

TDK Dolgozat

Szerzők

Bakos Dávid

Bezeczki Bence



Eszterházy Károly Főiskola
Matematikai és Informatikai Intézet
Természettudományi Kar

RoboLOGO projekt bemutatása különböző platformokon

Készítők:

Bakos Dávid
Programtervező informatikus Bsc.
3. évfolyam

Bezeczki Bence
Programtervező informatikus Bsc.
3. évfolyam

Témavezető:

Dr. Geda Gábor
Főiskolai docens

Eger, 2012

Tartalom

Bevezetés:	5
RoboLogo projekt IDE.....	6
Desktop IDE:	6
Az alábbi technológiákat használtuk fel:	6
Telefonon futó alkalmazás:	6
Desktop alkalmazás részletes bemutatása:.....	7
1. Design:	7
2. Logika:.....	8
SQLite áttekintés:	8
Funkciók bemutatása:	9
1. File menü:	9
Új projekt létrehozása:	10
2. Szerkesztés:	10
3. Futattás:	10
Program küldése robotra:	12
Rajzvászon törlése:	12
Eljárás/változók:	12
Eljárások definiálása:	12
Változók definiálása:.....	12
DataGridView áttekintés:	13
Robot felépítése	13
Arduino mikrokontroller:	13
Szoftver:.....	13
Hardver:.....	14
Bluetooth modul:	14
Léptető motor:	14
Bolygó kerekek:	14
Szervó motor:	14
Lítium akkumulátorok:	14
Robot jelenlegi kinézete:	15
Többnyelvűség – Multilanguage	16
Erőforrások – Resource Dictionary.....	17
Szintaxis Kiemelés – Syntax Highlight.....	19

CommonSyntaxProvider.....	21
LogoKeywords	22
SyntaxHighlight.....	22
RichTextBox komponens WPF alapokon	26
Kapcsolatteremtés – Bluetooth.....	29
BluetoothSupport.....	30
BluetoothHelper	30
A szimulátor	33
További tervek, megvalósításra váró funkciók.....	42
Autocompletion lehetőség – Automatikus kódkiegészítés	42
Szintaktikai elemző	42
Multiplatform	42
Irodalomjegyzék	43

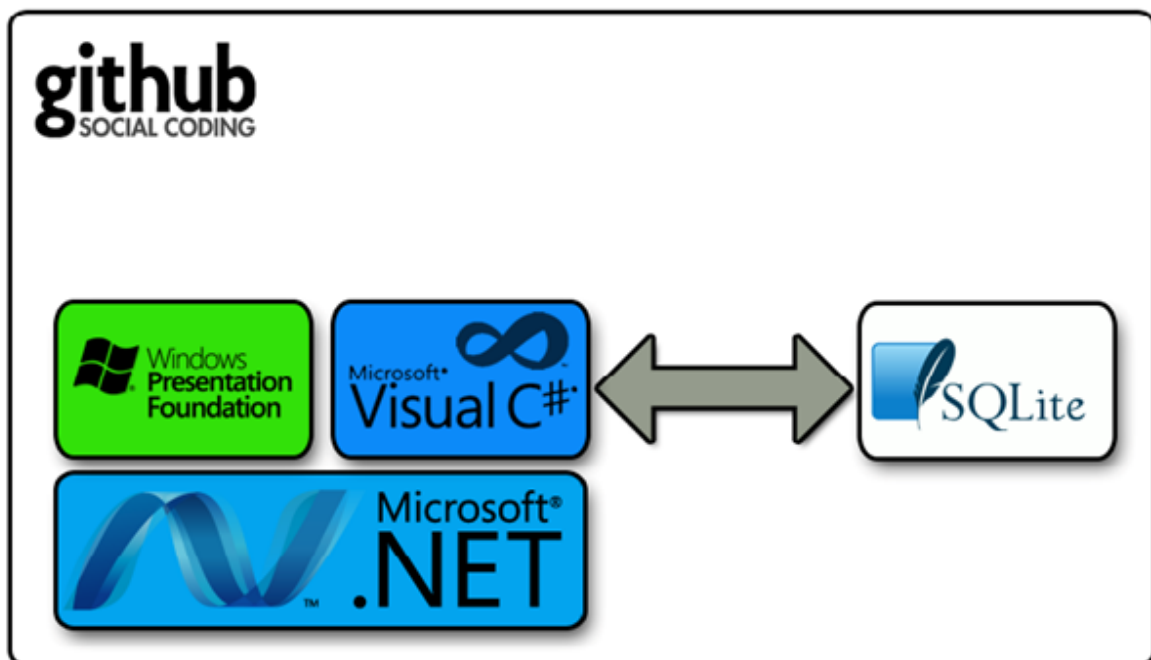
Bevezetés:

Dolgozatunk témája a RoboLogo projekt fejlesztői környezete és több platformon való bemutatása.

