

# Eseményvezérelt alkalmazások: 7. gyakorlat

## 1 Több fájl megnyitása lapokkal

Egészítsük ki a meglévő `DocuStatView` projektet, hogy egyszerre több fájlt is be tudjon olvasni, és ezeket külön lapokon meg tudja jeleníteni!

### 1.1 UI kiszervezése

Első lépésként szervezzük ki a korábban létrehozott felhasználói felület menü alatti részét saját vezérlőbe. Adjunk hozzá a projekthez egy új `UserControl`-t, legyen ennek a neve `DocuStatControl`. A felület egyszerű reprodukálásához célszerű lehet a meglévő elemeket a `DocuStatDialog`-ról átmásolni és beilleszteni, így a korábban beállított property-k megmaradnak. Az így elkészült `DocuStatControl` a lentebbi ábrán látható.

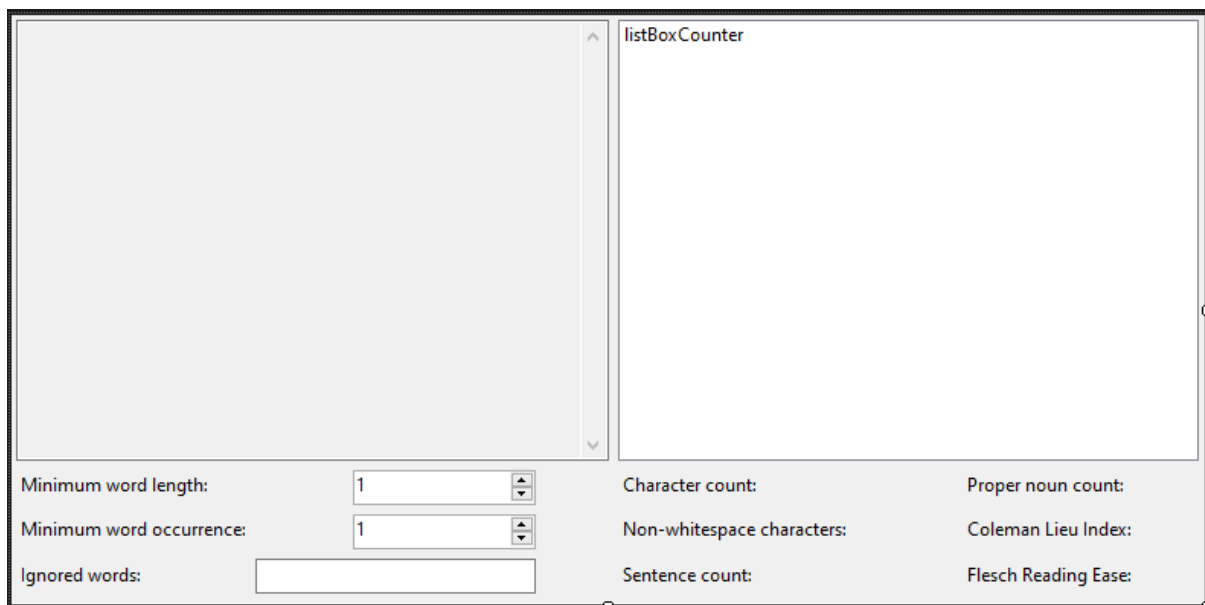


Figure 1: A létrehozott UserControl

Szervezzük ki a már meglévő felületet kezelő kódot is az alábbi osztálydiagramon látható módon.

- A `LoadFile` a kapott `IFileManager`-t felhasználva elvégzi a példányosítást és betöltést, majd az adatokat felhasználva frissíti a felületet. Itt ne végezzünk most hibakezelést, majd hívó oldalon fogjuk a kivételeket elkapni.
- Az `UpdateUI` feltölti a felületet a `_documentStatistics` adatait felhasználva. A korábbi `UpdateLabels`-hez képest ezt módosítsuk annyival, hogy itt végezzük el a `textBox` kitöltését is!
- A `CalculateStatistics` a korábbihoz hasonlóan most is töltse ki a jobb oldali listát.

