Eseményvezérelt alkalmazások: 4. gyakorlat

A munkafüzet bevezet a háromrétegű grafikus alkalmazás fejlesztésbe, a perzisztencia leválasztásába, továbbá ezzel kapcsolatban szemlélteti a NuGet Package, a függőségi befecskendezés és a gyártófüggvények használatát. A perzisztencia felel az adatok tárolásáért egy perzisztens (hosszú távú) adattárban.

1 Perzisztencia réteg leválasztása ^{KM}

Készítsünk egy `Persistence` könyvtárat a `DocuStat` projekten belül.

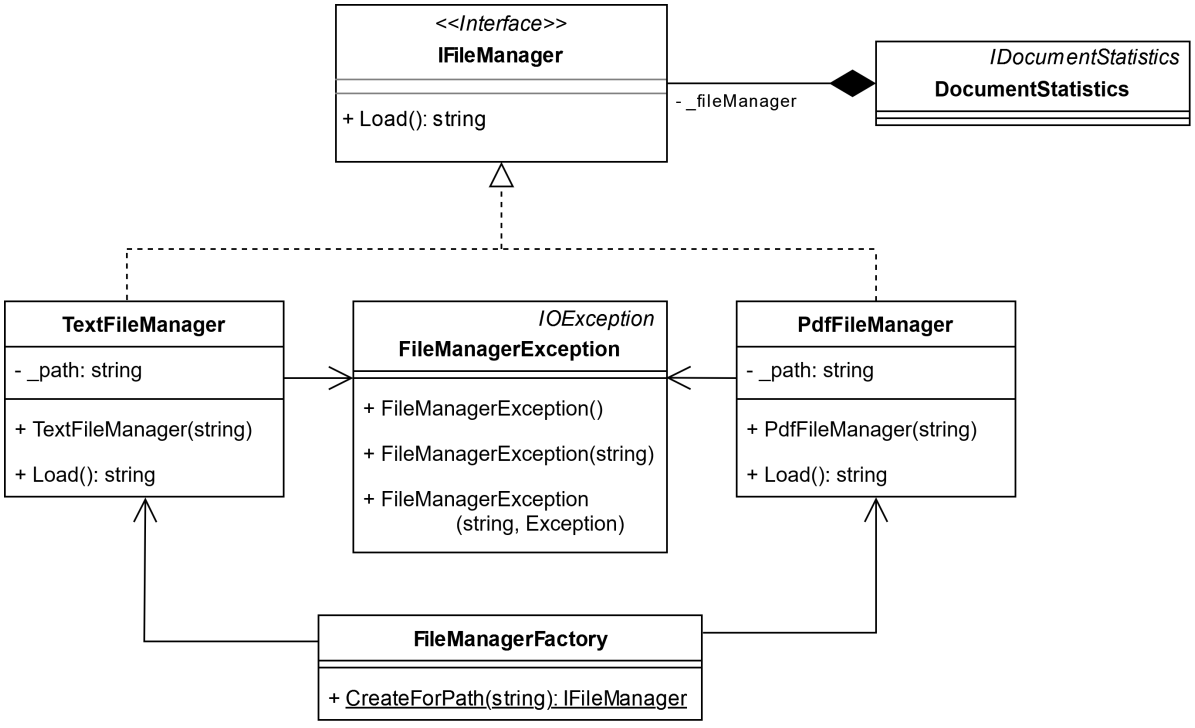

Figure 1: Perzisztencia osztálydiagramja

1.1 Szöveges fájlok kezelése

Hozzunk létre egy `Persistence` könyvtárat és benne az `IFileManager` interfészt, amely tartalmazzon egy `string Load()` metódust. Az interfészt valósítsa meg egy `TextFileManager` osztály, amely az elvárt metóduson felül még tartalmaz egy mezőt a fájl elérési útvonalával, amelyet a konstruktoron keresztül adhatunk meg. A `Load` függvény olvassa be az elérési útvonalhoz tartozó szöveges fájl tartalmát és adja vissza azt. Használhatjuk például a `System.IO.File.ReadAllText()` metódust.

