6. HF - Tervezési minták (kiterjeszthetőség)

A házi feladatban a kapcsolódó laboron (6. labor – Tervezési minták (kiterjeszthetőség)) elkezdett adatfeldolgozó/anonimizáló alkalmazást fogjuk továbbfejleszteni.

Az önálló feladat az tervezési minták előadásokon elhangzottakra épít: - "Előadás 08 - Tervezési minták 1" előadás: "Bővíthetőséghez, kiterjeszthetőséghez kapcsolódó alap tervezési minták" nagyfejezet: bevezető példa, Template Method, Strategy, Open/Closed elv, SRP elv, egyéb technikák (metódusreferencia/lambda) - "Előadás 09 - Tervezési minták 1" előadás: Dependency Injection minta

A feladatok gyakorlati hátteréül a 6. labor – Tervezési minták (kiterjeszthetőség) laborgyakorlat szolgál.

Az önálló gyakorlat célja:

- Kapcsolódó tervezési minták és egyéb kiterjeszthetőségi technikák alkalmazása
- Integrációs és egységtesztek koncepcióinak gyakorlása

A szükséges fejlesztőkörnyezetről itt található leírás. Ennél a házi feladatnál nincs szükség WinUI-ra (egy konzol alapú alkalmazás kontextusában kell dolgozni), így pl. Linux/MacOS környezetben is elvégezhető.

A beadás menete

- Az alapfolyamat megegyezik a korábbiakkal. GitHub Classroom segítségével hozz létre magadnak egy repository-t. A meghívó URL-t Moodle-ben találod (a tárgy nyitóoldalán a "*GitHub classroom hivatkozások a házi feladatokhoz*" hivatkozásra kattintva megjelenő oldalon látható). Fontos, hogy a megfelelő, ezen házi feladathoz tartozó meghívó URL-t használd (minden házi feladathoz más URL tartozik). Klónozd le az így elkészült repository-t. Ez tartalmazni fogja a megoldás elvárt szerkezetét. A feladatok elkészítése után commit-old és push-old a megoldásod.
- A kiklónozott fájlok között a `Patterns-Extensibility.sln`-t megnyitva kell dolgozni.
- ❗ A feladatok kérik, hogy készíts **képernyőképet** a megoldás egy-egy részéről, mert ezzel bizonyítod, hogy a megoldásod saját magad készítetted. **A képernyőképek elvárt tartalmát a feladat minden esetben pontosan megnevezi.** A képernyőképeket a megoldás részeként kell

beadni, a repository-d gyökérmappájába tedd (a neptun.txt mellé). A képernyőképek így felkerülnek GitHub-ra a git repository tartalmával együtt. Mivel a repository privát, azt az oktatókon kívül más nem látja. Amennyiben olyan tartalom kerül a képernyőképre, amit nem szeretnél feltölteni, kitakarhatod a képről.

- ❗ Ehhez a feladathoz érdemi előellenőrző nem tartozik: minden push után lefut ugyan, de csak a neptun.txt kitöltöttségét ellenőrzi. Az érdemi ellenőrzést a határidő lejárta után a laborvezetők teszik majd meg.

1. Feladat

A házi feladat megoldásának alapja a következő:

- A Strategy és a kapcsolódó Dependency Injection (DI) tervezési minta ismerete
- Ezen minták alkalmazásának pontos megértése a labor feladatának a kontextusában (anonimizáló)

A házi feladat kiinduló állapota megfelel a 6. labor végállapotának: ez a házi feladat solutionjében a "Strategy-DI" projekt. Futtatáshoz/debuggoláshoz be kell állítani, hogy ez legyen a startup projekt (jobb katt, "*Set as Startup Project*"). Ennek forráskódját alaposan nézd át és értsd meg.

- A `Program.cs` fájlban található három `Anonymizer`, eltérő strategy implementációkkal paraméterezve. Ráhangolódásképpen érdemes ezeket egyesével kipróbálni/futtatni, és megnézni, hogy valóban a választott strategy implementációknak megfelelően történik az anonimizálás és a progress kezelés (emlékeztető laborról: az anonimizáló bemenete "bin\Debug\net8.0" mappában levő us-500.csv, kimenete az ugyanitt található "us-500.processed.txt").
- Szintén érdemes a `Program.cs` fájlban kiindulva, töréspontokat elhelyezve végig lépkedni a kódon (ez is segítheti az ismétlést/teljes megértést).

> **Dependency Injection (manuális) vs. Dependency Injection Container**
>
> A labor során, és jelen házi feladatban a Dependency Injection egyszerű, manuális változatát használjuk (előadáson is ez szerepel). Ez esetben az osztály függőségeit manuálisan példányosítjuk és adjuk át az osztály konstruktorában. Alternatív és komplexebb alkalmazások esetében gyakran használt alternatíva egy Dependency Injection Container alkalmazása, melybe beregisztrálhatjuk, hogy az egyes interfész típusokhoz milyen implementációt kívánunk használni. Az MVVM labor során "mellékesen" használtuk ezt a technikát, de a DI konténerek alkalmazása nem tananyag. A manuális változata viszont az, és kiemelt fontosságú, hiszen enélkül nincs értelme a Strategy minta alkalmazásának.

