Szoftvertechnológia és-technikák
7. Gyakorlat

Kiterjeszthetőséghez kapcsolódó minták: Template Method és Strategy

A gyakorlat menete

A gyakorlat egy adott pontig vezetett, utána önálló munka. A feladat egy kiinduló kód átalakítása, hogy megfelelően alkalmazza a kiterjeszthetőséghez használható T*emplate Method* és *Strategy* mintákat, majd ennek továbbfejlesztése, a kiterjeszthetőség gyakorlati alkalmazása. A gyakorlat során egy C# konzol alkalmazásban dolgozunk.

Közös (vezetett) feladatok

Feladat 1: Kiinduló állapot megismerése

A feladatunk egy embereket adatait tartalmazó adatforrás (kezdetben: fájl) átalakítása, és kiírása más formátumban. Ehhez egy konzol alkalmazásból indulunk. Nyissuk meg a letöltött kiinduló állapotot, és keressük meg a Main függvényt a Program.cs osztályban.

Nézzük át a Main függvényt, és azonosítsuk a feldolgozás lépéseit:

1. Beolvasás
2. Átalakítás: most rendezés
3. Kiírás

```
// Ide kerulnek a beolvasott adatok
List<Person> peopleUnsorted = new List<Person>();

Beolvasás itt
// A teszt adatok egy CSV fajlban vannak
using (StreamReader reader = new StreamReader("us-500.csv"))
{
    // ...
}

Átalakítás itt
// Csoportositsuk az embereket az allam (state) alapjan
// Az egyenloseg jobb oldalat nem kell megerteni, csak lassuk, hogy a 'State' alapjan
List<Person> peopleByState = peopleUnsorted.OrderBy(p => p.State).ToList();

Kiírás itt
// Irjuk ki szoveges fajlba a sorrendezett emberek neveit az allammal egyutt
using (StreamWriter writer = new StreamWriter(@"emberek.txt"))
{
    // ...
}
```

Futtassuk le a programot. Az eredmény egy txt fájl lesz, ezt a solution alatti *bin* könyvtárban találjuk *emberek.txt* néven. Nézzük meg, hogyan néz ki.

Fussuk át a forráskód főbb részeit. A fájl beolvasás, a reguláris kifejezés, a rendezés, és a fájlba való kiírás pontos szintaktikáját nem kell mélyen megérteni, csak alapvetően a szándékot értsük meg. Itt nem a konkrét algoritmusokra és a C# kódra koncentrálunk, hanem a tervezésre.

Feladat 2: Osztályba szervezés

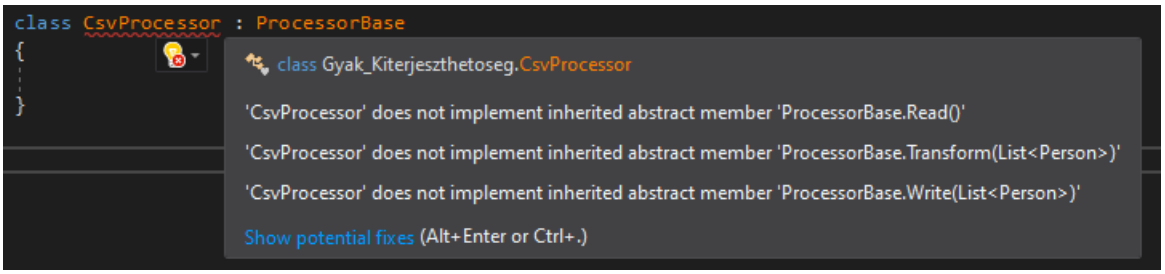
A program logikája „ömlesztve” van a *Main* függvényben. Szervezzük ki egy saját osztályba a logikát, bontsuk fel a lépések mentén kisebb darabokra. Eközben pedig gondoljunk arra is, hogy bővíthetővé tegyük a logikát: készítsünk egy közös feldolgozó vázat a jövőre készülve.

