Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Hankó Péter

2024

Szegedi Tudományegyetem
Informatikai Intézet

Multifunkcionális számológép programozóknak

Szakdolgozat

Készítette:
Hankó Péter
programtervező informatikus
szakos hallgató

Témavezető:
Dr. Iván Szabolcs
egyetemi docens

Szeged
2024

Feladatkiírás

Napjainkban szinte egy olyan számítógép vagy okostelefon sincs már, amire alapból ne lenne telepítve egy számológép alkalmazás. Ezek általában rendelkeznek tudományos üzemmóddal, azonban speciálisabb számítások elvégzéséhez, mint például mátrix műveletek, gyakran egy külön, erre specializálódott kalkulátor alkalmazás használatára van szükség. A hallgató feladata egy olyan számológép alkalmazás fejlesztése, amely a tudományos számításokon túl, olyan gyakori műveleteket is el tud végezni, amikre szüksége lehet egy informatikus hallgatónak, szakembernek, de akár matematikusok számára is hasznosnak bizonyulhat.

Fontos szempont a fejlesztés során, hogy a hallgató minél inkább megismerkedjen a Microsoft által létrehozott .NET fejlesztői platformmal; az általa készített grafikus alkalmazás a WPF vizuális felhasználói felület keretrendszert használja, és az MVVM architekturális mintára épüljön. Továbbá ismerje meg az ehhez kapcsolódó adatkötés, parancs, vezérlőelem értesítő és más egyéb fogalmakat. Végül egységtesztek segítségével tudja bizonyítani a kód helyes működését.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Multifunkcionális számológép programozóknak

- ***A megadott feladat megfogalmazása:***

A feladat egy olyan asztali alkalmazás elkészítése, amely az alapműveletes- és zsebszámológépekhez képest olyan funkciókkal is rendelkezik, amik a matematikában és az informatikában is hasznosak lehetnek. Ilyenek és ezekhez hasonlók a mátrixokkal, gráfokkal való műveletek, fájlból való számadatok beolvasása és ehhez kapcsolódó statisztikai kimutatások.

- ***A megoldási mód:***

A WPF keretrendszer segítségével, az MVVM architekturális mintához igazodva elkészíteni egy asztali alkalmazást, mely rendelkezik egységtesztekkel, különböző NuGet csomagokat használ, és ellátja a specifikáció szerinti funkciókat.

- ***Alkalmazott eszközök, módszerek:***

A fejlesztés Visual Studio 2022-vel történt C# nyelven a WPF felhasználó felület keretrendszer segítségével. Az gráfok reprezentációját és vizualizálását a QuikGraph és GraphShape végzi el, a függvények vizualizálását a LiveCharts.Wpf teszi lehetővé, végül a fejlesztés több pontját a később részletezett NuGet csomagok segítik. A projekt verziókezelését a Git végezte el.

- ***Elért eredmények:***

A számológép képes különféle számításokat elvégezni, melyek csoportjai, fajtái között a felhasználói felületen különböző üzemmódok kiválasztásával lehet váltani. Az alkalmazás lehetővé teszi a gráfok interaktív vizualizálását és különböző gráfalgoritmusok végrehajtását.

- ***Kulcsszavak:***

számológép, matematikai műveletek, WPF, MVVM, Entity Framework, gráfok

Tartalomjegyzék

Feladatkiírás 1
Tartalmi összefoglaló 2
Bevezetés 4
1. Háttér és elméleti alapok 6
1.1. C# nyelv jellemzői 6
1.2. WPF keretrendszer bemutatása 6
1.3. MVVM architekturális minta 7
1.3.1. Alapelvek és előnyök 7
1.4. NuGet csomagok és használatuk jelentősége 8
2. A számológép fejlesztése 9
2.1. Projekttervezés és követelmények 9
2.1.1. Funkcionális követelmények 9
2.1.2. Nem funkcionális követelmények 10
2.2. Fejlesztési környezet és eszközök 11
3. Az MVVM minta alkalmazása a gyakorlatban 13
3.1. A View réteg 13
3.2. A ViewModel réteg 16
3.3. A Model réteg 19
4. Megvalósított funkciók 21
4.1. Navigáció 21
4.2. Komplex kalkulátor 22
4.3. Mátrix kalkulátor 23
4.4. Gráf vizualizáció 26
4.5. Függvényábrázolás 29
5. Összegzés és jövőbeli terjeszkedési lehetőségek 30
5.1. A projekt összefoglalása 30
5.2. Lehetséges továbbfejlesztési irányok 30
5.3. Személyes tapasztalatok és tanulságok 30
Irodalomjegyzék 32
Nyilatkozat 33

Bevezetés

Középiskolás koromtól kezdődően gyakran szembesültem azzal, hogy mind a matematika, mind a szakmai informatika órák egy elengedhetetlen eszköze volt a számológép. Éppen 12. osztályos voltam, amikor az érettségit megelőzően vettem magamnak egy jobbnak számító számológépet, ami a Casio fx-991CE X volt. Lenyűgözőnek tartottam, hogy egy átlagos kalkulátorhoz képest mennyivel könnyebb kezelni, és az sem egy elhanyagolható tény, hogy 668 elérhető funkciója van.

Egyetemi tanulmányaim során szintén sokszor hasznát vettem ennek a számológépnek. Azonban gyakran azt tapasztaltam, hogy például a számítógépen elérhető számológép funkciói kifejezetten korlátozottak voltak, és a különböző feladatokhoz az interneten fellelhető online kalkulátorokra volt szükségem. Ekkor fogalmazódott meg bennem az a gondolat, hogy érdemes lenne egy olyan számológépet készítenem, amely a megszokott tudományos funkciókon túl képes elvégezni olyan számításokat, melyek az informatikus hallgatóknak, szakembereknek, de talán még a matematikusok számára is hasznosak lehetnek. Tehát az igény már megfogalmazódott, azonban találnom kellett valamilyen módszert ahhoz, hogy meg tudjam valósítani ezt az ötletemet.

A C# programozási nyelvvel középiskolában találkoztam először, és azóta is az egyik kedvenc nyelvemnek tekintem, így nem volt nehéz a választás. Már korábban is fejlesztettem kisebb grafikus felhasználói felülettel rendelkező alkalmazásokat, azonban azokhoz a már akkor is elavultnak számító Windows Forms-ot használtam. Éppen ezért jött az az ötlet, hogy a szakdolgozatomat a modernebbnek számító Windows Presentation Foundation (WPF) keretrendszer segítségével készítem el. Az alkalmazás elkészítésére tehát egy kiváló lehetőségként tekintek abból a szempontból is, hogy jobban megismerjem a C# nyelvi sajátosságait, továbbá a WPF keretrendszer használatát. Emellé társul az is, hogy a programozási készségem további fejlesztése végett az alkalmazást a Modell-Nézet-Nézetmodell (MVVM) architekturális mintára alapozzam, ami megfelelően illik a WPF keretrendszerhez. Ennek a mintának több jellemzője is van, de ezek közül talán az egyik legfontosabb alapelve a felhasználói felület elkülönítése az üzleti logikától, továbbá az adatkötések használata is nagy szerepet játszik.

A könnyebb fejlesztés érdekében NuGet csomagokat használok, amelyek további értéket adnak a projektnek azáltal, hogy lehetővé teszik a fejlesztési folyamat gyorsítását és egyszerűsítését. Ezen csomagok közül leginkább az MVVM Community Toolkit-et

érdemes kiemelni, amit a Microsoft adott ki azzal a szándékkal, hogy könnyebb legyen az MVVM mintára építeni a .NET-es alkalmazásokat, továbbá sok, felesleges, úgynevezett boilerplate kódok megszüntetését is szolgálja.

Mindazonáltal a szakdolgozatom nem csak a szoftverfejlesztés technikai aspektusait tárgyalja, hanem betekintést nyújt a tervezési minták, fejlesztési eszközök és tesztelési technikák alkalmazásába is, amelyek elengedhetetlenek a modern szoftverfejlesztési projektek sikeréhez.

1. Háttér és elméleti alapok

1.1. C# nyelv jellemzői

A C# nyelv a Microsoft által kifejlesztett, modern, objektumorientált programozási nyelv [1], amely a .NET platform részét képezi. Jellemzői között kiemelkedik a típusbiztonság, amely hozzájárul a futásidejű hibák csökkentéséhez, valamint a memóriakezelés, ami automatikus szemétgyűjtést biztosít, csökkentve ezzel a memóriaszivárgások és egyéb problémák esélyét. A C# platformfüggetlen, így az ezen a nyelven megírt kódok futtathatóak különböző operációs rendszereken, és sokoldalúságát tekintve alkalmas asztali, webes, mobilos, valamint felhőalapú alkalmazások fejlesztésére.

1.2. WPF keretrendszer bemutatása

A Windows Presentation Foundation (WPF) a Microsoft által kifejlesztett grafikus felhasználói felület keretrendszer, amely része a .NET platformnak. A WPF-et kifejezetten arra tervezték, hogy gazdag vizuális felhasználói felületeket hozzanak létre asztali alkalmazásokhoz [2]. A WPF egyik legfontosabb jellemzője a XAML (eXtensible Application Markup Language), egy XML-alapú nyelv, amely lehetővé teszi a felhasználói felület elemeinek deklaratív leírását. A XAML használatával könnyedén kialakíthatók komplex felületek, mint például animált menük és interaktív vezérlők, anélkül, hogy közvetlenül kódot kellene írni.