RoboLOGO projekt felépítése:

- Robot
- Android alkalmazás
- Desktop IDE (WPF-C#)
- RoboPreter: Logo
 - C# interpreter
 - Arduino interpeter

Desktop alkalmazás felépítése az 1. ábra látható:



1. ábra

Github verziókövető rendszert használtuk, ez ingyenes a nyílt forráskódú szoftverek számára. A GIT egy nyílt forráskódú elosztott verziókezelő szoftver másnéven egy szoftver forráskód kezelő rendszer. Eredetileg a Linux fejlesztője Linus Tórvard fejlesztette ki és egy kis öniróniával saját magáról nevezte el (git jelentése: ostoba, kellemetlen).

Számos nagy volumenű projekt használja manapság is a git verziókövetőt, talán legfontosabbak ezek közül:

- Linux rendszermag
- Gnome
- Samba

A RoboLOGO projekt a következő linken érhető el: <https://github.com/RoboLOGO> .

RoboLogo projekt IDE

A fejlesztői környezetünk elkészült Windows (desktop) és Androidos (mobil) platformra is. Ezzel elősegítve, hogy több irányban is elterjedhessen a szoftverünk.

Desktop IDE:

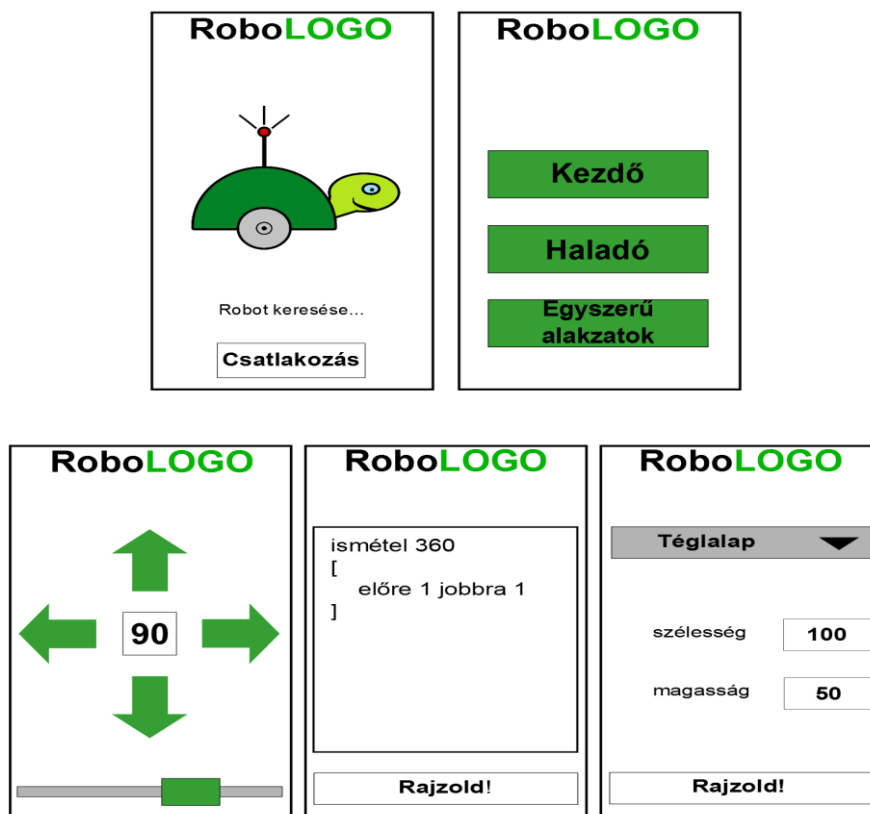
IDE (Integrated Development Environment), integrált fejlesztői környezetet C#-ban Visual Studio 2010 (.Net 4.0) majd később Visual Studio 2012 (.Net 4.5) segítségével készítettük el.

Az alábbi technológiákat használtuk fel:

- WPF (Windows Presentation Foundation)
- .Net 4.0, majd .Net 4.5
- SQLite adatbázis kezelő

Telefonon futó alkalmazás:

A telefonon futó alkalmazásnál az volt a fő célunk, hogy könnyen kezelhető legyen, tartalmazzon egy olyan funkciót amivel LOGO tudást nélkülözve is tudjuk irányítani a robotot, de e mellett még egy haladó funkciót is kifejlesztettünk ahol a hozzáértőek is kipróbálhatják tudásukat. A telefonon futó alkalmazás dizájnját a 2. ábra mutatja.