Készítsünk egy *ProcessorBase* ősosztályt, ami a feldolgozás általános lépéseit fogja össze. Ezek a lépések a beolvasás, átalakítás, és kiírás. Ezek mentén bontsuk függvényekre a feldolgozó logikát, és készítsünk egy *Run* függvényt, ami az egész feldolgozás logikáját összefogja. Ez egy absztrakt osztály lesz, mert a konkrét logika (mint a CSV fájl olvasása) nem ide fog kerülni.

```
abstract class ProcessorBase
{
    public void Run()
    {
        List<Person> input = Read();
        List<Person> processed = Transform(input);
        Write(processed);
    }

    protected abstract List<Person> Read();
    protected abstract List<Person> Transform(List<Person> people);
    protected abstract void Write(List<Person> people);
}
```

Ebből az osztályból leszármazva készítsünk egy *CsvProcessor* osztályt, amiben az eddigi logikákat megvalósítsuk. A *Main* függvényből átemelve rakjuk át a meglevő kódrészeket a megfelelő függvényekhez.

Célszerű először elkészíteni az új osztályt üresen. Majd a kódban jelöljük a leszármazási szándékunkat. Ekkor jelezni fogja a Visual Studio, hogy milyen függvények hiányoznak még a leszármazás miatt. A pirossal aláhúzott részre állva megnyílik az alább is látható segítség. Ugyanez a a Ctrl pont kombinációval is előhozható. Itt válasszuk az automata javítási lehetőséget, ami elkészíti számunkra az üres függvényeket.



A generált kódba:

- másoljuk át a Main függvény releváns részeit,
- ahol szükséges, egészítsük ki a szükséges return utasításokkal,
- ahol szükséges, nevezzük át az átmásolt kódban a változókat,
- gondoskodjunk a szükséges névterek using-olásáról.

Az osztályunk végül így néz ki:

```
using System.IO;
using System.Linq;
…

class CsvProcessor : ProcessorBase
{
    protected override List<Person> Read()
    {
        // átmásolt kód, peopleUnsorted→people átnevezéssel
        return people;
    }

    protected override List<Person> Transform(List<Person> people)
    {
        return people.OrderBy(p => p.State).ToList();    }

    protected override void Write(List<Person> people)
    {
        // átmásolt kód, peopleByState → people átnevezéssel
    }
}
```

A *Main* függvényünk pedig az alábbira egyszerűsödik:

```
static void Main(string[] args)
{
    Console.WriteLine("Program indul");

    CsvProcessor p = new CsvProcessor();
    p.Run();
}
```

Próbáljuk ki a programot, hogy még jól működik-e.

Konklúzió

A megoldásunkban gyakorlatilag már a ***Template Method*** tervezési mintát alkalmaztuk. Ezt a mintát a „06. ea. - Tervezési minták 1” előadás ismerteti részletesen, példákkal illusztrálva. Az alapelve a következő:

- Egy műveleten belül algoritmus vázat definiál, és az algoritmus bizonyos lépéseinek implementálását a leszármazott osztályra bízza. ***A példánkban ez a művelet/algoritmusváz a Run függvény, mely felhasználja a Read, Transform és Write függvényeket, de ezek megvalósítását a leszármazottakra bízza.***
- ***Ezáltal az ős CsvProcessor osztályban a Read, Tansform és Write függvények kiterjesztési pontok lesznek, hiszen a megvalósításuk nincs beégetve a CsvProcessor osztályba, a leszármazottak tetszőleges implementációt megadhatnak.***
- Más megközelítésben: a Run függvény, sablonszerű (vagyis template) metódus, egy „algoritmus” vázat definiál. A minden leszármazottra közös logikát tartalmazza egyetlen helyen, a közös ősben, így elkerüljük a duplikációját az alkalmazásban. A példánkban ugyan egyelőre egyetlen leszármazott/implementáció van, de ez rövidesen megváltozik.

Feladat 3: Bővítés *Template Method* minta alapján

A következőkben a Template Method mintát fogjuk tovább gyakorolni. Bővítsük ki az alkalmazásunkat egy másik fájlformátum támogatásával. Ahhoz, hogy az alkalmazásunk kódja karbantartható, könnyen kiterjeszthető maradjon, alkalmazzuk itt is a *Template Method* mintát a bővítéshez. Ezt már részben elő is készítettük az előző feladatban, így könnyebb dolgunk lesz.

