Szoftvertechnológia és-technikák
9. Gyakorlat

Tervezési minták 3

Command, Command Processor, Memento, Composite és Adapter minták

2020-

A gyakorlat menete

A gyakorlat során egy parancssori alkalmazást fogunk átalakítani és új funkciókkal kiegészíteni úgy, hogy az *Adapter*, *Command*, *Memento* és *Composite* minták használatát gyakoroljuk. A gyakorlat elején a laborvezető bemutatja a feladatot és átnézzük a kiinduló kódot, megbeszéljük milyen funkciókat szeretnénk megvalósítani.

A gyakorlat első fele vezetett: a cél, hogy először közösen beszéljük meg, hogyan tudjuk az egyes mintákat alkalmazni és miért előnyös a használatuk, ezután írjuk meg együtt a szükséges kódokat.

A gyakorlat második fele önálló munka. Itt is először megbeszéljük hogyan lehetne megvalósítani a kívánt funkciókat, de a megvalósítást önállóan kell megcsinálni.

Lehetséges, hogy a hallgatók még nem ismerik az XML és a JSON formátumokat. A laborhoz nincs is erre szükség, a labor vonatkozásában ezek csak különböző, szemmel is jól olvasható adatreprezentációs formátumok.

Megjegyzés: a kiinduló projektnek van egy külső, NuGet függősége. Objektumok JSON formátumba sorosítására a Newtonsoft.Json package-et használja. Ezt a Visual Studio solution megnyitása után az első build során tölti le hálózaton. A késleltetett letöltés miatt az első build sikertelen lehet, ekkor próbálkozzunk párszor tovább. Ha esetleg ez sem segítene, akkor a Solution Explorerben kattintsunk jobb gombbal a solutionön (legelső elem), és válasszuk ki a „Restore NuGet Packages” menüt.

0. Feladat A kiinduló állapot megismerése

A kiinduló állapot megismerése

A programról:

- A program egy egyszerű könyvadatbázis parancssori felülettel. Minden könyvnek van szerződje, ISBN száma és címe. Az ISBN számot nem lehet megváltoztatni.
- Az adatbázisba tudunk új könyveket felvinni, létező könyvek adatait (cím és szerző) megváltoztatni, a könyvadatbázist fájlba kimenteni, fájlból betölteni.
- Lehetőség van kilistázni az adatbázisban tárolt könyveket.

- Az adatbázismódosításokról bejegyzéseket (log) készítünk, amelyek kilistázhatók (megjegyzés: könyv törlésekor a logbejegyzései is törlődnek).
- A kezdeti alkalmazás még nem tud ilyet, de implementálni szeretnénk egy funkciót, amellyel az utolsó műveletek visszavonhatók.

Indítsuk el az alkalmazást:

- Egy parancssori felületet kapunk, ahol egysoros parancsokat tudunk kiadni.
- Írjuk be, hogy „súgó", így láthatjuk, milyen parancsok érhetők el.
- A TODO-val végződő parancsokon fogunk dolgozni a labor során.
- Nézzük meg az egyes műveleteket, amelyeket jelenleg ismer az alkalmazás (ezeket ki is lehet próbálni)

Példa parancs	**Leírás**
`súgó`	A súgó kiírása
`ment` *`valami.xml`*	Kimenti az adatbázist a valami.xml fájlba
`betölt` *`valami.xml`*	Betölti az adatbázist a valami.xml fájlból
`új` *`123`*	Létrehoz egy új könyvet az 123 ISBN számmal
`szerkeszt 123 cím Gyűrűk ura`	A 123-as ISBN számú könyv címét beállítja erre: „Gyűrűk ura"
`szerkeszt 123 szerző Tolkien`	A 123-as ISBN számú könyv szerzőjét beállítja erre: „Tolkien"
`kilép`	Kilép a programból
`lista`	Kiírja a könyvek listáját
`töröl 123`	Kitörli az adatbázisból az 123-as ISBN számú könyvet
`log`	Kiírja az összes logot

