Tervezés I.

# Tartalomjegyzék

# Bevezetés

Mivel a személyes vagyon nyilvántartása pénzügyi alkalmazások nélkül hosszú és rengeteg odafigyelést igénylő folyamat, így az emberek többsége nem is veszi a fáradságot e nyomon követések elkészítésére. Tovább bonyolítja a helyzetet, ha egy adott személy, aki szeretné nyomon követni a pénzügyeit, de tegyük fel, hogy több banknál is rendelkezik fiókkal, vagy egy család tagjai nem ugyan annál a banknál rendelkeznek számlával mégis egyszerűen és gyorsan szeretnék megtekinteni az összesített pénzügyi kimutatásaikat. Az effajta problémák áthidalására jöttek létre a személyes pénzügyi alkalmazások.

Tehát személyes pénzügyi szoftverek elsősorban arra a célra lettek kifejlesztve hogy egyszerűbbé tegyék a felhasználók számára pénzügyeik nyomon követését és bizonyos funkciók nyújtásával segítsék beosztani, esetleg jobbá tenni azt. Ezek a funkciók többek között lehetnek:

* Ha a banki tranzakciókat tekintjük, például számon tarthatjuk mennyi pénzünk van jelenleg a bankszámlánkon, mennyi pénzt tudtunk félretenni bizonyos hónapokban, milyen kategóriájú termékekre költöttünk a legtöbbet, hány tranzakció volt a számlánkon bizonyos hónapokban, napokban.
* Ha tőzsdei tranzakciók nyomon követéséről beszélünk az alkalmazás segítségével meg tudjuk jeleníteni a jelenleg birtokolt részvényeket és azokhoz tartozó darabszámokat illetve a jelenlegi részvényárfolyamot is. Le tudjuk kérni a birtokolt részvényekhez tartozó jelenlegi és egy bizonyos intervallumra visszamenő legmagasabb, legalacsonyabb, nyitó és záró árakat. Valamint a felhasználó által eladott részvényekhez nyereség-veszteség kiszámolására is van lehetőség.

Ezekből az adatokból olyan következtetéseket tud levonni a felhasználó, amelyek segítik a sikeres pénzbeosztását, esetleges spórolást illetve a tőzsdei nyereségek esélyét. De elsősorban segítik a pénzügyi tudatosság kialakulását.

Bizonyos applikációk felajánlanak pénzspórolási tanácsokat is, amiket a felhasználó tranzakcióinak statisztikai elemzéseiből nyernek ki. Habár e tanácsok többsége fizetős funkció.

A legnépszerűbb pénzügyi szoftverek legfőbb tulajdonsága hogy a felhasználók egy helyen láthatják az összes bankszámlájuk tranzakcióját, költési szokásukat és egyéb kimutatásokat. Így egy átfogó képet kapva a személyes pénzügyeikről, ezáltal segítve annak nyomon követését és esetleges jobbá tételét is.

Fontos megjegyezni, hogy manapság az alkalmazások nagy része elérhető telefonos applikáció formájában is. Mivel a mai emberek azonnali elérhetőséget kívánnak a nap bármely pontjában, főként, ha a személyes pénzügyeik alakulásáról van szó. Tehát a sikeres pénzügyi alkalmazások mind elérhetőek telefonos készülékre is.

Az első fejezetben a feladat rövid leírása található, leírva a fő problémát, annak okát, illetve tisztázva azokat a funkciókat, amivel a programnak rendelkeznie kell.

A második fejezetben bemutatásra fog kerülni pár elterjedtebb pénzügyi alkalmazás, azok funkcióival, pozitív illetve negatív tulajdonságukkal egyaránt. Illetve ezen alkalmazások összehasonlítása táblázat formájában.

A harmadik fejezetben bemutatásra kerülnek a szoftver képernyőtervei, illetve az azokhoz kötött funkcionális követelményeiket is, figyelembe véve a második fejezetben olvasottakkal.

A negyedik fejezetben leírásra kerülnek a program megírása alatt használt szoftvereket és e szoftverek ismertetése, illetve hogy miért ezeket az alkalmazásokat használtam a szakdolgozatok megírásához.

Az ötödik fejezetben felsorolásra kerülnek a jövőbeli tervek, fejlesztési irányok a jelenleg meglévő szoftvert illetően.

# Feladat leírása

A jelenleg piacon lévő pénzügyi alkalmazások mind nagyon különbözőek, mind megjelenítésben, mint nyújtott funkciók terén. De egy közös tulajdonsággal mind rendelkeznek, méghozzá azzal hogy mind képesek feldolgozni és megjeleníteni a felhasználó által beimportált tranzakciókat. Egy számlaaggregátor applikáció kifejlesztését megkönnyíti az a tény, hogy a felhasználó által beimportált fájlok banktól függetlenül ugyan azzal a formátummal rendelkeznek. Sajnos a Magyarországon jelenlévő bankoktól és tőzsdei programoktól letölthető tranzakciós fájlok felépítése nincs egységesítve. Tehát minden fájl felépítése, struktúrája más és más.

Célom egy olyan importáló szoftver írása, amely képes feldolgozni bármely magyar banktól, illetve Magyarországon elérhető tőzsdei programból exportált számlakivonatot tartalmazó fájl adatait. Majd ezen adatokat eltárolni egy egységes formátumban lévő adatbázisba. Majd azokat egy táblázat formájában megjeleníteni.

Valamint az eladott tőzsdei tranzakciók esetén megjeleníteni a nyereség illetve veszteség jelenlegi állását. Ahol a kiszámolási módszert a felhasználó választja ki.

A FinTech (pénzügyi technológia) egy fejlődésben lévő iparág, mely hatékonyabbá teheti a pénzügyi szektort a digitális kor innovációinak felhasználásával. Ennek hatására, kiszorítva azon pénzügyi szektorban jelenlévő, úgymond kevésbé informatikai szemléletben gondolkozó szereplőket.

A fintech cégek között megkülönböztetnünk két alkategóriát.

* Enabler: a hagyományos pénzügyi rendszer résztvevőit (bankok, biztosítok stb.) támogatják, segítik annak érdekében, hogy a FinTech cégek által támasztott kihívásoknak meg tudjanak felelni.
* Disruptor: hagyományos pénzügyi intézmények (bankok, biztosítok stb.) versenytársainak tekinthetők.

A bankszektor alapvetően konzervatív jellege miatt sokáig ellenkezett a technológiai újításokkal szemben. De a mai digitalizált világban a konzervatív nézéspont nem túl jó tulajdonság.

Senkinek sem okoz nagy meglepetést, hogy a banki szektor a leginkább szabályozott intézmények közé tartozik. Tehát egy-egy újítás nehezen építhető be a fennálló jogi környezetbe. Viszont új típusú fogyasztói igények megjelenése és a gyorsan fejlődő technológia azt eredményezi, hogy a FinTech iparág az egyik legnépszerűbb új technológia lett a befektetők körében (főként külföldön). Tehát a bankok most azzal a választással szembesülnek, hogy vagy alkalmazkodnak a digitális forradalomhoz, vagy elveszítik a piaci részesedésüket.

A FinTech előnyei annyira ígéretesek hogy szinte elfeledkezünk az esetleges hátrányaikról, ami nagy hiba lenne.

* Az online elérhető adatok könnyen ellophatók harmadik féltől. Más jövedelmező célokra vagy akár személyazonosság-lopáshoz is felhasználhatók

A FinTech cégek évtizedes kriptográfiai és biztonsági kutatások csúcspontjára épülnek és bár az SHA-256 titkosítást jelenleg nagyon erősnek tartják, még mindig vannak bizonyos sebezhető területek. Ami egy nagy hátrány, mivel az emberek hajlandóak szkeptikusak kezdetben a technológiai újításokkal szemben, főleg ha a pénzügyeikről van szó. De mentségükre szóljon, hogy veszélymentes rendszer nem létezik.

## Blokklánc technológia

A blokklánc technológia (blockchain) egy főként FinTech cégek által használt újítás, ami elsősorban a pénzügyi válság után vált ismerté. Fő előnye a nyílt forráskódú fejlesztés, ami lehetővé teszi a tranzakciók elszámolását közvetítők bevonása nélkül. Ebből következik, hogy töredékére csökkennek az elszámolási idők. Tehát ami eddig napokba telt, az a blokklánc technológiával közel valós időben teljesül.

## FinTech Magyarországon

Magyarországon a tavalyi év (2017) szeptemberében benyújtásra került egy törvényjavaslat a bankok és a FinTech cégek számára. Ha a PSD2 nevű irányelv elfogadásra kerül az a két fő dolgot jelent.

* Nagyobb versenyt a bankok számára.
* A bankolást megújító innovatív cégeknek pedig könnyebb piacra lépést.

A PSD2 irányelv elsődleges célja a belső piac további erősítése. Ami előnyös, mind a fogyasztók, mind a kereskedők és a vállalkozók számára is.

Az irányelv új típusú szolgáltatók megjelenését is lehetővé teszi. Az úgynevezett külső szolgáltatók kétféle új szolgáltatást nyújthatnak: számlainformációs és fizetéskezdeményezési opciót. Ezekhez az ügyfelek engedélyével banki adatokat lehet lekérdezni, vagyis a FinTech szolgáltatók hozzáférést kapnak a banki rendszerekhez.

Ez a gyakorlatban olyan számlaaggregátor szolgáltatók megjelenését jelenti, melyek alkalmasak a személyes pénzügyi alkalmazások által nyújtott funkciók megvalósítására.

# Irodalmi áttekintés

Mivel a személyes pénzügyek a legtöbb ember számára fontos, így nagy igény van ilyen szoftverekre, főleg napjainkban, ahol az emberek kényelemben szeretik élni mindennapjaikat. Tehát egy át fogó képet szeretnének látni pénzügyeikről, mindössze pár kattintás segítségével.

Természetesen számos ilyen alkalmazás létezik. Nem szokatlan hogy egy felhasználó egyszerre használ több pénzügyi alkalmazást, mivel amit az egyik változat tud, azt nem biztos, hogy a másik is.

Habár, a pénzügyi applikációk mindegyike rendelkezik Tranzakciók Importálása menüponttal, ahol pár kattintás segítségével új tranzakciókat tudunk felvinni a rendszerbe, valamint Táblázat menüponttal, ami tartalmazza a beimportált tranzakciókat. Illetve könnyedén előfordulhat, hogy egy család használja ugyan azt az alkalmazást és a közös kasszájukat követik nyomon. Tegyük fel hogy, a család minden tagja külön banknál rendelkezik számlával (ami könnyedén előfordulhat). Így egy ilyen szoftvernek képesnek kell lennie több bank adatait is feldolgozni. Ha egy alkalmazás nem rendelkezik e funkciókkal, nem lesz túl népszerű a felhasználók körében.

Természetesen egy pénzügyi szoftvert népszerűvé tehet az általa nyújtott funkciók sokasága is. Például hogy nem csak bankoktól származó adatok, hanem tőzsdei tranzakciók kezelésére is képes. Valamint nyereség-veszteség kiszámolására is használható. Gyakori a Pénztárca funkció is, ami segítségével a felhasználók saját kedvűkre megszabhatnak egy úgymond képzeletbeli korlátot bizonyos terméktípusokra, amit nem szabad túllépniük egy általuk megszabott ideig.

Habár ezen applikációk többségének kipróbálása ingyenes, természetesen léteznek fizetős változatok is. Fizetős funkciók között szerepelhet a számlabefizetés, szöveges értesítés telefonra bizonyos tranzakciók esetén stb..

Tehát rengeteg tulajdonság tehet egy pénzügyi alkalmazás sikeressé, avagy kevésbé népszerűvé. A következőekben négy darab szoftvert mutatok be azok pozitív illetve negatív tulajdonságait bemutatva.