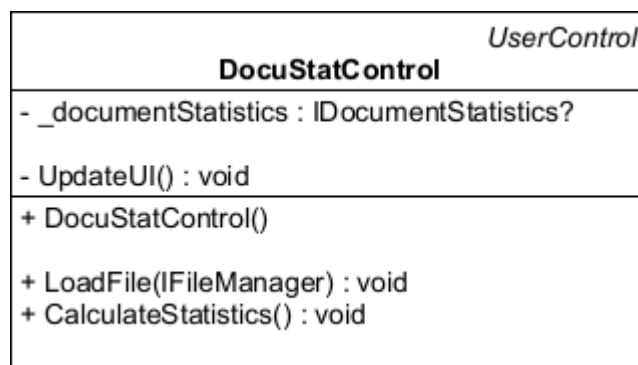


Figure 2: A DocuStatControl osztálydiagramja

## 1.2 Lapok létrehozása

Miután kiszerveztük a megjelenítést, töröljük le a `DocuStatDialog`-ról ezeket a vezérlőket, és vegyünk fel helyettük egy `TabControl`-t. A `(Name)` property legyen `tabControl`, a `Dock` legyen `Fill`, és töröljük az alapértelmezetten létrehozott `tabPage`-eket!

Kód oldalon módosítsuk az `OpenDialog` metódust, hogy az `OpenFileDialog` dialógus ablak `Multiselect` property-je igaz legyen, ez lehetővé teszi, hogy a felugró ablakban több fájlt is megadhassunk. Az így kapott útvonalakat a `FileNames` property-n keresztül érjük el (mint `string` tömb). A már megnyitott lapok egyszerű kezelése végett töröljük az eddigi tabokat a `tabControl.TabPages.Clear()` segítségével!

A lapok (*tabok*) létrehozásához vegyük fel a `AddTabPage` segédeljárást, ez a paraméterül kapott fájlvonalhoz létrehoz egy `IFileManager`-t, majd ezzel példányosít egy `DocuStatControl`-t és meghívja a `LoadFile`-t. Hozzunk itt létre egy `TabPage`-et is, aminek a `Controls` gyűjteményéhez adjuk hozzá a létrehozott `DocuStatControl`-t! A `TabPage`-nek célszerű azonosító címkét megadni, ezt akár a konstruktorában, akár a `Text` property-n keresztül tehetjük meg, ez lehet például a fájl neve, amit a kapott teljes útvonalból a `System.IO.Path.GetFileName` segítségével állíthatunk elő. Végezetül a `tabControl.TabPages` gyűjteményéhez a létrehozott `TabPage`-et hozzáadva megjelenik a felületen az új lap.

Itt kezeljük a korábbihoz hasonló módon a lehetséges kivételeket is, hiszen a `FileManagerFactory` és a `LoadFile` is hibát válthat ki, ebben az esetben térjünk is vissza, hogy ne kerüljön fel tényleges lap a felületre.

```
private void AddTabPage(string fileName)
{
    IFileManager? fileManager = FileManagerFactory.CreateForPath(fileName);
    if (fileManager == null)
    {
        // ...
        return;
    }

    try
    {
        DocuStatControl control = new DocuStatControl();
        control.LoadFile(fileManager);
        TabPage tabPage = new TabPage(System.IO.Path.GetFileName(fileName));
        tabPage.Controls.Add(control);
        tabControl.TabPages.Add(tabPage);
    }
    catch (FileManagerException ex)
    {
        // ...
        return;
    }
}
```

```
}

```

A `CalculateStatistics` eseménykezelőt alakítsuk át úgy, hogy az aktuálisan megnyitott laphoz (`SelectedTab`) tartozó `DocuStatControl` `CalculateStatistics` metódusát hívja meg, nem megfigyelve arról, hogy lehet, hogy nincsen lap kiválasztva, amit `SelectedTab == null` jelez. A `DocuStatControl` elővételéhez használjuk fel, hogy mi állítottuk össze a lapot, és tudjuk, hogy a `DocuStatControl` típusú elem az egyetlen vezérlő benne.

```
(tabControl.SelectedTab.Controls[0] as DocuStatControl).CalculateStatistics();

```

## 2 Párhuzamosítás

Az eddigi átalakítások után egyszerre több fájlt is meg tudunk nyitni, azonban ezek egymás után, sorrendben fognak betöltődni, és hosszú szöveg esetén a töltés alatt a felület is elveszíti a reszponzivitását, “lefagy”. A probléma orvoslására használjuk fel a C# aszinkron programozási lehetőségeit!

Az *async / await* minta lényege, hogy háttérszálakon el tudjuk párhuzamosan végezni az időigényes műveleteket *async* hívások segítségével, majd amikor szükségünk van az előállított eredményekre, az *await* operátor használatával meg tudjuk várni ezek befejeződését, mindezt olyan módon, hogy közben a hívást végző folyamat nem “blokkolódik”, tehát a felhasználói felület továbbra is kattintható és reszponzív marad.