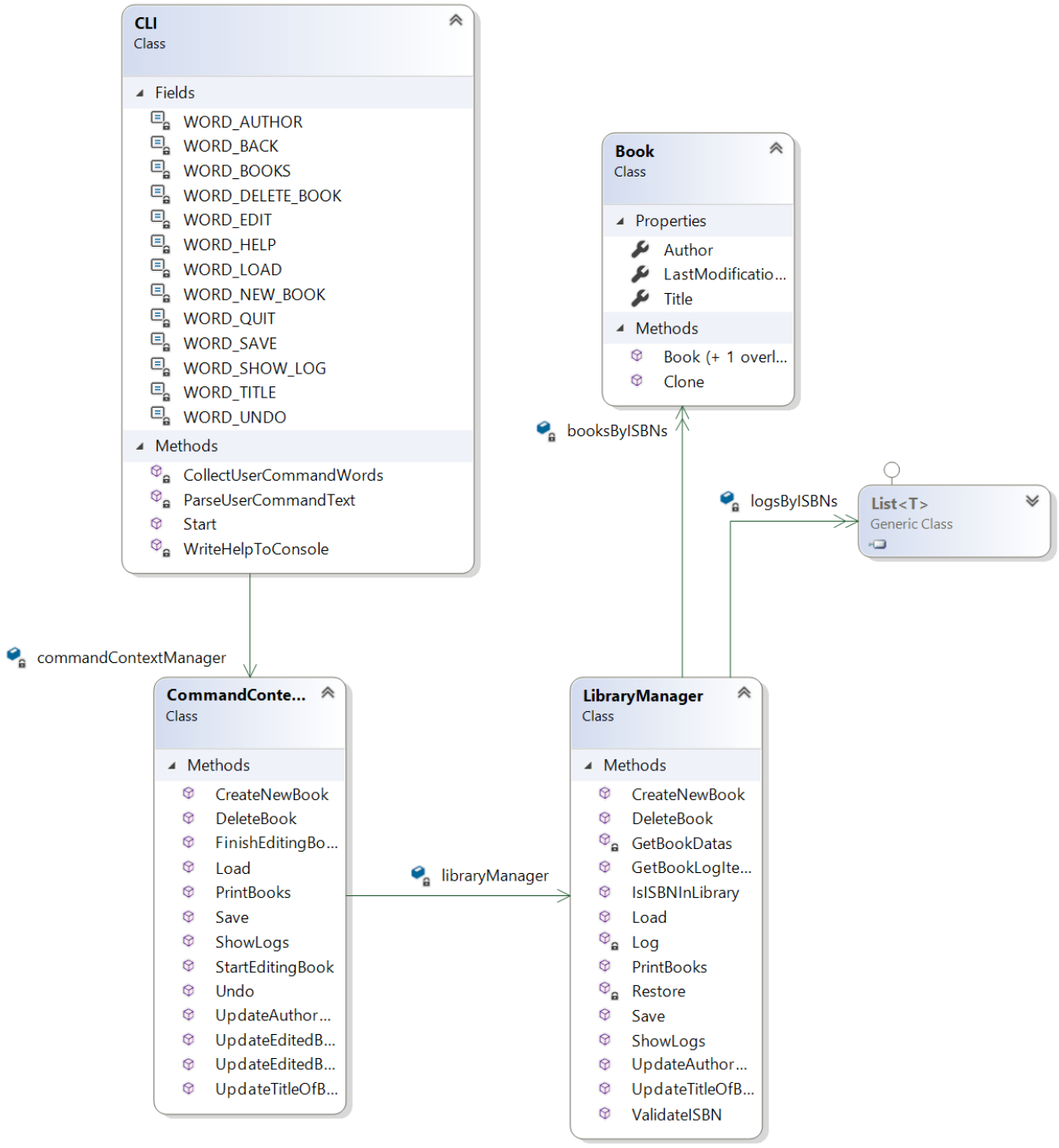
- A következő parancsok jelenleg nem működnek, de majd megvalósítjuk ezeket is:

Példa parancs	**Leírás**
`szerkeszt 123`	Összetett, kontextusfüggő szerkesztést indít (vagyis szerkeszti módba lépünk az 123 azonosítójú könyv vonatkozásában). Elkezdjük az 123-as ISBN számú könyv módosítását, ezután csak a következő parancsokat lehet kiadni, de azokból akármennyit.
`cím Gyűrűk ura`	Beállítjuk az aktuálisan szerkesztett könyv címét erre: „Gyűrűk ura"
`szerző Tolkien`	Beállítjuk az aktuálisan szerkesztett könyv szerzőjét erre: „Tolkien"
`visszavon`	az utolsó parancs visszavonása
`befejez`	Befejezzük az 123-as ISBN számú könyv szerkesztését.
`visszavon`	Visszavonjuk az utoljára kiadott parancsot. Amennyiben az egy összetett szerkesztés parancs volt, akkor annak összes alparancsát is visszavonjuk.

A kód alapvető felépítése (ezt nézzük végig közösen, beszéljük meg az egyes osztályok szerepét, hogy miért így épül fel az alkalmazás):