## Mint.com

A Mint.com egy olyan alkalmazás, amely többek között segít beosztani a pénzünket, emlékeztet a számlák befizetésére. Sőt, egy régi frissítésnek köszönhetően, szigorúan csak a felhasználó engedélyével, már be is fizethet helyettünk számlákat. Természetesen képes több bankszámlához származó adat eltárolására is. Csak a számlaforgalmat látja, azt kezelni nem tudja (számlabefizetésen kívül, ha a felhasználó igény tart erre a funkcióra).

A szoftver az első importálás után egy algoritmus segítségével megvizsgálja a felhasználó korábbi tranzakcióit, ezáltal megtudva költési szokásait. Így adva esetleges spórolási illetve költési tippeket a felhasználó számára. Az alkalmazás rendelkezik egy pontrendszerrel, ahova minden felhasználó összesített értékelése kerül be (spórolás, pénztárca korlát betartása stb. alapján kerül kiszámításra).

A Mint.com létezik telefonos alkalmazás formájában is. Sajnos a magyar bankok kezelésére nem képes, viszont a PayPal fiókokat támogatja, ami Magyarországon is elérhető.



Ábra: Mint.com kezdőoldala.

Az ábrán látható menüpont a Mint.com kezdőoldala. Láthatjuk, a szoftver e menüpontja tartalmazza az éppen aktuális Pénztárca állását, felhasználó pontozását. Továbbá láthatjuk az esedékes számlabefizetések dátumát (négy hétre előretekintve) illetve azok pénzösszegeit is megjelenítve. Valamint az elmúlt fél év statisztikáit jövedelem szempontjából egy grafikonon megjelenítve.

**Pozitív tulajdonságok**

* Könnyen átlátható, rengeteg oktató videó található az interneten.
* Többlépcsős beléptető rendszerrel rendelkezik.
* Használta ingyenes.
* Rendelkezik tőzsdei tranzakció importálás funkcióval.
* Létezik telefonos alkalmazás formájában is.

**Negatív tulajdonságok**

* Csak az Amerikai bankkártyák adatait tudja kezelni.
* Csak ellenösszeg fejében kapunk tanácsokat a spóroláshoz.

## YNAB (You Need A Budget)

A YNAB a nevéhez hűen, csak egy pénztárca létrehozásához és kezeléséhez jó. De arra tökéletesen megfelel. Feltéve, ha az ember elég időt szán az alkalmazás megismerésére. A program, regisztrálástól számítva 34 napig ingyenes. Majd havi 5 dollár az előfizetés. Főként telefonos applikációnak tervezték, de elérhető asztali alkalmazásként is.

Habár a felhasználók többsége szerint a szoftver hatékony használatának megtanulása sok időt emészthet fel, segítségünkre lehetnek az útmutató videók, amiből az interneten rengeteg található. A YNAB a pénzünk jelenlegi helyzetével foglalkozik, és nem tekint előre esetleges jövőbeli számlákra, fizetés stb. (a Mint.com-al ellentétben). Valamint nem rendelkezik tőzsdei tranzakciók importálásának lehetőségével.



Ábra: a YNAB kezdőoldala.

Az ábrán láthatjuk a felhasználó által létrehozott Pénztárca tulajdonságait. Ahol láthatjuk a pénztárcánk maximális korlátait, az ebből már elköltött pénzösszeget illetve a kategóriánkénti aktivitásunkat.

Bal oldalt láthatjuk a fontosabb adatokat:

* Jelenleg rendelkezünk 3,362.01 $-al
  + Amiből jövedelem 1,740.00 $
  + A spórolt pénzösszeg összesen 5,247.23 $
  + A tartozásunk (hitelkártyán) összesen $3,625.22$

**Pozitív tulajdonságok**

* Személyes „pénztárca” alkalmazás.
* Spórolásra koncentrál.
* Rendelkezik ingyenes próbaidővel.
* Sok oktató tartalom található róla az interneten.

**Negatív tulajdonságok**

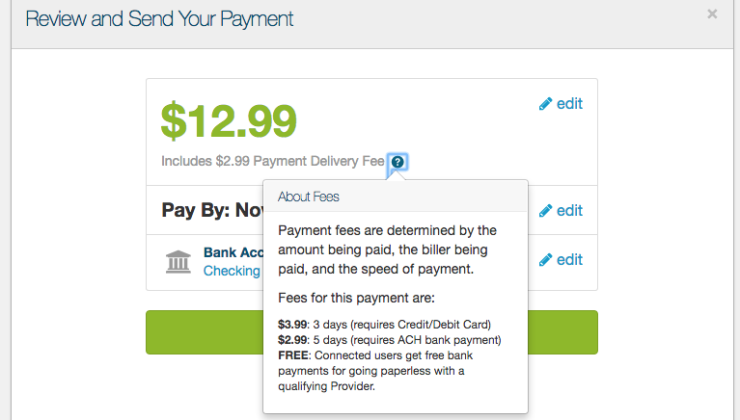
* Sok idő, míg a felhasználó meg tanulja hatékonyan használni a funkciókat.
* Nincs többlépcsős belépés.
* Főként költésre van kiélezve.
* Nem képes tőzsdei tranzakciók kezelésére

## Doxo

A Doxo egy olyan pénzügyi alkalmazás, ami főként a számla befizetésre és a banki számlához tartozó dokumentumok tárolására van kiélezve. Egy kifejezetten jó alkalmazás azon emberek számára, akik nem szeretnének többé papír alapú tranzakciós dokumentumokat (például számlákat) tárolni.

Habár az alkalmazás használata alapvetően ingyenes, rendelkezik fizetős funkciókkal is például számlabefizetés. Ami a Doxo egyik meghatározó funkciója.

Sajnos amilyen jól hangzik a számlabefizetés funkció, annyira rosszul működik. Mivel nagyon lassú. Illetve, esetleges lefagyások is bekövetkezhetnek, ami azt eredményezheti, hogy nem megy végbe az átutalás. Illetve az alkalmazáson keresztül való befizetésért számláktól függően külön kezelési költséget számol fel az alkalmazás. Valamint a kezelési költség csak a számla befizetésének végbemenetele után jelenik meg.



Ábra: Doxo számla befizetés funkciója.

**Pozitív tulajdonságok**

* Rendelkezik számla-befizetési opcióval.
* A használata nagy részben ingyenes.
* Tud tárolni blokkokat és egyéb dokumentumokat, amik a bankszámláinkhoz tartoznak.

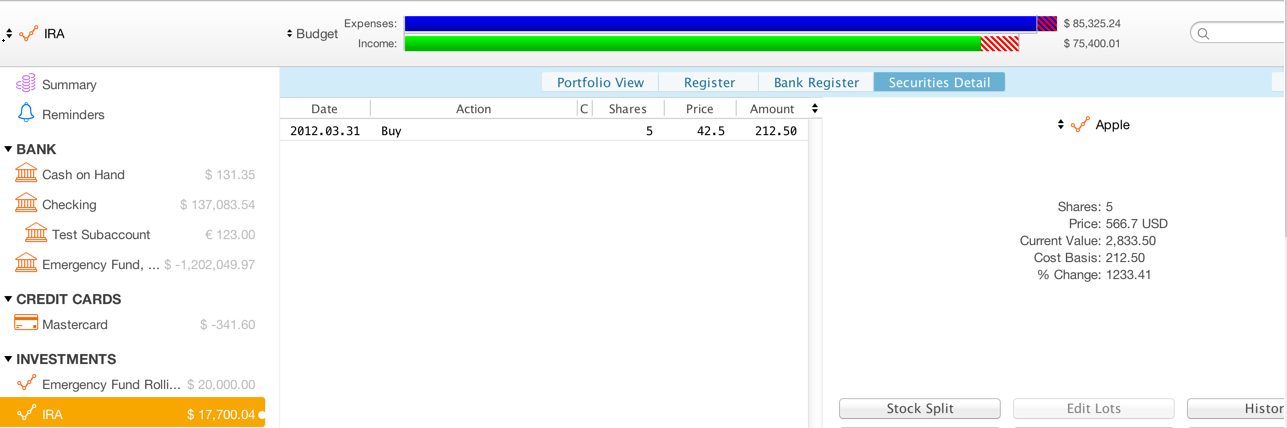
**Negatív Tulajdonságok**

* Lassú lehet bizonyos funkciók esetén például számla befizetés, tranzakciók importálás (bank függő).
* Sok negatív vélemény (felhasználók befizették a számláikat, de a pénz sosem érkezett meg).

## Quicken Premier

A Quicken Premier az egyik legrégebb óta piacon lévő pénzügyi alkalmazás. Legfőképp azon felhasználók számára ajánlott, akik már ismeretesek egyéb számlamenedzselős programokkal és tudják mit keresnek. A többi alkalmazáshoz hasonlóan mutatja a befizetéseket, a számla jelenlegi állását, pénztárca jelenlegi helyzetét.

Habár a felület talán kicsit barátságtalanabbnak tűnhet a többi piacon lévő alkalmazáshoz képest, több funkcióit, megjelenítési módot tartalmaz, mint konkurens társai. Valamint rendelkezik kiforrott befektetés követéssel is.



Ábra: Quicken Premier befektetés kezelés.

Az fenti ábrán láthatjuk, hogy a felhasználó 2012.03.31.-én vásárolt öt darab Apple részvényt, méghozzá darabonként 42,5 dollárért. Majd az oldal jobb szélén láthatjuk, hogy ha ezt az öt darab részvényt eladná 1233,41%-os nyereséget könyvelhetne el.

**Pozitív tulajdonságai**

* Rendelkezik befektetés követő funkcióval.
* Gyakran jönnek ki új verziói.
* Elsősorban asztali alkalmazásnak tervezték, de már elérhető telefonra is.
* Az egyik legrégebb piacon lévő személyes pénzügyi alkalmazás.

**Negatív tulajdonságok**

* Nincs ingyenes változata
* Nem kimondottan kezdők számára fejlesztették ki.

## Összehasonlító táblázat

A négy alkalmazás alapján összeállítottam egy táblázatot, ami szemlélteti az alkalmazások funkcióit, illetve egyéb tulajdonságait a többi alkalmazással szemben.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Alkalmazás neve | Univerzális importálás | Telefonos változat | Nyomtatás funkció | Fizetős funkció | Befektetések kezelése |
| Mint.com | + | + | + | Tanácsok | + |
| YNAB | + | + | - | csak 34 napig ingyenes használata | - |
| Doxo | + | + | - | számla befizetés | - |
| Quicken Premier. | + | + | + | éves díja 80$ | + |

# Személyes pénzügyi importáló tervezése

A szoftver tervezési fázisának elején fontos döntéseket kellet meghoznom. Elsősorban hogy milyen platformra tervezem létrehozni a szoftvert, illetve hogy milyen programozási nyelvet válasszak a fejlesztés közben fellépő problémák hatásos megoldásához. Mivel célom volt a platformfüggetlenség. Valamint egy könnyen változtatható, látványos grafikus felület létrehozása, ezért a C# programnyelvet választottam. Így csupán NET keretrendszerre van szükség a célszámítógépen, ami pedig rendelkezésre áll a felhasználók által leggyakrabban használt operációs rendszerek mindegyikén (Windows, Linux, Mac).

