmazások architektúrája

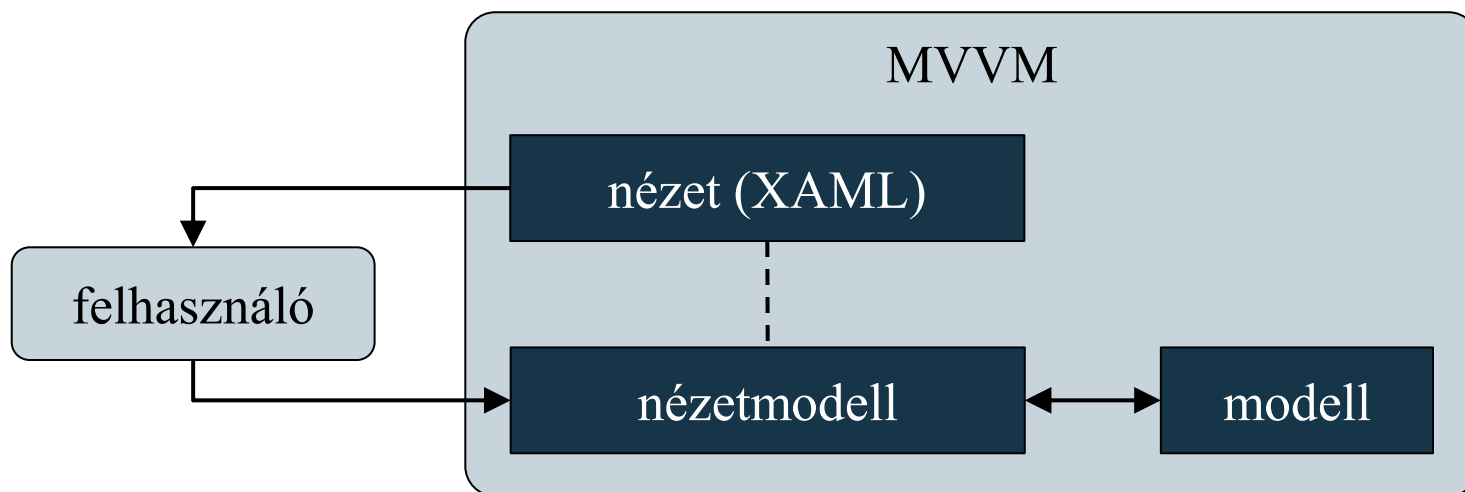
A nézet felbontása

- A nézeten belüli szeparációnak köszönhetően a felület megvalósítása elhatárolható két különálló részre:
 - a háttérkód a *programozó* feladata, aki ért a kódhoz, eszköze: *Microsoft Visual Studio*
 - a felületi kód a *grafikus* feladata, aki ért az ergonómiához, tervezéshez, eszköze: *Microsoft Blend* (korábbi nevén *Microsoft Expression Blend*)
- Ehhez szükséges, hogy a felületi kód és a háttérkód hasonlóan szeparálhatóak legyenek, mint a modell és a nézet
 - azaz ne legyenek összekötések (pl. eseménykezelő-társítás), amelyek mentén kettejük munkája összeakadhat

WPF alkalmazások architektúrája

A modell/nézet/nézetmodell architektúra

- A *modell/nézet/nézetmodell* (*Model/View/ViewModel*, *MVVM*) célja, hogy teljes egészében elvállassa a megjelenítést és a mögötte lévő tevékenységeket
 - köszönhetően egy közvetítő réteg (*nézetmodell*) közbeiktatásának, amely a háttérkód feladatát veszi át



WPF alkalmazások architektúrája

A modell/nézet/nézetmodell architektúra

- Az MVVM architektúrában
 - 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
- Előnyei:
 - a grafikus és a programozó tevékenysége elhatárolódik
 - a nézet, illetve a nézetmodell könnyen cserélhető, módosítható anélkül, hogy a másikat befolyásolná