A használatnak egy alapvető eleme a `Task<T>` osztály, ami egy `T` típusú eredményt előállító aszinkron folyamatot reprezentál (void esetén egyszerűen `Task`).

### 2.1 Perzisztencia és modell párhuzamosítása

A teljes alkalmazáslogikában a fájlok tartalmának betöltése az egyik szűk keresztmetszet hatékonyság szempontjából: egyrészt az I/O műveletek lassú volta miatt (memóriakezeléshez viszonyítva), másrészt a PDF fájlok feldolgozása is időigényes feladat lehet.

Alakítsuk át ezért először az `IFileManager` interfész és implementációinak `Load` metódusát. Az *async* kulcsszó az *await* operátor használatát engedélyezi a függvényen belül, a visszakapott `Task`-on keresztül pedig majd a művelet befejeződését tudjuk megvárni. A függvény aszinkron mivoltát elnevezési konvenció szerint *Async* utótaggal szokás jelezni. A metódus új szignatúrája legyen a következő (az interfészben az *async* kulcsszót ne írjuk ki):

```
public async Task<string> LoadAsync()
{
    // ...
}

```

- Szöveges (TXT) fájl feldolgozása esetén a `File` osztály már támogatja egy fájl tartalmának aszinkron betöltését, használhatjuk a `ReadAllTextAsync` eljárást ehhez.
- PDF fájlok esetén a felhasznált *iText7* szoftverkönyvtár nem rendelkezik aszinkron támogatással, de az erőforrásigényes oldalbetöltési műveletet egyszerűen azzá tehetjük, ha egy `Task`-ba foglaljuk. Ezt a `Task.Run` alkalmazásával egyszerűen megtehetjük, ez a paraméterül kapott lambda kifejezés végrehajtását elkezdi aszinkron módon, és visszatér az ezt reprezentáló `Task` objektummal.

```
return await Task.Run(() =>
{
    StringBuilder text = new StringBuilder();
    for (int i = 1; i <= document.GetNumberOfPages(); i++)
    {
        PdfPage page = document.GetPage(i);
        text.Append(PdfTextExtractor.GetTextFromPage(page));
    }
    return text.ToString();
});

```

*Megjegyzés:* az iText7 dokumentációja szerint egyszerre csak egyetlen szál férhet hozzá ugyanahhoz a PdfDocument objektumhoz. Ezért több oldal párhuzamos feldolgozásával nem foglalkozunk most.

A DocumentStatistics modell vonatkozásában a Load eljárást tegyünk aszinkronná, és kezeljük ennek megfelelően a perzisztenciát is. Mivel a szöveg metrikák előállítására számításigényes, nagyobb fájlokra akár több hosszabb időt (több másodpercet) is igénybe vehet, ezt is háttérfolyamatként, külön Task-ban futtassuk.

```
public async Task LoadAsync()
{
    FileContent = await _fileManager.LoadAsync();

    await Task.Run(() =>
    {
        // ...
    })
}
```

## 2.2 Nézet párhuzamosítása

A nézet rétegben minden DocuStatControl vezérlő saját modellel rendelkezik, amelyet be kell töltenie. Tegyük ezt aszinkron lehetővé, a LoadFile metódusának refaktorálásával:

```
public async Task LoadFileAsync(IFileManager fileManager)
{
    _documentStatistics = new DocumentStatistics(fileManager);
    await _documentStatistics.LoadAsync();
    UpdateUI();
}
```

Így már a DocuStatDialog ablak AddTabPage metódusát is átalakíthatjuk, hogy egy új DocuStatControl vezérlő felvétele aszinkron lehetséges legyen. A metódus új szignatúrája legyen a következő:

```
private async Task AddTabPageAsync(string fileName)
{
    // ...
}
```

A folyamatban a DocuStatControl vezérlő elkészítése és betöltésének párhuzamosított megkezdése után folytathatjuk a tablap összerakását, majd amikor már nincs más teendőnk, a kapott Task-ra await-et hívva megvárjuk a feladat befejezését. Magát a betöltés megkezdését és befejeztét jelezhetjük például a TabPage szövegének módosításával.

```
// ...
DocuStatControl control = new DocuStatControl();
Task loadTask = control.LoadFileAsync(fileManager);
TabPage tabPage = new TabPage("Loading...");
tabPage.Controls.Add(control);
tabControl.TabPages.Add(tabPage);

await loadTask;
tabPage.Text = System.IO.Path.GetFileName(fileName);
// ...
```

Legyen az AddTabPageAsync hívását végző OpenFileDialog is aszinkron! Ide is tegyük ki a private után az async kulcsszót, de a visszatérési érték maradjon void, mivel az eseménykezelő típusa megköveteli ezt.