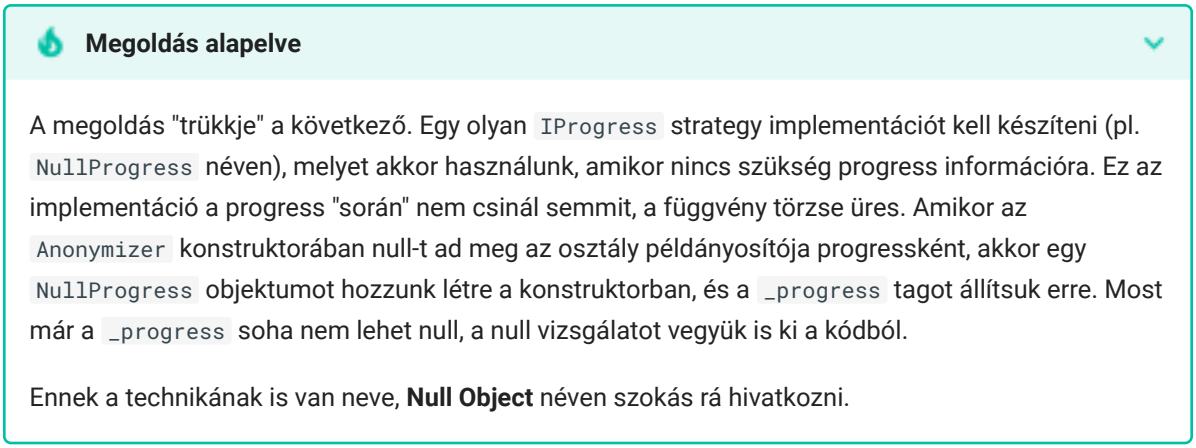
⚠️ Saját szavaiddal megfogalmazva adj rövid választ a *Feladatok* mappában található `readme.md` fájlban az alábbi kérdésekre:

- Mit biztosít a Strategy a DI mintával kombinálva a labor példa keretében, mik az együttes alkalmazásuk előnyei?
- Mit jelent az, hogy a Strategy minta alkalmazásával az Open/Closed elv megvalósul a megoldásban? (az Open/Closed elvről az előadás és laboranyagban is olvashatsz).

2. Feladat - Null Strategy

Az `Anonymizer` konstruktor paramétereit megvizsgálva azt látjuk, hogy progress stratégiának `null` is megadható. Ez logikus, hiszen lehet, hogy az `Anonymizer` felhasználója nem kíváncsi semmiféle progress információra. Ennek a megközelítésnek van egy hátránya is. Ez esetben az osztályban a `_progress` tagváltozó null lesz, és így az alkalmazása során szükség van a null vizsgálatra. Ellenőrizzük, hogy a `_progess` használatakor valóban van null vizsgálat a `?.` operátor alkalmazásával. De ez egy veszélyes játék, mert komplexebb esetben hacsak egyetlen helyen is lefelejtődik a null vizsgálat, akkor futás közben `NullReferenceException`-t kapunk. Az ehhez hasonló null hivatkozás hibák a leggyakoribbak közé tartoznak.

Feladat: Dolgozz ki egy olyan megoldást, mely a fent vázolt hibalehetőséget kizárja. Tipp: olyan megoldásra van szükség, melynél a `_progress` tag soha nem lehet null. A megoldásra először magadtól próbálj rájönni.



3. Feladat - Tesztelhetőség

Vegyük észre, hogy az `Anonymizer` osztály működésének van még számos aspektusa, melyeket valamelyik megoldásunkkal kiterjeszthetővé lehetne tenni. Többek között ilyen a:

- **Bemenet** kezelése: Most csak fájl alapú, adott CSV formátumot támogatunk.
- **Kimenet** kezelése: Most csak fájl alapú, adott CSV formátumot támogatunk.

Ezeket az SRP elve miatt illene az osztályról leválasztani, más osztályba tenni (ismételd át, mit jelent az SRP elv). A leválasztást nem feltétlen kiterjeszthető módon kellene megtenni, hiszen nem merült fel igény arra, hogy különböző bemenetekkel és kimenetekkel kellene tudni dolgozni. Így a leválasztás során nem alkalmaznánk a Strategy mintát.

Ugyanakkor van még egy kritikus szempont, melyről nem beszéltünk (és a régebbi, klasszikus design pattern irodalmak sem feltétlen emlegetik). Ez az egységtesztelhetőség.