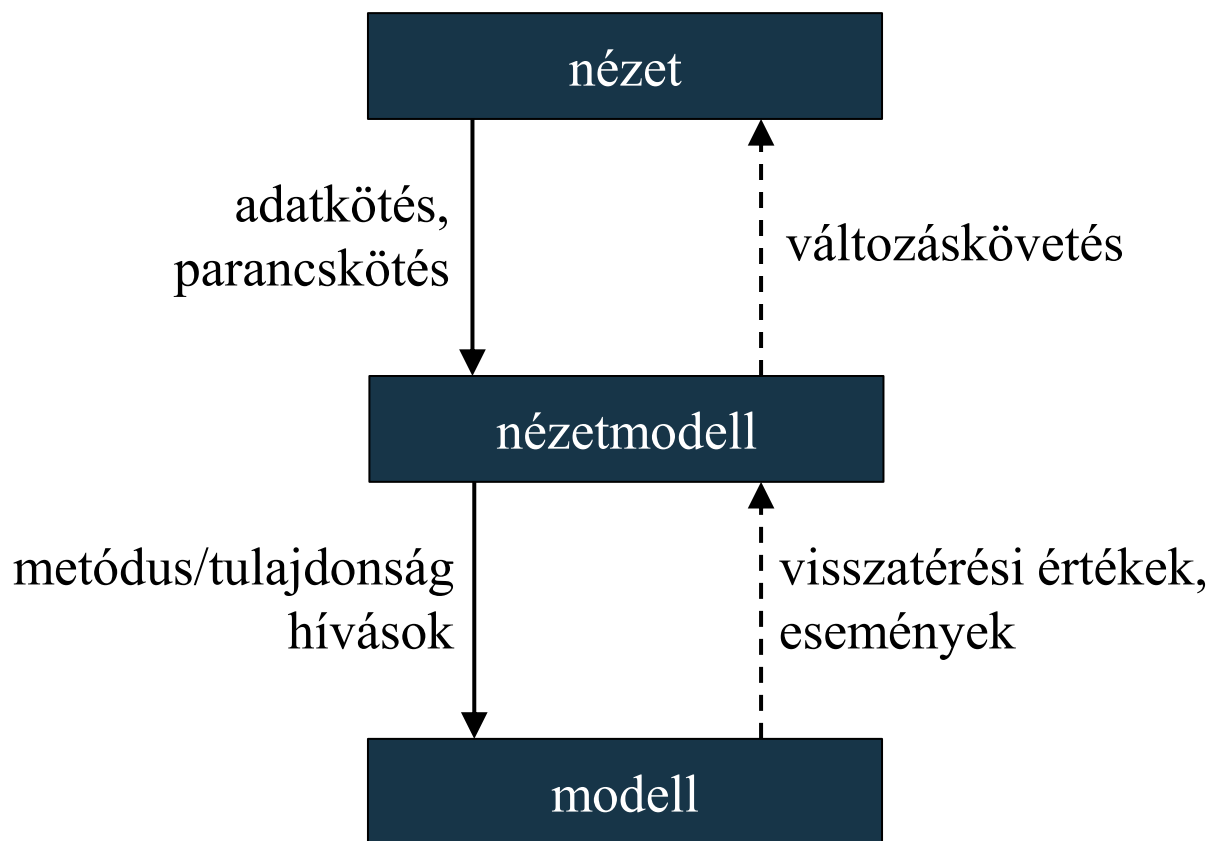
WPF alkalmazások architektúrája

A modell/nézet/nézetmodell architektúra

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

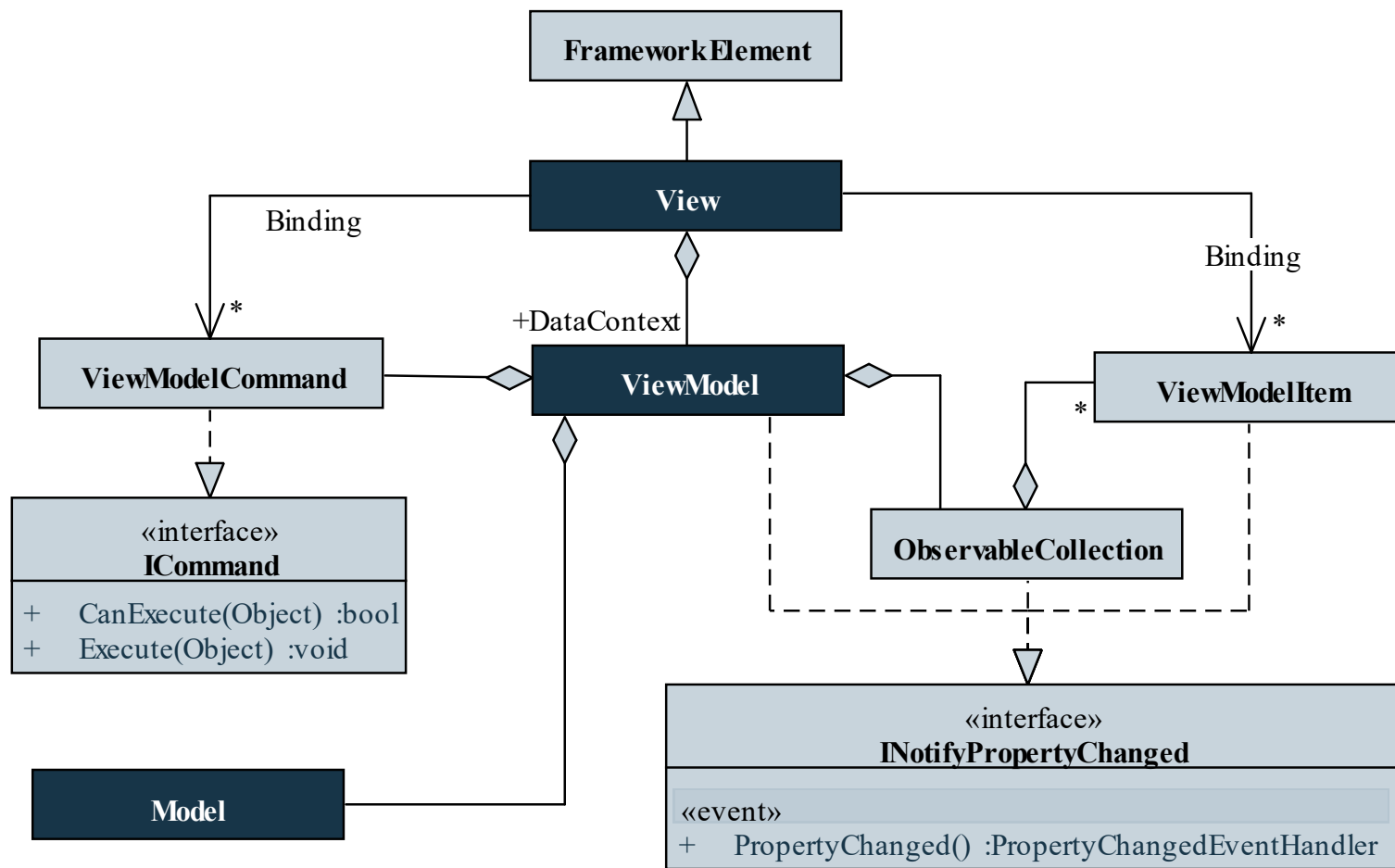
WPF alkalmazások architektúrája

A modell/nézet/nézetmodell architektúra



WPF alkalmazások architektúrája

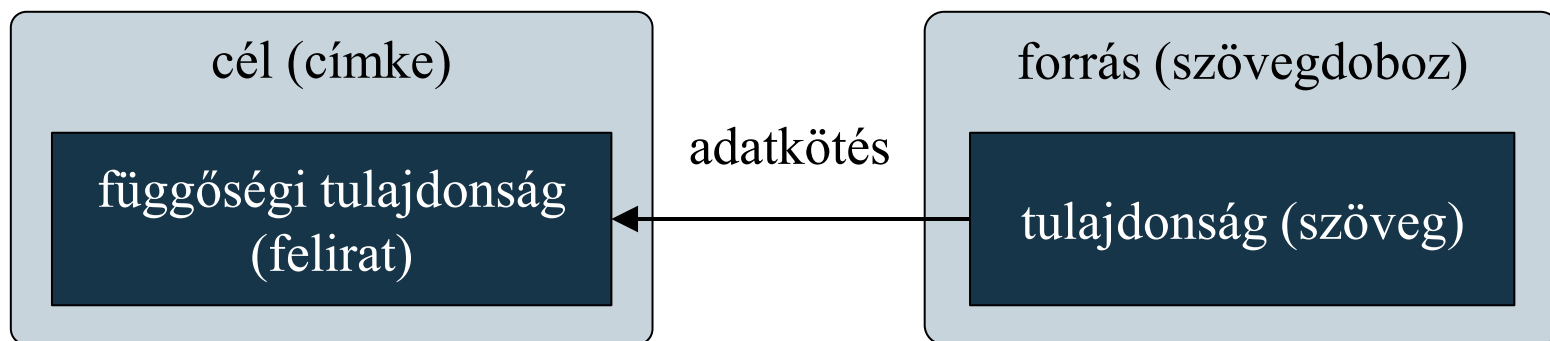
A modell/nézet/nézetmodell architektúra megvalósulása



WPF alkalmazások architektúrája

Adatkötés

- Az *adatkötés* (*data binding*) során függőségeket adhatunk meg a felületen megjelenő elemek tulajdonságaira
 - egy adott vezérlő valamilyen függőségi tulajdonságát (*cél*) tudjuk függővé tenni valamilyen objektumtól, vagy annak egy tulajdonságától (*forrás*)
 - így közvetett módon (anélkül, hogy a konkrét vezérlőhöz hozzáférésünk lenne) tudunk egy tulajdonságot állítani
 - pl. egy szövegdobozban tárolt szöveget kiírathatunk egy címkére



WPF alkalmazások architektúrája

Adatkötés

- A kötést (**Binding**) a függőségi tulajdonság értékeként hozzuk létre forrás objektum (**Source**, **ElementName**) és tulajdonság útvonal (**Path**) megadásával, pl.:

```
<TextBox Name="textBoxName" />
```

```
<!-- forrás (szövegdoboz) -->
```

...

```
<TextBlock Text="{Binding ElementName=textBoxName,  
                        Path=Text}" />
```

```
<!-- cél (címke), mindig azt a szöveget jeleníti  
      meg, ami a szövegdobozban van -->
```

...

```
<Button Content="{Binding ElementName=textBoxName,  
                    Path=Text}" />
```

```
<!-- cél (gomb) -->
```

WPF alkalmazások architektúrája

Adatkötés

- forrás lehet egy teljes objektum, vagy bármely tulajdonsága, vagy beágyazott tulajdonság, pl.:

```
<TextBlock
```

```
    Text="{Binding ElementName=textBoxName,  
                Path=Text.Length}" />
```

```
<!-- a címke a szöveg hosszát jeleníti meg -->
```

- amennyiben egy névvel rendelkező felületi elemhez kötünk, az **ElementName**, más objektumok, erőforrások esetén a **Source** tulajdonsággal adjuk meg a forrást
- a forrás értéke implicit konvertálódik a cél tulajdonság típusára, vagy mi adjuk meg az átalakítás módját (az **IValueConverter** interfész segítségével)

WPF alkalmazások architektúrája

Adatkötés paraméterezése

- A kötés többféleképpen paraméterezhető, pl.:
 - a kötés módja (**Mode**) lehet egyirányú (**OneWay**), kétirányú (**TwoWay**, ekkor mindkét objektum változása kihat a másikra), egyszeres (**OneTime**), ...
 - a cél frissítése (**UpdateSourceTrigger**) lehet változtatásra (**PropertyChanged**), fókuszváltásra (**LostFocus**), ...