Az új bemeneti fájlformátum egy olyan txt fájl, amiben az emberek adatai egyesével, soronként találhatóak meg, minden attribútum egy sor (lásd us-500-text.txt a forrásmappában). Ebben, és a meglevő CSV fájlban is közös, hogy szöveges fájlt dolgozunk fel. Továbbá a feldolgozás módja (a rendezés és a kiírás) változatlan marad.

Emeljük ki a meglevő *CsvProcessor* osztályunkból a közös részeket egy *FileProcessorBase* osztályba. Ez végezze el a *Transform* és a *Write* műveleteket, és a fájlkezelés nagy részét, csupán az egy ember beolvasását végző részt hagyjuk innen ki.

Az alábbi kód nagy részét érdemes az útmutatóból beemelni és utána röviden átfutni, a kulcsgondolatokat pedig hangsúlyozni:

```
abstract class FileProcessorBase : ProcessorBase
{
    private string path;

    public FileProcessorBase(string path)
    {
        this.path = path;
    }

    protected override List<Person> Read()
    {
        // A lista osszeallitasa es a fajlkezeles kozos resze marad itt
        List<Person> people = new List<Person>();
        using (StreamReader reader = new StreamReader(path))
        {
            Console.WriteLine("Fajl megnyitva");

            int personCount = 0;
            while (!reader.EndOfStream)
            {
                ++personCount;
                people.Add(readPerson(reader));
                Console.WriteLine($"{personCount}. ember beolvasva");
            }
        }

        return people;
    }

    protected abstract Person readPerson(StreamReader reader);

    // Az implementációt a CsvProcessor-ból mozgassuk ide
    protected override List<Person> Transform(List<Person> people) ...
    // Az implementációt a CsvProcessor-ból mozgassuk ide
    protected override void Write(List<Person> people) ...
}
```

A CsvProcessor osztályban a Read egy sablonszerű metódus („template method”), definiálja a személyek beolvasásának vázát, de nincs beégetve, hogy egyetlen személy beolvasása hogyan történjen: ezt egy readPerson absztrakt műveletre bízza, melyet a leszármazottak tetszés szerint megvalósíthatnak. Vagyis a példánkban a readPerson művelet egy kiterjesztési pont.

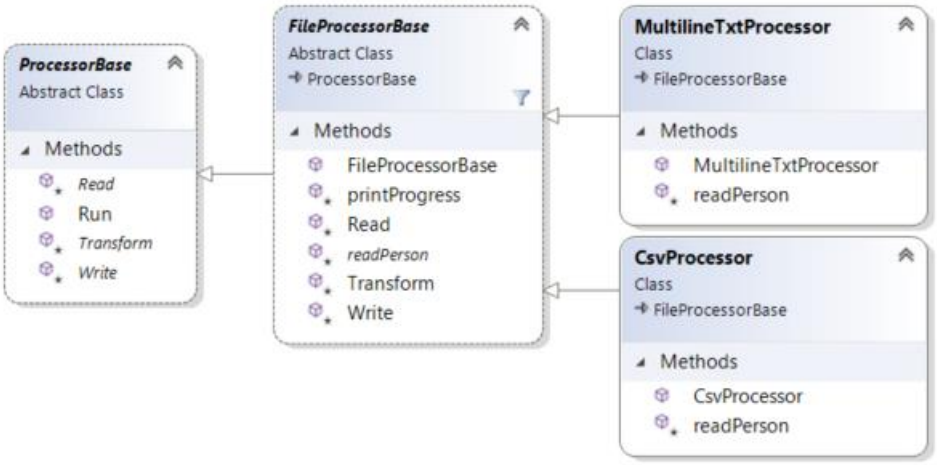
A *CsvProcessor* osztályunk a közös részek átmozgatása és a leszármazás után erre egyszerűsödik:

```
class CsvProcessor : FileProcessorBase
{
    public CsvProcessor()
        : base("us-500.csv") // A fajl nevet nem tudhatja az ososztaly, azt innen adjuk
    {
    }

    protected override Person readPerson(StreamReader reader)
    {
       var line = reader.ReadLine();
       System.Text.RegularExpressions.MatchCollection columns
          = new System.Text.RegularExpressions.Regex(
           "((?<=\")[^\"]*(?=\"(,|$)+)|(?<=,|^)[^,\"]*(?=,|$))"
         ).Matches(line);

       return new Person(firstName: columns[0].Value, lastName: columns[1].Value,
          companyName: columns[2].Value, address: columns[3].Value,
          city: columns[4].Value, state: columns[6].Value);
    }
}
```