- Az alkalmazás belépési pontja a `Program.cs`-ben található `Main` függvény.
- A `Models` könyvtár tartalmazza az adatbázisban tárolt adatokat leíró modelleket:
    - `Book`: egy könyv címe és szerzője (az ISBN szám szándékosan nincs itt, ld. később)
    - `BookLogItem`: bejegyzés egy könyv egy módosításáról. (Csak a dátumot és a módosítást leíró szöveget tároljuk.)
    - A fenti osztályokban hivatkozzuk a `BookData` és `BookLogItemData` osztályokat, ezekről majd később.
- `LibraryManager`:
    - Ez maga az adatbázis, itt tároljuk a könyveket és ezen keresztül módosítjuk az adatbázist.
    - A könyveket és a logokat egy-egy `Dictionary`-ben tároljuk (`booksByISBNs`, `logsByISBNs`), itt szerepelnek az ISBN számok, amik az előző modell osztályokban nem voltak benne.
    - Az adatbázison végrehajtandó műveleteknek megfelel egy-egy metódus: pl. `Save`, `Load`, `CreateNewBook`, `UpdateTitleOfBook` stb, érdemes ezeket végignézni.
    - Van néhány segédfüggvény: `IsISBNInLibrary`, `ValidateISBN` (az ISBN számok csak kötőjelet és számokat tartalmazhatnak), `restoreData`.
- Az adatbázis fájlba mentéséért és betöltéséért a `FileManager` osztály felelős. Ez reprezentálja az adatelérési réteget, ezért nem a modell osztályokat sorosítjuk közvetlenül, hanem az üzleti logika (`LibraryManager`) és az adatelérési között bevezetünk új osztályokat: `BookData`, `BookLogItemData`, illetve `LibraryData`.
    - Amikor kimentjük az adatbázist, létrehozunk egy `LibraryData` példányt, majd az adatbázis tartalmát átkonvertáljuk `BookData` és `BookLogItemData` objektumokká. A `FileManager` feladata a `LibraryData` sorosítása.
    - Sorosítás: erről csak annyit kell tudni, hogy a .NET támogatja memóriabeli objektumok adatfolyamba (pl. fájlba) írását és adatfolyamból (pl. fájlból való betöltését). Mindezt különböző adatformátumokra (XML, JSON, stb.). Ez sokkal egyszerűbb, mintha nekünk kézzel kellene az objektumok tagváltozóit egyesével kiírni és beolvasni.
    - Hasonlóan működik a visszaolvasás is. Érdemes az idevágó kódokat átnézni.
    - Amennyiben mentésnél XML fájlt adunk meg, akkor a `LibraryXmlSerializer` osztály segítségével XML szöveggé sorosítjuk a könyvtárat és azt mentjük ki a fájlba.
- A `CLI` osztály felelős a parancssori felületért: beolvassa a felhasználói parancsokat és ennek kellene meghívnia a `LibraryManager` megfelelő metódusait. A `CLI` tehát kizárólag a beolvasott szövegek értelmezésével foglalkozik, ehhez az osztályhoz nem kell nyúlni az órai munka során, csak a teljesség kedvéért van itt a kódja.
- Valójában a `CLI` nem közvetlenül a `LibraryManager`-t hívja meg, hanem bevezetünk egy köztes réteget: `CommandContextManager`. Mivel például az összetett szerkesztés parancs megvalósításához el kell tárolnunk bizonyos állapotinformációkat, ezért vezetjük be ezt a réteget.

Fontos, hogy a hallgatók átlássák az egész kódot, értsék, hogy miért így épül fel. A megértést az alábbi osztálydiagram is segítheti (benne van a kiinduló solution-ben is):



1. Feladat: összetett szerkesztés művelet (szerkesztési mód)

Valósítsuk meg az összetett szerkeszt műveletet, amikor egy paranccsal kijelölünk egy könyvet (pl. „`szerkeszt 123`”), majd azt tudjuk ezután további parancsokkal szerkeszteni, pl. „`cím Gyűrűk ura`”, „`szerző Tolkien`”, „`visszavon`”, „`befejez`” (ez utóbbi befejezi az adott könyv szerkesztését).

Ehhez szükségünk lesz állapotkezelésre a `CommandContextManager`-ben:

Hozzunk létre néhány egy tagváltozót és egy property-t:

```
private string editedISBN = null;
private bool IsInEditBookMode
{
    get { return !string.IsNullOrEmpty(editedISBN); }
}
```

Módosítsuk a Save metódust, hogy amennyiben *Edit* módban vagyunk, akkor ne lehessen menteni:

```
public void Save(string path)
{
    if (!checkIfInNormalMode()) return;

    libraryManager.Save(path);
}

private bool checkIfInNormalMode()
{
    if (IsInEditBookMode)
    {
        Console.WriteLine("Előbb fejezd be az aktuális könyv szerkesztését");
        return false;
    }
    return true;
}
```

Mint látható, bevezettünk egy checkIfInNormalMode segédfüggvényt: ezt azért tettük, mert a validációra több más függvényben is szükség van. Szúrjuk is be a

```
if (!checkIfInNormalMode()) return;
```

hívást az alábbi függvények elejére: Load, CreateNewBook, DeleteBook, UpdateTitleOfBook, UpdateAuthorOfBook, StartEditingBook.

Ezt követően be tudjuk fejezni a szerkeszt parancs hatására hívott StartEditingBook függvényt:

```
public void StartEditingBook(string isbn)
{
    if (!checkIfInNormalMode()) return;

    if (!libraryManager.IsISBNInLibrary(isbn))
    {
        Console.WriteLine("Ilyen ISBN számú könyv nincs az adatbázisban");
        return;
    }

    editedISBN = isbn;
}
```

Következő lépésben a befejez parancs hatására hívott FinishEditingBook függvényt írjuk meg. Ehhez bevezetünk egy checkIfInEditMode segédfüggvény, ugyanis annak validációját, hogy szerkesztési módban vagyunk-e, több helyről is meg fogjuk a későbbiekben tenni:

```
public void FinishEditingBook()
{
    if (!checkIfInEditMode()) return;

    editedISBN = null;
}

private bool checkIfInEditMode()
{
    if (!IsInEditBookMode)
    {
        Console.WriteLine("Előbb válassz ki egy könyvet szerkesztésre");
        return false;
    }
    return true;
}
```

Végezetül írjuk meg az `UpdateEditedBookTitle` és `UpdateEditedBookAuthor` metódusokat (ezeket hívjuk meg, amikor szerkesztési módban adjuk ki a `cím`, vagy `szerző` parancsokat)!