- Pl.:

```
<TextBox Name="textAnotherName"  
    Text="{Binding ElementName=textBoxName,  
        Path=Text, Mode=OneWay,  
        UpdateSourceTrigger=LostFocus}" />  
<!-- egyirányú kötés fókuszváltásra -->
```

WPF alkalmazások architektúrája

Adatkötés objektumértékekhez

- Adatkötés a felületi vezérlők mellett tetszőleges objektumra, kódban is megadható
 - kódban a cél **DataContext** tulajdonságának kell megadnunk a forrást
 - a teljes forrás kötése esetén a felületi kódban egy üres kötést adunk meg, pl.:

```
<TextBox Name="textBox" Text="{Binding}" />
```

```
<!-- a kötést megadjuk a felületen, de  
      tulajdonságait nem töltjük ki -->
```

...

```
textBox.DataContext = "Hello DataBinding!";  
// a forrást a kódban adjuk meg
```

WPF alkalmazások architektúrája

Adatkötés objektumértékekhez

- tulajdonság kötése esetén meg kell adnunk az útvonalat, pl.:

```
class Person {  
    public String FirstName { get; set; }  
    public String LastName { get; set; }  
}
```

...

```
Person person = new Person { ... };
```

```
textBox.DataContext = person;
```

```
    // a forrás a teljes objektum lesz
```

...

```
<TextBox Name="textBox"
```

```
    Text="{Binding Path=FirstName}" />
```

```
<!-- megadjuk a kötött tulajdonságot, röviden:
```

```
    {Binding FirstName} -->
```

WPF alkalmazások architektúrája

Adatkötés gyűjteményekre

- Az adatkötés gyűjteményekre is elvégezhető, ehhez olyan vezérlő szükséges, amely adatsorozatot tud megjeleníteni (pl. **ItemsControl**, **ListBox**, **GridView**, ...)
 - a vezérlők **ItemsSource** tulajdonságát kell kötnünk egy gyűjteményre (**IEnumerable**)
 - pl.:

```
<ComboBox Name="comboPersons"
            ItemsSource="{Binding}" />
...
List<String> persons = new List<String> { ... };
comboPersons.DataContext = persons;
// a teljes lista megjelenik a legördülő
// menüben
```


WPF alkalmazások architektúrája

Adatkötés tranzitivitása

- Az adatkötés tranzitív a elemek közötti kapcsolatokat leíró *logikai fán* a gyerek elemekre, így a tulajdonságok a beágyazott elemekben is elérhetőek
 - pl.:

```
<StackPanel Name="panelPersons">  
    <TextBlock Text="{Binding FirstName}" />  
    <!-- a FirstName-t kötjük hozzá -->  
    <TextBlock Text="{Binding LastName}" />  
    <!-- a LastName-t kötjük hozzá -->  
</StackPanel>  
  
...  
panelPersons.DataContext = person;  
    // az objektum két tulajdonsága jelenik meg
```
 - a tranzitivitás szűkíthető a tulajdonság megadásával

WPF alkalmazások architektúrája

Adatkötés öröklődése

- gyűjtemények esetén megadhatjuk az egyes elemek megjelenését
 - ehhez módosítanunk kell az adatok megjelenítési módját a vezérlőben az elemsablon (**ItemTemplate**) módosításával, amely egy adatsablont (**DataTemplate**) fogad
 - mind a teljes vezérlőre, mind az egyes elemek vezérlőire meg kell adnunk a kötést
 - az adatsablon bármilyen összetett vezérlőt tartalmazhat
 - pl.:

```
List<Person> persons = new List<Person> {...};  
comboPersons.DataContext = persons;  
// az elemek már összetett objektumok
```

WPF alkalmazások architektúrája

Adatkötés öröklődése

```
<ComboBox Name="comboPersons"
           ItemsSource="{Binding}" >
  <ComboBox.ItemTemplate>
    <!-- megadjuk az elemek megjelenítésének
           módját -->
    <DataTemplate>
      <TextBlock Text="{Binding FirstName}"/>
      <!-- minden elemnek a FirstName
           tulajdonsága jelenik meg -->
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

WPF alkalmazások architektúrája

Adatkötés a teljes felületre

- Az adatkötés egy teljes ablakra (**Window**) is elvégezhető
 - az adatkötést kódban adjuk meg, ezért az ablakot is kódban kell példányosítanunk és megjelenítenünk, pl.:
- az alkalmazás (**App**) indulásakor (**Startup**) kell végrehajtanunk a tevékenységeket, pl.:

```
MainWindow window = new MainWindow();  
window.DataContext = ...; // adatkötés az ablakra  
window.Show(); // ablak megjelenítése
```

```
public App() { // konstruktor  
    Startup +=  
        new StartupEventHandler(App_Startup);  
    // lekezeljük a Startup eseményt  
}
```

WPF alkalmazások architektúrája

Adatkötés változáskövetéssel

- Ahhoz, hogy a cél tükrözze a forrás aktuális állapotát, követni kell az abban történő változásokat
 - ehhez a forrásnak meg kell valósítania az **INotifyPropertyChanged** interfészt
 - ekkor a megadott tulajdonság módosításakor kiválthatjuk a **PropertyChanged** eseményt, ami jelzi a felületnek, mely kötéseknek kell frissíteni
 - az esemény elküldi a megváltozott tulajdonság nevét, ha ezt nem adjuk meg, akkor az összes tulajdonság változását jelzi
 - egyszerűsítésként felhasználhatjuk a **CallerMemberName** attribútumot, amely automatikusan behelyettesíti a hívó tag (tulajdonság) nevét

WPF alkalmazások architektúrája

Adatkötés változáskövetéssel

- pl.:

```
class Person : INotifyPropertyChanged {  
    ...  
    private String _firstName;  
  
    public String FirstName {  
        get { return _firstName };  
        set {  
            if (_firstName != value) {  
                _firstName = value;  
                OnPropertyChanged("FirstName");  
                //OnPropertyChanged();  
                // jelezzük a változást  
            }  
        }  
    }  
}
```

WPF alkalmazások architektúrája

Adatkötés változáskövetéssel

```
public event PropertyChangedEventHandler?  
    PropertyChanged; // implementált esemény  
  
public void OnPropertyChanged(  
    [CallerMemberName] String? name = null)  
    // ha paraméter nélkül hívták meg, a hívó  
    // nevét helyettesíti be  
{  
    if (PropertyChanged != null) {  
        PropertyChanged(this, new  
            PropertyChangedEventArgs(name));  
    } // eseménykiváltás  
}  
}
```


WPF alkalmazások architektúrája

Adatkötés változáskövetéssel

- a változáskövetés teljes gyűjteményekre is alkalmazható, amennyiben a gyűjtemény megvalósítja az **INotifyCollectionChanged** interfészt
- az **ObservableCollection** típus már tartalmazza az interfészek megvalósítását, ezért alkalmas változó tartalmú gyűjtemények követésére

- pl.:

```
ObservableCollection<Person> persons =  
    new ObservableCollection<Person> { ... };  
comboPersons.DataContext = persons;  
    // amennyiben a gyűjtemény, vagy bármely  
    // tagjának tulajdonsága változik, azonnal  
    // megjelenik a változás
```

WPF alkalmazások architektúrája

Példa

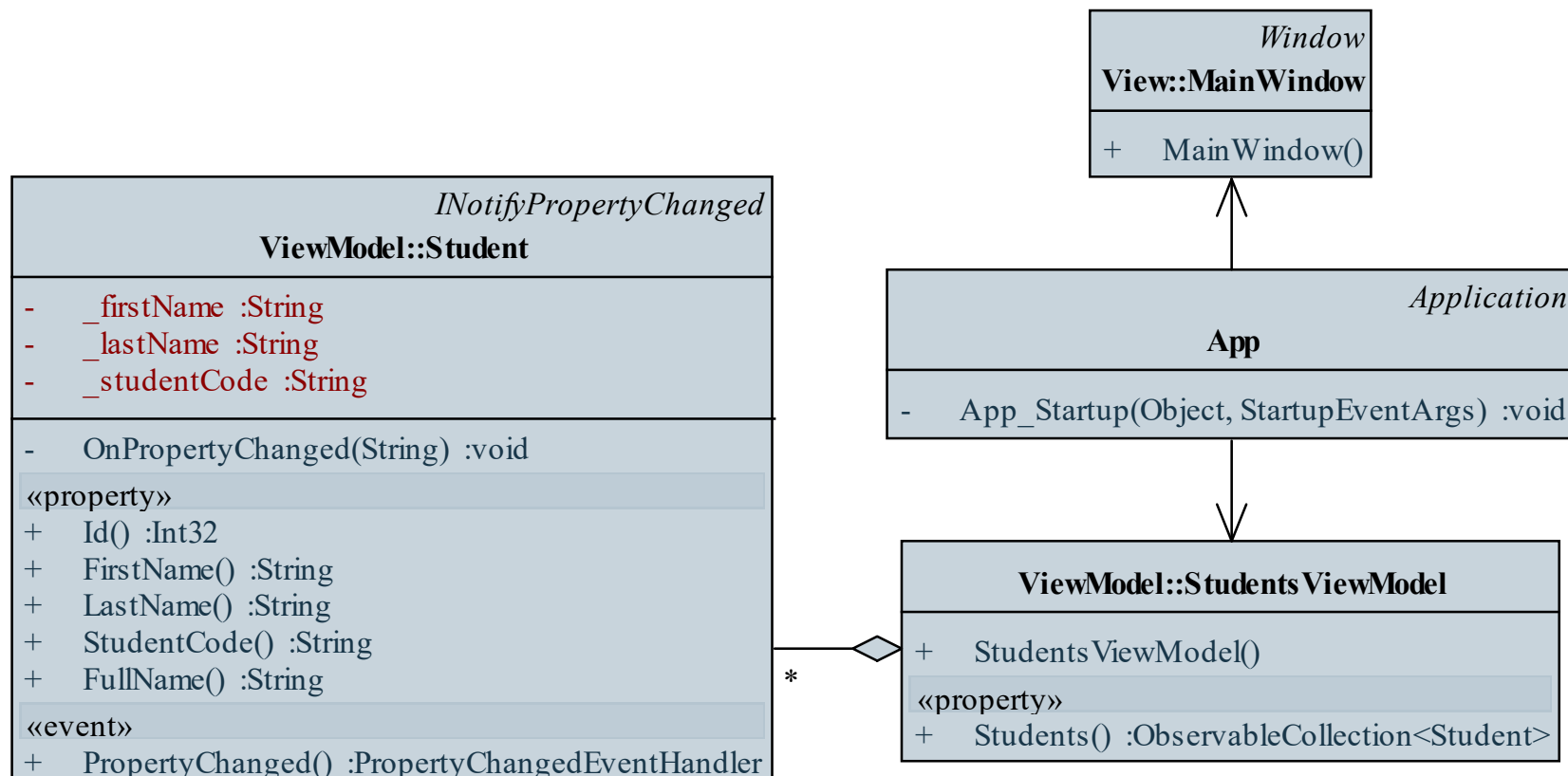
Feladat: Készítsünk egyszerű grafikus felületű alkalmazást, amellyel megjeleníthetjük, valamint szerkeszthetjük hallgatók adatait.

- a felületen a hallgató keresztnéve, vezetéknéve és Neptun-kódja külön szövegdobozba kerül, és egy szövegcímkében megjelenik a teljes neve, ezeket adatkötéssel fogjuk a hallgatóhoz (**Student**) kötni, amely jelezni fogja a változást (**INotifyPropertyChanged**)
- a nézetmodellben helyet kap a változásfigyelő gyűjtemény (**ObservableCollection**), és annak feltöltése
- a nézet és nézetmodell társítása az alkalmazásban (**App**) történik

WPF alkalmazások architektúrája

Példa

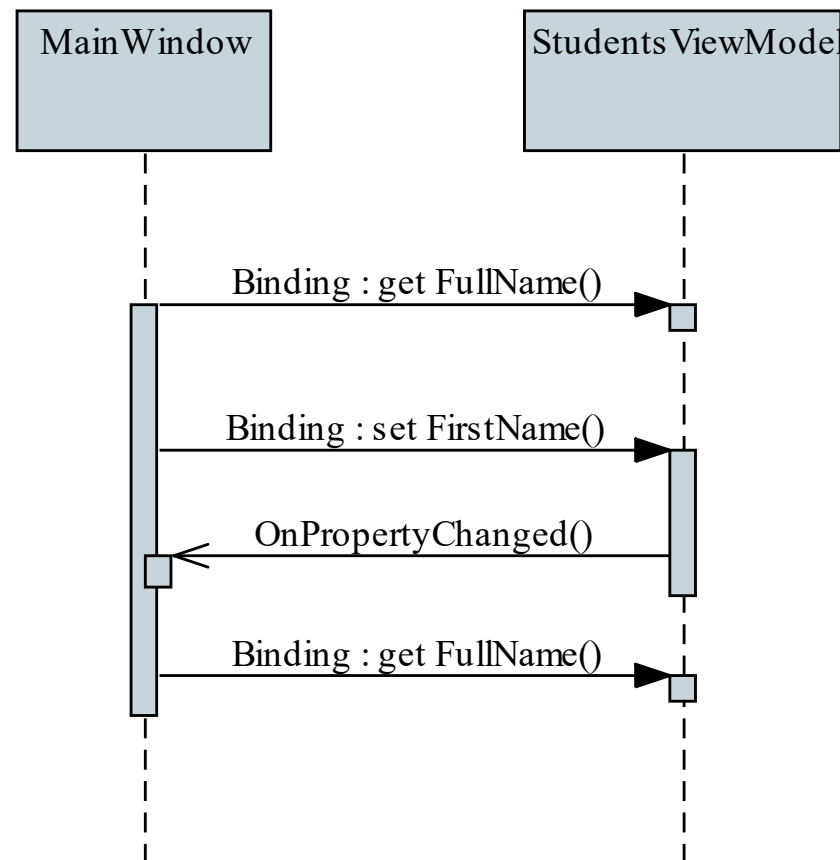
Tervezés:



WPF alkalmazások architektúrája

Példa

Tervezés:



WPF alkalmazások architektúrája

Példa

Megvalósítás (Student.cs):

```
class Student : INotifyPropertyChanged {  
    ...  
    public String FirstName {  
        get { return _firstName; }  
        set {  
            if (_firstName != value) {  
                _firstName = value;  
                OnPropertyChanged();  
                OnPropertyChanged("FullName");  
                // megváltoztak a FirstName és  
                // FullName tulajdonságok is  
            }  
        }  
    }  
    ...  
}
```

WPF alkalmazások architektúrája

Példa

Megvalósítás (App.xaml.cs):

...

```
private void App_Startup(...) {  
    MainWindow window = new MainWindow();  
    // nézet létrehozása  
    StudentViewModel viewModel =  
        new StudentViewModel();  
    // nézetmodell létrehozása  
    window.DataContext = viewModel;  
    // nézetmodell és modell társítása  
    window.Show();  
}
```

...

WPF alkalmazások architektúrája

Példa

Megvalósítás (MainWindow.xaml):

...

```
<ItemsControl
```

```
    ItemsSource="{Binding Students}">
```

```
    <!-- megadjuk az adatforrást -->
```

```
    <ItemsControl.ItemTemplate><DataTemplate>
```

```
        <!-- megadjuk az adatok reprezentációját -->
```

```
        <StackPanel Orientation="Horizontal">
```

```
            <TextBox Text="{Binding FirstName}"
```

```
                Width="100" Margin="5"/>
```

```
            <!-- adatkötés a tulajdonságokhoz -->
```

```
        ...
```

```
    </ItemsControl />
```

...

WPF alkalmazások architektúrája

Parancsok

- Mivel az eseménykezelők összekötnék a felületet a modellel, nem használhatóak az MVVM architektúrában
- Az eseménykezelők helyettesítésére a nézetmodellben *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)

WPF alkalmazások architektúrája

Parancsok

- Pl.:

```
public class MyCommand : ICommand {  
    public void Execute(object? parameter){  
        // tevékenység végrehajtása (paraméterrel)  
        MessageBox.Show(parameter);  
    }  
  
    public Boolean CanExecute(object? parameter){  
        // tevékenység végrehajthatósága  
        return parameter != null;  
    }  
  
    public event EventHandler? CanExecuteChanged;  
    // kiválthatóság változásának eseménye  
}
```

WPF alkalmazások architektúrája

Parancsok

- Pl.:

```
public class MyViewModel { // nézetmodell
    // parancs elhelyezése a nézetmodellben
    public MyCommand ClickCommand { get; set; }
    ...
}
...
<Button Content="Click Me"
        Command="{Binding ClickCommand}"
        CommandParameter="Hello, world!" />
<!-- parancs megadása adatkötéssel, valamint
paraméterrel -->
...
```

WPF alkalmazások architektúrája

Példa

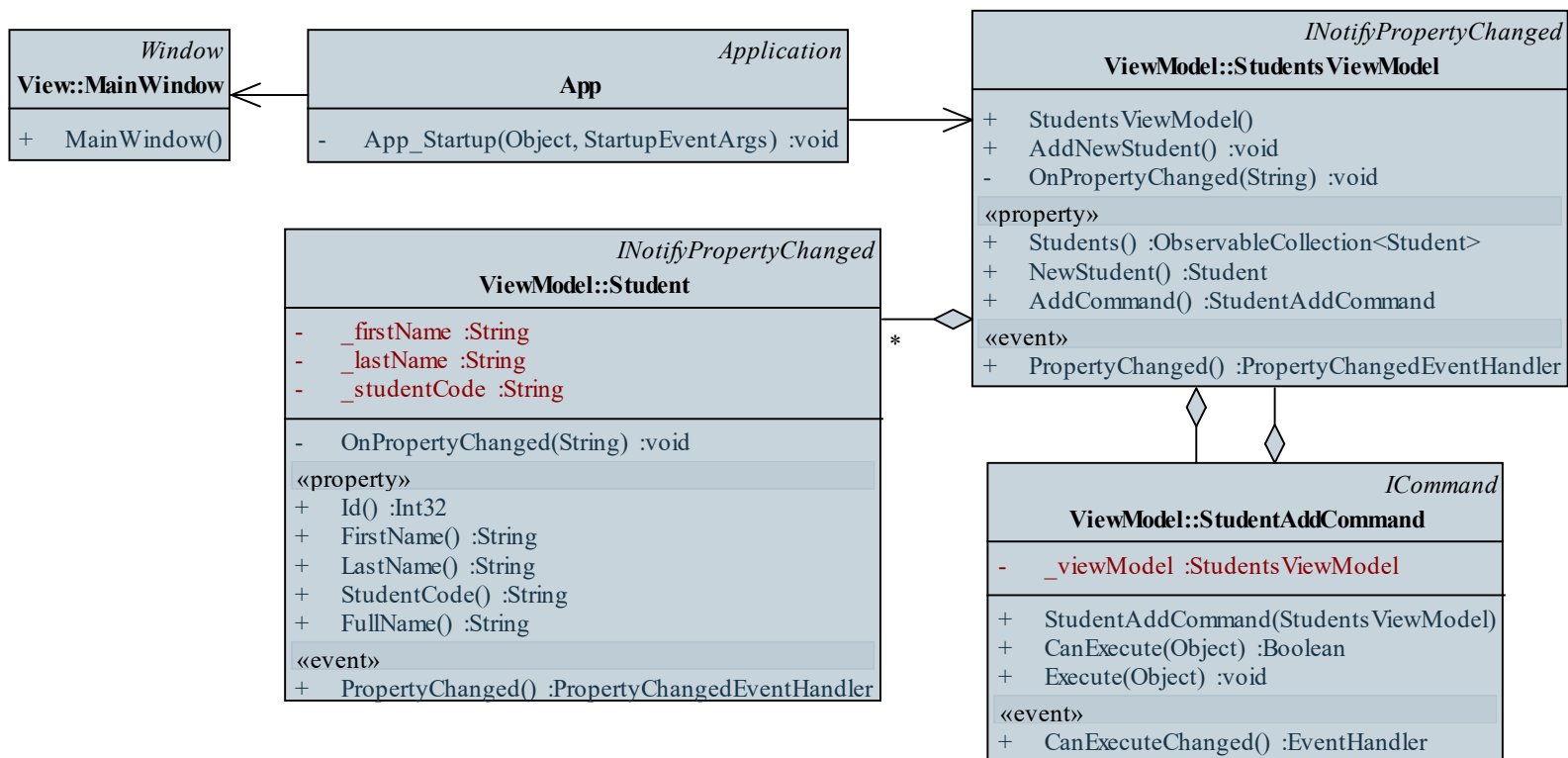
Feladat: Módosítsuk az előző alkalmazást úgy, hogy lehessen felvenni új hallgatót.

- a felületen három szövegdobozban megadhatjuk a hallgató adatait, majd egy gomb segítségével felvehetjük őket az alkalmazásba
- ehhez létrehozunk egy új parancs osztályt, amely a hallgató felvételét végzi (**StudentAddCommand**), és a végrehajtáskor felveszi a listába az új hallgatót
 - a parancsot tulajdonságként felvesszük a nézetmodellben
- magát az új hallgatót (**NewStudent**) is felvesszük a nézetmodellben, hogy lehessen mihez kötni a felületi adatokat

WPF alkalmazások architektúrája

Példa

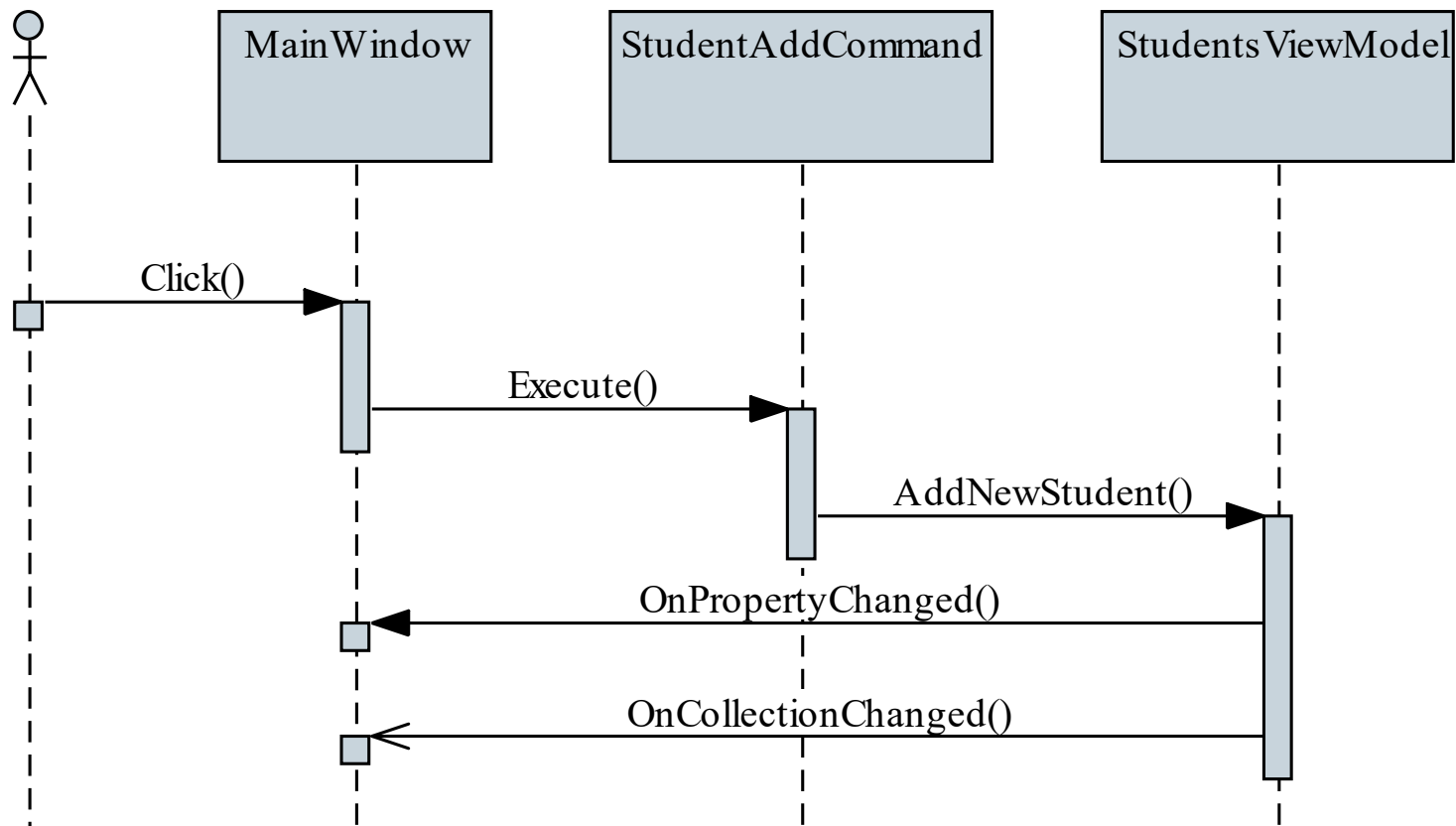
Tervezés:



WPF alkalmazások architektúrája

Példa

Tervezés:



WPF alkalmazások architektúrája

Példa

Megvalósítás (StudentAddCommand.cs):