Készítsünk egy saját kivételt `FileManagerException` névvel a `System.IO.IOException` osztályból származtatva, és ha a fájl beolvasása során hiba lép fel, akkor használjuk ezt a kivételt.

```
public class TextFileManager: IFileManager
{
    private readonly string _path;

    public TextFileManager(string path)
    {
        _path = path;
    }

    public string Load()
    {
        try
        {
            return File.ReadAllText(_path);
        }
        catch (Exception ex)
        {
            throw new FileManagerException(ex.Message, ex);
        }
    }
}
```

Az így létrehozott funkcionalitást függőségi befecskendezés (*Dependency Injection*) segítségével hivatkozzuk a `DocumentStatistics` osztályban: a konstruktor egy `IFileManager` típusú paramétert vár az elérési útvonal helyett. A megoldás előnye, hogy a modell réteg nem függ a perzisztencia konkrét megvalósításától, csak annak elvárt interfészétől.

```
private readonly IFileManager _fileManager;

public DocumentStatistics(IFileManager fileManager)
{
    _fileManager = fileManager;
    // ...
}
```

A `Load` függvényt alakítsuk át úgy, hogy a `IFileManager` interfész `Load` függvényét hívja. Továbbá, ne felejtsük el módosítani a `DocumentStatistics` felhasználási helyein módosítani a konstruktor-hívásokat és kezelni a `FileManagerException` kivételt.

1.2 PDF fájlok kezelése

Készítsünk egy `PdfFileManager` osztályt, amely szintén megvalósítja az interfészt, a fájl elérési útvonalát az előző implementációhoz hasonlóan a konstruktoron keresztül adjuk át és tároljuk el egy privát mezőben. A PDF fájlok kezeléséhez szükségünk lesz az `iText7` Nuget-csomagra. A projekthez tartozó csomagokat a *Project menü* > *Manage NuGet Packages...* menüpont alatt kezelhetjük.

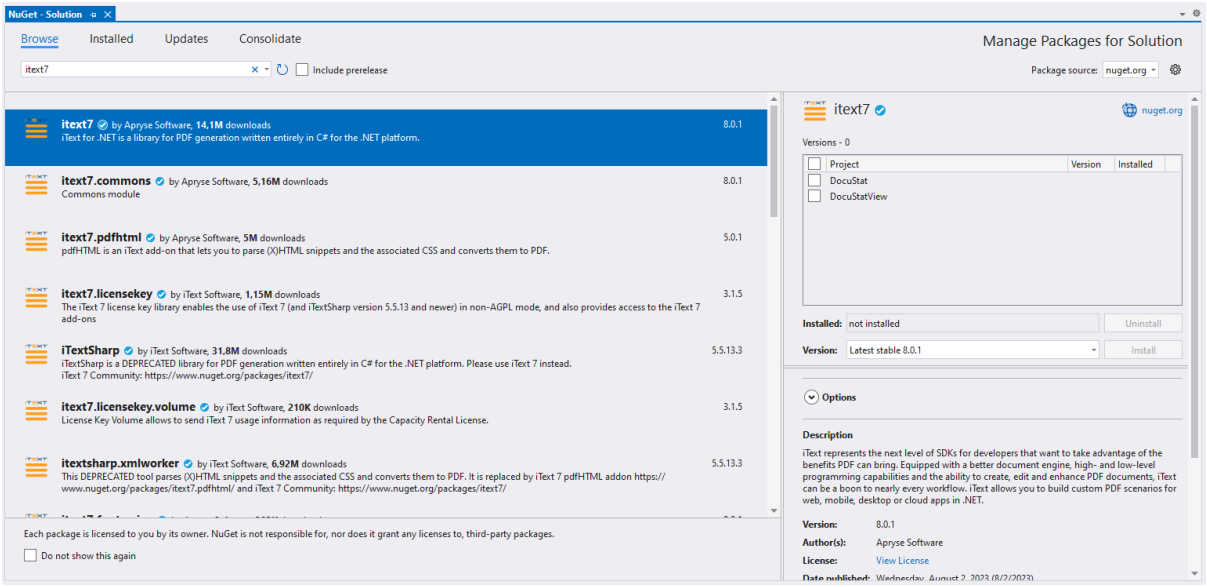

Figure 2: NuGet package keresése

1.2.1 Load metódus megvalósítása

A kiválasztott állományt a `iText.Kernel.Pdf` névtérben található osztályokkal fogjuk megnyitni és olvasni. Először a `PdfReader` osztályt példányosítsuk, amelynek konstruktora a fájl elérési útvonalát várja. Ezt követően példányosítsuk a `PdfDocument` osztályt és adjuk át a konstruktorának az imént létrehozott `PdfReader` példányt.