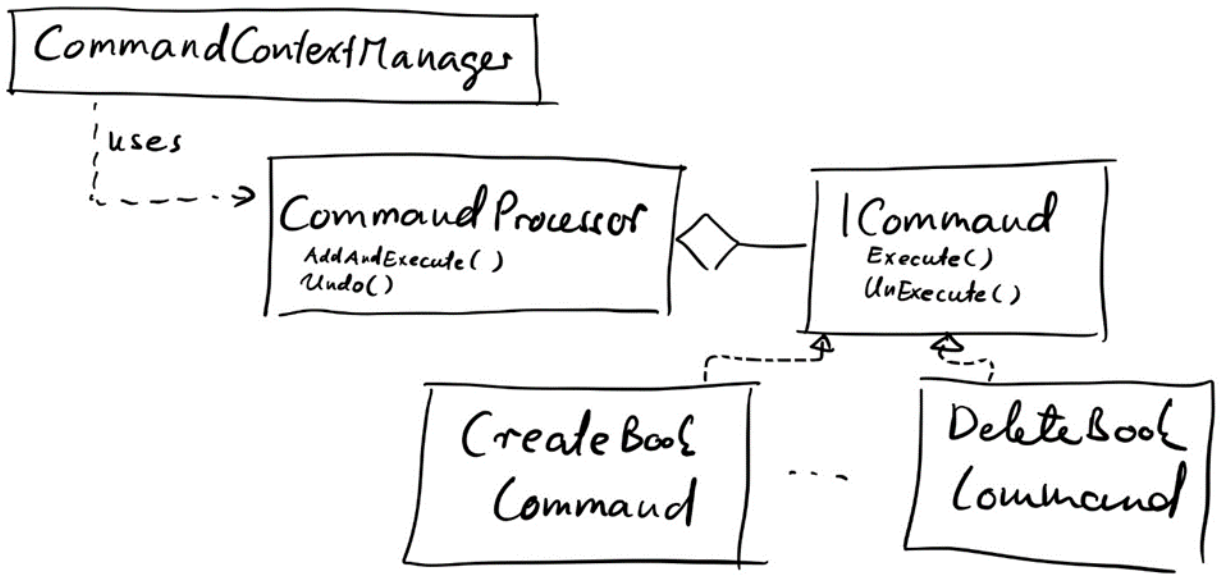
```
public void UpdateEditedBookTitle(string title)
{
    if (!checkIfInEditMode()) return;

    libraryManager.UpdateTitleOfBook(editedISBN, title);
}
```

```
public void UpdateEditedBookAuthor(string author)
{
    if (!checkIfInEditMode()) return;

    libraryManager.UpdateAuthorOfBook(editedISBN, author);
}
```

Futtassuk az alkalmazást, próbáljuk ki az összetett szerkesztés műveletet: vegyünk fel egy új könyvet (`új` parancs), lépjünk összetett szerkesztés módba (`szerkeszt`), módosítsuk a könyv szerzőjét (`szerző` parancs) és címét (`cím` parancs), majd lépjünk ki az összetett szerkesztés üzemmódból (`befejez`).

2. Feladat: Commandok bevezetése

A célunk az undo (visszavonás) funkció megvalósítása. Beszéljük meg a *Command*, pontosabban a *Command Processor* mintát. Itt azért fogjuk használni, hogy könnyen megvalósítható legyen az undo/redo funkció.



Vezessük be az `ICommand` interfészt! Hozzunk létre egy `Commands` mappát és abba dolgozzunk, `Commands/ICommand.cs`:

```
public interface ICommand
{
    void Execute();
    void UnExecute();
}
```