Az átalakítás azért volt célszerű, mert az új fájlformátumot beolvasó osztályt így egyszerű lesz megvalósítani (a közös kód egy helyen az ősben, itt csak a kiterjesztési logikát kell megírni):

```
class MultilineTxtProcessor : FileProcessorBase
{
    public MultilineTxtProcessor()
        : base("us-500-text.txt") // A fajl nevet nem tudhatja az ososztaly
    {
    }

    protected override Person readPerson(StreamReader reader)
    {
        // 13 egymas utani sort olvasunk be
        string[] columns = new string[13];
        for (int i = 0; i < 13; ++i)
            columns[i] = reader.ReadLine();

        // Minden sorbol egy ember adatot leiro osztaly peldanyt keszitunk
        return new Person(firstName: columns[0], lastName: columns[1], companyName:
columns[2], address: columns[3], city: columns[4], state: columns[6]);
    }
}
```

Végezetül az alábbi osztályhierarchiát kaptuk.



Ezen a ponton **próbáljuk ki az alkalmazást**. Mindkét feldolgozóval (*CsvProcessor*, *MultilineTxtProcessor*) teszteljük a működést. Ehhez a *Main* függvényben cseréljük ki a példányosítást.

Feladat 3/b: folyamat állapotának követése

A *FileProcessorBase Read* metódusban van egy kezdeményünk a folyamat állapotának mutatásához:

```
Console.WriteLine($"{personCount}. ember beolvasva");
```

Ez a sor konzolra kiírja, hányadik embernél tartunk. Ezt az aspektust is szervezzük ki, hogy kibővíthető és lecserélhető legyen. Például gondoljunk arra, hogy százalékosan is akarhatjuk kiírni a haladást.

A változtatást a *FileProcessorBase* osztályban kezdjük. Emeljük ki egy új függvénybe a haladás kiírását. A függvény legyen virtuális, azaz adjunk neki implementációt, de lehetőséget is adunk a felüldefiniálására a leszármazottaknak.

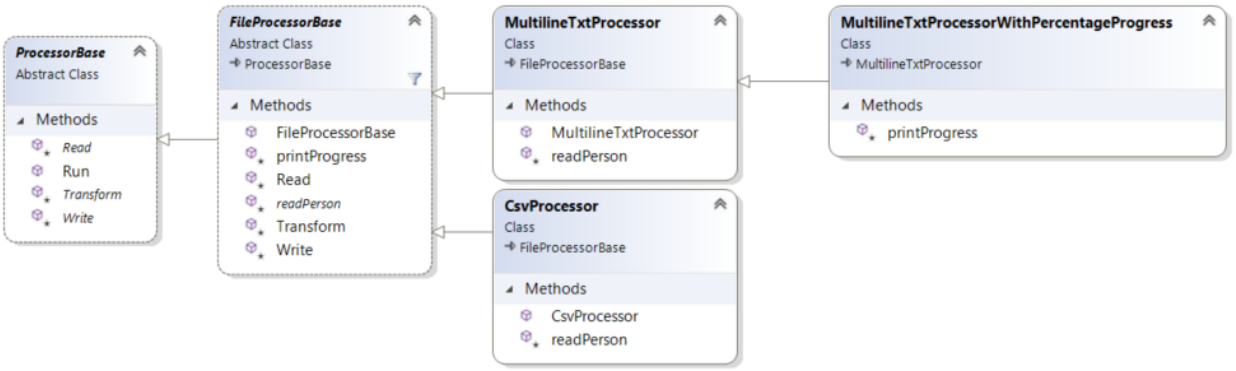
```
abstract class FileProcessorBase : ProcessorBase
{
    protected override List<Person> Read()
    {
        // A lista osszeallitasa es a fajlkezeles kozos resze marad itt
        List<Person> people = new List<Person>();
        using (StreamReader reader = new StreamReader(path))
        {
            Console.WriteLine("Fajl megnyitva");

            int personCount = 0;
            while (!reader.EndOfStream)
            {
                ++personCount;
                people.Add(readPerson(reader));
                printProgress(personCount);
            }
        }

        return people;
    }

    protected virtual void printProgress(int personCount)
    {
        Console.WriteLine($"{personCount}. ember beolvasva");
    }
```