> Megjegyzés: mivel a `PdfReader` és `PdfDocument` osztályok megvalósítják az `IDisposable` interfészt ezért javasolt a `using` statement/declaration használata az osztályok által használt erőforrások felszabadítása érdekében a beolvasást követően (pl. megnyitott fájlok bezárása). További részletekért ld. a dokumentációt.

A dokumentumot oldalaként fogjuk feldolgozni, az oldalszámot a dokumentum `int GetNumberOfPages()`, az aktuális oldalt pedig a `PdfPage GetPage(int)` metódussal kérhetjük le. Egy oldalt a `iText.Kernel.Pdf.Canvas.Parser.PdfTextExtractor` osztály `string GetTextFromPage(PdfPage)` statikus metódusával tudunk karaktersorozattá konvertálni. Készítsünk egy for-ciklust amely bejárja az oldalakat (az indexelés egytől kezdődik!) és gyűjtsük össze az oldalak szövegeit a `System.Text.StringBuilder` osztály egy példányába.

> Megjegyzés: a C# stringek módosíthatatlan típusok, ezért minden módosító művelet egy új példány létrehozásával jár. Ha sok műveletet végzünk egy karaktersorozaton (pl. ciklusmagban), akkor javasolt a `StringBuilder` osztály használata. A `StringBuilder` a háttérben egy módosítható bufferben tárolja a karaktereket, így új példány létrehozása nélkül megváltoztatható a tartalma. Ha a karaktersorozatok összefűzését követően kevés az aktuális buffer mérete, akkor automatikusan létrehoz egy nagyobb buffert és átmásolja oda a régi tartalmát. Egy új stringet az `Append` metódus használatával tudunk az aktuális tartalomhoz hozzáfűzni. Hagyományos stringet pedig a `ToString()` metódus meghívásával készíthetünk a `StringBuilder` aktuális tartalmából. További részletekért ld. a dokumentációt.

```
using PdfReader reader = new PdfReader(_path);
using PdfDocument document = new PdfDocument(reader);

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();
```

Ne feledkezzünk meg a beolvasás során fellépő kivételeket kezeléséről. A szöveges fájlok beolvasásához hasonlóan váltsunk ki egy `FileManagerException` kivételt hiba esetén.

Végül teszteljük a megoldásunkat: a szöveges fájlokat kezelő osztály helyett fecskendezzük be a modellbe az imént elkészített osztály egy példányát és olvassunk be egy PDF állományt!

2 Megvalósítás kiválasztása fájlformátumnak megfelelően ^{EM}

Az elvégzett módosításokkal a modell módosítása nélkül kicserélhetjük a perzisztencia megvalósítását. Viszont, jó lenne ha az összes elkészített megvalósítás rendelkezésre állna és az aktuális fájl típusának megfelelőt választhatnánk ki. Az osztályok példányosítása előtt megvizsgálhatjuk a kiválasztott fájl kiterjesztését és ennek megfelelően választhatjuk ki, hogy melyik fájlkezelő implementációt adjuk át a modellnek. A kiterjesztés vizsgálatát érdemes kiszervezni egy külön osztályba a kódismétlés elkerülésének érdekében.

2.1 Factory osztály

A `Persistence` könyvtárban hozzunk létre egy `FileManagerFactory` statikus osztályt, amely egy publikus statikus metódussal rendelkezik: `public static IFileManager? CreateForPath(string path)`. A metódusban vizsgáljuk meg a fájl kiterjesztését (`System.IO.Path.GetExtension`) és ennek megfelelően példányosítsuk a korábban létrehozott fájlkezelő osztályok valamelyikét. Ha a megadott fájl kiterjesztését nem támogatja a programunk, akkor `null` értékkel térjünk vissza.

2.2 Konzolos alkalmazás módosítása (DocuStat projekt)

A `Program` osztály `Main` metódusában módosítsuk az elérési útvonal beolvasását:

```
string path;
IFileManager? fileManager = null;
do
{
    Console.Write("Please enter a valid text or pdf file path: ");
    path = Console.ReadLine() ?? "";
    if (System.IO.File.Exists(path))
    {
        fileManager = FileManagerFactory.CreateForPath(path);
    }
}
while (fileManager == null);
IDocumentStatistics stat = new DocumentStatistics(fileManager);
```

A beolvasást addig ismétli a program amíg a felhasználó egy létező fájlt ad meg egy támogatott fájlformátummal. A korábbi megoldással ellentétben nem szükséges ismernünk a támogatott fájlformátumokat: a `FileManagerFactory.CreateForPath` csak akkor fog nem-null értékkel visszatérni, ha a megadott fájl kiterjesztése támogatott.