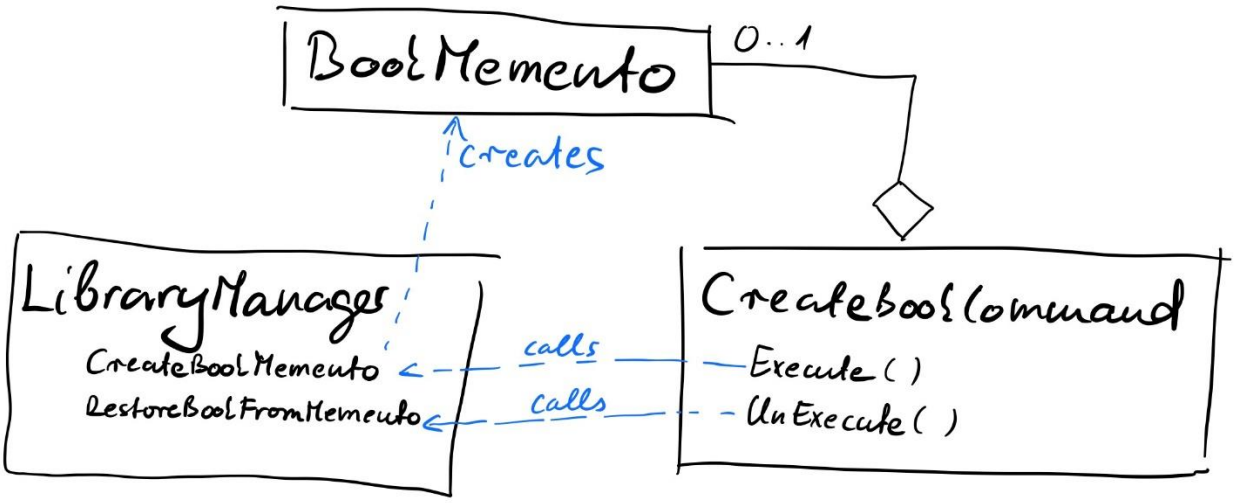
Szükség lesz egy, a commandokat tároló osztályra: `Commands/CommandProcessor.cs`

```
class CommandProcessor
{
    private Stack<ICommand> commands = new Stack<ICommand>();

    public void AddAndExecute(ICommand command)
    {
        commands.Push(command);
        command.Execute();
    }

    public void Undo()
    {
        if (commands.Any())
        {
            var command = commands.Pop();
            command.UnExecute();
        }
    }
}
```

Valósítsuk meg az új könyv létrehozását végző commandot: `Commands/CreateBookCommand.cs`

```
public class CreateBookCommand : ICommand
{
    protected LibraryManager library;
    protected string isbn;
    public CreateBookCommand(LibraryManager library, string isbn)
    {
       this.library = library;
       this.isbn = isbn;
    }
    public void Execute()
    {
        library.CreateNewBook(isbn);
    }

    public void UnExecute()
    {
        library.DeleteBook(isbn);
    }
}
```

Mi lesz, ha a könyv törléshez szeretnénk commandot létrehozni? Az lesz a gond, hogy az UnExecute-nál újra létre kell hozni majd a könyvet. Elő tudjuk állítani az eredeti könyvet, de mások lesznek a log bejegyzések, hiszen azokat nem mi szúrjuk be, hanem a `LogManager`. Erre a problémára a megoldás a *Memento* minta használata.



Pár gondolatvezető a Memento minta vonatkozásában (részletesen lásd az előadás anyagában):

- Esetünkben a logok a `LibraryManager`-ben védettek (private). De ha védett, nem férhetünk hozzá kívülről, pedig most erre lenne szükség, hiszen el kell menteni, majd a visszavonás során vissza kell állatani.
- Nem tesszük publikussá csak a visszavonás parancs támogatása miatt, sérülne az egységbezárás. Helyette a Memento mintát alkalmazzuk.
- A LibraryManager védett állapotát (logok) egy Memento objektumba csomagolva tesszük lekérdezhetővé (`CreateBookMemento`) és visszaállíthatóvá (`RestoreBookFromMemento`). A Memento egy objektum - esetünkben a LibraryManager - adott időpontbeli azon állapotát tárolja, melyet később vissza akarhatunk állítani, esetünkben a visszavonás parancs hatására.

Hozzunk létre egy Memento osztályt: `Memento/BookMemento.cs`

```
public class BookMemento
{
    public BookMemento(Book book, List<BookLogItem> logItems, string isbn)
    {
        Book = book;
        LogItems = logItems;
        ISBN = isbn;
    }

    public Book Book { get;set; }
    public List<BookLogItem> LogItems { get; set; }
    public string ISBN { get;set; }
}
```

Ne feledkezzünk meg róla, hogy alapból nem publikus (public) osztályt hoz létre a VS, igy ne felejtsük el azzá tenni, különben a következő metódusok láthatósági hibát fognak jelezni!

Egészítsük ki a `LibraryManager`t úgy, hogy tudjon mementót készíteni, illetve mementó alapján vissza tudja állítani az állapotot!

`LibraryManager.cs`

```
public BookMemento CreateBookMemento(string isbn)
{
    return new BookMemento(
        booksByISBNs.ContainsKey(isbn) ? booksByISBNs[isbn].Clone() : null,
        logsByISBNs.ContainsKey(isbn) ? new List<BookLogItem>(logsByISBNs[isbn]) : null,
        isbn
    );
}

public void RestoreBookFromMemento(BookMemento memento)
{
    if (memento.Book == null && booksByISBNs.ContainsKey(memento.ISBN))
        booksByISBNs.Remove(memento.ISBN);
    else
        booksByISBNs[memento.ISBN] = memento.Book;

    if (memento.LogItems == null && logsByISBNs.ContainsKey(memento.ISBN))
        logsByISBNs.Remove(memento.ISBN);
    else
        logsByISBNs[memento.ISBN] = memento.LogItems;
}
```

Fontos, hogy a kontruktorban ne csak a referenciákról készítsünk másolatot, hanem magukról az objektumokról is, ha azok megváltoztathatók:

- A `Book` objektum megváltoztatható, erről a `Clone` művelettel másolatot készítünk
- A `LogItems` lista megváltoztatható, így a listából újat hozunk létre a korábbi lista elemeivel inicializálva. Viszont a `BookLogItem` elemek nem megváltoztathatók (readonly tagjai vannak), így az elemeket nem kell klónozni.