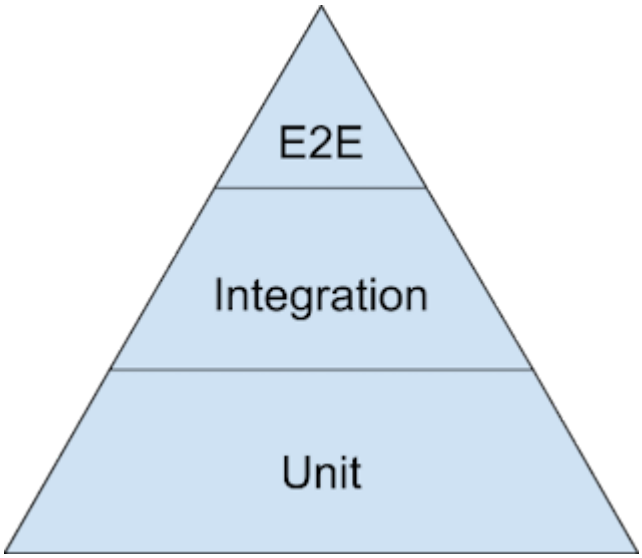
Jelen pillanatban az `Anonymizer` osztályunkhoz automata **integrációs teszteket** tudunk írni, automata **egységteszteket** nem:

- Az integrációs tesztek a teljes működést egyben vizsgálják: ebben benne van a bemenet feldolgozása, adatfeldolgozás, kimenet előállítása. Ez példánkban egyszerű: elállítunk bizonyos bemeneti CVS állományokat, és megnézzük, a várt kimeneti állomány állítódik-e elő.
- Az integrációs tesztek nagyon lassúak tudnak lenni: sokszor fájlokból, adatbázisokból, felhő alapú szolgáltatásokból veszik a bemenetet, illetve ezek szolgálnak kimenetként. Egy nagyobb termék esetében - mikor sok ezer teszt van - ez a lassúság korlátozó tényező, ritkábban tudjuk futtatni és/vagy nem tudunk jó tesztlefedettséget elérni.

A fentiek miatt sokszor nagyobb kódlefedettséget nem a lassabb integrációs, hanem nagyon gyorsan futó **egységtesztekkel** szoktunk/tudunk elérni. Ezek mindenféle **lassú fájl/adatbázis/hálózat/felhő elérés nélkül önmagában egy-egy logikai egységet tesztelnek a kódban**, ezt viszont így már villámgyorsan. Így sokat tudunk futtatni adott idő alatt, jó tesztlefedettséggel.

Tesztpiramis

Ezt egy tesztpiramissal szokás szemléltetni, melynek több formája terjedt el az irodalomban. Egy egyszerű variáns a következő:



Minél fentebb vagyunk a piramis rétegeiben, annál átfogóbbak ugyan a tesztek, de annál lassabbak és költségesebben is futtathatók. Így ezekből általában kevesebbet is készítünk (ezáltal kisebb kódlefedettséget is érünk el velük). A piramis csúcsán az automata E2E (End-to-end) vagy GUI tesztek vannak. Alatta vannak több egységet/modult egyben tesztelő integrációs tesztek. A piramis talapzatában az egységtesztek vannak, ezekből készítünk a legtöbbet (a piramis talapzata a legszélesebb).

Fun fact: Amikor egy termék fejlesztése során hosszú ideig elhanyagolják az egységtesztek készítését, akkor - mivel a kód szerkezete nem támogatja - már nagyon nehéz egységteszteket utólag készíteni. Így ezekből csak nagyon kevés lesz, némi integrációs tesztekkel kiegészítve, és jobb híján tesztelőcsapatok által elkészített sok-sok end-to-end/GUI teszttel (de ezzel sokszor nem lehet jó tesztlefedettséget elérni egy komplex termékben). Egy piramissal szemben ennek fagyitölcsér formája van, csak pár gombócot kell a tetejére képzelni. Szokás ezt fagyi "mintának" is nevezni (és ez nem az a fagyi, amit szeretünk). Azt azért érdemes megjegyezni, hogy mindent a helyén kell kezelni: vannak kivételek (olyan alkalmazások, ahol az egyes részekben alig van logika, az egész alkalmazásban az egyes nagyon egyszerű részek integrációja a hangsúlyos: ilyen esetben természetszerűen az integrációs tesztek túlsúlyosak).

Az osztályok kódja alapesetben sokszor nem egységtesztelhető. Jelen formájában ilyen az `Anonymizer` is. Ebbe be van égetve, hogy csak a lassú, fájl alapú bemenettel tud dolgozni. De amikor mi pl. a `Run` művelet logikáját szeretnénk egységtesztelni, teljesen mindegy, hogy fájlból jönnek-e az adatok (lassan), vagy egyszerűen kódból a `new` operátorral előállítunk néhány `Person`

objektumot a teszteléshez (több nagyságrenddel gyorsabban).

A megoldás - a kódunk egységtesztelhetővé tételéhez - egyszerű:

> ⚠️ *A Strategy (+DI) minta (vagy delegate-ek) alkalmazással válasszuk le az egységtesztelni kívánt osztályról a tesztelést akadályozó vagy lassító (pl. bemenet/kimenet kezelés) logikákat. Ezeknek készítünk a valódi logikát megvalósító implementációit, illetve tesztelést segítő, ún. mock implementációit.*

> ⚠️ *Ennek megfelelően a Strategy mintát sokszor nem azért használjuk, mert az ügyféligények miatt többféle viselkedést kell benevezni, hanem azért, hogy a kódunk egységtesztelhető legyen.*

Ennek megfelelően elkészítjük a megoldásunk egységtesztelésre is előkészített változatát, melyben a bemenet és kimenet kezelése is le van választva a Strategy minta alkalmazásával.