Tip: A kód külön metódusba emeléséhez használhatjuk a Visual Studio „Extract Method” refactoring szolgáltatását. Ehhez jelöljük ki a külön függvénybe kiemelendő kódrészt, jobb katt a kijelölésen, majd válasszuk ki a „Quick Actions and Refactorings” menüelemet (vagy ctrl + „.” billentyű kombináció). Ekkor egy preview jelenik meg, Extract method címkével, az Enter billentyű lenyomásával megtörténik a függvénybe való kiemelés. Ekkor gépeljük be a függvény nevét és Enter-rel véglegesítsük a változtatásokat. Ez követően igazítsuk megfelelően a függvény fejlécét (private static helyett protected virtual).

Éljünk a lehetőséggel, és valósítsunk meg egy olyan feldolgozó logikát is, amely százalékosan írja ki a haladást. Mivel leszármazáson keresztül van lehetőségünk a kiegészítésre a *Template Method* minta használatával, így egy új leszármazottra lesz szükségünk. A *MultilineTxtProcessor* osztályból leszármazva készítsük el az új módszert alkalmazó logikát:

```
class MultilineTxtProcessorWithPercentageProgress : MultilineTxtProcessor
{
    protected override void printProgress(int personCount)
    {
        float percentage = (float)personCount / 500 * 100;
        if (percentage % 5 == 0) // Csak 5 szazalekonkent irjuk ki
            System.Console.WriteLine($"Feldolgozas {percentage}%");
    }
}
```

Ezzel az új osztállyal bővülve így néz ki a hierarchia:



Próbáljuk ki ezt a megoldást is (a Main függvényben ezt az osztályt példányosítsuk).

Feladat 4: Áttérés a Strategy mintára

A T*emplate Method* minta alkalmazásával sikerült az eddigi bővítéseket megoldanunk. Azonban láthatjuk, hogy ha a CSV fájl beolvasó logikából is szeretnénk egy olyat, ahol a haladást százalékosan írjuk ki, akkor ismét új osztályra és leszármazásra lenne szükség. Ha pedig még tovább akarnánk bővíteni a változatokat, akkor hatalmas leszármazási hierarchiák és a sok kombinációnak megfelelően sok osztályra kellene készítenünk. Egyszerűbb esetekben a Template Method minta jól alkalmazható könnyen bővíthető megoldás készítésére, bonyolultabb esetekben – különösen, ha különböző aspektusok mentén szeretnénk bővíthetővé tenni a megoldásunkat – a gyakorlatban már használhatatlanná válhat.

Térjünk át a *Strategy* minta használatára: az aktuális funkcionalitást valósítjuk meg az aktuális kiterjeszthetőségi aspektusok mentén, de Template Method helyett a Strategy tervezési mintával.

Mielőtt ennek nekiállunk, célszerű az aktuális állapotot elmenteni, hogy otthon meg tudjuk nézni.

A strategy minta áttekintése (információ a gyakorlatvezetőknek): a Strategy minta előadáson részletesen, több példával illusztrálva is szerepel („06. ea. - Tervezési minták 1”). Az egyik előadáspéldához a tárgy honlapján található rövid ppt kivonat ábrával, mindössze 2 dia, az oktatói anyagok között: „6. gy. - Strategy segédlet (vetítéshez)”[1], ezt célszerű a gyakorlaton kivetíteni és a Strategy minta lényegét ez alapján pár percben ismétlésképpen áttekinteni.

A programunk két aspektus mentén rendelkezik többféle működési modellel: az emberek adatainak beolvasása a fájlból, és a haladás mutatása. Kezdjük először az előrehaladás átalakításával.

Készítsünk egy új mappát a projektünkben *Progresses* néven. Ebbe a mappába készítsünk egy interfészt *IProcessProgress* néven. Ez az interfész csak és kizárólag a felhasználót előrehaladásról való tájékoztatásáért felel, így csak egyetlen metódusa van. (Azért interfészt hozunk létre, mert nincs okunk osztályban gondolkodni itt.)