```
class StudentAddCommand : ICommand
{
    // parancs objektum

    private StudentsViewModel _viewModel;

    public void Execute(Object? parameter) {
        _viewModel.AddNewStudent();
        // új hallgató felvétele
    }
    ...
}
```


WPF alkalmazások architektúrája

Példa

Megvalósítás (MainWindow.xaml):

...

```
<StackPanel DataContext="{Binding NewStudent}"  
    Orientation="Horizontal" Grid.Row="1">
```

...

```
<TextBox Text="{Binding StudentCode}"  
    Width="100" Margin="5"/>
```

```
</StackPanel>
```

```
<Button Content="Add student"  
    Command="{Binding AddCommand}" Margin="5"  
    Grid.Row="2" />  
<!-- parancs hozzákötése -->
```

...

WPF alkalmazások architektúrája

Parancsok a nézetmodellben

- Mivel egy alkalmazásban számos parancsra lehet szükség, nem célszerű mindegyik számára külön osztályt készíteni
 - a parancsoknak egy tevékenységet kell végrehajtania, amely **Action<T>** típusú λ -kifejezéssel is megadható, míg a feltétel egy **Func<T, Boolean>** típusúval (vagy **Predicate<T>**)
 - a tényleges tevékenységet végrehajtó művelet elhelyezhető a nézetmodell 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

WPF alkalmazások architektúrája

Parancsok a nézetmodellben

- Pl.:

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

WPF alkalmazások architektúrája

Parancsok a nézetmodellben

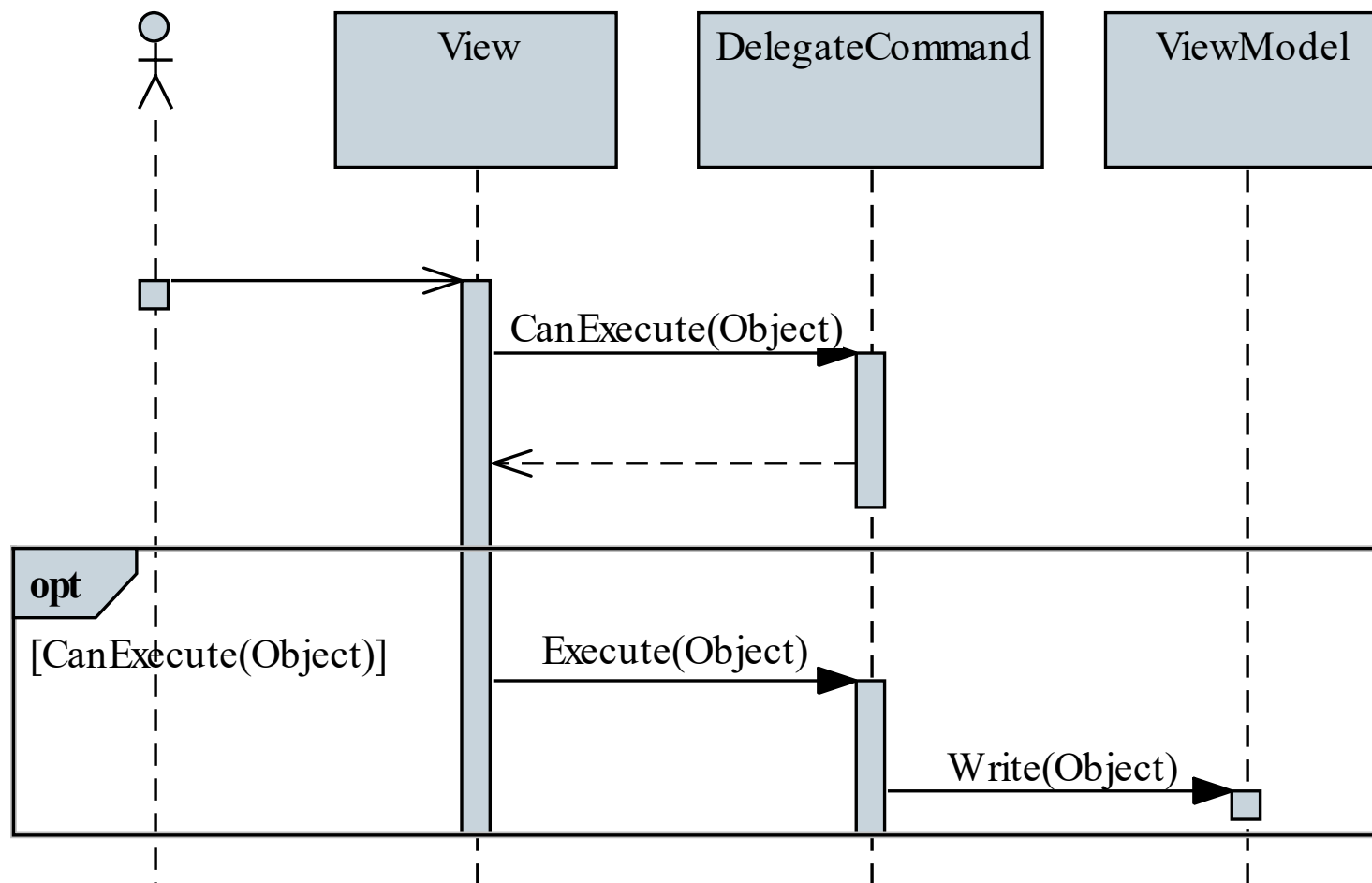
- Pl.:

```
public class MyViewModel
    : INotifyPropertyChanged // nézetmodell
{
    // parancs elhelyezése a nézetmodellben
    public DelegateCommand MyCommand { get; set; };

