10. gyakorlat

C#/.NET alapú grafikus alkalmazás fejlesztés

A heti feladat egy egyszerű kód szerkesztő elkészítése. A felület az eddig megszokott *Windows Forms* keretrendszer helyett **Windows Presentation Foundation** keretrendszerrel készül; valamint az eddigi *Modell - Nézet - Perzisztencia* architektúrát felváltja az **MVVM: Modell - Nézet - Nézetmodell**. Természetesen a Perzisztenciát ismét külön rétegbe szervezzük.

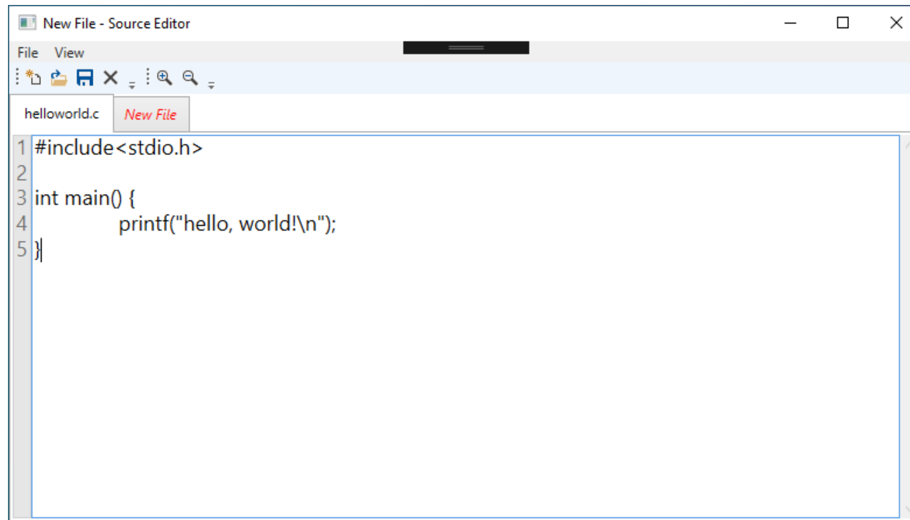

Figure 1: Kódszerkesztő alkalmazás

Funkcionalitás

- Sorszámozás – Miképp is mutatna egy magára valamit is adó kódszerkesztő nélküle?
- Betűméret csökkentése és növelése
- File betöltés és mentés
- File módosítás indikátor – Elmentettük-e már az aktuális állapotot?
- Eszköztár mutatása/elrejtése
- Több dokumentum egyidejű megnyitása lapok segítségével

1. Alapvető felület elkészítése

Készítsünk egy új *WPF App (.NET Core)* projektet Visual Studioban.

A főablak előre generált kódját a `MainWindow.xaml` állományban találjuk. Az állomány alapértelmezetten kód és vizuális nézetben egyszerre nyílik meg, ezt igény szerint tudjuk változtatni.

Tipp: eleinte ajánlom hogy az alapértelmezett "hibrid" nézetet használjuk. Az új elemeket grafikus felületen adjuk hozzá majd ismerkedjünk a generált kóddal.

A felületre először egy `DockPanel` elemet húzzunk! Látható, hogy a panel egyből kitöltötte a felületen a rendelkezésre álló teret. Ez az elem lesz a "konténer", ami az összes elem automatikus rendezéséért és méretezéséért felel majd.

Menu és eszközsáv

Ezután helyezzük el a `Menu` és `ToolBarTray` elemeket! Az elemek elrendezését elemenként a `DockPanel.Dock` mezőn tudjuk állítani.

A menübe `MenuItem`-eket, míg a `ToolBarTray`-be `ToolBar`-okat, azon belül `Button`-okat tudunk felvenni. A menübe következő elemeket vegyük fel, az eszközsávra ugyanezen funkcióknak megfelelő gombokat:

- File
  - New
  - Open
  - Save
  - Close
- View
  - Increase font size
  - Decrease font size

Opcionális: A menüben és az eszközsávban az akciókhoz rendeljünk képeket!

Tipp: Alapértelmezett felületi elemekhez a grafikákat érdemes mindig a platform fejlesztői tárházában keresni, így könnyedén készíthetünk a rendszer vizuális világába jól illeszkedő felhasználói felületet. Jelen esetben a Visual Studio Resource csomag áll rendelkezésünkre.