Feladat: Alakítsd át a Strategy-DI projektben található megoldást olyan módon, hogy az osztály egység tesztelhető legyen, mégpedig a Strategy minta segítségével. Részletesebben:

- Vezess be egy `InputReaders` mappát, melyben vezess be egy bemenet feldolgozó strategy interfészt `IInputReader` néven (egyetlen, `List<Person> Read()` művelettel), és az `Anonymizer` osztályból a Strategy mintát követve szervezd ki a bemenet feldolgozást egy `CsvInputReader` nevű strategy implementációba. Ez az osztály konstruktor paraméterben kapja meg a fájl útvonalát, melyből a bemenetét olvassa.
- Vezess be egy `ResultWriters` mappát, melyben vezess be egy eredmény kiíró strategy interfészt `IResultWriter` néven (egyetlen, `void Write(List<Person> persons)` művelettel), és az `Anonymizer` osztályból a Strategy mintát követve szervezd ki a kimenet írását egy `CsvResultWriter` nevű strategy implementációba. Ez az osztály konstruktor paraméterben kapja meg a fájl útvonalát, melybe a kimenetet bele kell írja.
- Bővítsd ki a `Anonymizer` osztályt, beleértve annak konstruktorát (Strategy + DI minta), hogy bármilyen `IInputReader` és `IResultWriter` implementációval használható legyen.
- A `Program.cs` fájlban alakítsd át az `Anonymizer` osztály használatát, hogy az újonnan bevezetett `CsvInputReader` és `CsvResultWriter` osztályok is át legyenek paraméterként átadva.

A következő lépés egységtesztek készítése (lenne) az `Anonymizer` osztályhoz. Ehhez olyan, ún. mock strategy implementációkat kell bevezetni, melyek nemcsak tesztadatokat szolgáltatnak (természetesen gyorsan, fájlkezelés nélkül), hanem ellenőrzéseket is végeznek (adott logikai

egység valóban jól működik-e). Ez most bonyolultnak hangzik, de szerencsére a legtöbb modern keretrendszerben van rá könyvtár támogatás (.NET-ben a moq). Ennek alkalmazása túlmutat a tárgy keretein, így a feladatunk egységtesztelhetőséghez kapcsolódó vonulatát ebben a pontban lezárjuk.

A feladat végeztével, a kimeneti fájl tartalmának ellenőrzésével mindenképpen győződj meg arról, hogy az anonimizálás valóban lefut!

3. feladat BEADANDÓ

- Illessz be egy képernyőképet, melyen az `Anonymizer` osztály konstruktora és a `Run` függvény implementációja látszik (`f3.1.png`).

4. Feladat - Delegate-ek alkalmazása

Napjainkban rohamosan terjed a korábban szigorúan objektumorientált nyelvekben is a funkcionális programozást támogató eszközök megjelenése, és az alkalmazásfejlesztők is egyre nagyobb szeretettel alkalmazzák ezeket (merthogy sokszor jelentősen rövidebb kóddal, kisebb "ceremóniával" lehet ugyanazt segítségükkel megvalósítani). Egy ilyen eszköz C# nyelven a delegate, és ehhez kapcsolódóan a lambda kifejezés.

Mint a félév során korábban láttuk, delegate-ek segítségével olyan kódot tudunk írni, melybe bizonyos logikák/viselkedések nincsenek beégetve, ezeket "kívülről" kap meg a kód. Pl. egy sorrendező függvénynek delegate formájában adjuk át paraméterként, hogyan kell két elemet összehasonlítani, vagy mely mezője/tulajdonsága szerint kell az összehasonlítást elvégezni (így végső soron meghatározni a kívánt sorrendet).

Ennek megfelelően a delegate-ek alkalmazása egy újabb alternatíva (a Template Method és a Strategy mellett) a kód újrafelhasználhatóvá/kiterjeszthetővé tételére, kiterjesztési pontok bevezetésére.

A következő lépésben a korábban Strategy mintával megvalósított progress kezelést alakítjuk át delegate alapúra (új funkciót nem vezetünk be, ez egy pusztán "technikai" átalakítás lesz).

Feladat: Alakítsd át a Strategy-DI projektben található megoldást olyan módon, hogy a progress kezelés Strategy helyett delegate alapon legyen megvalósítva. Részletesebben:

- Ne vezess be saját delegate típust (használd a .NET által biztosított `Action` típust).
- A meglévő `SimpleProgress` és `PercentProgress` osztályokat ne használd a megoldásodban(de ne is töröld ezeket!).