A WPF másik kiemelkedő jellemzője a hatalmas testreszabhatóság és a grafikus képességek. Támogatja a vektor alapú grafikát, ami a gyakorlatban azt jelenti, hogy a felületi elemek tisztán és élesen jelennek meg minden felbontásban. Emellett a WPF képes 2D és 3D grafikák kezelésére is, lehetővé téve ezzel a figyelemfelkeltő vizuális effektek létrehozását. Az animációk és átmenetek könnyen megvalósíthatók, ami dinamikusabbá és interaktívabbá teszi a felhasználói élményt.

A WPF egyik kulcsfontosságú eleme az adatkötés (angolul: data binding). Az adatkötés lehetővé teszi a felületi elemek, mint például a szövegmezők, a listák, és a mögöttes adatmodell közötti automatikus szinkronizációt. Ez azt jelenti, hogy az adatmodell változásai azonnal megjelennek a felhasználói felületen, és ugyanígy fordítva. Ez a funkcionalitás jelentősen csökkenti a szükséges kód mennyiségét, mivel nem kell manuálisan frissíteni a felületi elemeket minden egyes adatváltozásnál.

A WPF továbbá támogatja a stílusokat és sablonokat, amelyekkel egységes megjelenést lehet adni az alkalmazásoknak. A stílusok segítségével egyszerűen lehet definiálni és alkalmazni formázási szabályokat különböző felületi elemekre, míg a sablonokkal teljesen át lehet formálni a vezérlők megjelenését és viselkedését.

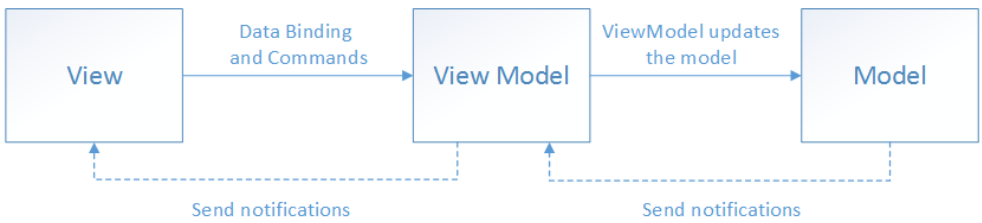
Végül, a WPF egyik legnagyobb előnye, hogy teljes mértékben integrálódik a .NET platformmal, így hozzá lehet férni a platform teljes körű funkcionalitásához, beleértve az adatbázis-kezelést, hálózati kommunikációt és sok más szolgáltatást. Ez lehetővé teszi, hogy kifinomult és teljes körű asztali alkalmazásokat hozzanak létre a programozók, amelyek nem csak látványosak, hanem funkcionálisan is gazdagok.

1.3. MVVM architekturális minta

Az MVVM, azaz a Model-View-ViewModel, egy modern architekturális minta, amelyet kifejezetten a grafikus felhasználói felületek hatékony fejlesztésére terveztek. Az MVVM mintát elsősorban a Microsoft fejlesztette ki, és szorosan kapcsolódik a WPF-hez, bár alkalmazható más technológiákhoz is. Az MVVM lényege, hogy elkülöníti az alkalmazás három fő komponensét: a Modellt (Model), ami az adatokat és az üzleti logikát tartalmazza; a Nézetet (View), ami a felhasználói felületet képezi; és a Nézetmodellt (ViewModel), ami egy összekötő réteg a Modell és a Nézet között. Ez a megközelítés elősegíti a fejlesztési munka hatékonyabbá és rendezettebbé tételét, különösen összetett és dinamikus felhasználói felületek esetén.

1.3.1. Alapelvek és előnyök

Az MVVM alapelvei az adatok és a felület szigorú elkülönítésén alapulnak [3]. A Model közvetlenül nem kommunikál a View-val, hanem a ViewModel-en keresztül. Ez lehetővé teszi, hogy a felületi logika, mint például a felhasználói interakciók kezelése, különváljon az üzleti logikától és az adatmodelltől (1.1. ábra).



1.1. ábra: MVVM minta vizualizálva [3] *1. ábra*

Ez a megközelítés jelentős előnyöket biztosít. Először is, javítja az alkalmazás karbantarthatóságát, mivel a kód strukturáltabbá és átláthatóbbá válik. Másodszor,

nagymértékben megkönnyíti az egységtesztek írását, mivel a ViewModel külön tesztelhető a felhasználói felülettől függetlenül. Harmadszor, az adatkötés (data binding) alkalmazása, ami az MVVM egyik kulcseleme, illetve automatizálja a felhasználói felület és az adatok közötti kommunikációt, így csökkentve a hibák lehetőségét, továbbá növelve a fejlesztés hatékonyságát. Végül, de nem utolsósorban, az MVVM támogatja a rugalmas felület-fejlesztést, lehetővé téve, hogy a felhasználói felületet könnyen módosíthassuk anélkül, hogy a mögöttes logikát vagy adatmodelleket érintenénk, ami különösen hasznos lehet nagyobb csapatokban dolgozva, vagy amikor a felület gyakori frissítése szükséges.

1.4. NuGet csomagok és használatuk jelentősége

A modern szoftverfejlesztésben az egyik legnagyobb kihívás a hatékony kódszerkesztés. Itt jön képbe a NuGet, ami egy, a Microsoft által üzemeltetett csomagkezelő platform a .NET keretrendszerhez. A NuGet lényegében egy központi tárház, amely számos újrafelhasználható kódot és könyvtárat tartalmaz, amelyeket fejlesztők osztanak meg a világ minden tájáról. Ezek a csomagok különböző funkciókat és szolgáltatásokat kínálnak, kezdve az egyszerű segédfüggvényektől egészen a bonyolult keretrendszerekig.

A NuGet használatának jelentősége elsősorban abban rejlik, hogy jelentősen csökkenti a fejlesztési időt és erőforrásigényt. Amikor egy új funkcionalitást szeretnénk beépíteni az alkalmazásunkba, akkor nem szükséges nulláról megírni az összes kódot. Ehelyett érdemes szétnézni a NuGet csomagok között, és nagy valószínűséggel már létezik egy olyan csomag, ami megvalósítja a kívánt funkcionalitást. Ez nemcsak időt takarít meg, hanem növeli a kód megbízhatóságát is, mivel ezek a csomagok gyakran jól teszteltek és karbantartottak.

Ezen kívül a NuGet csomagok segítségével könnyen kezelhetővé válnak az alkalmazás függőségei. A csomagok verziókezelése lehetővé teszi a fejlesztők számára, hogy pontosan meghatározzák, mely verziójú kívánják használni, így elkerülhetőek a kompatibilitási problémák. A NuGet kezeli az összes függőség automatikus letöltését és frissítését, így nem kell manuálisan követni a könyvtárakat és azok frissítéseit.

2. A számológép fejlesztése

2.1. Projekttervezés és követelmények

A fejlesztés megkezdése előtt fontosnak tarottam a specifikáció meghatározását. Ebben részletesen összegyűjtöttem azokat a funkciókat, amiknek az elvégzését elvárom a számológéptől. A könnyebb eligazodás érdekében több kisebb-nagyobb listát írtam össze, melyek a számológép különböző üzemmódjaihoz tartoznak, így ezáltal az MVVM minta Nézeteit és Nézetmodelljeit is meg tudom határozni, és egyben ezeket elkülöníteni egymástól. Az ismert szoftverfejlesztési folyamatok közül az iterációs (inkrementális) modellt választottam, az esetlegesen változó követelmények miatt, de emellett próbáltam megismerni az agilis szoftverfejlesztési módszereket is.

2.1.1. Funkcionális követelmények

A funkcionális követelményeket a számológép különböző üzemmódjainak megfelelően osztottam fel, mindazonáltal majd a Nézetek közötti navigáció is egy alapvető funkció lesz. Az első legfontosabb és legalapvetőbb funkciók az általános számológéphez kapcsolódnak. Ennek feladata, hogy a felhasználó számára biztosítson egy olyan minimális grafikus felületet, amelyen el tudja végezni a négy alapműveletet, tud négyzetre emelni számokat, ugyanígy négyzetgyököt vonni számokból, reciprokot számolni, meg tudja változtatni a számok előjelét, és végül nemcsak egész számokkal, hanem tizedestörtekkel is tud számolni.

A következő üzemmód a tudományos számológép. Az általános számológépnél felsorolt funkciók mindegyikét el tudja végezni, és ezeken felül lehetővé teszi további műveletek elvégzését. Ezek közé tartozik a zárójelek használata, abszolútérték kiszámítása, felső- és alsó egészrész meghatározása, trigonometrikus, logaritmikus függvények, n-edik hatvány kiszámolása, fokok és radiánok közötti átváltás, n-faktoriális kiszámítása és az ehhez tartozó kombináció (nCr), variáció (nPr) kiszámítása. A matematikában gyakran használt e és π számok közelítéséhez is rendelkezésre áll két konstans, végül a programozók számára hasznosnak mondható kettes, nyolcas, tízes, tizenhatos számrendszerek közötti váltásokra is lehetőség van.

Az előbb felsorolt funkciókat a legtöbb számítógépre és okostelefonra telepített számológép el tudja végezni, azonban az alkalmazás legelső nem szokványos üzemmódja a mátrix kalkulátor. Ez egy A és egy B mátrix között tudja elvégezni a szokványos összeadást, kivonást, szorzást, továbbá a mátrixok esetén még elvárt művelet az

elemenkénti szorzás és a mátrix skalárral való beszorzása. Ezenkívül fontos feladat még a determináns kiszámolása, a transzponálás, az inverz és sajátérték kiszámítása, a hatványozás, és végül az LU felbontás is egy hasznos funkció.