Illetve a C# platform előnyei között szerepel:

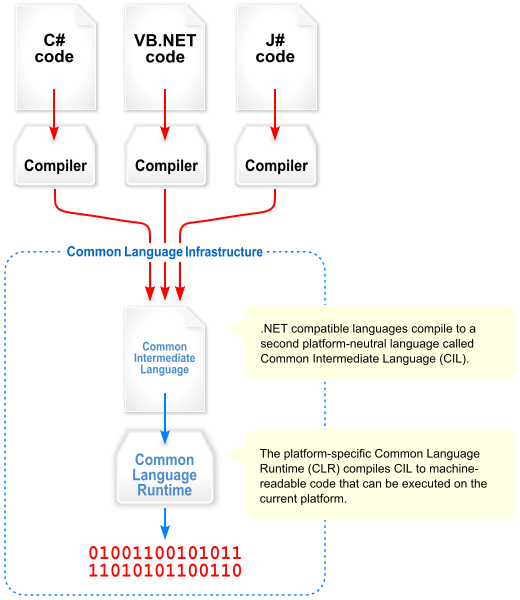
* Objektum orientált programozási nyelv, amely megkönnyíti a programozó dolgát bizonyos problémák könnyű megoldásában, illetve program struktúrájának átlátásában.
* Támogatja a többszálú alkalmazások létrehozását.
* Rengeteg ingyenes könyvtárra rendelkezik.
* Számtalan oktató illetve ismeretterjesztő tartalom található az interneten.

Valamint az sem volt az utolsó szempont hogy egy eddig ismeretlen programozási nyelvet tanuljak és sajátítsak el a szakdolgozatom megírása által.

## Használt technológiák

### .NET keretrendszer

A Microsoft .NET rendszere többféle programozási nyelven is programozható - feltéve, ha az adott programozási nyelvhez van olyan fordítóprogram, amely alkalmazza a Common Language Specification elveit és ajánlásait, valamint képes a .NET keretrendszer virtuális kódjára fordítani. Sok ilyen programozási nyelv létezik. Például a Microsoft által direkt támogatott nyelvek a Visual Basic, J# , C#, Visual C++.



Ábra: A .NET programkódok fordítási folyamata.

(A Common Language Specification olyan szabályok halmaza, melyek leírják azokat az alapvető normákat, amelyre a programozási nyelveknek illeszkedniük kell.)

Mivel a .NET keretrendszerben mindegy, hogy milyen nyelven programozunk, az egyik programozási nyelven megírt eljárást a másik nyelvből is meg lehet hívni és használni. Fontos hogy ez a funkció csak abban az esetben valósítható meg, ha a nyelvek megegyeznek bizonyos alapvető elvekben. Ilyen elvek közé tartozik a tömbök és rekordok, ábrázolása és használata, alapvető típusok (Integer, Double, String, stb.) reprezentációja a memóriában.

A Common Language Runtime (CLR) egy végrehajtási környezet. Ami egy rétegként működik az operációs rendszer és a .NET nyelveken írt alkalmazások között, (azon nyelvek, melyek megfelelnek a Common Language Specification (CLS) szabványnak). A Common Language Runtime (CLR) fő funkciója a kezelt kód natív kóddá való átalakítása, majd a program végrehajtása. A program végrehajtása során a CLR a memóriát, szálkezelés, memória felszabadítást (Garbage Collection), kivételkezelést, jogosultsági rendszert kezelést és egyéb rendszerszolgáltatásokat kezel.

### C# programozási nyelv

A C# az a programozási nyelv, ami a legközvetlenebb módon tükrözi az alatta működő, minden .NET keretrendszeren futtatható programot. A nyelv adattípusai az objektumok. Több korlátozást és továbbfejlesztést is tartalmaz a C és C++ nyelvekhez képest. Például:

* A legtöbb objektum-hozzáférés csak biztonságos hivatkozásokon keresztül tehető meg.
* Rendelkezik Garbage Collection funkcióval, ami automatikusan felszabadítja azon memória területeket, amire már nincs hivatkozás.
* A C++-tól eltérően C# csak egyszeres öröklődést enged meg. Viszont egy osztály több interfészt is megvalósíthat
* A C# sokkal típus biztosabb mint a C++. Az egyetlen implicit konverzió a biztonságos konverzió. Úgy, mint az egészek tágabb intervallumba konvertálása vagy a leszármazott osztályok alaposztályba konvertálása. Nem lehetséges az implicit konverzió a logikai (boolean) és az egész típusok között, vagy a felsorolás tagok és az egészek között.

Mivel a C# kétféle grafikus felhasználói felület létrehozását is támogatja:

* WPF (Windows Presentation Foundation)
* WinForm (Windows From Designer)

Meg kellet vizsgálnom, hogy melyik felületet választásával tudnám elérni a legjobb grafikus felületet létrehozását.

### WPF (Windows Presentation Foundation) vagy WinForm (Windows From Designer)

A fő különbség a két felület között hogy a WinForm csak egy felületet biztosít az alapvető képernyőelemek fölé (például egy Szövegdoboz). Viszont a WPF et használva felépíthetjük akár a semmiből saját készítésű képernyőeleminket. Egy jó példa erre a különbségre egy Gomb elem készítése, amiben van egy kép és egy Szövegdoboz is, ez nem egy beépített képernyőelem, így a WinForm nem ajánlja fel nekünk ezt a lehetőséget. Tehát nekünk kell implementálnunk az ezt megvalósító metódust, ami a következőképpen lehetséges

* Létre kell hoznunk egy olyan gombot, ami támogatja a képek megjelenítését.
* Vagy importálunk kell egy harmadik féltől származó kiegészítést.

Ellenben a WPF felületen egy Gomb komponens tartalmazhat bármit, mivel ez az elem végülis egy keret, ami bizonyos állapotokkal rendelkezhet (például érintetlen, kattintott, letiltott stb.).

Ez a tulajdonság nem csak a Gombról mondható el, hanem az összes WPF-ben létrehozható elemről is. Ha az akarjuk megoldani WPF felületen, hogy egy Gomb tartalmazzon egy képet és egy Szövegdobozt nincs más dolgunk, mint a Gomb tartalmába beilleszteni ezeket az elemeket. Viszont ennek a kötetlenségnek és a semmiből való felépítésnek az a hátránya, hogy bizonyos esetekben több időt kell fordítanunk olyan elemek létrehozására, amik a WinForm korlátozott felületén alapvető képernyőelemként elérhetőek.

#### XAML

Az XAML az XML nyelv egy változata, amit a Microsoft fejlesztett ki a grafikus felhasználó felület leírására WPF megjelenítési felület esetén. A WinForm grafikus elemei és azok tulajdonságai, változásai ugyan azon a nyelven íródtak, mint maga a program. Viszont XAML esetén külön van választva a megjelenítésért felelő kód és a programkód, ami segíti elkülöníteni a programozók munkáját.

#### Databinding

Röviden, a Databinding egy olyan technológia, ami a képernyőn megjelenített elem tulajdonságát összeköti egy programkódban szereplő objektum értékével.



Ábra: Databinding fajtái WPF esetén.

A cél adat jelenti a grafikus felületlen megjelenített adatot, míg a forrás a programkódban lévő objektumot. A cél és a forrás adat összekötése databinding használatával van megvalósítva.

* Egyszeri: (Az ábrán nincs illusztrálva) E kötés esetén, ha a cél adat megváltozik, arról a forrás nem fog tudomást szerezni.
* Egyszeri (Az ábrán OneWayToSource): Az Egyszeri kötés fordítottja, tehát akkor frissíti egyszer a forrás értéket, mikor a cél érték megváltozik.
* Egy irányú: (Az ábrán OneWay) Az egy irányú összerendelés a cél érték automatikus megváltoztatását eredményezi, abban az esetben, ha megváltoztatjuk a forrás adatot, de a célobjektum tulajdonságainak módosítása nem hat vissza a forrás értékre.
* Kétirányú: (Az ábrán TwoWay) a Kétirányú összerendelés megváltoztatja a forrásadatot vagy a céladatot a másik frissítése esetén. Tehát ha a kódban változtatunk egy tulajdonságot, az kihatással lesz a felhasználói felületen megjelenő elem tulajdonságára is, valamint ugyan ez fordítva is igaz.

Összehasonlítás

WPF előnyei:

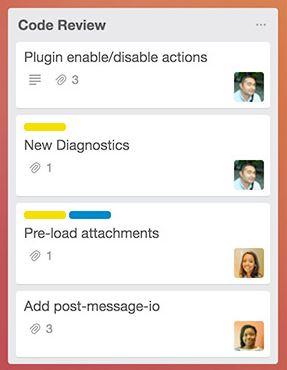
* Újabb, tehát jobban szinkronban van az új technológiákkal.
* Rugalmas, több dolgot csinálhatunk anélkül, hogy egyéb, harmadik féltől származó kiegészítőket implementálnunk kellene.
* XAML segítségével könnyen el tudjuk különíteni a megjelenítését és a programozás részét a programunknak.
* Rendelkezik Dabinding-el, ami segítségével össze tudjuk kötni a programkódunkban lévő objektumokat a grafikus felületen megjelenített elemek tulajdonságaival.

WinForm előnyei:

* Régebbi technológia, ezáltal többet tesztelték és próbálták.
* Sok harmadik féltől származó kiegészítés létezik, amit ingyen letölthetünk.

### Trello

A Trello egy web alapú projektmenedzsment alkalmazás, aminek segítségével nyomon tudjuk követni hogy állunk a számunkra kitűzött célok elérésében. A Trello alapvetően egy weboldal, amely vízszintesen felsorolt ​​listákat tartalmaz, így egy madártávlatos képet kaphatunk a projektről, amin dolgozunk. A listákon belül kártyák helyezhetőek el, amik a részfeladatok és az ahhoz tartozó információkat tartalmazzák. Ezen kártyák mozgathatok, szerkeszthetőek.



Ábra: Trello folyamatok állapotának áttekintése.

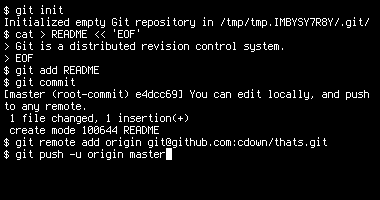
Az egyes kártyák tartalmazhatnak ellenőrző listákat, képeket, csatolmányokat, határidő dátumokat, színes címkéket és vitafelvételeket másoktól, akik osztoznak a projecten. Annyi sajátkészítésű táblát csinálhatunk, amennyit csak szeretnénk. Az egyik például "Otthoni teendők" a másik "Szakdolgozat" és így tovább.

Ha több fős a project csapattal rendelkezünk, akkor különböző emberekhez rendelhetünk bizonyos részfeladatokat, erről ők is értesítést kapnak email-en

### Verziókövetés - Git

A Git röviden egy nyílt forráskódú verziókezelő rendszer, számítógépes fájlok módosításainak nyomon követéséhez. Valamint segít koordinálni a munkát e fájlokon több ember között. Elsősorban a forráskód kezeléshez használják szoftverfejlesztésben, de felhasználható fájlok bármely változatának nyomon követésére is.

A konkurens verzióvezérlő rendszerhez hasonlóan a Git esetében is minden számítógépen megtalálható a Git könyvtár teljes körű archívuma, a teljes változtatási előzményekkel és verziókövetési lehetőségekkel, függetlenül a hálózati hozzáféréstől.



Ábra: verziókövetés Git használatával

### Pencil Project

A Pencil nevű mockup lehetővé teszi, hogy gyorsan és egyszerűen hozzunk létre szórakoztató és interaktív grafikus felületet, maketteket. Az alkalmazás rengeteg beépített elemmel rendelkezik és nagyon könnyű kezelni. Valamint egy egyszerű behúzós felülettel, segítségével tudunk új elemeket hozzáadni a készülő felülethez.

A Pencil Project természetesen engedi, hogy saját képeinket is beimportáljunk. Letölthető asztali felületre, de használható böngészőn keresztül is. Habár a telepített verziósokkal több képernyő elemmel rendelkezik.

### Live-Charts

Mivel a WPF grafikus felülete nem rendelkezik alapértelmezett grafikon elemmel. Viszont egy ilyen grafikus elem sajátkezű implementálása túl sok időt emésztett volna fel, ezért a legcélszerűbb opciónak azt találtam, hogy egy harmadik féltől származó könyvtárt fogok használni a grafikonok megjelenítésére. A választásom a Live-Charts könyvtárára esett.