Szövegbevitel és sorszámozás

Helyezzünk fel egy `ScrollViewer` elemet, abba ismét egy `DockPanel` konténert, amibe további kettő `TextBox`-ot. A két szövegdoboz dokkolási irányát most bal értékre állítsuk, így a két elem egymás mellett helyezkedik majd el.

A két szövegdoboz sortörését kapcsoljuk ki, betűtípus családjukat állítsuk azonosan *Modern* értékre.

A baloldali szövegdoboz lesz a sorszám jelzőnk. Ezt a felületről nem szabad szerkeszteni, háttérszíne legyen semleges, betűszíne halvány! Legyen jobbra zárt!

A jobboldali szerkesztőben engedjük a tab és enter billentyűk használatát (ld. `AcceptsReturn`)!

2. Nézetmodell

Adjunk hozzá a projekthez a rendezettség kedvéért egy `ViewModels` mappát! Ahhoz, hogy a nézetmodellünk mezőit hozzá tudjuk rendelni a felületi elemekhez és azok változásairól a felület értesüljön, implementálnunk kell az `INotifyPropertyChanged` interfészt.

Annak érdekében, hogy ezt az implementációt ne kelljen minden nézetmodellben megtenni, készítsünk egy absztrakt `ViewModelBase` osztályt! Szükségünk lesz a `PropertyChanged` eseményre:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Valamint, egy `OnPropertyChanged` metódusra, mely biztonságosan hívja az eseményre feliratkozott eseménykezelőket:

```
protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
  if (PropertyChanged != null)
  {
    PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

Érdekesség: A `[CallerMemberName]` annotáció azt jelöli, hogy a megjelölt `string` paramétert a fordító tölti ki fordítás közben a függvényt aktuálisan közvetlen meghívó függvény nevével. Vigyázat, amikor nem az adott *property setter*-éből kerül hívásra, elveszhet az információ!

Készítsünk egy `EditorViewModel` osztályt, ami a `ViewModelBase` absztrakt osztályt terjeszti ki! Vegyünk fel egy-egy szöveges mezőt a két szövegdoboz tartalmának, valamint egy `double` mezőt, a betűméret számára (`CurrentText`, `LineNumbers`, `FontSize`)!

Nézetmodell és nézet összekapcsolása

Nyissuk meg az `App.xaml` állományt és töröljük a `MainWindow` indítására vonatkozó sort! Ezzel kikapcsoltuk a nézet automatikus indítását, lehetővé téve, hogy manuálisan vezéreljük.

Nyissuk meg az `App.xaml.cs` állományt, és osztály adattagnak vegyünk fel egy `_viewModel` és egy `_window` tagot megfelelő típussal!

Hozzuk létre az alábbi szignatúrájú `App_Startup` metódust!

```
void App_Startup(object sender, StartupEventArgs eventArgs)
```

A konstruktorban rendeljük az imént készített eseménykezelőt a `Startup` eseményhez!

Az eseménykezelőben hozzuk létre a nézet és nézetmodell tagokat és a nézet `DataContext` mezőjéhez rendeljük hozzá a nézetmodellt.

Nyissuk meg a nézetet! Rendeljük hozzá a nézetben a két mezőt a két szövegdoboz `Text` tagjához `{Binding CurrentText}` formában! Rendeljük a `FontSize` változót a szövegdobozok `FontSize` mezőihez!

A nézetmodellben az eddig felfett *property*-ket módosítsuk úgy, hogy mind egy-egy a *property*-vel azonos típusú privát adattag felett működjenek!

Figyeljünk oda, hogy a *property*-k *setter*-ei hívják az `OnPropertyChanged` metódust! A begépelt szöveget reprezentáló mező setterében számítsuk ki és frissítsük a sorszámokat tartalmazó mezőt!

Próbáljuk ki az eddigi alkotásunkat, és vegyük észre, hogy a sorszámok nem azonnal frissülnek!

Annak érdekében, hogy a `CurrentText` *property* minden módosításkor azonnal frissüljön, az alábbira módosítsuk a felületi szövegdoboz elem `Text` mezőjének értékét:

```
{Binding CurrentText, UpdateSourceTrigger=PropertyChanged}
```

Tipp: A nézet XAML kódja változtatható futtatás közben is. Apróbb változtatások kipróbálásához ajánlott futtatás közben kísérletezni.

Parancskötések

Ahhoz, hogy a felhasználói felület elemeihez akciókat rendeljünk, parancsokat (*command*) és parancskötést kell használnunk A parancsok az `ICommand` interfészt implementálják.

Készítsük el saját parancs implementációnkat, ez lesz a `DelegateCommand` osztály! (Azonos az előadás mintapéldákban szereplő `DelegateCommand` osztállya.)

Tipp: A Visual Studio segít nekünk az interface-k implementálásában. Amikor az osztályunk neve mellé írjuk az interface nevét, a "villanykörte" ikon segédlete felajánlja számunkra az implementációt, ezáltal legenerálva az interface által megkövetelt metódusok csonkjait.

Szükségünk lesz az alábbi adattagokra:

```
private readonly Action<Object> _execute;
private readonly Func<Object, Boolean> _canExecute;
```

Készítsük úgy el az implementációt, hogy ezekben az adattagokban tárolt függvény objektumokat hívja a `CanExecute` és az `Execute` tagfüggvény.

Érdemes egy két paraméteres konstruktort is készíteni, mely a két adattagot inicializálja.

Tipp: A `_canExecute` gyakorta `null` marad, mivel legtöbbször feltétel nélkül végrehajtandó akciókat reprezentálnak a parancsok. Érdemes a konstruktor paramétereit ennek megfelelően rendezni és a feltétel kiértékelésére hivatott paraméternek alapértelmezett értéket `null` adni.

Visszatérve a nézetmodellhez, vegyünk fel kettő `DelegateCommand` tagot, melyet konstruktorban inicializáljunk úgy, hogy a betűméretet ne lehessen csökkenteni, mikor már 1 értéken áll.

Ezt a két parancsot rendeljük a menü valamint az eszközsáv megfelelő elemeihez parancskötésen (*command binding*) keresztül!

3. Fájl mentése és betöltése

Eddig csak nézetünk és nézetmodellünk volt, most készítsünk egy modellt is (`EditorModel`), mely nyilvántartja majd a megnyitott fájljainkat (jelenleg még csak egy van), valamint interaktál a perzisztenciával és eseményeken keresztül küldi a frissítéseket a felület irányába.

A modell legyen alkalmas a fájl tartalmának tárolására, a fájl nevének és elérési útvonalának nyilvántartására! Az elérési útvonal alapértelmezetten legyen `null`, és legyen egy publikus függvény annak lekérésére, hogy tartozik-e már elérési útvonal a fájlhoz!

A modell publikáljon kettő eseményt:

```
public event EventHandler<FileOperationEventArgs> FileOpened;
public event EventHandler<FileOperationEventArgs> FileSaved;
```

Hozzuk létre a `FileOperationEventArgs` osztályt az `EventArgs`-ból származtatva, mely tartalmaz egy fájl elérési útvonalat és a fájl tartalmát!

A modell tartalmazzon egy publikus aszinkron eljárást a mentéshez a fájl tartalmával és opcionálisan az elérési útvonallal, mint paraméterekkel!

A modell tartalmazzon egy publikus aszinkron eljárást a betöltéshez, mely paraméterül kapja az elérési útvonalat!

Készítsünk egy `FilePersistence` osztályt, mely implementálja a betöltés és mentés aszinkron műveleteket, melyeket a model közvetlenül hívhat majd. A különböző hibák lekezelésére készítsünk egy `FileOperationException` `Exception`-ből származó osztályt!

A model a sikeres műveletek esetén küldje ki a megfelelő eseményt!

Vegyük fel az `App` osztályba a `_model` adattagot is, inicializáljuk, konstruktor paraméterben egy új perzisztencia objektumot átadva!

Vegyünk fel publikus eseménykezelőket a modell két eseményéhez az `EditorViewModel`-ben!

Az `App_Startup`-ban rendeljük a modell két eseményéhez az imént létrehozott két eseménykezelőt!

Most már nincs más hátra, mint a mentést és betöltést kiváltó felhasználói interakciók kezelése!

Mivel a nézetből nem szeretnénk közvetlenül hívni a modellt, így kettő eseményt veszünk fel a nézetmodellbe melyeket majd a gombokhoz rendelt `DelegateCommand`-ok fognak kiváltani:

```
public event EventHandler OpenFile;
public event EventHandler<FileOperationEventArgs> SaveFile;
```

Az `App` osztályban hozzunk létre eseménykezelőket a nézetmodell új eseményeire! Ezeket az eseménykezelőket az `App_Startup`-ban rendeljük az eseményeikhez!

Figyeljünk arra, hogy fájl mentésekor nem tudjuk, hogy a fájlnak létezik-e már elérési útvonala a modell-ben, így azt kérdezzük le a modelltől! Amennyiben nem létezik, `SaveFileDialog` segítségével kérjünk be egy elérési útvonalat a felhasználótól!

A betöltés kezelésénél az alkalmazás nyisson meg egy `OpenFileDialog` ablakot, kérje be a betölteni kívánt útvonalat, majd továbbítsa a kérést a modell felé!

Módosítás indikátor

Vegyünk fel egy `IsDirty` mezőt a nézetmodellbe! Ezt a *property*-t felhasználva a felületen több-féle képpen is jelezhetjük a fájl állapotát, például a címsorban egy csillag karakterrel.

Ez az indikátor segítségünkre lehet abban, hogy elkerüljük a felesleges mentéseket. Módosítsuk a mentés parancsot úgy, hogy csak akkor hajtódjon végre, ha az `IsDirty` igaz!

Módosítsuk a nézetmodell modell felől érkező mentés eseményt kezelő metódusát úgy, hogy frissítse az indikátort!

Kerüljük el a módosítások eldobását abban az esetben, ha mentetlen állapotra a felhasználó betöltést kér! Amennyiben a felhasználó szeretné a változásokat eldobni, legyen lehetősége tovább haladni! Ehhez a `FileOperationEventArgs` osztályba vegyünk fel egy indikátor *property*-t, melyben a nézetmodell továbbíthatja a betöltés eseménynél az aktuális fájl állapotát! Az `App` osztály eseménykezelője szükség esetén nyisson meg egy visszaigazolást kérő párbeszédablakot!

4. Több fájl megnyitása lapok használatával

Az `EditorViewModel` nézetmodell laponkénti adatait ábrázoló változóit szervezzük ki külön `TabViewModel` osztályba! A nézetben használjunk `TabControl` elemet, melynek `ItemsSource` mezőjét a nézetmodell egy `ObservableCollection`

tagjára állítsuk! Ez a kollekció tartalmazza a lapokat ábrázoló `TabViewModel` objektumokat!

Az `EditorModel` modell egy fájlra vonatkozó adattagjait szervezzük ki egy `FileModel` osztályba! A modell egy `Dictionary`-ben tárolja a fájlokat reprezentáló modelleket! A fájlok azonosítására (névütközés elkerülése és fájlnév változás támogatása végett) GUID típusú azonosítókat használjon! A fájl azonosítóit a `TabViewModel` valamint az események is tartalmazzák, az egyértelmű kommunikáció érdekében!

5. További funkciók

Implementáljuk az új fájl funkciót, mely egy üres fájlba, mentés nélkül kezd írni, majd igény szerint ment!

Implementáljuk a lap bezárása funkciót, mely megakadályozza a mentetlen munka eldobását (megerősítést kér), majd a modell-ből is kiveszi a fájlt!

Implementáljuk az ablak bezárása esetén történő módosítások elvesztése elleni védelmet! A figyelmeztetés párbeszédablakán a felhasználónak legyen lehetősége megszakítani a bezárást!

Használjunk billentyűkombinációkat a menüből elérhető funkciókhoz! A menüben tűntessük fel az ikonokat és a billentyűkombinációkat is! Példa:

```
<Window.InputBindings>
<KeyBinding Command="{Binding NewFileCommand}" Key="N" Modifiers="Ctrl"/>
</Window.InputBindings>
```

Megjegyzés

Az elkészített kódban figyeljük meg, hogy az eseményekkel való kommunikáció a nézet, nézetmodell és modell között egyirányú, "lentről-felfelé", minden réteg a közvetlenül alatt lévőt ismeri, és annak állapotára valamint változásaira hagyatkozhat.

A `ViewModelBase`, valamint a `DelegateCommand` osztályokat otthoni gyakorlás esetén érdemes lehet minden alkalommal saját kézzel implementálni, így közelebbi megértést szerezni a koncepcióról.