Megtehetnénk, hogy egyszerűen await AddTabPageAsync(fileName) hívással megvárjuk az aktuális fájl betöltődését, és addig nem kezdjük el a többit, viszont ezzel továbbra sem használnánk ki a párhuzamosság adta lehetőségeket, ehelyett jobb megközelítés, ha egy List<Task> konténerbe először

eltesszük ezeket a feladatokat, ezzel elindítva mindet, majd utána várjuk be az összeset. Ez utóbbit végessük a `Task.WhenAll(tasks)` statikus módszerrel, ami visszaadja az összes kapott `Task` befejezettségét jelző `Task`-ot, ezt `await`-elve tudjuk megvárni az összes lap betöltését.

```
// ...
List<Task> tasks = new List<Task>();
foreach (string fileName in openFileDialog.FileNames)
{
    tasks.Add(AddTabPageAsync(fileName));
}
await Task.WhenAll(tasks);
// ...
```

Innentől kezdve a lapok betöltése közben továbbra is használható lesz a felület, például láthatjuk a már betöltött tabokat, viszont ebben a köztes állapotban az újabb betöltés indítása, vagy a `CalculateStatistics` hívása egy még befejezetlen tabon hibát okozhat. Emiatt az eseménykezelőkhöz tartozó menüpontokat (`openFileDialogMenuItem`, `countWordsMenuItem`) a betöltés idejére tegyük inaktívvá az `Enabled` property beállításával, majd pedig a végén kapcsoljuk ezeket vissza!

### 2.3 UI frissítés háttérszárlól

Ebben az állapotban a programot futtatva futási idejű kivételt kaphatunk, ugyanis az aszinkron metódusok (`async`) szinkronban futnak, amíg el nem érik az első várakozási kifejezést (`await`), utána ez már nem biztosított. A Windows Forms asztali grafikus keretrendszerben minden vezérlőt csak az őt létrehozó szál kezelhet, különben *cross-thread operation* miatti kivételt kaphatunk (`InvalidOperationException`).

- Ezt megoldandó minden `Control` objektum rendelkezik egy `Invoke` módszerrel, ami a paraméterül kapott lambda kifejezést a felületet birtokló szálon hajtja végre. Pl. a `DocuStatControl` esetében:

```
public Task LoadFileAsync(IFileManager fileManager)
{
    _documentStatistics = new DocumentStatistics(fileManager);
    await _documentStatistics.LoadAsync();
    Invoke(() => UpdateUI());
}
```

- Alternatívaként a UI frissítést láncolhatjuk újabb `Task`-ként a betöltés után a `ContinueWith()` eljárással. Ez esetben egy `TaskScheduler` objektummal végezhető el a szálak közötti szinkronizáció. A `FromCurrentSynchronizationContext` módszerrel egyszerűen kérhető az aktuális szárhoz egy `TaskScheduler` objektum:

```
public Task LoadFileAsync(IFileManager fileManager)
{
    _documentStatistics = new DocumentStatistics(fileManager);
    await _documentStatistics.LoadAsync()
        .ContinueWith(_ => UpdateUI(), _scheduler);
}
```

*Tipp:* mivel végső soron a tényleges megjelenítést muszáj ugyanannak a szálnak végeznie, ha például egy nagyon hosszú szöveges fájlt próbálunk betölteni, a teljes tartalom megjelenítése `TextBox`-ban hosszú időt igénybe tud venni, ami alatt az alkalmazás irreszponzív. Ezen minimális munkával egészen sokat tud gyorsítani, ha a szövegdoboz `WordWrap` property-jét hamisra állítjuk, ezzel nem számolódnak ki az automatikus sortörések.

*Megjegyzés:* A tabok betöltésének párhuzamosításához használhatjuk a `Parallel.Foreach` metódust is. Ebben az esetben minden ciklus kör egy új szálon fog futni. A `Parallel` osztály biztosítja számunkra, hogy az indított szálak mindegyike végrehajtsdjon, mielőtt a futás tovább folytatódik. Mivel a túl sok szál indítása szintén le tudja terhelni a CPU-t, érdemes korlátozni a párhuzamos szálak számát. Ezt `MaxDegreeOfParallelism` property beállításával tudjuk szabályozni.