Azért erre a könyvtárra esett a választásom mivel nagyon szépen kidolgozott, könnyen skálázható elemekkel bővíti ki a grafikus elemek listáját. Valamint WPF felületen történő megjelenítést és a DataBindigot is támogat. Illetve remek oktató anyagok találhatóak a projekt publikus GitHub oldalán.

### IEX Trading

A bizonyos időintervallumra visszamenő részvényekhez tartozó adatokat (nyitó, záró, legmagasabb, legalacsonyabb ár) lekéréséhez találnom kellet egy olyan API-t (Application Programming Interface***)*** amely lehetővé tesz ezen adatok egyszerű és gyors lekérését.

Ezen feltételeket figyelembe véve, az IEX alkalmazást választottam. Ami lehetővé teszi a részvényadatok lekérését csv fájl formátumban. Egy lekéréshez két adatot kell szolgáltatnunk az alkalmazás számára, a tőzsdei szimbólumot, például az Apple Inc. esetén a tőzsdei szimbólum AAPL. Valamint egy intervallumot, ami az aktuális naptól számított időtől

* 5,2,1 év
* 6,3,1 hónap
* 1 nap

Valamint nagyszerű dokumentáció és példa kódok szerepel az alkalmazás hivatalos oldalán.

## Követelmények

A következő alapvető követelményeket állítottam fel a programmal szemben a tervezési fázis elején.

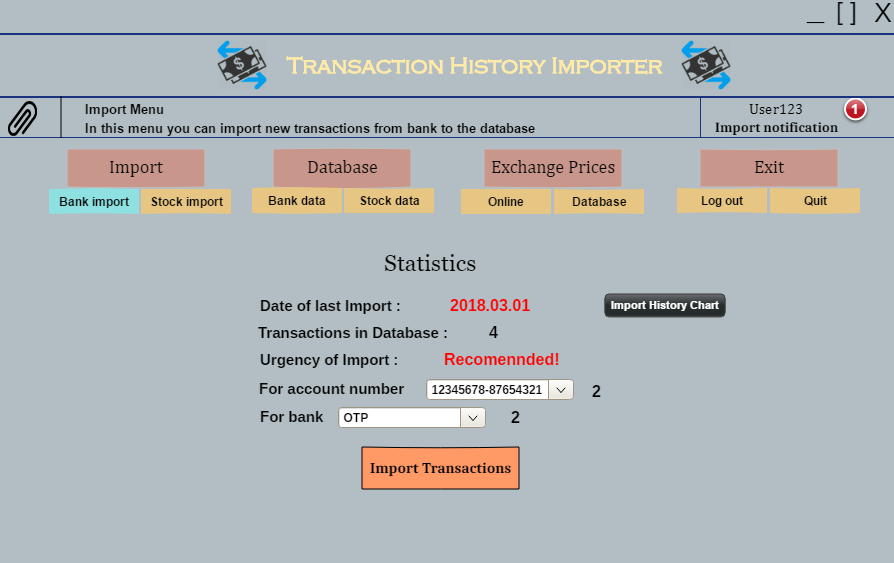
### Alapvető követelmények

* A program támogassa mind az Excel, valamint a CSV kiterjesztésű fájlok beolvasását.
* A felhasználónak legyen lehetősége egyszerre több fájlt importálni. Ezalatt az értendő, hogy az import fájl kiválasztásra során a control billentyű lenyomásával több fájl is kijelölhető egyszerre.
* Legyen lehetőség a grafikus felületre való ,,behúzással" (Drag and Drop) által importálni.
* A felhasználó számára bejelentkezés után jelenjen meg egy statisztikát tartalmazó rész, ahol láthatja az általa beimportált fájlokból kinyert statisztikai adatokat.
* Banki tranzakciók esetén egy felhasználóhoz tartozhasson több számlaszám is.
* Banki tranzakciók esetén egy beolvasott tranzakció csak abban az esetben lesz eltárolva, hogyha legalább az alábbiakban felsorolt információk szerepelnek benne:
  + Tranzakció dátuma
  + Tranzakció ára
  + Tranzakciós fájlhoz tartozó bankszámlaszám
* Ezen felül az alábbi adatok kiolvasására és eltárolására lesz lehetőség:
  + Tranzakcióhoz tartozó leírás
  + Számlaegyenleg
* A tőzsdei tranzakciók esetén egy beolvasott tranzakció csak abban az esetben lesz eltárolva, hogyha az alábbiakban felsorolt információk szerepelnek benne:
  + Tranzakció dátuma
  + Termék neve (Vásárolt/Eladott részvény neve)
  + Tranzakció ára
  + Tranzakció típusa (Eladás vagy Vásárlás)
  + Vásárolt/eladott mennyiség
* A hatékonyság érdekében, minden menüből a program futása során csak egy példány jöjjön létre. Amit Singleton tervezési minta segítségével oldok meg.

### Grafikus felület

Kezdetben a grafikus felület megtervezésére fókuszáltam, mivel az, az alkalmazás egyik legmeghatározóbb része. Az alábbi követelményeket állítottam fel a grafikus felülettel szemben:

* Mivel célomban állt, elkülöníteni, a megjelenítéshez tartozó kódot a tényleges programkódtól, ezért a grafikus felületen elhelyezkedő elemek animálását ahol csak lehet, ott XAML szinten oldom meg.
* A rendszer bizonyos funkciói eléréséhez tartalmazzon gombokat, amikre rákattintva a felhasználó meg tudja jeleníteni a kattintott menü tartalmát. Ezen gombok helyezkedjenek el egy menüben.
* A felhasználó a fontosabb eseményekről az oldal közepére kapjon egy felugró szövegdoboz (Message Box) értesítést.



Ábra: korai képernyőterv a banki tranzakciók importálásához.

### Tranzakciók importálása

Mivel az új tranzakciók importálása volt a fő funkciója a készülő programnak ezért annak tervezésével töltöttem el a legtöbb időt. Az alábbi követelményeket állítottam fel az importálás funkcióval szemben:

* Abban az esetben fog eltárolásra kerülni egy importált banki tranzakció, ha annak adatai még nem szerepelnek az eltárolt adatok között. Kivéve, ha a felhasználó mindenképpen importálni szeretné a tranzakciót.
* .A tőzsdei tranzakciók esetében a felhasználó választhassa ki a nyereség - veszteség kiszámolási módszert.
* A felhasználó kapjon egy felugró szöveges értesítőt mind banki, mind tőzsdei fájl importálásának végén, hogy melyik fájlból hány darab tranzakciót sikerült felvinnie, illetve ha volt egyezés az eltárolt adatok és az importált adatok között azt is tudassuk vele.
* Mind tőzsdei, illetve baki tranzakciók esetén a rendszer adatbázisában ugyan abban a sorrendben kerüljenek eltárolásra a tranzakciók, ahogyan az importált fájlban voltak.
* Az alkalmazás rendelkezzen egy automatizált, illetve egy felhasználó által deklarált importálási lehetőséggel.

#### Automatikus importálás

Ebben az esetben a szoftver olyan algoritmusokat használjon, ahol az importált fájlt megvizsgálva megpróbálja beazonosítani azon oszlopokat illetve cellákat, amelyek egy érvényes tranzakció beolvasásához szükségesek. Ha az algoritmusok által nem sikerül beolvasni az importált fájl adatait, akkor a felhasználó kap egy értesítést, hogy használja a Felhasználó által deklarált importálást. A rendszer sikeres importálás végén elmenti a kiolvasott tranzakciókat az adatbázisba.

#### Felhasználó által deklarált importálás

Ha a felhasználó az ő általa vezérelt importálási folyamatot választja, akkor megjelenik számára egy új oldal. Ahol látni fog szövegdobozokat illetve legördülő listákat, amik használatával tudatnia kell a szoftverrel mely sorokból, oszlopokból olvassa ki a tranzakciókhoz szükséges adatokat. Ha a felhasználó kitöltötte az összes szövegdoboz, egy gombra kattintva jelzést ad a szoftvernek hogy az kezdheti az adatok kiolvasását az általa megadott adatokat használva.

### Tárolt adatok megtekintése

* A tárolt adatok megjelenítésé esetén minden esetben táblázatot fogok használni, ahol az oszlopok a tranzakcióhoz tartozó tulajdonságok.
* Ha a táblázat egy cellájának maximális mérete kisebb, mint a megjelenítendő adat hossza, akkor legyen lehetőség egy szövegbuborék segítségével megjeleníteni a teljes adatot.
* Tőzsdei tranzakciók esetén, ha nyereséges volt az adott részvény eladása, akkor azt jelezzük a táblázatban zöld színnel, ellenkező esetben piros színnel.
* Ha szükséges (a táblázat elég adatot tartalmaz ahhoz, hogy a hossza nagyobb legyen teljes grafikus felület hossza) a táblázat jobb oldalán jelenjen meg egy görgető csúszka.
* Mind banki és tőzsdei tranzakciókat tartalmazó táblázat esetén minden adatnál jelenjen meg az importálás dátuma.

### Részvényadatok lekérése Internetről

* A felhasználónak legyen lehetősége beírni az általa lekért részvény szimbólumát egy szöveges dobozba. Majd megadnia, hogy milyen dátumra visszamenőleg szeretné lekérni az adatokat.
* A lekért részvény adatait jelenítse meg a felhasználó számára egy gráf, ahol az X tengely fogja tartalmazni a dátumokat, az Y tengely pedig az aktuális árakat.
* A gráfon látható metszéspontok tartalmazzák konkrét árat is.
* A lekért adatok kerüljenek elmentésre a rendszer adatbázisába.

### Binding és EvenHandler használata

* Ahol lehet ott Binding segítségével fogom megoldani az események bekövetkezését, adatok lekérését, változtatását. Például a gombokhoz rendelt parancsok, táblázatokhoz tartozó adatok, legördülő lista elemei, aktuálisan kiválasztott elem stb. Ehhez segítségemre lesz az MVVM modell, aminek működési elvét követve külön osztályba fogom tenni a grafikus felülettel kapcsolatban lévő elemeket az üzleti logikát tartalmazó elemektől.

Ha nem lehetséges megoldani a Binding segítségével, akkor EventHandlert fogok használni.

A Bindingot részesítem az EventHandlerrel szemben előnyben. Mivel ha EventHandlert használok egy bizonyos esemény kezelésére, akkor az EventHandler függvénynek mindenképpen abban az osztályban kell lennie ahol az elem létre van hozva (Az elemek az osztály adattagjai képezik), így egy olyan kód keletkezik ahol keveredik az üzleti, és a megjelenítésért felelő kód.

### Importált adatok elmentése

* Az alkalmazás egy belső SQL adatbázisba tárolja el az importált adatokat. Külön adatbázis kell a felhasználók, a tranzakciókhoz tartozó információkhoz, Illetve az internetről lekért részvényadatokhoz is.

Fontos alapkövetelménye az adatok mentésnek hogy a folyamatból semmit ne érzékeljen a felhasználó. Ezen feltétel alatt azt értem, hogy külön szálon fusson az adatbázisba való írás, hogy ne akadályozza a felhasználó tevékenységét eseteleges ,,megfagyásokkal" .

# Személyes pénzügyi importáló megvalósítása

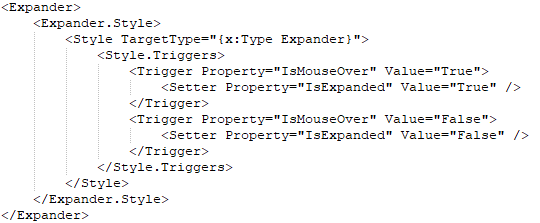
Ebben a fejezetben be fogom mutatni az elkészült Személyes pénzügyi importáló program megvalósítását, illetve a bizonyos részekhez tartozó implementációkat.