A következő, a mátrix kalkulátorhoz képest egyszerűbbnek számító üzemmód a komplex számokkal elvégezhető műveletek. Ez esetben az összeadás, kivonás, szorzás, osztás alapműveletek állnak rendelkezésre, továbbá a komplex számok esetén a konjugálás és az abszolútérték meghatározása is egy fontos művelet.

A látványosabb funkció-csoportok közé tartozik a függvények ábrázolását lehetővé tevő üzemmód. Ez lehetőséget biztosít a deriválásra, integrálásra, interpolációra, regresszióra, és végül ha van egy vagy több zérushely, akkor ezeket is megpróbálja meghatározni az alkalmazás.

A számítástudomány egy érdekes területe a gráfelmélet, és az ahhoz kapcsolódó algoritmusok és problémák. Az ehhez tartozó üzemmód interaktív funkciókkal rendelkezik, mint például a gráf egyéni bővítése, csúcsok mozgatása, és több népszerűbb algoritmus, mint a szélességi keresés, mélységi keresés, Hamilton-út keresés, a megfelelő gráfok esetén pedig minimális feszítőfa meghatározása és Dijkstra algoritmus a legrövidebb út megtalálásához.

Legvégül számológépekhez képest nem szokványos módon adatok kezelésére is lehetősége van a felhasználónak. Txt, csv vagy SQLite-os db kiterjesztésű fájlokat olvashat be, melyen végre tudja hajtani a létrehozás, szerkesztés, megtekintés-listázás, törlés műveleteket, amelyeket a szakzsargon együttesen CRUD műveleteknek nevez. Később a beolvasott adatokról lehet kérni statisztikát, amibe beletartozik a minimum, maximum, átlag, összeg, produktum, szórás, variancia, medián és módusz értékek meghatározása.

2.1.2. Nem funkcionális követelmények

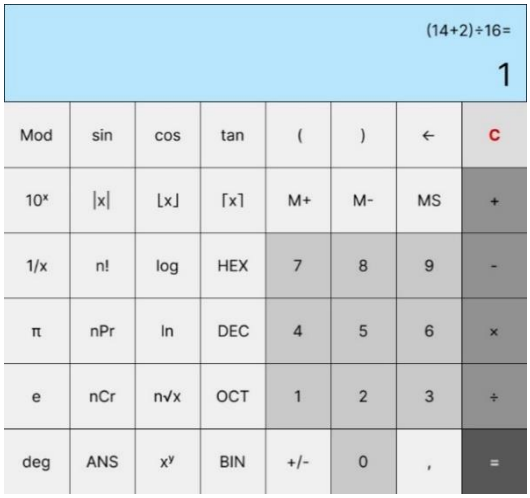
Ezen követelmények közül az egyik legfontosabb a felhasználóbarát interfész. Alapvetően asztali alkalmazást készítek, de mivel a számítógépek esetén is lehet beszélni eltérő kijelzőméretről, ezért érdemes reszponzívvá tenni ezt az alkalmazást is a webalkalmazásokhoz hasonlóan. Mindezen túl a teljesítmény, megbízhatóság és karbantarthatóság is egy fontos szempont számomra az alkalmazás megtervezése, és a későbbi fejlesztése során. Végül úgy tervezem a számológépet, hogy könnyen lehessen bővíteni a jövőben több funkcióval is, hiszen az előző részben felsorolt műveleteken túl számos más műveletet elvégzésére is igény lehet.

2.2. Fejlesztési környezet és eszközök

A fejlesztési folyamat során a Visual Studio 2022 integrált fejlesztői környezetet (IDE) választottam. Ez a környezet kedvezően támogatja a C# programozási nyelvet és a WPF keretrendszert, biztosítva ezzel a szükséges eszközöket és funkciókat az alkalmazásom számára. A Visual Studio 2022 átfogó fejlesztői eszközkészlete lehetővé teszi, hogy hatékonyan írjak, teszteljek kódot, és hogy hibát keressek abban, valamint egyszerűen tudjam kezelni a projekt függőségeit.

A kódverzió-kezelés szempontjából a Git-et és a GitHub-ot használom. A Git biztosítja a hatékony és biztonságos verziókezelést, ami elengedhetetlen a projekt fejlesztési folyamatának menedzseléséhez. A GitHub pedig egy különálló platform a kód tárolására és megosztására, lehetővé téve a munka nyomon követését. Mivel a Visual Studio 2022 rendelkezik beépített verziókezelési lehetőségekkel, ezért könnyen együtt tudnak működni ezek az eszközök.

A felhasználói felület (UI) tervezéséhez a Figma-t választottam, ami egy modern és intuitív eszköz a grafikai tervezéshez. Mivel korábban még nem használtam ezt az eszközt, ezért részletesebben megismerkedtem a felhasználói interfész/felhasználói élmény (UI/UX) fogalmával [4], és annak jelentőségével, továbbá az idézett könyv által a Figma használatát is meg tudtam tanulni. Az alkalmazás segítségével könnyen készíthetek részletes és jól kinéző terveket a számológép felhasználói felületéhez. Ez könnyebbé teszi a fejlesztés folyamatát, hiszen így nem kódolás közben kell a nehezebb, lassabb úton megterveznem a felhasználói felületet, hanem már előre rendelkezésemre állnak a megfelelő tervrajzok. Erre egy példa a tudományos számológéphez készített felülettervem (2.1 ábra).



2.1. ábra: felületterv a tudományos számológéphez

Legvégül egyéb eszközök közé sorolnám a Sourcetree nevű alkalmazást, ami segít vizualizálni a Git branch-eket, és ezzel is el tudom végezni viszonylag könnyen a verziókezeléshez kapcsolódó műveleteket. Továbbá a Visual Studio Code programozói szövegszerkesztőnek is hasznát vettem a megfelelő kódrészletek beszúrásához.

3. Az MVVM minta alkalmazása a gyakorlatban

Létrehoztam Visual Studio 2022-ben a megfelelő projektsablon kiválasztásával a WPF-es alkalmazást, majd pedig legelőször az általános számológépet valósítottam meg a könnyűsége végett. A következő fejezetben ezen az üzemmódon keresztül mutatom be az MVVM minta megvalósítását. Mivel minden egyes üzemmód elkülönül egymástól, ezért az általános számológép is egy külön nézetet kapott. Általában ilyenkor az MVVM-nek megfelelően külön létrehoznak a fejlesztők egy Views, ViewModels és Models mappát, ezért én is ezt a konvenciót követtem. Fontos kiemelni, hogy a fájlokat is hasonló elven kell elnevezni, így mindegyik végére oda kell írni az MVVM megfelelő egységének nevét.

3.1. A View réteg

Egy *UserControl*-t hoztam létre ehhez az üzemmódhoz. Az utóbbi nem egy teljes ablakot (*Window*-ot) kíván reprezentálni, hanem egy olyan vizuális vezérlőelem, ami rendelkezik az ehhez illő tulajdonságokkal és viselkedéssel, továbbá saját maga is egy vagy több vezérlőelemből épül fel. Gyakorlati szempontból ez az üzemmódok közötti navigációnál fog hasznos lenni, de ezt később fogom részletezni.

Új *UserControl* létrehozásakor legenerálódik egy XAML és egy C# kódot tartalmazó fájl, melyek közül az előbbi a nézet leírását foglalja magában, az utóbbi pedig egy mögöttes kódként (code-behind) szolgál. Első lépésként, hogy megfelelően működjön az adatkötés, a XAML fájlban meg kell adni egy adatkontextust (3.1. ábra), ami a később létrehozandó ViewModel szerepet betöltő osztályra hivatkozik XML névtéren keresztül:

```
<UserControl x:Class="AdvancedCalculator.Views.BasicCalculatorView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:viewmodel="clr-namespace:AdvancedCalculator.ViewModels"
             mc:Ignorable="d"
             MinHeight="300" MinWidth="300" MaxHeight="700" MaxWidth="700">
    <UserControl.DataContext>
        <viewmodel:BasicCalculatorViewModel />
    </UserControl.DataContext>
    <Grid></Grid>
</UserControl>
```

3.1. ábra: adatkontextus megadása XAML-ben

Ezt követően a nézethez tartozó mögöttes kód konstruktorában is meg kell adni az adatkontextust a *ViewModel* példányosítása által (3.2. ábra):

```
public partial class BasicCalculatorView : UserControl
{
    public BasicCalculatorView()
    {
        InitializeComponent();
        DataContext = new BasicCalculatorViewModel();
    }
}
```

3.2. ábra: adatkontextus beállítása a code-behind részből

Mindazonáltal most már megfelelően fog működni az adatkötés a Nézet és Nézetmodell között. Ezt azért is érdemes beállítani, mert a Visual Studio 2022 rendelkezik egy úgynevezett IntelliSense-szel, ami segít a kódkiegészítésben, és ha például elgépelés miatt nem működne az adatkötés, akkor figyelmeztetni fog az IDE, továbbá akár fel is sorolja nekünk a Ctrl + Space kombináció lenyomásakor a köthető tulajdonságokat és metódusokat.