Alakítsuk át a korábbi `CreateBookCommand` osztályt úgy, hogy a mementót használja:

```
public class CreateBookCommand : ICommand
{
    protected BookMemento memento;
    protected LibraryManager library;
    protected string isbn;
    public CreateBookCommand(LibraryManager library, string isbn)
    {
       this.library = library;
       this.isbn = isbn;
    }
    public void Execute()
    {
        memento = library.CreateBookMemento(isbn);
        library.CreateNewBook(isbn);
    }

    public void UnExecute()
    {
        library.RestoreBookFromMemento(memento);
    }
}
```

Még mielőtt megírnánk a többi commandot: a többi commandnál a fenti kódhoz nagyon hasonló osztályokra lesz szükség, csak az Execute második sora fog hiányozni, ezért érdemes egy ősosztályt készíteni:

```
public abstract class BookCommand : ICommand
{
    protected BookMemento memento;
    protected LibraryManager library;
    protected string isbn;
    protected BookCommand(LibraryManager library, string isbn)
    {
        this.library = library;
        this.isbn = isbn;
    }

    public virtual void Execute()
    {
        memento = library.CreateBookMemento(isbn);
    }

    public virtual void UnExecute()
    {
        library.RestoreBookFromMemento(memento);
    }
}
```

Beszéljük át az `abstract`, `virtual` kulcsszavakat!

Ismét írjuk át a `CreateBookCommand`-ot, hogy a fenti ősosztályból származzon!

```
public class CreateBookCommand : BookCommand
{
    public CreateBookCommand(LibraryManager library, string isbn) : base(library, isbn)
    {
    }
    public override void Execute()
    {
        base.Execute();
        library.CreateNewBook(isbn);
    }
}
```

Ezután írjuk meg a többi commandot! (*Esetleg lehet önálló munka*.)

Commands/DeleteBookCommand.cs

```
public class DeleteBookCommand : BookCommand
{
    public DeleteBookCommand(LibraryManager library, string isbn) : base(library, isbn)
    {
    }
    public override void Execute()
    {
        base.Execute();
        library.DeleteBook(isbn);
    }
}
```

Commands/UpdateAuthorCommand.cs

```
class UpdateAuthorCommand : BookCommand
{
    private string author;
    public UpdateAuthorCommand(LibraryManager library, string isbn, string author) :
base(library, isbn)
    {
        this.author = author;
    }

    public override void Execute()
    {
        base.Execute();
        library.UpdateAuthorOfBook(isbn, author);
    }
}
```

Commands/UpdateTitleCommand.cs

```
class UpdateTitleCommand : BookCommand
{
    private string title;
    public UpdateTitleCommand(LibraryManager library, string isbn, string title) :
base(library, isbn)
    {
        this.title = title;
    }

    public override void Execute()
    {
        base.Execute();
        library.UpdateTitleOfBook(isbn, title);
    }
}
```

Ezután a CommandContextManager-be fel kell venni egy CommandProcessor tagot, majd át kell átalakítani úgy a CommandContextManager-t, hogy a commandokat használja:

```
public class CommandContextManager
{
    private readonly CommandProcessor commandProcessor = new CommandProcessor();
    …

}
```

```
public void CreateNewBook(string isbn)
{
    …

    //libraryManager.CreateNewBook(isbn);
    commandProcessor.AddAndExecute(new CreateBookCommand(libraryManager, isbn));
}
```

```
public void UpdateTitleOfBook(string isbn, string title)
{
    …

    //libraryManager.UpdateTitleOfBook(isbn, title);
    commandProcessor.AddAndExecute(new UpdateTitleCommand(libraryManager, isbn, title));
}
```

```
public void UpdateAuthorOfBook(string isbn, string author)
{
    …

    //libraryManager.UpdateAuthorOfBook(isbn, author);
    commandProcessor.AddAndExecute(new UpdateAuthorCommand(libraryManager, isbn,
author));
}
```

```
public void UpdateEditedBookTitle(string title)
{
    …

    //libraryManager.UpdateTitleOfBook(editedISBN, title);
    commandProcessor.AddAndExecute(new UpdateTitleCommand(libraryManager, editedISBN,
title));
}
```

```
public void UpdateEditedBookAuthor(string author)
{
    …

    //libraryManager.UpdateAuthorOfBook(editedISBN, author);
    commandProcessor.AddAndExecute(new UpdateAuthorCommand(libraryManager, editedISBN,
author));
}
```

```
public void DeleteBook(string isbn)
{
    …

    //libraryManager.DeleteBook(isbn);
    commandProcessor.AddAndExecute(new DeleteBookCommand(libraryManager, isbn));
}
```

Végezetül, most már az Undo metódust is tudjuk implementálni:

```
public void Undo()
{
    commandProcessor.Undo();
}
```