## Grafikus felület

Mint ahogy a tervezés fejezetében írtam, kezdetben a grafikus felület megvalósítására fókuszáltam. A grafikus felület animációs megvalósításához csak XAML nyelvet használtam. Fontosnak tartottam a modern megjelenést, így egy olyan előugró menüt dolgoztam ki, ami abban az esetben lesz teljesen látható, ha a felhasználó ráviszi az kurzort. Szerencsére az implementációt nagymértékben megkönnyítette XAML-ben lévő Tag-ek sokasága.

Természetesen gondoltam az almenükre is, az almenük abban az esetben jelennek meg egy főmenü alatt, ha a felhasználó ráviszi az egerét a főmenüre, viszont ha egy új menüpontra húzza át az egeret, akkor automatikusan bezáródik a régi almenü és megnyílik az új.

Az almenü abban az esetben is bezáródik, ha a felhasználó elviszi az egeret az alkalmazás menüjéről.



Ábra: Az almenük megjelenése és eltűnéséért felelős XAML kód

Az Expander egy olyan XAML Tag, amit használva az elemek tartalmazhatnak magukban több elemet is. Alapértelmezett módon kattintás segítségével nyithatjuk le őket. De mivel én nem ezt a megoldást szemléletem ki, ezért egy egyedi stílust (Style) kellet létrehoznom, aminek meg kell adni egy célt, hogy mire hajtsa végre a stílusváltozást (TargetType), aminek megadtam magát az Expandert (hogy az egész elemre kihatással legyen).

Ezek után meg kell adnom, hogy a stílusváltozás egy esemény bekövetkezésekor lépjen életbe (Style.Trigger). Ez az esemény nem más, mint az IsMouseOver állapot, ami akkor lesz igaz, ha az adott elem, amiben az expander van, a kurzor alatt helyezkedik el.

Ha az IsMouserOver igaz, automatikusan kibontjuk az elemet, úgy, hogy az elem IsExpanded tulajdonságát (Property) expliciten igazra (True) állítjuk.

Ha az IsMouseOver hamis, tehát a kurzor alatt már nem az aktuális elem helyezkedik el. Így annak az IsExpanded tulajdonságát expliciten hamisra (False) állítjuk, ezzel bezárjuk az aktuális Expandert.

A következő feladatom volt az előugró menü kódjának leimplementálása.



Ábra: A főmenü előugrásának animációja.

Amint látható a fenti ábrán, a menü (StackPanel) alapértelmezett helyzete -90 pixel távolságra helyezkedik el a bal margótól, tehát a felhasználó számára láthatatlan. Fontos megjegyezni hogy, ha a StackPanelben elhelyezkedik egy elem, és a StackPanelt tulajdonságát változtatom, változni fog vele a benne lévő elem is. Ezt a tulajdonság változást kihasználva, elég a StackPanelhez hozzárendelni egy stílust, ami kihatással lesz a benne lévő elemekre is.

Tehát a StackPanelhez hozzárendelt stílus (Style) változtatásának célpontja (TargetType) maga a StackPanel. Ezen belül elhelyezkedik két eseményfigyelő (EventTrigger):

* MouseEnter, ami akkor lép életbe, ha a jelen esetben a StackPanelre történik egy kurzor belépés. Ilyenkor elkezdődik egy Storyboard. A Storyboard egy olyan Tag, amelyben eseményeknek időkorlátot, késleltetést állíthatunk be. A Storyboard Tagen belül a ThicknessAnimation Tag-et használtam, ami lehetővé teszi elemek eltolását, jelen esetben az eltolás cél tulajdonsága a Margó (Storyboard.TargetProperty) amit -90 pixeltől eltolunk 0 pixelig, tehát a végülis a StackPanel bal oldali margóját eltoljuk jobbra 90 pixellel, illetve ehhez beállítottam egy fél másodperces lefutási időt (Duration
* MouseLeave, ami akkor lép életbe, ha a jelen esetben a StackPanel felületről ,,lehúzzuk" a kurzort. Ilyenkor elkezdődik egy Storyboard, majd ismét ThicknessAnimation-t használtam, ami segítségével a bal margót 0 pixeltől eltolunk -90 pixelig, itt is fél másodperces késleltetéssel.

Mivel a különböző menükhöz különböző oldalak tartoznak, de a program csak egy ablakkal fog rendelkezni, ezért a főoldalban kijelöltem egy részt, ami a tartalmi részét jelenítni majd meg az oldalnak. Ezt a területet bele tettem egy Frame Tag-be amit elneveztem MainFrame-nek.

<Frame x:Name="MainFrame" Margin="150,70,0,0"/>

Így amikor megnyitunk egy új menüpontot, csak meghívjuk a főoldal objektumának MainFrame adattagját és átállítjuk a Content tulajdonságát az új oldal tartalmára. Ami a következő módon lehetséges:

main.MainFrame.Content = importPage.getInstance();

Ahol main a főablak egy objektumát jelöli. Fontos hogy ez csak akkor működik, ha az importPage öröklődik a Page osztályból (System.Windows.Controls névtér).

A Gombokhoz tartozó parancsokat Binding-al oldottam meg. Ami a következőképpen néz ki XAML szempontjából.

<Button Command="{Binding Path=import}"/>

Látható Command tulajdonságához az Import nevű Binding tartozik. A kódba ezt úgy kell lekezelni, hogy írni kell hozzá egy osztályt, ami implementálja az ICommand interfészt.

public ICommand import{

get{

return new CommandHandler("import");

}

}

public class CommandHandler : ICommand{

private string commandName;

public CommandHandler(string value){commandName=value}

public void Execute(){

if(commandName=="import"){

//do something

}

}

public bool CanExecute(object parameter){

return true;

}

}

A fenti kódból láthatjuk, hogy van egy ICommand típusú objektumunk, aminek a neve az XAML-ben lévő gomb command tulajdonságában szereplő nevével. Ha ezt az objektumot lekértjük (rákattintunk a gombra) akkor a következő dolog történik.

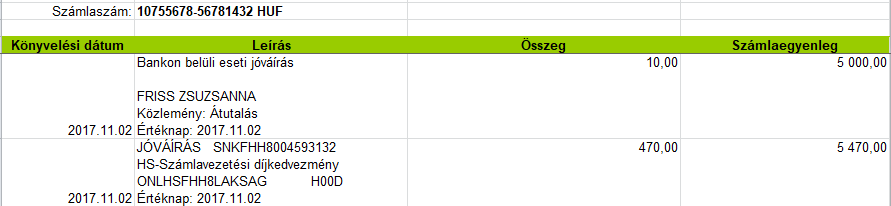
* Létrehozunk egy példányt a CommandHandler osztályból, átadva egy sztringet paraméterként. amit a konstruálás során elment az objektum. Majd automatikusan meghívódik az Execute() függvény ahol az objektum megvizsgálja a commanName adattagjának értékét.

Láthatjuk, hogy a CommandHandler osztálynak még van egy CanExecute függvénye, ami mindig Igazat térít vissza, ezt természetesen köthetjük valamiféle logikához, de a személtetéshez erre nem volt szükség.

## Banki tranzakcióik importálása

A tranzakciók importálását először a banki tranzakciók beolvasásával kezdtem el megvalósítani. Első feladatom volt, hogy minél több számlakivonatot tartalmazó fájlt szerezzek. Ami nem volt egyszerű feladat. Egyrészt azért nem, mivel az emberek bizalmatlanok a pénzügyi dolgaikat érintően (ami érthető). Másrészt mivel célom volt minél többféle banktól származó fájl megszerzése ezért olyan embereket kellet találnom, akik más és más bankoknál rendelkeznek fiókkal. És egy idő után gyakori volt az a probléma hogy nem találtam már olyan embert, aki olyan banknál lenne, ahonnét még nem rendelkezek, tranzakciós fájlal

A Következőekben bemutatok két fájl formátumot, egyik a CIB banktól származik, a másik pedig az FHB banktól (a bankszámlaszámok nem valódiak).



Ábra: CIB banktól letölthető tranzakciós fájl.

A fájlok szerkezete nagyon különbözőnek bizonyult, egyes fájlok nem rendelkeztek egyenleg oszloppal, vagy két oszlopban helyezkedett el a tranzakció összege (Jóváírás,Terhelés). Mindegyik fájltípus más formában jelenítette meg a tranzakció dátumát, több olyan oszlop volt, ami megfelel tranzakció leírásának (Közlemény, Leírás, Információ stb.). Bátran kijelenthetem, hogy az egyik legbarátságosabb formátummal a CIB bank rendelkezik több okból is kifolyólag.

A legpozitívabb tulajdonságnak az bizonyult, hogy összesen négy oszloppal rendelkezik a fájl, plusz egy cella, amiben a számlaszám helyezkedik el. Ahogy a korábbiakban leírtam fájlok nagy részében több oszlop tartalma is megfelelhet tranzakció leírásának, itt nem lép fel ez a probléma mivel csak egy Leírás nevű oszlop van. Valamint egy oszlopban helyezkedik el a tranzakció összege, illetve a fájlban megtalálható számlaegyenleg oszlop is. Továbbá pozitívum hogy a tranzakciós sorok között nincs üres sor.



Ábra: FHB banktól letölthető tranzakciós fájl.

Amit elsőként észre lehet venni a fenti ábrát nézve, hogy jóval több oszloppal rendelkezik mint CIB-es fájl. Több oszlop tartalma is megfelelhet tranzakció leírásnak, például Tranzakció típusa, Közlemény, Leírás.

Látható hogy a tranzakció összege két oszlopra van szétszedve (Jóváírás, Terhelés). Valamint az egyenleg oszlopban csak bizonyos sorokban szerepel érték, ami elég összezavaró. Valamint ami a beolvasást jelenti nagy problémát jelent, hogy rengeteg üres cella szerepel bizonyos sorokban.

### Felhasználó által deklarált importálás

Az importálás implementációját a felhasználó által deklarált importálás létrehozásával kezdtem. Létre kellet, hoznom egy olyan felületet ahova a felhasználó betudja írni az összes lehetséges formátumú fájlok adatait.

Fontosnak tartottam, hogy ha a felhasználónak egy oszlopszámát kell beírnia, hogy azonosítson bizonyos oszlopokat az általa importált fájlban, írhassa be azt egész számként, illetve karakter formájában is (Excelben az oszlopok karakterekkel vannak jelölve). Természetesen, ha a felhasználó karakter formátumban írja be azt kódban át kell alakítani egy számra, amit a következőképpen oldottam meg:

public static int ExcelColumnNameToNumber(string columnName){

columnName = columnName.ToUpper();

int sum = 0;

for (int i = 0; i < columnName.Length; i++){

sum \*= 26;

sum += (columnName[i] - 'A' + 1);

}

return sum;

}

A felhasználó által kitöltendő adatok a következőek:

(Ahol több lehetősége van a felhasználónak egy legördülő listából kell kiválasztania az aktuálisan megfelelő opciót)

* Tranzakciók kezdősora, egyszerű egész szám.
  + Azt a sort jelöli, az importált fájlban ahol a tranzakciók adatainak felsorolása kezdődik.
* Számlaszám (az importáló számlaszáma), itt három lehetősége van a felhasználónak:
  + egy cellában helyezkedik. Például B3 ahol a B az oszlopot azonosítja (2. oszlop), 3 pedig a sort azonosítja.
  + a számlaszám oszlopban helyezkedik el (minden tranzakció mellet szerepel a felhasználó számlaszáma).
  + az Excel munkafüzet neve tartalmazza a számlaszámot.
* Tranzakció dátumát tartalmazó oszlop
  + Egy oszlop számát kell beírnia a felhasználónak.
* Tranzakció összegét tartalmazó oszlop, két lehetősége van a felhasználónak
  + Egy oszlopban tartalmazza a fájl az összeget, egy oszlop szám beírására van lehetőség
  + Két oszlopban tartalmazza a fájl az összeget (Jóváírás, Terhelés) , két oszlop szám beírására van lehetőség
* Számlaszám egyenlegét tartalmazó oszlop, két lehetősége van a felhasználónak
  + Ha szerepel a fájlban ilyen oszlop, akkor az oszlop számát írja be
  + Nem szerepel ilyen oszlop, nem szükséges beírni ezt az adatot
* Tranzakcióhoz tartozó leírás oszlop
  + Itt fontosnak tartottam azt, hogy a felhasználónak több oszlop beírására is legyen lehetősége. Tehát felvihet akár több oszlopszámot is, azokat vesszővel elválasztva (például A,C,E).
* Bank neve
  + Mivel ezt az adat a fájlokban nem szerepel, ezért a felhasználónak saját kézzel kell ezt az adatot beírnia egy szövegdobozba.