Hogy reszponzív legyen az általános számológép felülete, ezért a vezérlőelemeket egy rácsos elrendezésbe helyezem el <*Grid*>-ek között, mivel ennek segítségével tudok definiálni sorokat és oszlopokat, így ezáltal lesz egy táblázatosnak mondható elrendezésem, aminek tudok hivatkozni a soraira, oszlopaira (akár ezeket össze is lehet vonni) a különböző elemek elhelyezésekor.

A számológép kijelzőjének egy <*TextBox*>-ot használtam, aminél az *IsReadOnly* attribútumnak *True* értéket adtam, ezáltal a felhasználó közvetlenül nem tud írni ebbe a speciális beviteli mezőbe, csakis kódból lehet megváltoztatni a tartalmát. Ennél a résznél használtam legelsőnek az adatkötést a következő módon (3.3 ábra):

```
<TextBox
    Text="{Binding Display, Mode=OneWay}"
    Grid.Row="1"
    Grid.Column="1"
    Grid.ColumnSpan="4"
    IsReadOnly="True" />
```

3.3. ábra: számológép kijelzőnek beállítása (*rövidített kódrészlet*)

Jól látható a 3.3. ábrán, hogy a szövegdoboz *Text* attribútumán keresztül tudom beállítani a szövegdoboz tartalmát, és ez esetben nem egy előre begépelt szöveg lesz a tartalma, hanem a *Display* ViewModel-ben megtalálható *string* típusú tulajdonságban eltárolt karakterlánc. A *Binding* mellett be lehet állítani az adatkötés irányát is a *Mode*

segítségével, ami a *OneWay* értéket kapta, ezáltal a nevéből sejthető módon ez egy egyirányú kötést fog jelenteni. Végül még látható, hogy a szövegdoboz a rácsrendszer 1-es indexű sorában és oszlopában lett elhelyezve, továbbá a számológép kijelzője nagyobb méretű lesz a következőnek tárgyalt gombokhoz képest, így 4 oszlopot összevontam a *ColumnSpan* segítségével.

Az adatkötés egy másik fajtája az, amikor nem tulajdonságot, hanem egy *ICommand* implementációt kötünk valamilyen vezérlőelemhez. Ennek egyik legkézenfekvőbb példája az, amikor egy gomb lenyomására szeretnénk valamilyen eseményt kezelni.

```
<Button
    Content="1"
    Command="{Binding NumberPressedCommand}"
    CommandParameter="1"
    Grid.Row="6"
    Grid.Column="1"
    Style="{StaticResource BasicButton}" />
```

3.4. ábra: parancs beállítása a számológép 1-es gombjához

A 3.4. ábrán lehet látni, hogy a gomb *Command* attribútumának egy olyan ICommand implementációt adtam meg, ami a *CommandParameter*-ben megadott 1-es számot fel tudja majd dolgozni a megfelelő módon. Az utóbbi azért is hasznos, mert így az összes többi numerikus gombhoz lehet ugyanazt a parancsot használni, csupán a paraméter értékét kell megváltoztatni. A nem-MVVM megközelítés az lenne, hogy a *Click* attribútumhoz generálunk egy metódust, azonban ez két ponton is sérti az MVVM szabályrendszerét: a View nincs leválasztva a logikától, mikor annak csakis a megjelenésért kellene felelnie; nem valósul meg semmiféle adatkötés a View és ViewModel között, sőt a ViewModel-nek nincs is szerepe ebben az esetben. Az sem elhanyagolható szempont, hogy így minden egyes numerikus gombhoz külön metódust kellene generálni, ami rontaná a kód átláthatóságát és tesztelését, azonban az előbb említett *Command* és *CommandParameter* megadásával mindezeket a problémákat ki lehet küszöbölni.

A stílus beállítása az utolsó részlet amit érdemes megemlíteni a Nézeteknél. Bár meg lehet adni a vezérlőelemek attribútumainál is többféle stílust, azonban ha ezekből túl sok van, továbbá több olyan elem is van, ami ugyanazt a stílust használja, akkor érdemes egy külön fájlba átvinni a stílust, és a kívánt helyen pedig csak elég rá hivatkozni. Erre kínál megoldást a <*ResourceDictionary*>, ami a *UserControl*-hoz hasonlóan ugyanúgy

egy XAML típusú fájlban foglal helyet. Külön nincs ajánlás arra, hogy a külső stílusokat hova kell elhelyezni, azonban a jó elkülöníthetőség végett egy *Themes* nevű mappába helyeztem el, mivel ez nem tekinthető egy Nézetnek, így biztosan nem a *Views* mappába való. A *ResourceDictionary*-n belül több *Style* objektum elem is megadható, de ha nem szeretnénk például minden egyes gombra ugyanazt a stílust beállítani, akkor az *x* XAML-névtér *Key* hivatkozással megadhatunk neki egy kulcsot, amivel később tudunk az adott stílusra hivatkozni.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style x:Key="ClearButton" TargetType="Button">
        <Setter Property="Foreground" Value="Red" />
        <Setter Property="FontSize" Value="18" />
        <Setter Property="FontWeight" Value="Bold" />
    </Style>
    <Style x:Key="EqualsButton" TargetType="Button">
        <Setter Property="Background" Value="#575757" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="FontSize" Value="18" />
        <Setter Property="FontWeight" Value="Bold" />
    </Style>
</ResourceDictionary>
```

3.5. ábra: külső stílus definiálása gombokra (*rövidített kódrészlet*)

A 3.5. ábrán lehet látni, hogy például a törlés (C) gombhoz beállítottam, hogy a tartalma legyen piros színű, 18 pontos betűméretű, félkövér. Az előbb említetteknek megfelelően minden stílusnak adtam nevet is, így a megfelelő nézetben a *ResourceDictionary* importálása után a *StaticResource* megadásával (3.4. ábra legutolsó sora) tudok név szerint hivatkozni a stílusra.

3.2. A ViewModel réteg

A nézet megtervezését követően a *ViewModels* mappában létrehoztam az általános számológéphez a Nézetmodellt, amire többször is hivatkoztam az előző (3.1.) alfejezetben. Ehhez elég létrehozni egy egyszerű C#-os osztályt, azonban a kód megírása előtt letöltöttem a Microsoft MVVM Community Toolkit-jét. Ez lehetőséget biztosít arra, hogy könnyebben tudjam az MVVM mintára építeni a kódomat, és ne kelljen túlságosan sok „boilerplate”-nek nevezett kódot írnom, ami által szintén javul a kód olvashatósága és karbantarthatósága. A csomagot a NuGet csomagkezelő segítségével tudom letölteni, ami a projektet leíró fájlban elhelyez egy hivatkozást erre, így bármelyik fájlból el tudom érni a Toolkit-et. Érdemes kiemelni, hogy nemcsak a könnyebb fejlesztés, hanem jobb

teljesítmény miatt is megéri használni ezt a csomagot, mert maga a Microsoft írta meg, és az így generált kódok számos olyan optimalizációt [5] tartalmaznak, amit egy átlagos fejlesztő nem tudna ennyire hatékonyan megvalósítani. Első lépésként egy *partial* jelentés módosítóval kell ellátni a *ViewModel* osztályát, ami azt teszi lehetővé, hogy ne csak egy, hanem kettő vagy több forrásfájlban helyezkedjen el az osztály. Ennek köszönhetően a generált fájlok forráskódjai is tudnak majd hivatkozni a ViewModel osztályra, szóval az általam írt és a generált kódok elválasztódnak egymástól, mégis logikailag összetartoznak. Következő lépésként az *ObservableObject* osztályból kell öröklődnie az osztályomnak, ami egy olyan előre implementált osztály, aminek a tulajdonságai (property-jei) megfigyelhetők (observable-ek). Ez a gyakorlatban azt jelenti, hogy amikor a tulajdonság értéke megváltozik, akkor értesíti a megfelelő vezérlőelemet erről, és az új érték fog megjelenni az adatkötésnek köszönhetően. Ennek a viselkedésnek az eléréséhez eredetileg az *INotifyPropertyChanged* interfészt kellene implementálni egy alap *ViewModelBase*-ben, amiből aztán minden *ViewModel* öröklődne, azonban az *ObservableObject* ezt megoldja helyettünk.

```
public partial class BasicCalculatorViewModel : ObservableObject
{
    private BasicCalculatorModel _basicCalculator = new();

    [ObservableProperty]
    private double _operand1;

    partial void OnOperand1Changed(double value)
    {
        _basicCalculator.Operand1 = value;
    }
}
```

3.6. ábra: Megfigyelhető osztály és tulajdonság (*rövidített kódrészlet*)

A 3.6. ábra mutatja be az eddig említettek megvalósítását az általános számológép üzemmódhoz. Az *ObservableObject* után szükségünk van olyan tulajdonságokra, amik kihasználják az osztály ezen adottságát, és itt jönnek képbe az *ObservableProperty*-k. C#-ban általában egy egyszerűbb tulajdonság (property) úgy néz ki, hogy adott egy privát láthatóságú adattag, amit csakis egy publikus láthatóságú tulajdonságon keresztül tudunk elérni és módosítani - így lehet megvalósítani a C# nyelven az objektumorientált programozás egységbezárás alapelvét. Ez ebben az esetben sem néz ki máshogy, azonban az *_operand1* privát láthatóságú adattag fölé kell írni szögletes zárójelek közé, hogy *ObservableProperty*, amit ebben a formában attribútumnak nevezünk. Az attribútumok

valamilyen metaadatot vagy deklaratív információt kötnek össze a kóddal. Jelen esetben ez ismét egy generált kódot fog eredményezni, ami a bonyolult optimalizálási lépéseken túl azt a feladatot látja el, hogy a tulajdonságot megfigyelő *View*-ban található vezérlőelemeket értesíti az adatkötésen keresztül minden, a tulajdonsághoz köthető változásról. Olyan lehetősége is van a fejlesztőknek, hogy akár ők maguk kézzel is tudnak reagálni az így bekövetkező változásokra, és erre lehet látni egy példát a 3.6. ábra *OnOperand1Changed* metódusában, ami a modell osztályból készített példány megfelelő tulajdonságának változtatja meg az értékét. Nemcsak az osztályok, hanem a metódusok is rendelkezhetnek *partial* módosítóval, aminek köszönhetően ismét egy generált kódrészletet tudok használni az osztályomon belül.

Utoljára pedig a 3.1. alfejezetben említetteket idézném fel: a gombok eseménykezelését parancsokon keresztül valósítjuk meg, és erre nyújt még egy megoldást a Community Toolkit.