1 https://www.aut.bme.hu/Upload/Course/SzTT/oktatoi_jegyzetek/Tervez%c3%a9si%20mint%c3%a1k%20gy1%20-%20Strategy.pptx

```
interface IProcessProgress
{
    void PrintProgress(int personCount);
}
```

Valósítsuk meg ezen interfész két implementációját az alábbiak szerint. Két új osztályt készítsünk!

```
class CountProgress : IProcessProgress
{
    public void PrintProgress(int personCount)
    {
        Console.WriteLine($"{personCount}. ember beolvasva");
    }
}
```

A másik pedig hasonlóan készül.

```
class PercentageProgress : IProcessProgress
{
    public void PrintProgress(int personCount)
    {
        float percentage = (float)personCount / 500 * 100;
        if (percentage % 10 == 0) // Csak 5 szazalekonkent irjuk ki
            System.Console.WriteLine($"Feldolgozas {percentage}%");
    }
}
```

A beolvasó műveletekkel azonos lépések mentén járjunk el. Készítsünk egy új mappát *PersonReaders* néven. Ebbe a mappába készítsünk egy *IPersonReader* interfészt, amely a korábbi megoldásunkhoz hasonlóan egy ember adatainak beolvasásáért felel:

```
interface IPersonReader
{
    Person Read(StreamReader reader);
}
```

Itt is látható, hogy milyen egyszerű az interfészünk, mert csak egyetlen aspektusra koncentrálunk. A két implementáció a korábbi osztályokból átmásolható az alábbiaknak megfelelően. Figyeljük meg, hogy az osztályok is egyszerűek lesznek (nincs se ős konstruktor hívás, se függvények felüldefiniálva).

```
class CsvPersonReader : IPersonReader
{
    public Person Read(StreamReader reader)
    {
        var line = reader.ReadLine();

        System.Text.RegularExpressions.MatchCollection columns
          = new System.Text.RegularExpressions.Regex(
           "((?<=\")[^\"]*(?=\"(,|$)+)|(?<=,|^)[^,\"]*(?=,|$))"
         ).Matches(line);

       return new Person(firstName: columns[0].Value, lastName: columns[1].Value,
          companyName: columns[2].Value, address: columns[3].Value,
          city: columns[4].Value, state: columns[6].Value);
    }
}

// ****** külön fájl

class MultilineTxtPersonReader : IPersonReader
{
    public Person Read(StreamReader reader)
    {
        string[] columns = new string[13];
        for (int i = 0; i < 13; ++i)
            columns[i] = reader.ReadLine();

        return new Person(firstName: columns[0], lastName: columns[1],
           companyName: columns[2], address: columns[3],
           city: columns[4], state: columns[6]);
    }
}
```

És most kitörölhetjük a feleslegessé vált osztályainkat:

- CsvProcessor
- MultilineTxtProcessor
- MultilineTxtProcessorWithPercentageProgress

A *FileProcessorBase* osztályunkat nevezzük át *FileProcessor*-ra, hiszen már nem fogunk belőle leszármazni. A kódja pedig az alábbiak szerint változik. Konstruktorban kapja az előbb készített két interfészt, és a *Read* függvényben az absztrakt és virtuális függvények helyett ezeket használja. (Ügyeljünk arra, hogy a létrehozott interfészek a mappáknak megfelelő névterekbe kerültek, így azt is ki kell írnunk az osztály nevénél.)

```
class FileProcessor : ProcessorBase
{
    private string path;
    private readonly PersonReaders.IPersonReader personReader;
    private readonly Progresses.IProcessProgress processProgress;

    public FileProcessor(string path, PersonReaders.IPersonReader personReader,
Progresses.IProcessProgress processProgress)
    {
        this.path = path;
        this.personReader = personReader;
        this.processProgress = processProgress;
    }

    protected override List<Person> Read()
    {
        // A lista osszeallitasa es a fajlkezeles kozos resze marad itt
        List<Person> people = new List<Person>();
        using (StreamReader reader = new StreamReader(path))
        {
            Console.WriteLine("Fajl megnyitva");

            int personCount = 0;
            while (!reader.EndOfStream)
            {
                ++personCount;
                people.Add(personReader.Read(reader));
                processProgress.PrintProgress(personCount);
            }
        }

        return people;
    }

    protected override List<Person> Transform(List<Person> people) ...
    protected override void Write(List<Person> people) ...

    // Töröljük a korábbi printProgress és readPerson metódusokat !!!
}
```

Vegyük észre, hogy a fenti FileProcessor osztály már nem absztrakt, a „class” elől mindenképpen **vegyük ki az abstract kulcsszót**! Valamint töröljük az osztályból a már nem használt printProgress és readPerson metódusokat.