Nem lenne túl nagy népszerűsége a felhasználó által deklarált importálásnak, ha minden importálásnál újra be kellene írni azon fájlokhoz tartozó oszlopok számát vagy a tranzakciós adatok kezdősorát, amit már egyszer importáltunk.

Tehát hogy gyorsítsuk az importálás folyamatát, a felhasználó által beírt oszlopszámokat és egyéb adatokat elmentjük az alkalmazás adatbázisába. Majd miután a felhasználó újra a deklarált importálást választja, akkor megvizsgáljuk a korábban eltárolt adatokat és megvizsgáljuk melyik illeszkedik legjobban az aktuális fájlra és automatikusan kitöltjük a felhasználó számára a szövegdobozokat és legördülő listákat.

Habár a felhasználó számára a legjobban egyező adatokkal lesznek feltöltve a szövegdobozok, illetve legördülő listák. Lehetősége lesz betöltenie az összes tárolt adatot is.

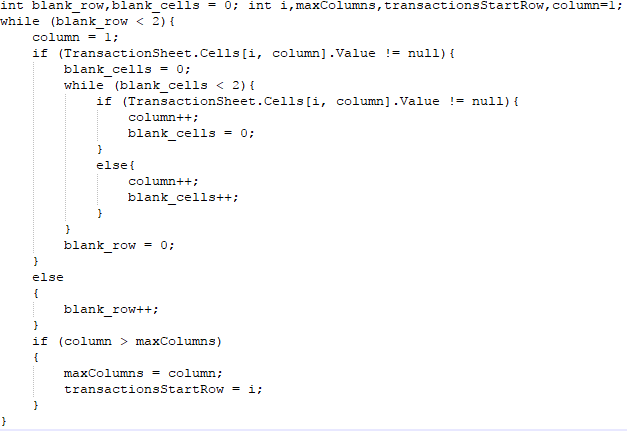
### Automatikus importálás

Mivel minden szoftverben fontos az automatizált működés, ezért fontosnak tartottam egy olyan importálás implementálását ahol az alkalmazás algoritmusokat használva megpróbálja beazonosítani a kellő adatokat tartalmazó oszlopokat illetve cellákat. Majd ezen kiolvasott oszlopokat illetve egyéb adatokat felhasználva, az alkalmazás kiolvassa az importált fájlban lévő tranzakciókat.

Mint ahogy korábban bemutattam a fájlok formátuma, struktúrája nagyon különbözik. Az egyetlen közös dolog az, hogy minden fájl ugyan azon adatokat hivatott megjeleníteni (tranzakciókhoz tartozó ár, dátum, bankszámlaszám, leírás, egyenleg). Tehát arra döntésre jutottam, hogy reguláris kifejezéseket fogok használni a bizonyos oszlopok beazonosításához.

Ehhez elsősorban meg kell adni az algoritmus számára, hogy melyik sorban kezdje az oszlopok vizsgálatát. Tehát meg kell találni azt a sort, ahol a tranzakciók adatainak felsorolása kezdődik. Amit a következőképpen oldottam meg:

Mivel a tranzakciók abban a sorban kezdőnek ahol a fájlban a legtöbb oszloppal rendelkezik egy sor, Így nincs más dolgunk, mint végig menni a fájlon és meg találni ezt a sort, ami egy egyszerű minimum keresés segítségével megoldható.



Ábra: a leghosszabb sor megtalálása

A TransactionSheet objektum jelöli az éppen aktuális Excel munkafüzetet, Cells[i,j] hivatkozása pedig a munkafüzet egy celláját, ahol i a cella sorát azonosítja, j pedig az oszlopát.

Mivel már tudtjuk melyik sort kell vizsgálnunk, elkezdjük a reguláris kifejezések használatát.

kód

Tehát most már megtaláltuk azt a sort ahol a tranzakciók elkedődnek.

bankszámlára regex

összeg regex

dátum regex

leírás, egyenleg.

## Tőzsdei tranzakciók importálása

### Felhasználó által deklarált importálás

### Automatikus importálás

Mint a banki tranzakciók beolvasásánál, itt is fontosnak tartottam egy olyan algoritmus megírását, ami automatikusan ismeri fel a tranzakció adatainak beolvasásához szükséges oszlopokat. Amit a következő módon implementáltam :

A dátum felismeréséhez a következő reguláris kifejezéseket alkalmaztam

^ jelenti hogy a vizsgált elem elejétől illeszkedjen a reguláris kifejezés

\d{2} jelenti hogy várunk két decimális számot (0-9)

* ^\d{4}.\d{2}.\d{2}
  + \d{4}. jelenti hogy négy darab decimális számot várunk utána egy ponttal (ez azonosítja az évet)
* ^d{4}-\d{2}-\d{2}
  + \d{4}- jelenti hogy négy darab decimális számot várunk utána egy kötőjellel (ez azonosítja az évet)
* ^\d{4}.\s\d{2}.\s\d{2}
  + \d{4}\s jelenti hogy négy darab decimális számot várunk white space karaktert például szóköz, tabulátor stb. (ez azonosítja az évet)

Következőnek a cég nevét kell beazonosítanunk, amelytől a vásároltuk/eladtuk a részvényt. Mivel minden tőzsdén lévő cégnek viselnie kell a cég típusát, legyen az Nyrt, Inc, Company, Co, AG stb.

Így nincs más dolgunk, mint hogy írjunk egy reguláris kifejezést az ilyen végződésű sztringek felismerésére.

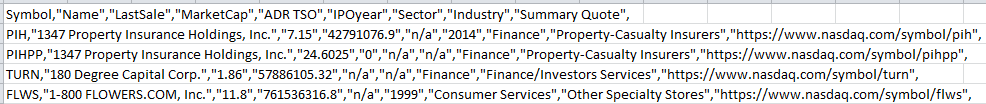
A részvény áránál viszont már komolyabb komplikációk merültek fel, mivel ott nem alkalmazhatunk olyan beazonosítást, hogy megvizsgáljuk egy cella tartalmát, és ha az egy szám, akkor valószínűleg az lesz a részvényhez tartozó ár, mivel semmi nem garantálja, hogy ha egy cella értéke tényleg egy szám, akkor az a szám az valóban a részvény vételi vagy eladási árát takarja. Ezt a problémát megoldva olyan többlépcsős vizsgálatot alkalmaztam, amivel egyértelműen be lehet azonosítani a tranzakciókhoz tartozó ár cellát.

Az első lépés hogy sikeresen beazonosítottuk a dátum és részvény nevéhez tartozó oszlopokat.

Második lépés hogy minden céghez elmentjük a legrégebbi tranzakció dátumát.

Harmadik lépés : Az internetről le tudjuk kérni a tőzsdei árakat bizonyos időre visszamenőleg, ha rendelkezünk a kellő a tőzsdei szimbólummal.

Mivel a tőzsdei tranzakciókat tartalmazó fájlokban mindig a cég teljes neve szerepel. Ezért meg kellett találnom a bizonyos cégekhez tartozó tőzsdei szimbólumot ( például Apple Inc. tőzsdei szimbóluma AAPL) Amit úgy oldottam meg, hogy a NASDAQ-nak (ami egy részvénypiac az Amerikai Egyesült Államokban) létezik egy letölthető csv kiterjesztésű fájlja, ami tartalmazza az összes NASDAQ és NYSE (New York Stock Exchange) tőzsdén jelenlévő cég nevét, és a hozzájuk tartozó tőzsdei szimbólumot.



Ábra: NASDAQ oldaláról letölthető cégneveket, szimbólumokat tartalmazó fájl

Tehát nincs más dolgom, mint a programon keresztül letölteni, és beolvasni ezt a csv fájlt. Amit a következőképpen oldottam meg:

string companyNames="";

using (var web = new WebClient()){

var url = $"http://www.nasdaq.com/screening/companies-by-industry.aspx?render=download";

companyNames = web.DownloadString(url);

}

A fenti ábrából láthatjuk, hogy az első ,,sor" ( csv-ben nincsenek sorok, csak vesszővel elválasztott értékek, de az Excel képes sorokra felosztani ) tartalmazza, hogy melyik ,,oszlopban" milyen adat szerepel. Amik közül nekem Symbol és a Name oszlop értékeire van szükségem. Az ábráról látható hogy csv kiterjesztéshez hűen itt is vesszővel vannak elválasztva egymástól az értékek.

Következő célom az volt hogy felosszam ezt a hosszú sztringet úgy, hogy csak a cégnevek és a hozzájuk tartozó szimbólumokat tartsam meg, majd ezt a két adatot beletegyem egy tömbbe. Tehát kézenfekvő dolognak tűnt a companyNames sztringre meghívni a split metódust. Ami annyit tesz, hogy a függvény argumentumaként megadott karakter alapján felszabdalja a sztringet, majd visszatér az eredményt tartalmazó tömbbel. Így a függvény paraméternek beírtam, hogy vesszők szerint ossza fel a sztringet. Majd rájöttem, hogy ez nem jó megoldás. Mivel a cégek nevein belül is használva vannak vesszők, nem csak az értékek elválasztása esetben.

Majd észrevettem, hogy habár az értékek vesszővel vannak elválasztva egymástól, minden érték idézőjelek között is szerepel, tehát reguláris kifejezés segítségével kell felosztani a sztringet. Ami nem lehetséges a split funkció használatával. Ezért a Regex osztály függvényeit kellet alkalmaznom a feldaraboláshoz.

Regex reg = new Regex("\"([^\"]\*?)\"");

string [] matches = reg.Matches(companyNamesCSV).

Cast<Match>().

Select(m => m.Value).

ToArray();

Miután sikeresen feldaraboltam a sztringet kisebb sztringekre, nem volt más dolgom, mint futtatni egy keresést a cégnevekre, ha van egyezés, akkor lekérjük a céghez tatozó tőzsdei szimbólumot (ami a mellette lévő elem a tömbben).

Mivel gyakorinak találtam azt az eshetőséget, hogy a tőzsdei tranzakciókat tartalmazó fájlban illetve a NASDAQ oldaláról lekért adatok között egy vessző vagy pont eltérés szerepel ezért, egy karakter eltérését megengedtem a fájl és a lekért cégnevek között. Amit Levenshtein távolság függvény implementálásával oldottam meg:

public static int levenshteinDistance(string s, string t){

int n = s.Length;

int m = t.Length;

int[,] d = new int[n + 1, m + 1];

if (n == 0){

return m;

}

if (m == 0){

return n;

}

for (int i = 0; i <= n; d[i, 0] = i++);

for (int j = 0; j <= m; d[0, j] = j++);

for (int i = 1; i <= n; i++){

for (int j = 1; j <= m; j++){

int cost = (t[j - 1] == s[i - 1]) ? 0 : 1;

d[i, j] = Math.Min(

Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1),

d[i - 1, j - 1] + cost);

}

}

return d[n, m];

}

Amely vár két sztringet paraméterként, majd visszatér a paramétereként átadott két szó közötti eltérések számával.