- Legyen lehetősége az `Anonymizer` használójának továbbiakban is `null`-t megadni a konstruktorban, ha nem kíván semmiféle progress kezelést használni.
- A `Program.cs` fájlban kommentezd ki az eddigi `Anonymizer` használatokat. Ugyanitt vezess be egy új példát az `Anonymizer` olyan használatára, melyben a progress kezelés lambda kifejezés formájában van megadva, és a lambda kifejezés pontosan a korábbi "simple progress" logikáját valósítja meg. A "percent progress"-re nem kell hasonlót megvalósítani, azt ebben a megoldásban nem kell támogatni (a következő feladatban térünk vissza rá).

Tippek

- A delegate alapú megoldás alapelve nagyon hasonlít a Strategy-hez: csak nem strategy-ket kap és tárol az osztály tagváltozókban (interfész hivatkozásokon keresztül), hanem delegate-eket, és az ezek által hivatkozott függvényeket hívja a kiterjesztési pontokban.
- Ehhez hasonlót már csináltál is a 2. házi feladatban a ReportPrinter részben ;).

4. feladat BEADANDÓ

- Illessz be egy képernyőképet, melyen az `Anonymizer` osztály konstruktora és a `Run` függvény implementációja látszik (`f4.1.png`).
- Illessz be egy képernyőképet, melyen a `Program.cs` fájl tartalma (különösen az új részek) látszik (`f4.2.png`).

5. Feladat - Delegate-ek alkalmazása újrafelhasználható logikával

Az előző feladatban feltettük, hogy a "simple progress" és a "percent progress" logikáját csak egyszer használtuk, így nem kellett újrafelhasználhatóvá tenni. Ennek megfelelően pl. a "simple progress" logikáját a lehető legegyszerűbb formában, egy lambda kifejezéssel adtuk meg (nem kellett külön függvényt bevezetni rá). Amennyiben az `Anonymizer` létrehozásakor a delegate-nek mindig más és más implementációt adunk meg, akkor ez a lambda alapú megoldás tökéletes.

Viszont mi a helyzet akkor, ha a fenti példában szereplő "simple progress" logikát több helyen, több `Anonymizer` objektumnál is fel szeretnénk használni? Súlyos hiba lenne a lambda kifejezést copy-paste-tel "szaporítani", kódduplikációhoz vezetne (ellentmondana a "**Do Not Repeat Yourself**", röviden **DRY** elvnek).

Kérdés: van-e megoldás arra, hogy delegate-ek esetében is újrafelhasználható kódot adjunk meg? Természetesen igen, hiszen delegate-ek esetében nem kötelező a lambda kifejezések használata,

lehet velük közönséges műveletekre (akár statikus, akár nem statikusakra is), mint azt korábban a félév során láttuk, és számos esetben alkalmaztuk is.

Amennyiben a "simple progress" és/vagy "percent progress" logikát/logikákat újrafelhasználhatóvá szeretnénk tenni delegate-ek alkalmazásakor, tegyük ezeket egy külön függvényekbe valamilyen, az adott esetben leginkább passzoló osztályba/osztályokba, és egy ilyen műveletet adjuk meg az `Anonymizer` konstruktornak paraméterként.

Feladat: Bővítsd ki a korábbi megoldást úgy, hogy a "simple progress" és "percent progress" logikája újrafelhasználható legyen. Részletesebben:

- A "simple progress" és "percent progress" logikákat egy újonnan bevezetett `AllProgresses` nevű statikus osztály két statikus műveletében valósítsd meg (az osztály a projekt gyökerébe kerüljön).
- Vezess be két olyan új `Anonymizer` használatot a `Program.cs` fájlban a meglévők mellé, melyek az `AllProgresses` két műveletét használják (itt ne használj lambda kifejezést).
- A meglévő `IProgress` interfészt és ennek implementációi törölhetők lennének (hiszen ezek már nincsenek használatban). De NE töröld őket annak érdekében, hogy a korábbi megoldásodhoz tartozó progress logika is ellenőrizhető legyen.

Elkészültünk, értékeljük a megoldást:

- Kijelenthető, hogy a delegate alapú megoldás a Strategy-nél kisebb ceremóniával járt: nem kellett interfészt és implementációs osztályokat bevezetni (a beépített `Action` és `Func` generikus delegate típusokat tudtuk használni).
- A teljesen "eseti" logikát lambda kifejezés formájában legegyszerűbb megadni. Ha újrafelhasználható logikára van szükség, akkor viszont vezessünk be "hagyományos", újrafelhasználható függvényeket.

5. feladat BEADANDÓ

- Illessz be egy képernyőképet, melyen az `AllProgresses.cs` fájl tartalma látszik (`f5.1.png`).
- Illessz be egy képernyőképet, melyen a `Program.cs` fájl tartalma (különösen az új részek) látszik (`f5.2.png`).

Refaktorálás (Refactoring) fogalma

A labor és a házi feladat megvalósítása során számos olyan lépés volt, mely során a kódot úgy alakítottuk át, hogy az alkalmazás külső viselkedése nem változott, csak a belső felépítése.