```
[RelayCommand]
public void OnOperationPressed(string operation)
{
    Operation = operation;
}

[RelayCommand]
public void OnClearPressed()
{
    Operand1 = 0;
    Operand2 = 0;
    Operation = "";
    Display = "0";
}
```

3.7. ábra: parancsok generálása a Nézetmodellben

Ahhoz, hogy le tudjuk generáltatni a megfelelő parancsokat, egy *RelayCommand* attribútumot kell a metódusaink fölé írni (3.7. ábra). Fontos részlet, hogy ezeknek nem lehet visszatérési értékük, ezért *void* vagy üres *Task* lehet kizárólag a metódus visszatérési típusa. A 3.1. alfejezetben szó esett a *CommandParameter*-ről, aminek a hasznát az *OnOperationPressed* metódus esetén lehet látni, hiszen a View adatkötésen keresztül nemcsak meg tudja hívni a megadott parancs végrehajtását végző *Execute* metódust, hanem adhat annak paramétert is, ami jelen esetben az *operation* (műveleti jel) lesz. Ez a gyakorlatban azért is hasznos, mert így nem kell külön metódust írnom, majd parancsot generáltatnom minden egyes műveleti jelhez, hanem lehet egy metódusom és parancsom,

ami a paraméterben kapott műveleti jelet fogja kezelni a megfelelő módon. Látható az *OnClearPressed* metódusnál, hogy az módosítja az osztály tulajdonságainak az értékét, de mivel ezek megfigyelhető tulajdonságok, ezért a számológép kijelzőjén is látható lesz a változás, azaz a C gomb megnyomásakor alaphelyzetbe kerül a számológép. Érdemes kiemelni, hogy a generált parancsok minden esetben a *RelayCommand* attribútummal ellátott metódus neve után írják a „*Command*” szót, tehát például nem az *OnClearPressed*-del tudom megadni adatkötésnél a parancsot, hanem az *OnClearPressedCommand*-dal. Ez biztosítja számunkra, hogy a kötésnél a megfelelő *ICommand*-ot implementáló típust adhassuk meg.

3.3. A Model réteg

A modell réteget tartománymodellnek szokták mondani, ami általában tartalmazza az alkalmazáshoz szükséges objektumorientált adatmodelleket, vagy pedig az üzleti- és validációs logikát [3].

Az általános számológép modell osztálya ennek a rétegnek megfelelően egy számológépet hivatott reprezentálni, aminek van állapota: eltárolja a művelet két operandusát és a hozzá tartozó operátort (3.8. ábra).

```
public class BasicCalculatorModel
{
    public double Operand1 { get; set; }
    public double Operand2 { get; set; }
    public string? Operation { get; set; }
}
```

3.8. ábra: a modell osztály és a benne lévő tulajdonságok

Ezen kívül még van két metódusa is ennek az osztálynak, amelyek feladata a bináris (3.9. ábra) vagy unáris operandusú műveletek eredményének kiszámítása:

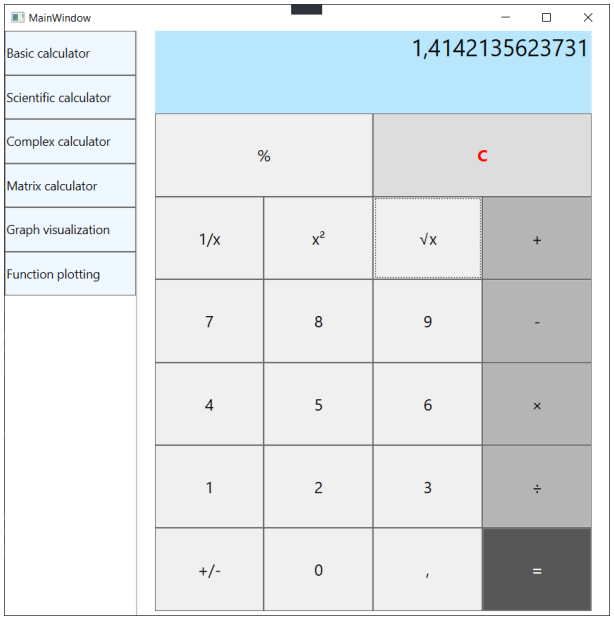
```
public double Calculate()
{
    switch (Operation)
    {
        case "+":
            return Operand1 + Operand2;
        // ...
        default:
            return 0;
    }
}
```

3.9. ábra: bináris operandusú művelet eredményének kiszámítási logikája

A két metódushoz érdemes a *switch-case* vezérlési szerkezetet használni az olvashatóbb kód, és a gyorsabb végrehajtás miatt. Természetesen az osztásnál oda kell figyelni arra az esetre is, amikor a bal oldali (2-es) operandus értéke nulla, azonban egy-egy *case* ágba lehet összetettebb kódot is írni, ezért a felhasználót egy felugró ablak (*MessageBox*) tájékoztatja arról, hogy 0-val nem lehet osztani.

Összegezve ennek a rétegnek nem feladata, hogy feldolgozza az adatkötésből jövő információt, hanem egy általános számológépet reprezentál, ami el tudja végezni a különböző számításokat. A 3.6. ábrán lehet látni, hogy Nézetmodellben van egy példánya a Modell osztálynak, így látható, hogy a Modell közvetlenül nem kommunikál a Nézettel.

Az elkészült általános számológép a 3.10. ábrán látható:



3.10. ábra: általános számológép a $\sqrt{2}$ kiszámolása után

4. Megvalósított funkciók

4.1. Navigáció

Az előző fejezetben tárgyalt általános üzemmód fejlesztése után fontosnak tartottam a navigáció megvalósítását, mivel a projektterv szerint egy oldalsó menüből lehet váltani a különböző funkciók között. Az alkalmazás elindításakor megnyílik egy új ablak, aminek *MainWindow* a neve, és benne lévő tartalmat a <*Window*> elemek között lehet elhelyezni. Mivel nem szándékozok minden egyes funkcióhoz egy teljesen új ablakot megnyitni, ezért a főablakban úgynevezett *UserControl*-okat fogok cserélni. Ehhez egy *NavigationService*-re lesz szükségem. Legelsőnek egy *INavigationService* nevű interfészt hoztam létre, aminek annyi a feladata, hogy leírja, milyen műveleteket tud majd az implementációja elvégezni. A *NavigationService* osztály az előbbi interfész implementációja, ami egy *ObservableProperty* segítségével megállapítható teszi, hogy éppen aktuálisan melyik NézetModell van kiválasztva, továbbá el is tud navigálni egy másik NézetModellre (4.1. ábra).

```
public partial class NavigationService : ObservableObject, INavigationService
{
    private Func<Type, ObservableObject> _viewModelFactory;
    [ObservableProperty]
    private ObservableObject _currentView;

    public NavigationService(Func<Type, ObservableObject> viewModelFactory)
    {
        _viewModelFactory = viewModelFactory;
        _currentView = new BasicCalculatorViewModel();
    }

    public void NavigateTo<TViewModel>() where TViewModel : ObservableObject
    {
        ObservableObject viewModel =  _viewModelFactory.Invoke(typeof(TViewModel));
        CurrentView = viewModel;
    }
}
```

4.1. ábra: NavigationService implementációja

Mindezek után ahhoz, hogy használni tudjam a navigációt, szükségem lesz a Depedency Injection tervezési minta használatára. Ennek esetemben az a lényege, hogy az osztályoknak nem szükséges közvetlenül létrehozniuk a függőségeiket, hanem kívülről kapják meg őket [6]. A Microsoft-nak ehhez van egy saját megoldása, ami NuGet csomagként érhető el a következő néven: Microsoft.Extensions.DependencyInjection.

Ebből nekem az *IServiceCollection*-re lesz szükségem, ami lehetőséget biztosít a szolgáltatások regisztrálására, majd pedig az *IServiceProvider* segítségével tudom azokat létrehozni és biztosítani [7]. Az *App.xaml* fájlban *DataTemplate*-ek között tudom felsorolni a minden egyes funkcióhoz tartozó *ViewModel*-t, a kód mögötti részben pedig az előbb említett Dependency Injection-höz szükséges beállításokat végeztem el, ahol szintén szükséges felsorolni a *ViewModel*-eket és az *IServiceCollection* példányomhoz az *AddSingleton* metódusával tudom hozzáadni őket. Legvégül még egy *MainViewModel* nevű osztályt is létrehoztam, ami tartalmaz egy *INavigationService* típusú adattagot. Érdemes megfigyelni, hogy az osztálynak a konstruktora nem egy konkrét implementációt vár paraméterben, hanem egy olyan típust, ami az előbbi interfészt implementálja (4.2. ábra).