Negyedik lépés, a negyedik lépés elkezdéséhez teljesülnie kell a harmadik lépésnek. Minden céghez tartozó legrégebbi dátumot megvizsgálunk, hogy az alábbi kategóriák közül melyik osztályba tartozik

* legfeljebb egy hónapja történt
* legfeljebb három hónapja történt
* legfeljebb hat hónapja történt
* legfeljebb egy éve történt
* legfeljebb három éve történt
* legfeljebb öt éve történt

Erre azért van szükségünk, mivel az IEX API számára ebbe a hat időintervallumba eső dátumok lekérése van lehetőségünk és nem akarunk több adatot lekérni a kelleténél a gyorsabb feldolgozás érdekében.

Mivel már rendelkezünk a cég nevéhez tartozó tőzsdei szimbólummal és a besoroltuk a dátumot a fent látható idő intervallumokba. A következő dolgom volt IEX API-t használva lekérni a bizonyos dátumhoz tartozó legmagasabb illetve legalacsonyabb részvényárat. A következő parancs segítségével.

string result="";

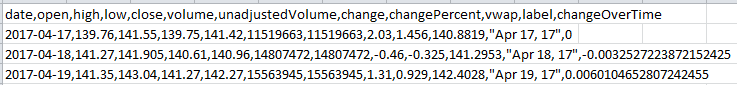
using (var web = new WebClient()){

string url = $"https://api.iextrading.com/1.0/stock/SYMBOL/chart/1Y?format=csv";

result = web.DownloadString(url);

}

Ahol a jelenleg keresett tőzsdei szimbólum SYMBOL. Valamint egy évre (1Y?) Visszamenően kérjük le az adatokat az aktuális naptól számítva. A fenti kód lefutása után a result nevű sztring tartalmazza az egy évre visszamenőleg minden aktív tőzsdei naphoz tartozó legmagasabb, legalacsonyabb, záró, nyitó árat stb. az adott szimbólumhoz.



Ábra: IEX API-tól lekérhető, tőzsdei információkat tartalmazó csv formátuma

A látható adatok közül szükségünk lesz a dátumra, a ,,High" ami az adott naphoz tartozó legmagasabb árat jelenti és a ,,Low" értékekre ami az adott naphoz tartozó legalacsonyabb árat jelenti.

Mivel a result nevű sztringbe az ábrán látható csv fájl adatait írattam bele. Mivel ez egy hosszú sztring jelenleg, ezt fel kell szeletelnünk kisebb szavakká. Mivel a csv-ben vesszővel vannak elválasztva az értékek, ezért a sztringerre meghívtam a split metódust, ami annyit tesz, hogy a függvény argumentumaként megadott karakter alapján felszabdalja a sztringet, majd visszatér az eredményt tartalmazó tömbbel. Tehát a függvény argumentuma jelenleg egy vessző karakter lesz.

double high,low=0;

string[] words = csvString.Split(',');

string regex = "\d{4}-\d{2}-\d{2}";

for (int i = 11; i < words.Length - 1; i ++){

if ((Regex.IsMatch(words[i], regex))){

string[] date = words[i].Split('\n');

if (date[1] == transactionDate){

high = double.Parse(words[i + 1].Replace('.', ','));

low = double.Parse(words[i + 4].Replace('.', ','));

break;

}

}

}

Az először is írunk egy reguláris kifejezést, hogy megtaláljuk a dátumokat, mivel nem áll szándékunkban minden elemet megvizsgálni. Az első 11 karakter nem vizsgáljuk, mivel azok a csv leírásáért felelnek (első ,,sor"). Ha a tömb aktuális elemében szerepel egy dátum, akkor arra az elemre meghívjuk a split függvény, most egy sortörés karaktert átadva paraméterként (\n).( Azért fogalmaztam úgy, hogy csak szerepel, mivel a sorok végén nem szerepel vessző, így minden sor utolsó eleme és a következő sor első eleme egy értékként fog szerepelni a tömb elemeként egy sortöréssel köztük.) A felszeletelés után megkapjuk a dátumot. majd megvizsgáljuk, hogy egyezik-e az általunk keresett dátummal (a részvényhez tartozó legrégebbi dátum az importált fájlból). Ha egyezik, akkor lekérjük a mellette jobbra elhelyezkedő elemet, ez lesz az aznap legmagasabb részvényhez tartozó ára, illetve a tőle néggyel jobbra elhelyezkedő elemét, ami a nap legalacsonyabb ára lesz.

Ötödik lépés, megvizsgáljuk, hogy a tranzakciós fájl aktuális sorában, ahonnét a dátumot, illetve a céghez tartozó tőzsdei szimbólumot kiolvastuk, létezik-e olyan cella, aminek az értéke nagyobb vagy egyenlő, mint a kiolvasott legkisebb érték, és kisebb egyenlő, mint a kiolvasott legnagyobb érték.

Ha létezik ilyen érték, akkor megtaláltuk az árhoz tartozó oszlopot, ha nem jártunk sikerrel, akkor megvizsgáljuk a következő céget ugyan ezzel a módszerrel.

Mennyiség oszlop megtalálása regurális kifejezések segítségével. Ha valamelyik oszlopnévre illeszkedik a Mennyiség vagy a Quantity reguláris kifejezés, megvizsgáljuk, hogy valóban egy nullánál nagyobb egész érték helyezkedik el a megtalált oszlop celláiban.

Tranzakció típusának megtalálása, reguláris kifejezés segítségével. Vizsgált illeszkedések : Buy,Sold,Vásárolt,Eladott,Bought,Sold.

Ha minden oszlopot megtaláltunk, akkor kezdődhet a tranzakciók tényleges kiolvasása a fájlból. Illetve ha a kellő oszlopok beazonosítása körül akár egy nem sikerül egy értesítést fog kapni a felhasználó, hogy erre a fájlra alkalmazzon Felhasználó által deklarált importot.

### Nyereség-veszteség kiszámítás

A nyereség-veszteség kiszámításának három módszerét tudja kiválasztani a felhasználó az általa importált tőzsdei tranzakciókra. (Még importálás folyamata előtt ki kell majd választania)

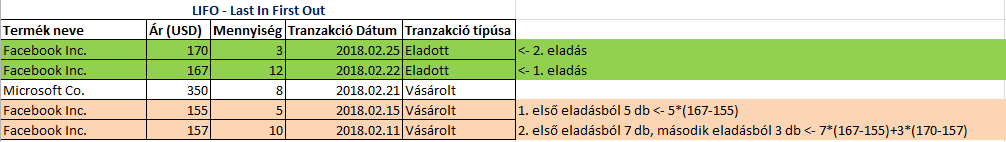
1. **FIFO (First In-First Out)** azaz amit időrendi sorrendben előbb vásároltunk meg részvényt, azt fogjuk először "felhasználni" tehát eladni. Az alábbi ábrán, egy egyszerű példával szemléltetem a FIFO elv működését.



Ábra: FIFO elv működése

* Mivel FIFO elvet alkalmazunk, ezért a vásárolt tranzakciókat alulról-felfelé dolgozzuk fel. A fenti ábrából láthatjuk, hogy 2018.01.22.-én vásároltunk tíz darab Apple Inc. részvényt, darabonként 170 dollár áron, hívjuk el ezt a tranzakciót Első tranzakciónak. Majd 2018.01.25.-én vásároltunk öt darab Apple Inc. részvényt, darabonként 165 dollár áron hívjuk el ezt a tranzakciót Második tranzakciónak, illetve 2018.02.02.-án vásároltunk nyolc darab Coca Cola Co. részvény darabonként 350 dollár áron, de mivel ezeket a részvényeket nem adtuk el ezért a későbbiekben nem foglalkozunk vele. Majd láthatjuk, hogy eladtunk tizenkét darab Apple részvényt 2018.02.02.-án méghozzá 180 dollár értékben. Ha FIFO elvet alkalmazunk a nyereség-veszteség kiszámolására, akkor azt a részvényt adtuk el amit legelőször megvettünk. Tehát mivel tizenkettő darab részvényt adtunk el ezért mind az Első tranzakció illetve a Második tranzakcióval számolnunk kell, mivel az Első tranzakcióhoz csak tíz darab részvény tartozik. Mivel FIFO elvet alkalmazunk ezért az Első tranzakcióból adtunk el tíz darab részvényt, aminek a nyeresége a következő: 10\*(180-170) ahol a 10 jelenti az eladott részvények számát, a 180 az eladási árat és 170 a vételi árat. Még maradt két eladott részvényünk az első eladásból, amit a Második tranzakcióból adtunk el, mivel az Első tranzakcióból eladtuk minden részvényt. Ennek az eladásnak a nyeresége a következő: 2\*(180-165) ahol kettő az eladott részvények száma, 180 a részvények eladási ára, 165 a Második tranzakcióból származó vételi ár.
* Tehát a 12 darab eladott Apple Inc. részvény nyeresége összesen 10\*(180-170)+2\*(180-165) ami egyenlő száz-harminccal.
* Az első eladás nyereség-veszteség kiszámolása után van összesen három darab Apple Inc. részvényünk a Második tranzakcióból. De a második eladás során pont három darabot adunk el. Így a második eladáshoz tartozó nyereségét a következő módon számoljuk ki: 3\*(182-165) ahol három az eladott részvények száma, 182 a részvények eladási ára 165 a részvények vételi ára.
* Tehát második eladáshoz tartozó nyereség egyenlő ötveneggyel. Illetve rendelkezünk még nyolc darab Coca Cola Co. részvénnyel.

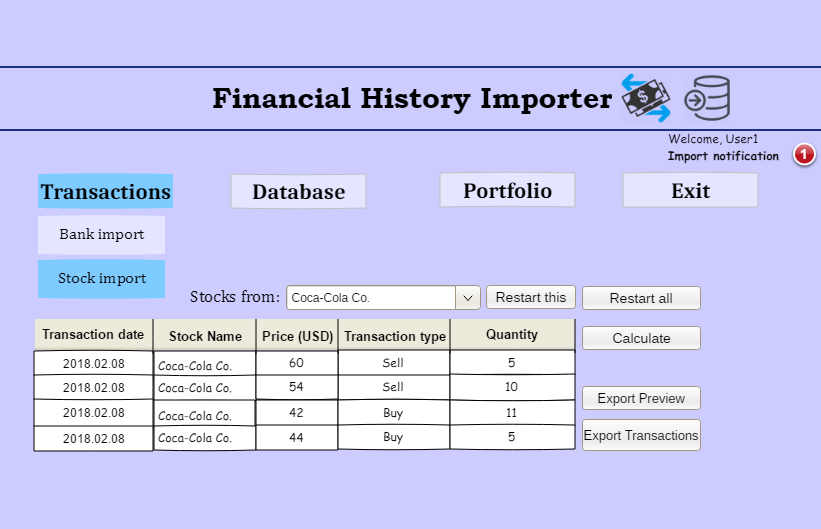
1. **LIFO (Last In-First Out)** azaz amit időrendi sorrendben később vásároltunk meg részvényt, azt fogjuk először "felhasználni" tehát eladni. Az alábbi ábrán, egy egyszerű példával szemléltetem a LIFO elv működését is.