Mégpedig annak érdekében, hogy valamilyen szempontból jobb kódminőségi jellemzőkkel rendelkezzen. Ezt a kód `refaktorálásának` (angolul `refactoring`) nevezzük. Ez egy nagyon fontos fogalom, a mindennapi munka során nagyon gyakran használjuk. Külön irodalma van, a fontosabb technikákkal a későbbiekben érdemes megismerkedni. A komolyabb fejlesztőeszközök beépítetten támogatnak bizonyos refaktorálási műveleteket: a Visual Studio ebben nem a legerősebb, de azért pár alapműveletet támogat (pl. Extract Method, Extract base class stb.). Manuálisan gyakoroltuk, ennek kapcsán külön feladatunk nem lesz, de a Refaktorálás fogalmát ismerni kell.

6. Opcionális feladat - Integrációs teszt készítése

A feladat megoldásával +1 IMSc pont szerezhető.

A korábbi, 3. feladat során ismertetésre került az integrációs teszt fogalma. Jelen opcionális feladat célja ennek gyakorlása, jobb megértése egy egyszerű feladaton keresztül.

Készíts egy integrációs tesztet az `Anonymizer` osztályhoz, a következők szerint:

1. A Solutionben a `Test` mappában előkészített `IntegrationTest` projektben dolgozz. Ez egy NUnit teszt projekt.
2. Ebben a projektben már előre felvettünk egy projekt referenciát a `Strategy-DI` projektre, így látjuk a `Strategy-DI` projektben levő (publikus) osztályokat. Értelemszerűen ez előfeltétele annak, hogy tudjuk tesztelni őket. Ellenőrizd a projekt referencia meglétét (Solution Explorerben a projekt alatt a Dependencies/Projects csomópont).
3. Az `AnonymizerIntegrationTest` osztályban már van egy `Anonymize_CleanInput_MaskNames_Test` nevű tesztelést végző művelet (a teszt műveleteket `[Test]` attribútummal kell ellátni, ez erre a műveletre már elő van készítve). A művelet törzse egyelőre üres, ebben kell dolgozni a következő lépésekben.
   a. Hozz létre egy `Anonymizer` objektumot, mely
      - a `@"TestFiles\us-500-01-clean.input.csv"` bemenettel dolgozik (ez megtalálható a projekt *TestFiles* mappájában, nézd meg a tartalmát),
      - a kimente legyen a `@"us-500-01-maskedname.processed.txt"` fájl,
      - "***" paraméterű `NameMaskingAnonymizerAlgorithm`-t használ.
   b. Futtasd az anonimizálót a `Run` műveletének hívásával, hogy álljon elő a kimenti állomány.
   c. Az `Assert.AreEqual` hívással ellenőrizd, hogy az anonimizálás során előállt kimeneti állomány tartalma megegyezik-e a várt tartalommal. A várt tartalom a `@"TestFiles\us-500-01-maskedname.processed-expected.txt"` fájlban érhető el (ez

megtalálható a projekt `TestFiles` mappájában, nézd meg a tartalmát). Tipp: egy fájl tartalmát pl. a `File.ReadAllBytes` statikus művelettel egy lépésben be lehet olvasni.

4. Ellenőrizd, hogy az integrációs teszt hiba nélkül lefut.
   a. Buildeld meg a projektet
   b. Nyisd meg a Test Explorert (Test/Test Explorer menü)
   c. A teszt futtatására a Test Explorer nézet tetején található eszközsávon levő gombokkal van lehetőség. De a teszt debuggolására is van lehetőség, jobb gombbal a tesztre kattintva és a Debug menü kiválasztásával: ez nagyon hasznos tud lenni, ha a tesztünk hibásan fut, és szeretnénk töréspontok segítségével a kódon lépkedni, illetve a változók értékét megnézni.
   d. Ha a teszt hiba nélkül fut le, a teszthez tartozó ikon zöld lesz. Ha hibával, akkor piros, és a hibaüzenetről a tesztet kiválasztva Test Explorer nézet alján kapunk bővebb információt.

7. Opcionális feladat - Unit teszt készítése

A feladat megoldásával +2 IMSc pont szerezhető.

A korábbi, 3. feladat során ismertetésre került az egységteszt fogalma. Jelen opcionális feladat célja ennek gyakorlása, jobb megértése egy feladaton keresztül.

Előkészítés:

1. Vegyél fel a solution-be egy új "NUnit Test Project" típusú projektet "UnitTest" néven (jobb katt a Solution-ön a Solution Explorerben/Add/New Project).
2. Ebben az új projektben vegyél fel projekt referenciát a `Strategy-DI` projektre, hogy a projektben elérhetők legyenek a `Strategy-DI` -ben definiált típusok (jobb katt a Unit Test projekt Dependencies csomópontján/Add Project Reference, a megjelenő ablakban pipa a `Strategy-DI` projekten, "OK").
3. A projektben születik egy `UnitTest1.cs` állomány, benne egy `Test` osztály. Ezeket célszerű `AnonymizerTest` -re nevezni.