Teszteljük a parancsok és a visszavonás működését az alábbi parancsok kiadásával:

lista
új 123
szerkeszt 123 szerző Tolsztoj
lista
szerkeszt 123 szerző Petőfi
lista
visszavon
lista
visszavon
lista
visszavon

3. Feladat: Composite Command minta a könyvek kezelésére (részben önálló feladat)

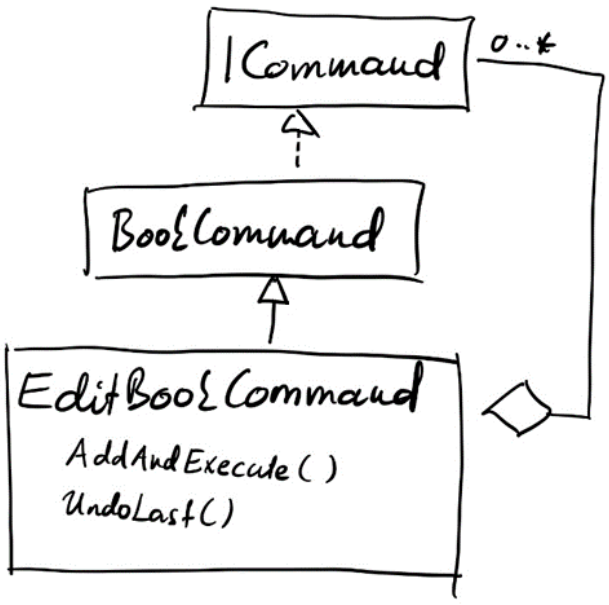
A feladatdefiníciót, a megoldás elveit, illetve a megoldás első lépéseit közösen beszéljük meg és csináljuk.

Az első feladatban megvalósított, kontextusfüggő összetett szerkesztés funkciót alakítsuk át a következőknek megfelelően:

- Jelen pillanatban a szerkesztési módban minden változtatás azonnal érvényre jut. Ezt változtassuk meg úgy, hogy csak a `befejez` parancs hatására történjen meg a szerkesztési módban kiadott cím és szerzőváltoztatások érvényesítése a szerkesztett könyvön. Vagyis a `befejez` parancs kiadása előtt a `lista` parancs az adott könyvön a szerkesztési módban végrehajtott változtatásokat ne mutassa (csak a `befejez` parancs kiadása után kiadott `lista` parancs mutatja)
- A szerkesztési módban végrehajtott változtatások `befejez` paranccsal való érvényesítése után a `visszavon` parancs a szerkesztési módban végrehajtott változtatások mindegyikét egyben visszavonja.
- **A fentieket úgy valósítsuk meg, hogy a `végrehajtás` és a `visszavonás` funkciókat a `CommandContextManager` osztályban a már bevált `commandProcessor.AddAndExecute(<cmd>)` parancsfuttató és `commandProcessor.Undo()` műveletekkel (vagyis a többi paranccsal egységes módon hajtsuk végre).**
- Szerkesztési módban a változtatások visszavonását első körben nem kell támogatni (opcionálisan megvalósítható, de csak ha marad rá idő).

A megvalósítás elvei:

- Az összetett szerkesztést is a többi paranccsal egységesen, commandként kezeljük: `EditBookCommand`
- Ez a command **összetett** lesz, amely további más commandokat is tartalmaz.
- A `CommandContextManager` szemszögéből (commandProcessor.AddAndExecute és commandProcessor.Undo) egységesen szeretnék kezelni a többi, elemi paranccsal: alkalmazzuk a ***Composite*** mintát a commandokon, ez a *Composite Command* minta.
- Amikor elkezdünk egy könyvet szerkeszteni, a `CommandContextManager`ben el kell tárolni az aktuális `EditBookCommand` parancsobjektumot (szerkesztési módban ehhez adjuk a commandokat), ezért ott is ki kell egészíteni az állapotkezelést.



Eddig beszéljük meg közösen a terveket, de innen a megvalósítás önálló munka, innen a hallgatók önállóan dolgoznak!

A következőkben némi segítséget adunk a megoldáshoz, de első körben célszerű teljesen önállóan próbálkozni.

EditBookCommand osztály felvétele

- BookCommand-ból származik
- List<ICommand> tagváltozó bevezetése a tartalmazott parancsok tárolásához
- Az Execute meghívja az ős execute-ját, majd minden tartalmazott command-ra Execute-ot hív
- Az Undo-t nem kell implementálni a leszármazottban
- AddCommand művelet bevezetése új parancs felvételéhez a listába
- LibraryManager library, string isbn paraméterű konstruktor bevezetése, ős konstruktor hívása

`CommandContextManager` osztályban a megoldás elve:

- Be kell vezetni egy `EditBookCommand editBookCommand` tagváltozót.
- Ez a hivatkozás kezdetben legyen null, szerkesztési üzemmódba lépéskor (StartEditingBook metódous) egy új `EditBookCommand` objektumra állítjuk, a befejez parancs során (`FinishEditingBook`) ezt a parancsot kell futtatni a `commandProcessor` segítségével, és az `editBookCommand` tagváltozót/hivatkozást null-ba kell állítani.
- Szerkesztési módban a cím és szerző módosításakor (`UpdateEditedBookTitle` és `UpdateEditedBookAuthor`) nem a `commandProcessor`-t használjuk, hanem az editBookCommand tag által hivatkozott composite commandhoz veszünk fel egy új `UpdateTitleCommand`, illetve `UpdateAuthorCommand` parancsot.
- A megoldás szépsége, hogy az `Undo` művelethez nem kell hozzányúlni, a Composite mintának köszönhetően!