2.3 Grafikus felületű alkalmazás módosítása (DocuStatView projekt)

A `DocuStatDialog` osztály `OpenDialog` metódusában kérjük le a dialógus által megnyitott fájlnak megfelelő implementációt és ellenőrizzük, hogy null értéket kaptunk-e vissza:

```
IFileManager? fileManager = FileManagerFactory.CreateForPath(openFileDialog.FileName);

if (fileManager == null)
{
    MessageBox.Show("File reading is unsuccessful!\nUnsupported file format.",
    "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    return;
}
```

A fájltallózó dialógus által támogatott formátumok közé pedig vegyük fel a `.pdf` állományokat is:

```
openFileDialog.Filter = "Text files (*.txt)|*.txt|PDF files (*.pdf)|*.pdf";
```

3 Kitekintés az IoC tárolókra ^{OP}

Összetettebb, sok függőség befecskendezését igénylő alkalmazásoknál a .NET keretrendszerre is bízhatjuk a példányosítást.

Az *IoC tároló* (*IoC container*) egy olyan *Inversion of Control* paradigmájú komponens, amely lehetőséget ad szolgáltatások megvalósításának dinamikus (futási idejű) betöltésére:

- egy központi regisztráció, amelyet minden programkomponens elérhet, és felhasználhat;
- a típusokat (elsősorban) interfész alapján azonosítja, és az interfészhez csatolja a megvalósító osztályt a tárolóba történő regisztrációkor megadjuk a szolgáltatás interfészét és megvalósításának típusát (vagy példányát);
- a szolgáltatást interfész alapján kérjük le, ekkor példányosul a szolgáltatás vagy kapunk egy már létező példányt;
- amennyiben a szolgáltatásnak függősége van, a tároló azt is példányosítja.

A .NET keretrendszerben az IoC tároló paradigmájára a `ServiceCollection` osztály ad implementációt, amelynek egy példányába mindkét interfészt (`IDocuemntStatistics`, `IFileManager`) és a hozzájuk tartozó implementációkat is regisztrálnunk kell. A szolgáltatások regisztrációja során meg kell adni azok élettartamát is:

- `AddTransient`: minden kérés alkalmával új példány jön létre az osztályból.
- `AddSingleton`: egy példány készül az osztályból, utána minden egyes alkalommal ugyanaz a példány adódik át.
- `AddScoped`: létrehozhatunk úgynevezett hatóköröket (scope), amelyeken belül minden kérésre ugyanazt a példányt adja át. Leggyakrabban ASP.NET Core webalkalamzások esetében találkozhatunk ezzel az élettartammal, ahol minden oldalbetöltéshez automatikusan létrejön egy hatókör.

Ha el akarjuk érni a szolgáltatásokat, akkor nem szükséges "manuálisan" példányosítanunk az osztályt amit használni szeretnénk, helyette a regisztrált osztályokat a `serviceProvider.GetRequiredService<T>()` függvény segítségével érhetjük el. Ilyenkor az adott osztály függőségei is automatikusan injektálásra kerülnek a konstruktoron keresztül.

Az egyszerűség kedvéért most csak a `DocuStat` projektben szemléltetjük a konténer használatát, a `DocuStatView`-ban nem szükséges implementálni.

```
static int Main(string[] args)
{
    string path;
    IFileManager? fileManager = null;
    // ...
    // Az elérési út beolvasása és fájlkezelő kiválasztása
    // ...

    ServiceCollection services = new ServiceCollection();
    // Regisztráció az interfész és az implementáció típusainak megadásával
    services.AddSingleton<IDocumentStatistics, DocumentStatistics>();
    // Regisztráció egy példány megadásával
    services.AddSingleton<IFileManager>(fileManager);
    using ServiceProvider serviceProvider = services.BuildServiceProvider();

    // Szolgáltatás lekérése
    IDocumentStatistics stat = serviceProvider.
        GetRequiredService<IDocumentStatistics>();

    // Számítások
    // ...
}
```