Készíts egy egységtesztet az `Anonymizer` osztályhoz, mely ellenőrzi, hogy a `Run` művelete pontosan azokkal a személy adatokkal hívja meg sorrendhelyesen az anonimizáló algoritmust, melyeket az `Anonymizer` a bemenetén beolvas (amennyiben nincsenek trimmelendő városnevek).

- A tesztfüggvény neve legyen `RunShouldCallAlgorithmForEachInput` .
- ❗ Alapvető fontosságú, hogy nagyon gyors egységtesztet kell írni, nem integrációs tesztet: tehát csak a `Run` logikáját akarjuk önmagában tesztelni, mindenféle fájlfeldolgozás nélkül. A

megoldásban semmiféle fájlkezelés nem lehet!

- Tipp: Memóriában hozz létre 2-3 `Person` objektumot, ezekkel dolgozz bemenetként.
- Tipp: Olyan bemenő személyadatokkal dolgozz, melyekre a `TrimCityNames` függvénynek nincs hatása (vagyis nincsenek benne áltávolítandó adatok), ez egyszerűbbé teszi a tesztelést.
- Tipp: Olyan `IInputReader`, `IAnonymizerAlgorithm` implementációkat hozz létre (és az `Anonymizert` ezekkel használd), **melyek megfelelő tesztadatokat biztosítanak, és/vagy futás közben adatokat gyűjtenek annak érdekében, hogy a futás után ellenőrizni tudd ezen adatok alapján, hogy a tesztelendő feltételek teljesülnek**. Ezeket a strategy implementációkat mindenképpen a teszt projektben vedd fel, mert csak a tesztelést szolgálják.

További gyakorlásképpen készíthetsz egy olyan másik egységtesztet, mely azt ellenőrzi, hogy minden bemeneti személyadat eljut-e a kimenetre is.

Összegzés

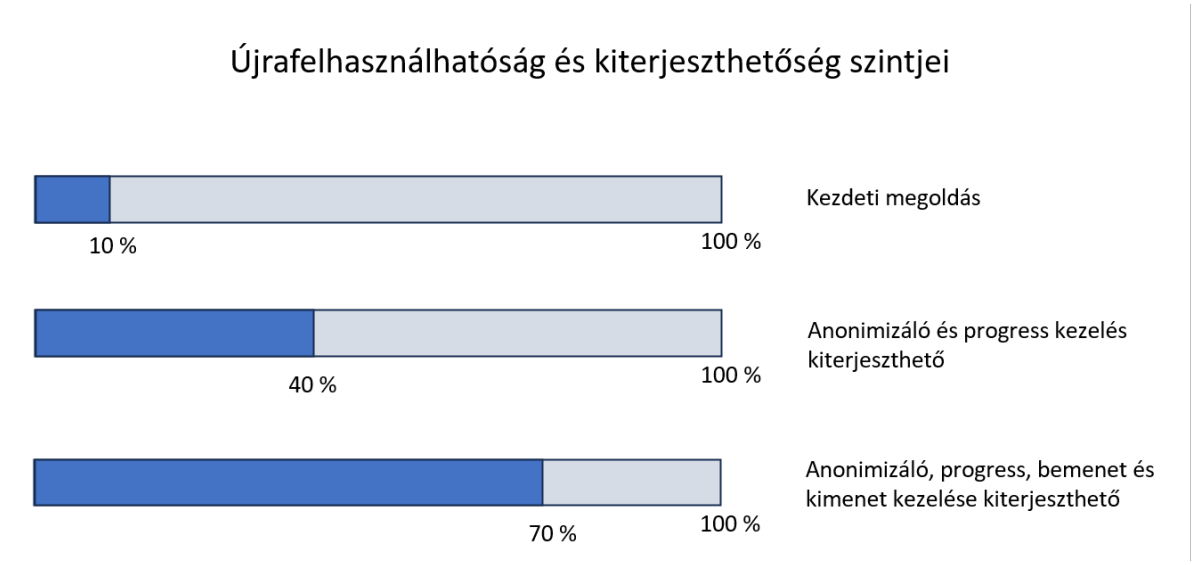
Több feladat nem lesz 😊. De ha kíváncsi vagy pl. arra, hogy jelen megoldás mennyire tekinthető "tökéletesnek"/hiányosnak, illetve mikor érdemes Template Methoddal, Strategyvel, vagy inkább delegate-ekkel dolgozni, akkor érdemes elolvasnod az alábbiakat, melyben értékeljük a laboron elkezdett és a házi feladat keretében befejezett megoldást.

A munkafolyamatunk áttekintése

- A változó igények során organikusan jelentek meg tervezési minták, és vezettünk be egyéb technikákat a refaktorálások során. Ez teljesen természetes, a gyakorlatban is sokszor így dolgozunk.
- Egy komplexebb feladat esetében egyébként is sokszor - különösen ha nem rendelkezünk sokéves tapasztalattal - egy egyszerűbb implementációval indulunk (ezt látjuk át elsőre), és alakítjuk át olyanra, hogy az adott kontextusban kívánt kiterjeszthetőségi/ újrafelhasználhatósági paraméterekkel rendelkezzen.