A *Main* függvényünkben pedig az alábbi módon hozhatunk létre egy felparaméterezett feldolgozót. **Próbáljuk ki több kombinációban is** (egyik fajta fájl, másik fajta előrehaladást mutató osztály).

```
var p = new FileProcessor("us-500-text.txt",
     new PersonReaders.MultilineTxtPersonReader(),
     new Progresses.PercentageProgress());
p.Run();
```

Önálló feladatok

Az alábbi feladatokat önállóan oldjuk meg. A probléma megértése után kövessük a javasolt lépéseket. Ne felejtsük el minden feladat végén kipróbálni a programunkat!

Feladat 5: Eredmény kiírás leválasztása *Strategy* minta használatával

Az eredmények kiírását a *FileProcessor* osztály végzi. Emeljük ki ezt a logikát is a kiterjeszthetőség végett egy interfész mögötti osztályba. Kövessük az alábbi lépéseket.

1. Készítsünk egy új mappát a projektben *Writers* néven.
2. Készítsünk ide egy új *IResultWriter* interfészt, amely `void Write(List<Person> people)` deklarációval rendelkezik.
3. Implementáljuk az interfészt egy új *TxtWriter* nevű osztályban. Ne felejtsük el az osztály deklarációban megadni az interfészt implementálását! (`class TxtWriter : IResultWriter`). Emeljük át a *FileProcessor* osztályból a *Write* függvényt ebbe az új osztályba (itt már nincs szükség az override kulcsszóra és protected helyett public a láthatóság).
4. **A *ProcessorBase* osztályunkat tegyük paraméterezhetővé az új *IResultWriter* stratégiával**:
    a. készítsünk egy új konstruktort, amely paraméterben várjon egy *IResultWriter*-t, és ezt mentse el egy tagváltozóba.
    b. A *ProcessorBase* osztály a *Run* függvényében ne a leszármazott *Write* függvényét hívja, hanem a konstruktorban kapott interfészt használja a kiíráshoz. A *Write* függvényt a *ProcessorBase* osztályból és a *FileProcessor* leszármazottból is töröljük ki.
5. A *FileProcessor* leszármazott konstruktorát ki kell bővíteni, hogy az új *IResultWriter* interfészt is várja paraméterben, és át tudja azt adni az ősének (amit pár lépéssel korábban írtunk oda). Ez így kell kinézzen:

```
public FileProcessor(string path,
   PersonReaders.IPersonReader personReader,
   Progresses.IProcessProgress processProgress,
   Writers.IResultWriter writer)
: base(writer)
```

6. Végezetül ne felejtsük a Main függvényben javítani a példányosítást, hiszen most már egyel több paramétere van a konstruktornak.

Figyeljük meg, hogy ugyan csak egy implementációja van az *IResultWriter* interfésznek, de így is van értelme a leválasztásnak. Felkészülünk a későbbi lehetséges kiterjesztésre, valamint egyszerűbb osztályaink lesznek (amely egyébként a tesztelést is elősegíti majd).

Továbbá lássuk azt is, hogy az *IResultWriter*-t nem a *FileProcessor*-ba, hanem az ősébe tettük. A kiterjesztést több helyen is megtehetjük egy leszármazási hierarchiában (így valójában kombináltuk a *Template Method* és *Strategy* mintákat).

Próbáljuk ki a változtatásainkat!

Feladat 6: Rendezés leválasztása, rendezéshez használt attribútum kiválasztása delegate segítségével

A *FileProcessor* osztály a nevével ellentétben nem csak a fájlok feldolgozásához szigorúan kapcsolódó funkciókért felel, hanem még a *Transform* művelet is itt van. Az előző feladatban látottakhoz hasonlóan válasszuk le és szervezzük ki külön osztályba a feldolgozást. Kövessük az alábbi lépéseket.