    public void Write(Object? parameter) {
        MessageBox.Show(parameter); // tevékenység
    }
    ...
    MyCommand = new DelegateCommand(x => Write(x));
    // tevékenység tényleges megadása
    ...
}
```

WPF alkalmazások architektúrája

Parancsok a nézetmodellben



WPF alkalmazások architektúrája

Parancsok végrehajthatósága

- A parancs bármikor jelezheti, hogy állapota megváltozott a **CanExecuteChanged** eseménnyel
 - amennyiben nem végrehajtható, a vezérlő kikapcsolt állapotba kerül
 - az eseményt egy megfelelő metódus segítségével válthatjuk ki (pl. **RaisePropertyChanged**), vagy automatizálhatjuk az állapotfigyelést a **CommandManager** osztály **RequerySuggested** statikus eseménye segítségével
 - automatikusan meghívja a rendszer, amikor beavatkozás szükségességét érzi (pl. ha valamilyen tevékenység fut a felületen)
 - az egyik eseményt elfedhetjük a másikkal, ehhez az esemény feliratkozását/leiratkozását kell megváltoztatnunk

WPF alkalmazások architektúrája

Példa

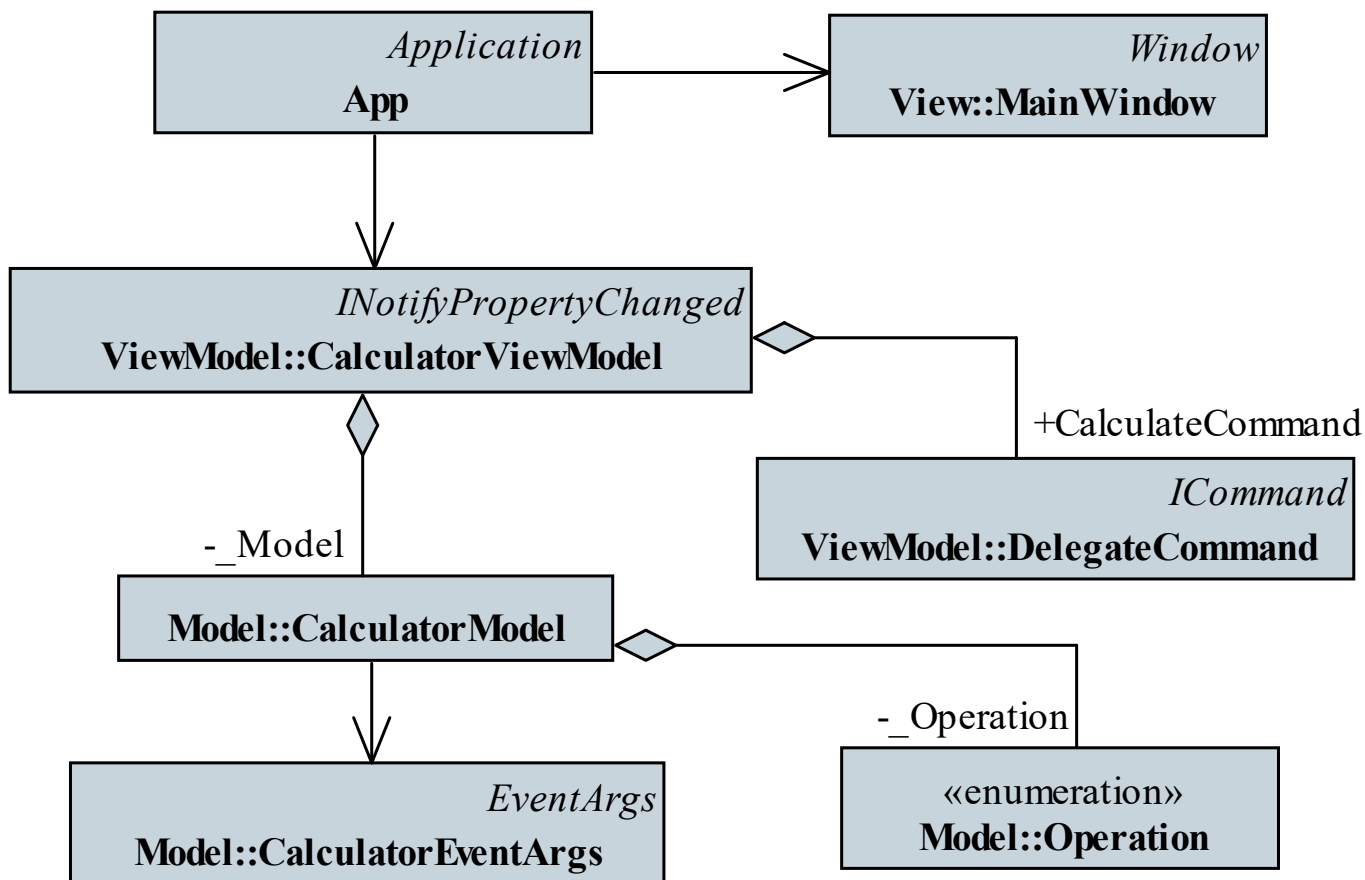
Feladat: Készítsünk egy egyszerű számológépet, amellyel a négy alapműveletet végezhetjük el, illetve láthatjuk korábbi műveleteinket is.

- az alkalmazást MVVM architektúrában valósítjuk meg
- a nézetmodell (**CalculatorViewModel**) tárolja a szöveges értéket (**NumberFieldValue**), az eddigi számításokat (**Calculations**), valamint a számítás parancsát (**CalculateCommand**), utóbbit egy általános parancsból (**DelegateCommand**) felépítve
- a nézetben (**MainWindow**) a végrehajtó gombokat társítjuk a parancshoz, amelynek paraméterként adjuk át a végrehajtandó műveletet

WPF alkalmazások architektúrája

Példa

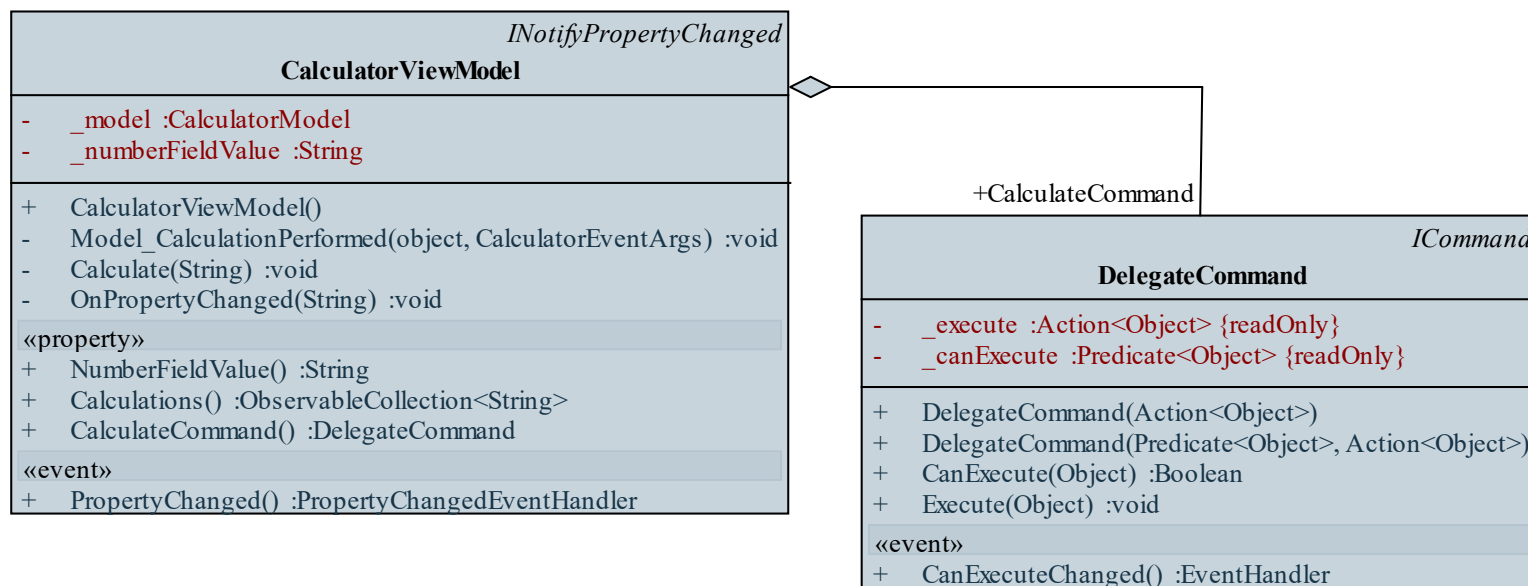
Tervezés:



WPF alkalmazások architektúrája

Példa

Tervezés:



WPF alkalmazások architektúrája

Példa

Megvalósítás (MainWindow.xaml):

```
<Grid>
```

```
    <TextBox Name="_textNumber" Height="42"
      VerticalAlignment="Top"
      Text="{Binding NumberFieldValue,
        UpdateSourceTrigger=PropertyChanged}"
      FontSize="28" TextAlignment="Right"
      FontWeight="Bold" />
```

```
    <!-- a szövegdobozhoz úgy kötjük a tartalmat,
      hogy minden módosításra mentsen -->
```

```
    <Button Command="{Binding CalculateCommand}"
      CommandParameter="+"
      Content="+" Height="60" ... />
```

```
...
```

WPF alkalmazások architektúrája

Példa

Megvalósítás (CalculatorViewModel.cs):