```
public partial class MainViewModel : ObservableObject
{
    [ObservableProperty]
    private INavigationService _navigationService;

    public MainViewModel(INavigationService navigationService)
    {
        _navigationService = navigationService;
        NavigationService.NavigateTo<BasicCalculatorViewModel>();
    }

    [RelayCommand]
    public void NavigateToBasicCalculator()
    {
        NavigationService.NavigateTo<BasicCalculatorViewModel>();
    }
}
```

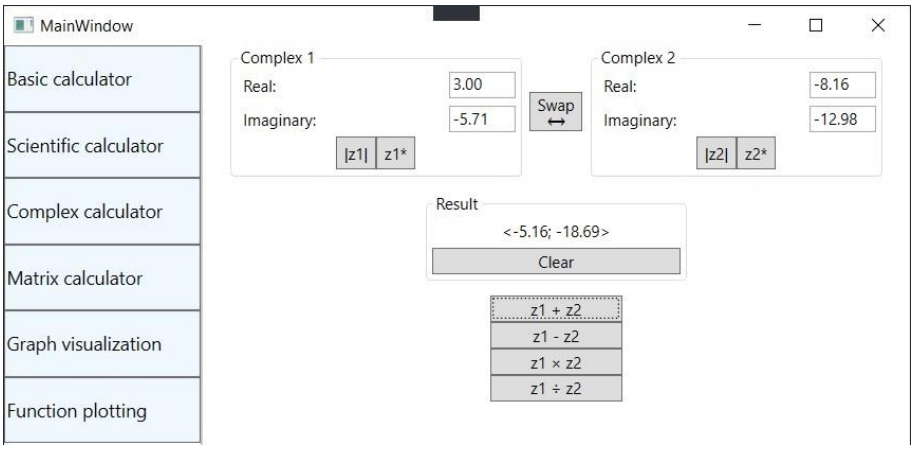
4.2. ábra: MainViewModel osztály egy részlete

Ezt követően már csak annyi dolgom lesz a további funkciók megvalósításakor, hogy mindegyikhez készítsek egy metódust a *NavigateToBasicCalculator* mintájára, a *MainWindow.xaml*-ből pedig adatkötéssel tudom megadni *Command*-ként.

4.2. Komplex kalkulátor

A komplex számok között elvégezhető műveletekre alkalmas kalkulátor fejlesztése során nem volt szükséges egy újabb *Model* osztályt létrehoznom, mivel a Microsoft ezeknek a reprezentálását már megvalósította a .NET Runtime-on belül a System.Numerics névtérben [8]. Ezen a funkción belül két oszlop található, amikben meg lehet adni a két komplex szám valós és képzetes részét beviteli mezőkön keresztül. A jobb felhasználói

élmény érdekében lehetőséget biztosítottam arra is, hogy további gépelés nélkül csupán a *Swap* gombbal meg lehessen cserélni a két komplex számot. Az utóbbi megvalósítását kifejezetten egyszerűvé tette a C# 7.0-ban bevezetett *tuple* típus.



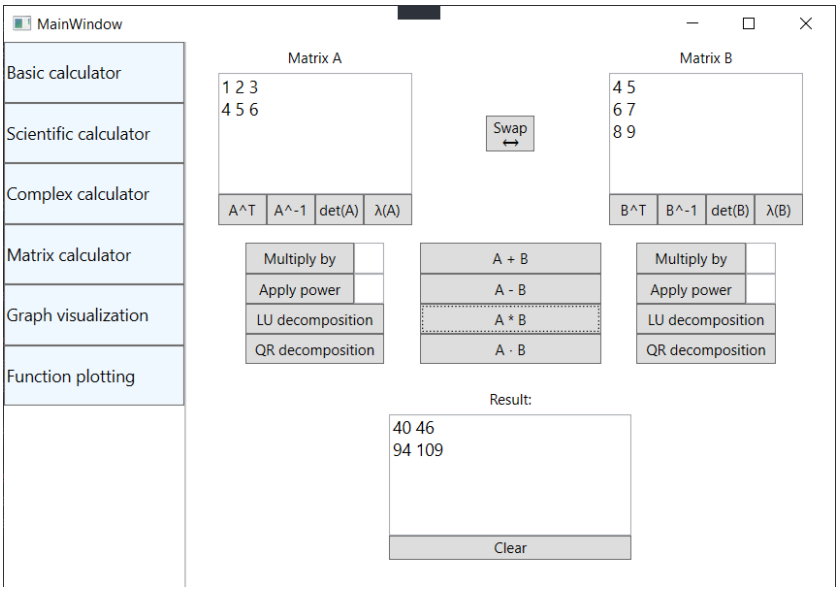
4.3. ábra: komplex kalkulátor működés közben

A 4.3. ábrán lehet látni, hogy milyen felhasználói felületet kapott a komplex kalkulátor, továbbá az előzőleg említett navigáció is látható a kép bal oldalán. Mind a két komplex számra külön-külön elvégezhető az abszolút érték és konjugálás művelet, melyek nemcsak az eredmény címkét módosítják, hanem a megfelelő 1-es vagy 2-es sorszámú komplex szám valós és képzetes részét is módosítják az eredménynek megfelelően. Lehet látni, hogy az eredmény alatt van egy *Clear* gomb is, ami alaphelyzetbe helyezi a komplex kalkulátort, azaz törli a két komplex számot és az eredményt is lenullázza. Végül a kép alján látható négy gombbal lehet elvégezni az alapműveleteket. Ha esetleg nem ír be valamilyen értéket a felhasználó a beviteli mezőkbe, akkor az alkalmazás a 0 értékekkel fog dolgozni.

4.3. Mátrix kalkulátor

A sokak által ismert MathNet.Numerics NuGet csomag segítette végig a fejlesztés folyamatát a mátrix kalkulátor esetén, mivel ez a csomag rendelkezik mátrix reprezentációval, és az ehhez megfelelő műveletek elvégzésére is alkalmas. Az egyszerűbb kezelhetőség végett létrehoztam egy *MatrixModel* nevű osztályt a Modell rétegbe, amit vagy a mátrix dimenzióinak megadásával tudok példányosítani, vagy pedig egy mátrix objektum átadásával elvégez egy másolást a két mátrix között. A komplex kalkulátorhoz képest a *View* és *ViewModel* megvalósítása nehezebb feladatnak bizonyult a mátrixok vizualizálása miatt, azonban egy olyan megoldásra jutottam, hogy egy WPF-be beépített *TextBox Text* attribútumának kötés segítségével adtam meg a mátrixot

karakterlánccá alakítva. Emiatt szükséges volt két olyan metódust megvalósítanom a *ViewModel*-en belül, ami lehetővé tette a mátrixszá és karakterlánccá konvertálást mind a kettő irányban.



4.4. ábra: két mátrix összeszorzása a mátrix kalkulátorban

A 4.4. ábrán lehet látni a mátrix kalkulátort működés közben, és egyben a felhasználói interfész kinézetét is. A komplex kalkulátorhoz hasonlóan két oszlopban helyezkedik el a két mátrix, amiket szintén beviteli mezőn keresztül lehet módosítani. A két mátrix között szintén van egy *Swap* gomb, ami könnyebbé teszi a felhasználó számára az A és B mátrix értékeinek megcserélését. Mind a két mátrix alatt találhatóak gombok, melyeknek a feladatai a következők: transzponálás, inverz kiszámítása, determináns meghatározása, valós sajátértékek kiszámítása. Az előbbi kettő művelet nemcsak a *Result* mezőbe írja bele az eredményt, hanem a megfelelő mátrix értékeit is megváltoztatja. A determinánshoz és sajátértékhez kapcsolódó funkciók nem módosítják a mátrixokat, csupán az eredményt írják ki. Középen helyezkednek el az alapműveleteket elvégző gombok, és közülük a legutolsó az elemenkénti szorzást végzi el (4.5. ábra).