Kiegészítő önálló feladat

Ezzel csak akkor foglalkozzunk, ha nagyon jól állunk az időben, egyébként inkább a következő, Adapter mintára térjünk rá. A példa inkább az otthoni önálló gyakorlást szolgálja.

Feladat: oldjuk meg, hogy szerkesztési módban is lehessen parancsokat visszavonni.

Lépések: az `EditBookCommand` osztályban vezessünk be egy `UndoLast` műveletet, és a `CommandContextManager` osztályban ezt hívjuk, ha szerkesztési módban vagyunk.

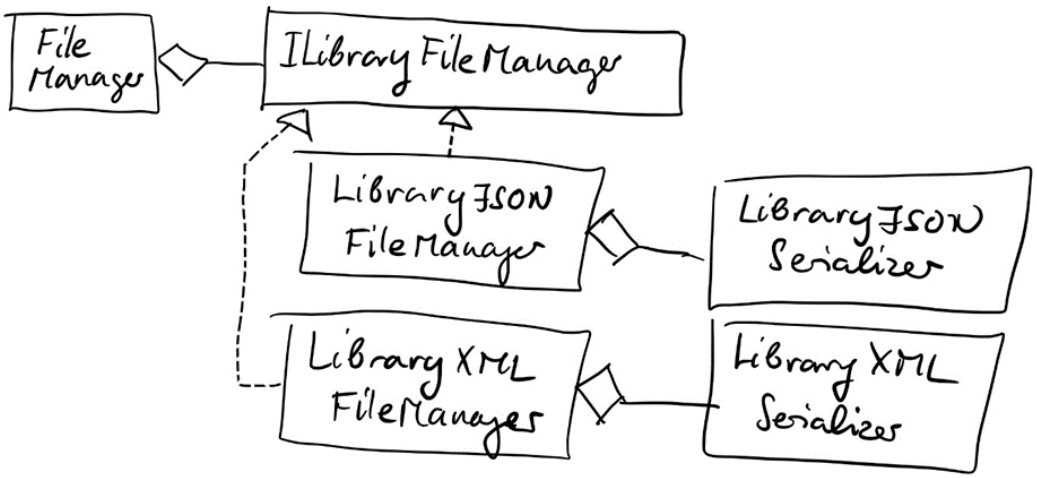
4. Feladat : Adapter minta (kezdeti átbeszélést követően önálló feladat)

Az adapter minta használata független az előzőektől. Megbeszélhetjük előre a terveket, vagy feladhatjuk teljesen önálló feladatnak (időtől függően).

Célunk az, hogy ne csak XML-be, hanem JSON formátumba is tudjunk sorosítani. Ehhez rendelkezésre áll egy `LibraryJSONSerializer`. (Beszélhetünk röviden az XML és JSON formátumokról, amennyiben szükséges, de a megvalósításhoz nem kell ezeket ismerni.)

- Beszéljük meg, hogy a `FileManager` `StoreLibrary` és `LoadLibrary` metódusait kell kiegészíteni az XML kezeléshez hasonló módon.
- Vegyük észre, hogy mindkét függvényben egy-egy új `if`-et kell létrehozni.
- Nem tudjuk egységesen kezelni a `LibraryJSONSerializer`-t és a `LibraryXMLSerializer`-t, mert mások a függvény nevek.
- Az új `if` belseje nagyon hasonló lesz az előző `if`-hez és ha további serializerek lennének sok ismétlődő kódot kellene írni.

Ismételjük át az adapter mintát! Beszéljük meg, hogyan lehetne itt hatékonyan alkalmazni, tervezzük meg a felhasználandó interfészeket!



Érdemes lehet átbeszélni, hogy a meglévő `StoreLibrary` és `LoadLibrary` metódusok mely részeit kell kiszervezni a technológiafüggő osztályokba.

A FileManager átalakításával kapcsolatban ismertessük az alábbi segítséget, miszerint

- a FileManager már fel van készítve arra, hogy a log fájl kiterjesztésétől függően legyen képes sorosítót választani. Ezt a tulajdonságát szeretnénk megtartani, ezért célszerű Dictionary adatszerkezetben eltárolni benne a rendelkezésre álló FileManager-eket, hogy a választott kiterjesztés szerint egyszerűen lehessen alkalmazni az implementációt! Pl. ezt írjuk be a FileManager osztályba:

```
private readonly Dictionary<string, ILibraryFileManager> fileManagers = new
    Dictionary<string, ILibraryFileManager>();

public FileManager()
{
     fileManagers.Add("json", new LibraryJsonFileManager());
     fileManagers.Add("xml", new LibraryXmlFileManager());
}
```

Illetve ebben egy adott elem elérése a kiterjesztés alapján: `fileManagers[extension]`