1. Készítsünk egy új mappát a projektben *PeopleProcessors* néven.
2. Készítsünk ide egy új *IPeopleProcessor* interfészt, amely `List<Person> Transform(List<Person> people)` deklarációval rendelkezik.
3. Implementáljuk az interfészt egy új *SortProcessor* nevű osztályban. Emeljük át a *FileProcessor* osztályból a *Transform* függvényt ebbe az új osztályba. Ne felejtsük el az osztály deklarációban megadni az interfészt implementálását! Szükségünk lesz az új osztály elején a `using System.Linq;` névtér importra is (az *OrderBy*) függvényhez.
4. A *ProcessorBase* osztályunkat tegyük paraméterezhetővé az új *IPeopleProcessor* stratégiával az előző feladattal analóg módon: tagváltozó bevezetése az új stratégia tárolásához, valamint *ProcessorBase* konstruktor kiegészítése.
5. A *ProcessorBase* osztály a *Run* függvényében ne a leszármazott *Transform* függvényét hívja, hanem az előző lépésben a konstruktorban átvett interfészt használj. A *Transform* függvényt a *ProcessorBase* osztályból és a *FileProcessor* leszármazottból is töröljük ki.
6. A *FileProcessor* konstruktorát ki kell bővíteni az új interfésszel, ahogy korábban is csináltuk.
7. Végezetül ne felejtsük a Main függvényben javítani a példányosítást, hiszen még egyel több paramétere van a konstruktornak.

A *SortProcessor* osztály rendezést végez, de „bele van égetve”, hogy mi alapján. Ezt is tegyük testreszabhatóvá egy *delegate* használatával.

8. A *SortProcessor* osztálynak készítsünk egy konstruktort, amely így néz ki:

```
public SortProcessor(Func<Person, string> sortSelector)
```

9. A konstruktor tárolja el tagváltozóban ezt a *sortSelector*-t, és a *Transform* függvényében az *OrderBy* hívásnak ezt adja át:

```
public List<Person> Transform(List<Person> people)
{
      List<Person> peopleSorted = people.OrderBy(this.sortSelector).ToList();
      return peopleSorted;
}
```

10. A *Main* függvényben *SortSelector* létrehozásakor meg kell adnunk a rendezési elvet. Ez egyszerű a következő szintaktikával:

```
new PeopleProcessors.SortProcessor(x => x.State)
```

Próbáljuk ki a változtatásainkat! Próbáljuk ki a rendezést más attribútum szerint is!

Extra feladat

Ezek a feladatok a gyakorlat teljesítésébe nem számítanak bele, gyakorlásnak szánjuk őket. A feladatok itt már szándékosan csak a célt mondják meg, a megoldás kitalálása a feladat része.

Feladat 7: Naplózás megvalósítása

A programban itt-ott vannak konzolra történő kiírások – az előrehaladást mutató állapot mellett is. Ilyen például a *Main* függvény első sora. A naplózást tipikusan a környezettől függően más-más módon kell megvalósítani. Fejlesztéshez praktikus a konzolra történő írás, de egy valós üzemben futó alkalmazásnál inkább fájlokban szokás a naplót tárolni (hogy ne vesszen el).

Valósítsunk meg naplózást a *Strategy* minta - pontosabban egy ahhoz hasonló technika - használatával. Legyen egy olyan naplózás, amely továbbra is a konzolra ír ki, és legyen egy fájlba kiíró naplózás is. A fájlba való íráshoz használjuk a *System.IO.File.AppendAllText* függvényt, amely egy fájl végére ír. A megvalósítás során *ne* kövessük szigorúan a Strategy mintát: naplózást *ne* a korábban látott módon a konstruktorban adjuk át a feldolgozó logikának, hanem a *Run* függvény paraméterében. A *Run* pedig adja tovább, ahova szükséges!

Használjunk intenzívebb naplózást! A program az előrehaladtáról írjon a naplóba üzeneteket (pl. kezdődik a beolvasás, befejezte a rendezést, megnyitotta a fájt a kiíráshoz stb.).