```
[RelayCommand]
private void PointwiseMultiplyMatrices()
{
    UpdateMatrices();
    var result = MatrixA.Data.PointwiseMultiply(MatrixB.Data);
    ResultString = MatrixToString(result);
}
```

4.5. ábra: elemenkénti szorzásért felelős kódrészlet

Lehet látni a fenti ábrán, hogy a szorzás elvégzése előtt aktualizálásra kerül a két mátrix értéke, ami az előbb említett konverziós műveletek közül a karakterlánc-mátrix irányú átalakítást végzi el.

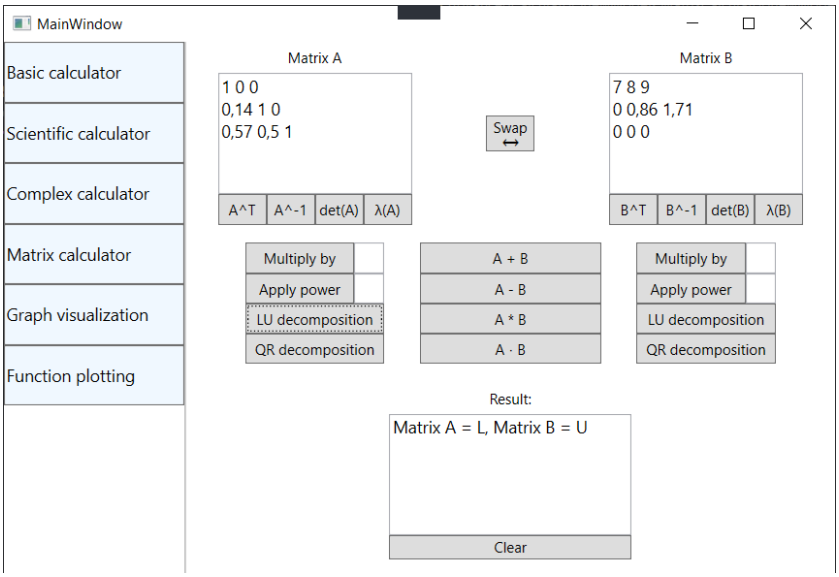
A középső sorban az alapműveletek csoportjának két oldalán találhatóak még olyan műveletek, amik specifikusan elvégezhetők a mátrixokra. A legelső a mátrix skalárral történő szorzását végzi el. Ha a felhasználó nem ad meg értéket, akkor a 0 értékkel kerül beszorzásra a mátrix. Az alatta lévő gomb a mátrixot egy 0 vagy annál nagyobb egész hatványra emeli; üres beviteli mező esetén az első hatványra emeli a mátrixokat. Az utolsó két gomb az A vagy B mátrix LU és QR felbontását végzi el.

```
[RelayCommand]
private void DecomposeLU(string name)
{
    UpdateMatrix(name);

    if (name == nameof(MatrixA))
    {
        var decomposition = MatrixA.Data.LU();
        MatrixA.Data = decomposition.L;
        MatrixB.Data = decomposition.U;
    }
    else if (name == nameof(MatrixB))
    {
        var decomposition = MatrixB.Data.LU();
        MatrixA.Data = decomposition.L;
        MatrixB.Data = decomposition.U;
    }

    MatrixStringA = MatrixToString(MatrixA.Data);
    MatrixStringB = MatrixToString(MatrixB.Data);
    ResultString = "Matrix A = L, Matrix B = U";
}
```

4.6. ábra: az LU-felbontást végző metódus

A 4.6. ábrán látható metódus kerül meghívásra abban az esetben, ha valamelyik mátrixot szeretné felbontani a felhasználó az LU módszer szerint. A metódus paraméterben egy nevet vár, ami az A vagy B mátrix azonosítására szolgál. Több olyan művelet is ezt a logikát alkalmazza, amelyek esetén csak egy mátrixszal szeretnénk végezni műveleteket – ennek megfelelően létezik egy *UpdateMatrix* nevű metódus is. ami csak egy mátrix értékét aktualizálja. A felbontás az előbbi műveletekhez képest az eredmény egyedi módon jelenik meg olyan értelemben, hogy az A vagy B mátrix LU vagy QR felbontását úgy végzi el, hogy az A mátrixba helyezi a felbontásnak megfelelő

L vagy Q mátrixot, míg a B mátrixba helyezi el az U vagy R mátrixot hasonló módon. Ezekben az esetekben a *Result* mező nem az eredményt jeleníti meg, hanem csupán a felhasználót tájékoztatja, hogy a felbontás részeit hova helyezte el (4.7. ábra).



4.7. ábra: egy 3×3-as mátrix LU-felbontása

4.4. Gráf vizualizáció

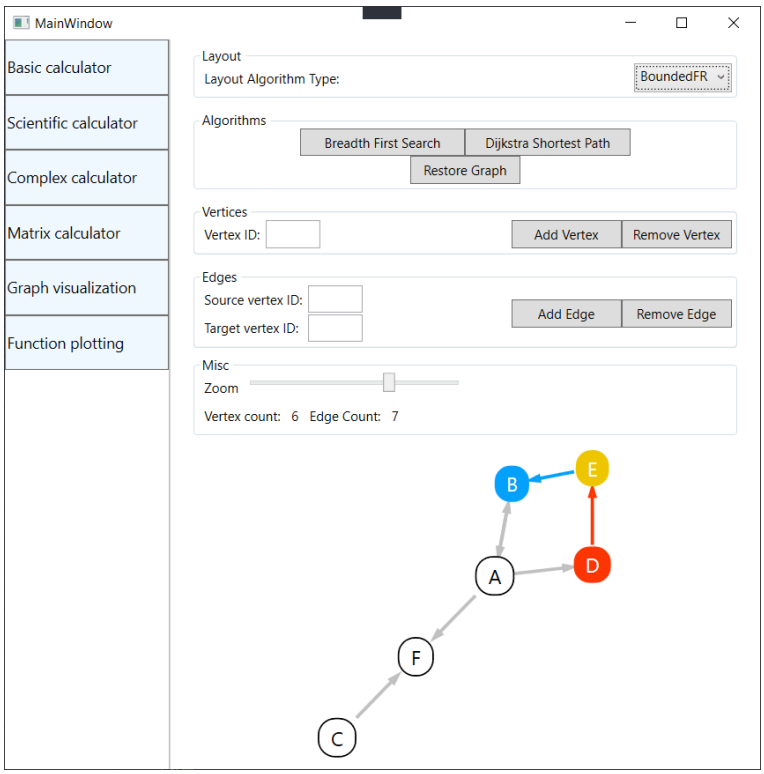
A gráfok könnyű reprezentálásához a QuikGraph nevű csomagot használom, aminek a segítségével tudok megadni csúcsokat, közöttük lévő éleket, továbbá beépített algoritmusokat is tartalmaz a csomag. Mivel azt szeretném, hogy a csúcsok tároljanak el egy azonosítót, az élek esetén pedig azonosítóra, súlyra és színre van szükségem, ezért több koncepciót igazoló (Proof of Concept - PoC) osztályt is létrehoztam, amelyek kibővítik a beépített osztályok funkcionalitását. Nem elég csak az élekhez és csúcsokhoz készítenem *PoC* prefixummal rendelkező saját osztályokat, hanem magához a gráf reprezentálásához is szükségem van egy ilyenre (4.8. ábra):

```
public class PocGraph : BidirectionalGraph<PocVertex, PocEdge>
{ }
```

4.8. ábra: *PocVertex*-et és *PocEdge* típust használó kétirányú gráf

A reprezentáción túl vizualizálni is kell a gráfokat, amihez a GraphShape és GraphShape.Controls NuGet csomagokat használtam. Az utóbbi vizualizációs csomagok használata a megszokotthoz képest egy nehezebb feladat volt, mivel ezek nem rendelkeznek dokumentációval, hanem a készítőjük tett közzé egy mintaprojektet a GitHub nevű oldalon [9]. Az előbb említett PoC prefixumú osztályokat azért is hoztam létre, hogy könnyebben testre tudjam szabni a gráf megjelenítését, így el tudtam azt érni,

hogy a csúcsok egy szöveget megjelenítő körök legyenek nyilakkal összekötve. A GraphShape-ben be lehet állítani átfedést megszüntető- és elrendezési algoritmusokat is, amelyek közül az előbbi azért felel, hogy a csúcsok ne takarják ki egymást, az utóbbi pedig a gráf csúcsainak elhelyezkedését befolyásolja különböző algoritmusok segítségével.



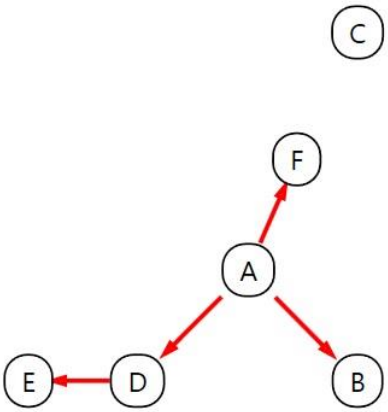
4.9. ábra: 6 csúcsú gráf CompoundFDB elrendezési algoritmussal, „E” csúcs kijelölve

A fenti ábrán lehet látni a gráf vizualizációért felelős üzemmód felhasználói felületét. Észrevehető a képen, hogy a B, D és E csúcsok és köztük lévő élek más színnel jelennek meg. Ennek oka, hogy a GraphShape külön kiemelési algoritmusokkal is rendelkezik, aminek köszönhetően sárgára színezi azt a csúcsot, amin az egérmutató van. Kék lesz az az él és csúcs, amelyiknek a kezdőpontjában van a kijelölt csúcs, fordított esetben pedig piros, ha a kijelölt csúcs a végpont.

Mivel nincs túlságosan sok elrendezési algoritmus, ezért az összeset elhelyeztem egy lenyíló listában a gráf fölötti menü legfelső részében, aminek a lenyitásával ki tudja választani a felhasználó, hogy milyen algoritmussal szeretné a gráfot megjeleníteni, így azt kiválasztva interaktívan megváltozik a gráf csúcsainak a pozíciója, amit a csúcson történő bal egérgomb lenyomásával és az egér mozgatásával is el lehet érni Hogy megfelelően működjön a lenyíló lista esetén az adatkötés, és továbbra is igazodjak az

MVVM mintához, ezért *ObservableCollection*-t használtam, amelyre eddig még nem volt szükségem a korábbi funkcióknál.

A 4.9. ábrán az „Algorithms” csoportban három gomb helyezkedik el, melyek közül a legelső a szélességi keresést hajtja végre egy megadott csúcsból, míg a következő a legrövidebb utat keresi meg Dijkstra módszerrel két csúcs között, végül a „Restore Graph” gomb az eredeti állapotába állítja vissza a gráfot.



4.10. ábra: „A” csúcsból indított szélességi keresés eredménye

A 4.10. ábrán látható, ahogyan a 4.9. ábrán bemutatott gráfon végrehajtotta az alkalmazás a szélességi keresést. Ilyenkor az utat tartalmazó éleket pirosra színezi, a többit pedig törli, továbbá egy *MessageBox*-ban is megjeleníti az alkalmazás az utat karakterlánc formában az élek azonosítóját használva. A szélességi keresés esetén a „Vertex ID”-t kell megadni a lenti beviteli mezőkben, míg a Dijkstra algoritmus a „Source Vertex ID” és „Target Vertex ID” mezők kitöltését várja el. Ennek elmulasztása esetén szintén egy *MessageBox* figyelmeztet.