```
public CalculatorViewModel() {  
    CalculateCommand = new DelegateCommand(param =>  
        Calculate(param.ToString())) ;  
    ...  
}  
...  
private void Calculate(String operatorString) {  
    ...  
    switch (operatorString) {  
        case "+":  
            _model.Calculate(value, Operation.Add) ;  
            break ;  
        ...  
    }
```

WPF alkalmazások architektúrája

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 (**MouseButton**), vagy a kombinációt (**Gesture**)

- Pl.:

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

WPF alkalmazások architektúrája

Egyedi vezérlők

- Lehetőségünk van egyedi vezérlők létrehozására öröklődéssel, vagy létező vezérlők összeállításával felhasználói vezérlővé (**UserControl**)
 - vezérlőként felhasználható más vezérlőkben (külön adatkötései, erőforrásai lehetnek)
 - a saját vezérlőket névtér hivatkozáson keresztül érjük el, pl.:

<Window ...

xmlns="http://schemas.microsoft.com/winfx/..."

xmlns:view="clr-namespace:MyApp.View" ... >

<!-- megadjuk a névteret -->

<view:MyControl ... >

<!-- példányosítjuk az egyedi vezérlőt -->

</Window>

WPF alkalmazások architektúrája

Példa

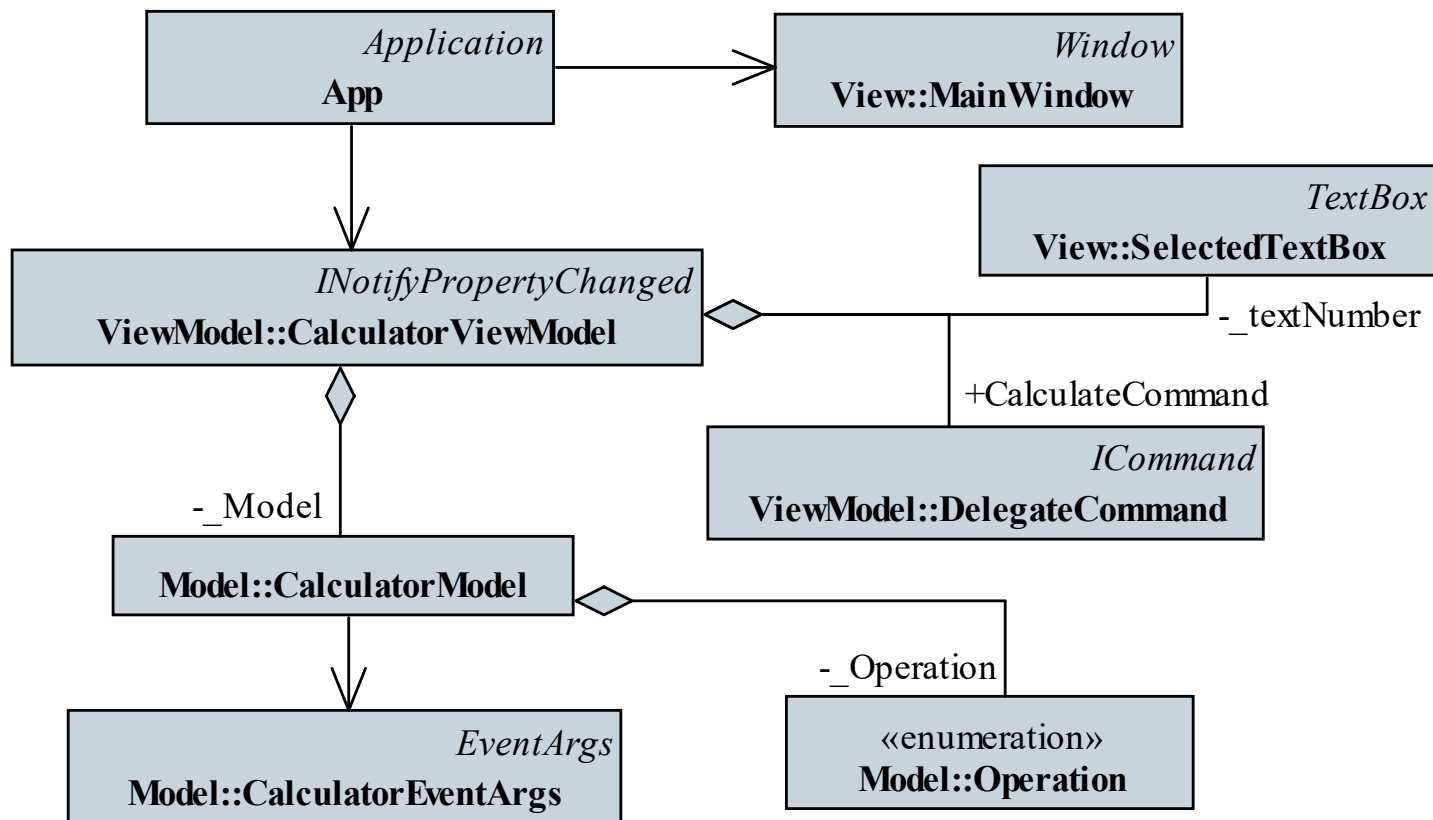
Feladat: Készítsünk egy egyszerű számológépet, amellyel a négy alapműveletet végezhetjük el, illetve láthatjuk korábbi műveleteinket is.

- lehessen a billentyűzetet is használni a műveletek megadásához
- a fókuszt automatikusan állítsuk a szövegdobozra (ehhez a nézetben használnunk kell a **FocusManager** osztályt)
- a szövegdoboz szövegét teljesen kijelöljük, ehhez felüldefiniáljuk a szövegdoboz billentyűzetkezelését
 - mivel ez nem végezhető el a nézetben, létrehozunk egy új vezérlőt a szövegdoboz leszármazottjaként (**SelectedTextBox**), amelyet felhasználunk a felületen

WPF alkalmazások architektúrája

Példa

Tervezés:



WPF alkalmazások architektúrája

Példa

Megvalósítás (MainWindow.xaml):

...

```
<Window.InputBindings>
```

```
  <!-- billentyűparancsok megfelelő  
    paraméterrel -->
```

```
  <KeyBinding Key="Enter" Command="{Binding  
    CalculateCommand}" CommandParameter="=" />
```

```
  <KeyBinding Key="Add" Command="{Binding  
    CalculateCommand}" CommandParameter="+" />
```

...

```
</Window.InputBindings>
```

...

WPF alkalmazások architektúrája

Példa

Megvalósítás (MainWindow.xaml):

```
<view:SelectedTextBox x:Name="_textNumber"
    Height="42" VerticalAlignment="Top"
    Text="{Binding NumberFieldValue,
        UpdateSourceTrigger=PropertyChanged}"
    FontSize="28" TextAlignment="Right"
    FontWeight="Bold" />

...

<Button Command="{Binding CalculateCommand}"
    CommandParameter="+"
    Content="+"
    FocusManager.FocusedElement="{Binding
        ElementName=_textNumber}" Height="60" ... />

...
```

WPF alkalmazások architektúrája

Példa

Megvalósítás (SelectedTextBox.cs):

```
private void SelectedTextBox_KeyUp(object sender,
                                   KeyEventArgs e) {
    switch (e.Key) {
        case Key.Add: // az akcióbillentyűkre
        case Key.Subtract:
        case Key.Enter:
        case Key.Multiply:
        case Key.Divide:
            SelectAll();
            // minden szöveget kijelölünk
            break;
    }
}
```

WPF alkalmazások architektúrája

Architektúra programcsomagok

- Az alapvető MVVM támogató konstrukciók a nyelvi könyvtárban nem elegendők a hatékony, gyors fejlesztésre
 - interfészek vannak (pl. **INotifyPropertyChanged**, **ICommand**), de nincsenek ösosztályok, gyűjtőosztályok
- Több olyan programcsomag került forgalomba, amely az MVVM alapú fejlesztést megtámogatja, pl.:
 - *MVVM Toolkit*: a *Microsoft .NET Community Toolkit* része, támogatja az MVVM architektúrát, a többrétegű modellt, komponensek közötti üzenetküldést, alkalmazás környezet kialakítását (korábbi nevén *MVVM Light Toolkit*)
 - *Prism Library*: támogatja a modul alapú fejlesztést, az MVVM architektúrákat, nézet-dekompozíciót és cserét