Ábra: LIFO elv működése

* Az ezelőtti példához hasonlóan itt is nevezzük el a vásárolt tranzakciókat. De mivel most LIFO elvet alkalmazunk a vásárolt tranzakciókat fordított időrendi sorrendben fogjuk feldolgozni. Az ábráról leolvashatjuk, hogy 2018.02.21.-én vásároltunk nyolc darab Microsoft Co. részvényt háromszáz-ötven dollár értékben, de mivel ezt nem adtuk el, nem foglalkozunk vele a későbbiekben. Láthatjuk 2018.02.15.-én vásároltunk Facebook Inc. részvényt, öt darabot, darabonként száz-ötvenöt dollár áron, nevezzük el Első tranzakciónak. Valamint 2018.02.11.-én is vásároltunk tíz darab Facebook Inc. részvényt, darabonként száz-ötvenhét dollár áron, nevezzük el Második tranzakciónak. Észrevehetjük, hogy 2018.02.22.-én eladtunk tizenkét darab Facebook Inc. részvényt, ennek a nyereség-veszteség értékét a következő módon tudjuk kiszámolni LIFO elv használatával: mivel tizenkét darab részvényt adtunk el, és az Első tranzakció csak öt darab részvényt tartalmaz. Ezért az első öt darab eladott részvényhez számolt nyereség a következő: 5\*(167-155) ahol öt az eladott részvények száma, 167 az eladott részvények ára és 155 a vásárolt részvények ára. Mivel tizenkét darabot adtunk el és eddig csak az első ötre számoltuk ki a nyereséget a Második tranzakcióból kell kiszámolnunk maradék hét eladott részvény nyereségét. Az első eladásból vett hét részvény és a Második tranzakcióból számolt nyereség a következő: 7\*(167-157) ahol hét az eladott részvények száma, 167 az eladott részvények ára, 157 a vásárolt részvények ára.
* Tehát az első eladáshoz tartozó nyerség egyenlő 5\*(167-155)+7\*(167-157) ami egyenlő százharminccal.
* A második eladáshoz tartozó nyereség kiszámolása a következő módon történik: mivel a Második tranzakcióhoz már csak három részvény tartozik, és a második eladás során három részvényt adtunk el így a következő képletet alkalmazhatjuk: 3\*(170-157) ahol három az eladott részvények száma, 170 az eladott részvények ára és 157 a vásárolt részvények ára.
* Tehát a második tartozó nyerség egyenlő harminckilenccel. Illetve rendelkezünk még nyolc darab Microsoft Co. részvénnyel.

1. **CUSTOM** nyereség-veszteség, Ha a felhasználó a CUSTOM nyereség-veszteség kiszámítási módot választja az alábbi táblázat és a gombok segítésével a saját íze-kedve szerint köthet össze eladott és vásárolt tranzakciókat.



Ábra: CUSTOM nyereség-veszteség kiszámítás képernyőterve

* Az oldalon láthatunk majd egy táblázatot, ami tartalmazza a frissen beimportált tranzakciókat. Méghozzá öt különböző adatot megjelenítve minden tranzakció esetében:
  + Tranzakció dátuma: a tranzakció végbemenetelének dátuma.
  + Termék neve: Azonosítja a céget, amely a részvények kibocsájtásáért felel.
  + Ár: a részvény akkori árát jeleníti meg amikor megvásároltunk vagy eladtuk.
  + Tranzakció típusa: a részvény típusa, két értéket vehet fel, eladott vagy vásárolt.
  + Mennyiség: megjeleníti az adott áron vett vagy eladott részvény mennyiségét.
* A táblázat felett elhelyezkedik egy legördülő lista, amely az összes különböző cégekhez tartalmazó részvény nevét tartalmazza, amelyeket a szoftver a felhasználó által beimportált fájlból olvas ki. A táblázat csak attól a cégtől származó részvényt fogja megjeleníteni, amit a felhasználó kiválasztott a legördülő listából. Tehát a legördülő lista kiválasztott elemének változtatása hatással van a táblázatban megjelenített adatokra.
* A legördülő lista mellet van egy gomb, amin az "Újrakezdés erre" feliratot láthatja a felhasználó. Ha rákattint erre a gombra, akkor az adott részvényhez tartozó mennyiségek visszakerülnek az eredeti állapotukba. Viszont a legördülő listából az összes többi nem kiválasztott részvénynévhez tartozó adatok maradnak a jelenlegi állapotukban. Tehát erre a gombra akkor ajánlott rákattintania a felhasználónak, ha újra szeretné kezdeni a nyereség-veszteség kiszámolását az aktuálisan kiválasztott részvény termékre.
* Az "Újrakezdés erre" gomb mellet az "Újrakezdés mindenre" gomb helyezkedik el. Ha erre a gombra kattint rá a felhasználó, akkor az összes részvény termék mennyisége a kezdeti állapotba vált vissza. Tehát erre a gombra akkor ajánlott rákattintania a felhasználónak, ha újra szeretné kezdeni a nyereség-veszteség kiszámolását az összes beolvasott részvényre.
* A részvényeket tartalmazó táblázat mellet láthatunk majd egy gombot, ami a "Kiszámolás" feliratot viseli. Ez a gomb alapértelmezetten kikapcsolt állapotban van, tehát a felhasználó csak bizonyos feltételek teljesítése által tud erre a gombra rákattintani. A gomb engedélyeséhez a felhasználónak ki kell jelölnie a táblázatban egy eladott és egy vásárolt részvényt. Amit az egér kattintása és CONTROL billentyűzet lenyomása segítségével tud megtenni. Ezért arra van szükség mivel így a felhasználó tudja kijelölni melyik eladást melyik vásárlással akarja összekötni. De egy bizonyos részvény nyereség-veszteség kiszámolásához kötelező kijelölni egy vásárolt és egy eladott részvényt. Tehát a gomb addig lesz kikapcsolt állapotban, míg ez a feltétel nem teljesül.
* A "Kiszámolás" gomb alatt láthatunk majd egy "Export előnézet" feliratú gombot, amire rákattintva a felhasználónak megjelenik egy új táblázat a részvényeket tartalmazó táblázat helyén, ami tartalmazni fogja a már kiszámolt részvényeket azon esetleges nyereség illetve veszteségét is egy Profit nevű oszlopban. Ahol a nyereséges eladások profit cellái a felhasználó segítésére zöld hátterű cellával jelennek meg, a veszteséges eladások pedig piros hátterű cellával. Ez a nézet segítség a felhasználó számára a későbbi adatfeldolgozás illetve megjelenítés szempontjából. Továbbá amint a felhasználó rákattint "Export előnézet" gombra, annak felirata megváltozik "Vissza"-ra. Ezzel jelezve a felhasználónak, hogy az eredeti nézetbe akkor juthat vissza, ha újra rákattint a gombra.
* Az "Export előnézet" gomb alatt helyezkedik a "Tranzakciók exportálása" gomb, amire rákattintva a felhasználó véglegesíti a nyereség-veszteség kiszámolást, és a tranzakciók, azon adataival elmentésre kerülnek a rendszer adatbázisába.

## Adatbázisban tárolt adatok megjelenítése

Mivel az alkalmazásba beimportált tranzakciók eddig különböző formátumú illetve struktúrájú fájlokban voltak eltárolva, fontosnak tartottam egy egységes táblázat formájában megjeleníteni a rendszerben eltárolt adatokat. Kihasználva a WPF és binding adta lehetőségeket. Egy oszlop létrehozásáért XAML környezetben az alábbi kód felel.

<DataGridTextColumn Header="Column Name" Width="150" Binding="{Binding columnData}"/>

Ahol Column Name az oszlop neve, és columnData a programkódban lévő adat, ami össze van kötve a cella értékével. Mivel a táblázatban lévő cellák változó hosszúságú adatokat tartalmazhatnak, fontosnak tartottam hogy ha egy cella tartalma hosszabb mint amit az általa maximálisan megjeleníthető adatmennyiség, akkor a felhasználónak legyen lehetősége rávinni kurzort a cellára hogy egy szövegbuborék segítségével kiolvashassa a teljes tartalmat. Az egyszerűség kedvéért a szövegbuborékot XAML segítségével implementáltam (XAML nyelvben a ToolTip tulajdonság felel a szövegbuborék létrehozásáért).

<DataGridTextColumn.CellStyle>

<Style TargetType="DataGridCell">

<Setter Property="ToolTip" Value="{Binding columnData}" />

</Style>

</DataGridTextColumn.CellStyle>

Mivel a táblázat az adatforrását a programkódból fogja kapni, ezért bindingal oldottam meg az adatok összekapcsolását, amiért a következő sor felel XAML szinten.

<DataGrid ItemsSource="{Binding Path=tableAttributes, Mode=OneWay, UpdateSourceTrigger=PropertyChanged }"/>

Ahol az ItemSource a táblázat (DataGrid) adatforrását jelenti, a binding útvonala, tableAttributes, jelenti a programkódban lévő objektumot, aminek adatait a táblázat oszlopaihoz kötjük. A binding módja jelen esetben OneWay, aminek korábban leírtam a működését, habár így a mód deklarálását ki is hagyhatnánk, mivel a OneWay mód az alapértelmezett beállítás.

Az UpdateSrouceTrigger tulajdonság beállítása nagyon fontos, mivel ezzel tudatjuk a grafikus felülettel hogy a programban változott a grafikus elemhez kötött változó értéke. Az értesítés függvényét a programban is implementálnunk kell. Ehhez először is implementálnunk kell az INotifyPropertyChanged interfészt a kötési objektumot tartalmazó osztályban (az interfész felel a grafikus felület értesítésért).

Az interfész implementálása megkövetelő egy PropertyChangedEventHandler típusú objektum létrehozását. Amit a következő módon deklarálunk, illetve használunk fel.

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName){

PropertyChangedEventHandler handler = PropertyChanged;

if (handler != null){

handler(this, new PropertyChangedEventArgs(propertyName));

}

}

Az OnPropertyChanged függvény segítségével tudathatjuk a grafikus felülettel, hogy a bizonyos binding értékei megváltoztak. Ehhez csak annyit kell tennünk hogy, átadjuk a függvény paraméterének az aktuálisan megváltozott objektum nevét. Egy példa az OnPropertyChanged függvény meghívására.

public List<Transaction> \_tableAttributes;

public List<Transaction> tableAttributes{

get{

return \_tableAttributes;

}

set{

if (\_tableAttributes != value){

\_tableAttributes = value;

OnPropertyChanged("tableAttributes");

}

}

}

Tehát ha ezen túl új értéket adunk a programkódban a tableAttributes változónak (set), akkor a grafikus felület tudomására adjuk hogy frissítse az ahhoz tartozó megjelenített értéket (OnPropertyChanged).

Az objektum változtatásának tudatása a grafikus felülettel azért fontos, ahogy az alkalmazás követelményeiben is leírtam, minden menü grafikus felületéből mindösszesen egy példány fog létrejönni a program futása során. Tehát ha új tranzakciókat importálunk, természetesen az új adatokat is hozzá szeretnénk adni a táblázatban megjelenített elemekhez.

### Banki adatok

A banki adatok megjelenítéséhez egy hatoszlopú táblázatot jelenítek meg a felhasználó számára. Az oszlopok az alábbi adatokat jelenítik meg:

* Tranzakció dátuma
* Tranzakció leírása
* Tranzakció összege
* Importált fájlhoz tartozó számlaszám
* Importált fájlhoz tartozó bank neve
* Importálás dátuma

### Tőzsde adatok

A tőzsdei adatok megjelenítéséhez egy hatoszlopú táblázatot jelenítek meg a felhasználó számára. Az oszlopok az alábbi adatokat jelenítik meg:

* Tranzakció dátuma
* Termék neve (Részvény kibocsájtója)
* Tranzakció összege
* Tranzakció típusa (Eladás/Vásárlás)
* Mennyiség
* Profit
* Importálás dátuma

A könnyebb átláthatóságért, ha a profitált a felhasználó az egyes részvények eladásából, akkor az aktuális profit cella háttérszíne zöldre vált, ellenkező esetben piros színt vesz fel.

Mivel nem akartam túl zsúfolni a tranzakciókat tartalmazó táblázatot, ezért az eladott tranzakciók esetén a felhasználó úgy tudja megnézni, hogy milyen nyereség-veszteség módszer által lett kiszámolva a megjelenített profit, hogy ráviszi a kurzort az adott cellára. Majd megjelenik egy szövegbuborék, ami tartalmazza a kiszámítási módszer nevét.

## Részvény árfolyamok

### internetről

### adatbázisból