```
[RelayCommand]
private void RemoveVertex() {
    var vertexToRemove = Graph.Vertices.FirstOrDefault(v => v.ID == VertexId);
    if (!string.IsNullOrWhiteSpace(VertexId) && vertexToRemove != null) {
        Graph.RemoveVertex(vertexToRemove);
        OnPropertyChanged(nameof(Graph));
        VertexId = string.Empty;
    } else {
        MessageBox.Show("The vertex cannot be found.", "Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

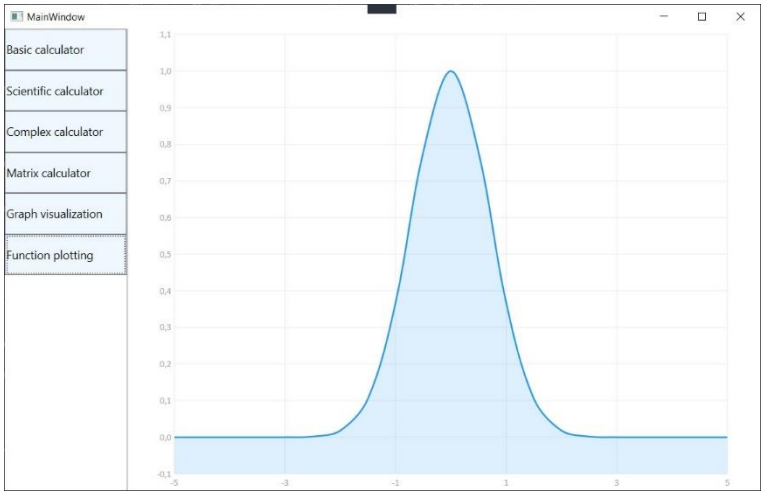
4.11. ábra: csúcs törléséért felelő metódus

Törekedtem arra, hogy minél inkább interaktív legyen ez a funkció, ezért a „Vertices” és „Edges” csoportokban lehet csúcsokat és éleket létrehozni, továbbá törölni. Odafigyeltem a hibakezelésre is, ezért a mezők üresen hagyása esetén figyelmeztet az alkalmazás, Továbbá ugyanolyan nevű csúcsokat sem lehet létrehozni, és két csúcs közé sem lehet kettő, ugyanolyan irányú élt. Hasonló ellenőrzések vannak törlés esetén is (4.11. ábra).

Legvégül kényelmi funkció gyanánt lehet állítani a zoom mértékét egy csúszkával, illetve az alatta elhelyezkedő címkéken lehet látni a gráf csúcsainak és éleinek számát, melyek minden egyes felhasználói interakció esetén frissülnek.

4.5. Függvényábrázolás

A függvényábrázolás funkció egyelőre csak a kódba beírt függvényeket tudja vizualizálni egy megadott intervallumban. Ehhez a LiveCharts.Wpf nevű NuGet csomagot használom, melynek a fejlesztésében sokat segített ennek a dokumentációja [10].



4.12. ábra: függvény vizualizálása

A fenti ábrán egy módosított Gauss-függvény ábrázolása látható a [-5, 5] intervallumon.

5. Összegzés és jövőbeli terjeszkedési lehetőségek

5.1. A projekt összefoglalása

A számológépem fejlesztésének folyamata során egy olyan alkalmazást hoztam létre, amely nemcsak az elvárásaimnak felel meg, hanem úgy gondolom, hogy a mindennapi életben is egy hasznos eszköz lehet. Az alkalmazás képes különböző matematikai műveletek elvégzésére, kezdve az alapvető számításoktól egészen a bonyolultabb tudományos funkciókig, mátrixműveletekig, komplex számok kezeléséig, illetve gráfok vizualizálására. A felhasználói felület tiszta és intuitív, a funkcionalitás pedig széles körű, ami miatt a későbbiekben mind az oktatásban, mind pedig programozók körében hasznos eszköz lehet az általam készített alkalmazás.

5.2. Lehetséges továbbfejlesztési lehetőségek

A projekt jelenlegi állapota mellett sok továbbfejlesztési lehetőséget is látok. Egyik ilyen irány a multiplatform támogatás bevezetése, ami lehetővé tenné az alkalmazás futtatását nem csak Windows operációs rendszeren, hanem más rendszereken is. Az Avalonia UI egy kiváló lehetőség ebben a tekintetben, mivel lehetővé teszi az alkalmazások fejlesztését egyszerre több platformra, beleértve a Linux, macOS, Android és iOS rendszereket. Másik lehetőségként a Microsoft Blazor technológiájával interaktív webalkalmazássá is alakíthatnám a projektemet, így ezzel is szélesítve a potenciális felhasználói bázist.

A további funkciók bevezetése szintén fontos terjeszkedési lehetőség számomra. Például egy *ModalService* implementálásával lehetővé tenném a párbeszédablakok kezelését és megjelenítését. Ennek a szolgáltatásnak köszönhetően néhány meglévő és további funkciók esetén könnyebben tudnék a felhasználótól adatot bekérni, és azt feldolgozni a megfelelő módon.

A gráfvizualizációs funkciónál további gráfalgoritmusokat szeretnék használni, mint például a Hamilton-út és kör megkeresése egy gráfban, A* algoritmus megvalósítása, súlyozott gráf esetén súly megjelenítése az élen tooltip helyett, gráf megfelelő frissítése, elrendezése változtatások esetén.

5.3. Személyes tapasztalatok és tanulságok

A projekt során számos értékes tapasztalatra tettem szert leginkább a Microsoft ökoszisztémán belül. Az egyik legfontosabb ezek közül a WPF használata és az MVVM minta alkalmazása volt, ami lehetővé tette számomra, hogy hatékonyan és

rendszerezetten építsem fel az alkalmazás felhasználói felületét. A NuGet csomagok hasznosságát is megtapasztaltam, amelyek jelentősen leegyszerűsítették a fejlesztési folyamatot, és hozzájárultak a projekt gyors és hatékony előrehaladásához.

A verziókezelés használata a Git és GitHub segítségével szintén nélkülözhetetlen volt a projekt menedzselésében, különösen olyan esetben, amikor az alkalmazás nem megfelelő módon működött, vagy mégsem voltam elégedett a végeredménnyel, akkor könnyen vissza tudtam állítani egy korábbi állapotot. Végül, de nem utolsósorban, a programozás során alkalmazott különböző minták és elvek használata, mint például a függőséginjektálás vagy az adatkötés tovább mélyítették a tudásomat, és hozzájárultak a szakmai fejlődésemhez.

Irodalomjegyzék

[1] Microsoft hivatalos dokumentációja a C# programozási nyelvről:
https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
Utolsó megtekintés dátuma: 2023. december 15.

[2] M. MacDonald, Pro WPF 4.5 in C#: Windows Presentation Foundation in .NET 4.5, Springer Science+Business Media New York: Apress, 2012.

[3] Microsoft hivatalos dokumentációja az MVVM-ről:
https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm
Utolsó megtekintés dátuma: 2023. december 20.

[4] F. Staiano, Designing and Prototyping Interfaces with Figma: Learn essential UX/UI design principles by creating interactive prototypes for mobile, tablet, and desktop, Birmingham: Packt Publishing Ltd, 2022.

[5] Microsoft: Bevezetés az MVVM Toolkit használatához:
https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/
Utolsó megtekintés dátuma: 2024. január 5.

[6] M. Seemann és S. van Deursen, Dependency Injection Principles, Practices, and Patterns, Shelter Island, New York: Manning Publications, 2019.

[7] Microsoft hivatalos dokumentációja a dependency injection-ről:
https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-usage
Utolsó megtekintés dátuma: 2023. december 22.

[8] A Complex.cs implementációja a .NET Runtime-on belül:
https://github.com/dotnet/runtime/blob/5535e31a712343a63f5d7d796cd874e563e5ac14/src/libraries/System.Runtime.Numerics/src/System/Numerics/Complex.cs
Utolsó megtekintés dátuma: 2024. március 29.

[9] GraphShape hivatalos mintaalkalmazás:
https://github.com/KeRNeLith/GraphShape/tree/master/samples/GraphShape.Sample.
Utolsó megtekintés dátuma: 2024. február 05.

[10] LiveCharts.Wpf dokumentációja:
https://v0.lvcharts.com/App/examples/Wpf/start
Utolsó megtekintés dátuma: 2024. május 15.

Nyilatkozat

Alulírott Hankó Péter Programtervező Informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Számítástudomány Alapjai Tanszékén készítettem, Programtervező Informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Dátum: 2024. január 25.

Hankó Péter

.....

Aláírás