Újrafelhasználhatósági és kiterjeszthetőség szintjei az egyes megoldásokban

Megpróbálhatjuk ábrába önteni, hogy vált a megoldásunk az egyes iterációkkal egyre inkább újrafelhasználhatóvá és kiterjeszthetővé:



Természetesen a % szinteket nem szabad túl komolyan venni. Mindenesetre a fejlődés jól megfigyelhető.

Miért "csak" 70%-os a végső megoldásnál mutatónk?

Felmerülhet a kérdés, miért adunk jelem megoldásra kb. 70%-ot? Többek között:

- Az `Anonymizer` osztályba az adattisztítás módja mereven be van égetve (trimmelés adott oszlopra adott módon).
- Nem követtünk egy nagyon fontos általános alapelvet: a UI és a logika különválasztását. A kódunk több pontban konzolra ír, így például egy grafikus felülettel nem használható!
- Bizonyos az anonimizáló algoritmusaink nagyon specifikusak. Lehetne olyan általánosabb algoritmusokat készíteni, melyek tetszőleges mezőket kicsillagoznak (nem csak a nevet beégetetten), illetve tetszőleges mezőket sávosítanak (nem csak az életkort).
- Jelen megoldás csak `Person` objektumokkal tud működni.
- Nem lehet egyszerre alkalmazni kombinálni különböző anonimizáló algoritmusokat.

Kiterjesztési technikák áttekintése

- **Template Method**: Egyszerű esetben, ha a viselkedések különböző aspektusainak nem kell sok keresztkombinációját támogatni, nagyon kényelmes és egyszerű megoldást ad, különösen, ha egyébként is kell használjuk a származtatást. De nem, vagy csak nehezen egységtesztelhető alaposztályt eredményez.
- **Strategy**: Nagyon rugalmas megoldást biztosít, és nem vezet kombinatorikus robbanáshoz, ha

több aspektus mentén kell az osztályt kiterjeszteni, és több keresztkombinációban is szeretnénk ezeket használni. Sok esetben csak azért alkalmazzuk, hogy az osztályunkról interfészek segítségével leválasszuk a függőségeit, és így egységtesztelhetővé tegyük az osztályunkat.

- **Delegate/lambda**: Ez a megközelítés kisebb ceremóniával jár, mint a Strategy alkalmazása, ugyanis nincs szükség interfészek és implementációs osztályok bevezetésére, emiatt egyre inkább (rohamosan) terjed a használata a modern objektumorientált nyelvekben is. Különösen akkor jönnek ki az előnyei, ha a viselkedéseket nem akarjuk újrafelhasználhatóvá tenni (mert ekkor csak egy-egy lambda kifejezéssel megadjuk ezeket, mindenféle új osztályok/külön függvények bevezetése nélkül).

Érdemes összeszedni, hogy a Strategy-nek mikor lehet/van van előnye a delegate-ekkel szemben:

- Ha kiterjesztendő osztály adott aspektusához több (minél több, annál inkább) művelet tartozik. Ilyenkor a strategy interfész ezeket "magától" szépen összefogja, csoportosítja (mint a példánkban az `IAnonymizerAlgorithm` interfész az `Anonymize` és `GetAnonymizerDescription` műveleteket). Ezek értelemszerűen az interfész implementációkban is együtt jelennek meg (delegate-ek esetében nincs ilyen csoportosítás). Ez átláthatóbbá teheti, sok művelet esetén egyértelműen azzá is teszi a megoldást.
- Az adott nyelv pusztán objektumorientált, nem támogatja a delegate/lambda alkalmazását. De ma már a legtöbb modern OO nyelv szerencsére támogatja valamilyen formában (Java és C++ is).
- A strategy implementációk a tagváltozóikban állapotot is tudnak tárolni, melyet létrehozásukkor meg tudunk adni. Ezt használtuk is (a `NameMaskingAnonymizerAlgorithm` esetében ilyen volt a `_mask`, a `AgeAnonymizerAlgorithm` esetében a `_rangeSize`). Ez nem azt jelenti, hogy ilyen esetben egyáltalán nem tudunk delegate-eket használni, hiszen:
  - ezeket az adatokat akár újonnan bevezetett függvény paraméterben is átadhatjuk az egyes delegate hívások során,
  - illetve, lambda használata esetén a "variable capture" mechanizmus segítségével a lambda függvények tudnak állapotot átvenni környezetükből.

  De ezek a megoldások nem mindig alkalmazhatók, vagy legalábbis körülményes lehet az alkalmazásuk.

Mindenképpen meg kell említeni, hogy nem csak jelen gyakorlatban említett néhány minta szolgálja a kiterjeszthetőséget és újrafelhasználhatóságot, hanem gyakorlatilag az összes. Most kiemeltünk párat, melyek (még p. az Observert/Iteratort/Adaptert ide sorolva) talán a leggyakrabban, legszélesebb körben alkalmazhatók és bukkannak is fel keretrendszerekben.

Ha idáig olvastad, mindenképpen jár egy extra thumbs up 👍!

2024-05-11 Szerzők