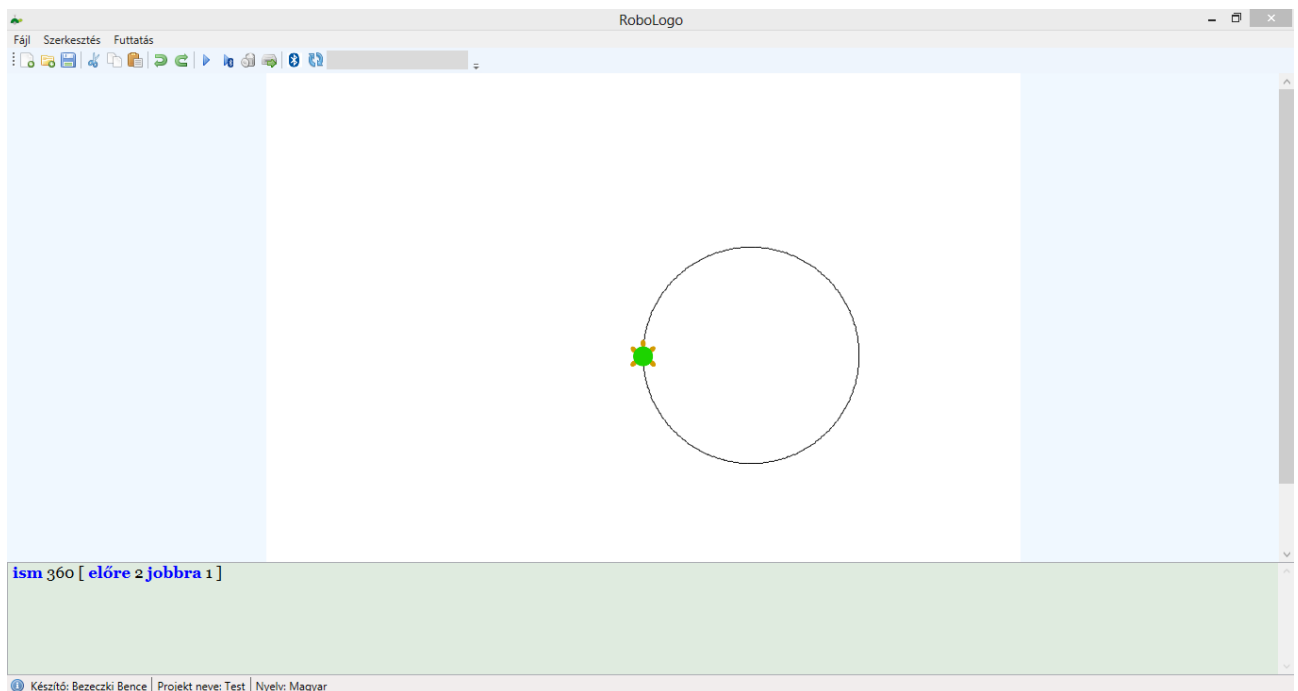
2. ábra

Desktop alkalmazás részletes bemutatása:

1. Design
2. Logika

1. Design:

Windows Presentation Foundation segítségével készült el. Azért esett erre a választásunk, mert viszonylag egyszerűen, gyorsan látványos eredményt lehet vele elérni. A .Net 4.5-re történő váltásról meg még később fogok írni. A kinézetet a 3. ábra mutatja.



3. ábra

WPF áttekintés:

A Windows Presentation Foundation a .NET keretrendszer egyik grafikus alrendszere mely a XAML¹ leíró nyelv segítségével egyszerűen építhetünk grafikus felhasználói felületet (GUI). A XAML egy XML² alapú deklaratív nyelv melyet a Microsoft fejlesztett ki, a Visual Studio 2008-al együtt jelent meg.

Következetes programozási modellt ad az alkalmazásfejlesztőknek teljes mértékben elősegítve a logika és a megjelenés szétválasztását.

A grafikai elemei nem az elavult raszteres, hanem már a vektorgrafikus megoldást támogatják. A vektorgrafika előnye, hogy az elemek bátran minőségromlás nélkül átméretezhetőek, skálázhatóak, torzíthatók, forgathatók és ezen felül még a velük összemérhető raszteres elemeknél kisebb tárhelyet igényelnek, hátrány pedig az, hogy bizonyos esetekben megemelkedhet a számítási teljesítmény.

¹Extensible Application Markup Language

²Extensible Markup Language

Nagy előnye a stílusok használata. Ezeket a stílusokat úgy kell elképzelni, mint a HTML rangsorolt stíluslapjait a CSS³-t. De a WPF stílusai újra hasznosíthatóak, sokkal egységesebbek és sokrétűbbek. Lehetővé teszik az összes stíluselem beállítását pl.: külső, belső margók helyét, színét stb. A stílusok használhatóak nem látható tulajdonságok beállításaira is.

2. Logika:

A programunk teljesen egy SQLite adatbázisra épül rá.

Adatbázis táblák:

- Methods (neve, metódus törzse)
- Options (projekt neve, canvas méret)
- Variables (neve, értéke)

Adatbázis táblák létrehozása az 1. kódrészben látható.

```
private void TablesInit()
{
    string methodsSQL = "CREATE TABLE Methods (Name VARCHAR(25) UNIQUE, Method TEXT)";
    string optionsSQL = "CREATE TABLE Options (Name VARCHAR(25) UNIQUE, Value TEXT)";
    string variablesSQL = "CREATE TABLE Variables (Name VARCHAR(25) UNIQUE, Value NUMBER)";
    sqlitewriter.ExecuteNonQuery(methodsSQL, sqliteCon);
    sqlitewriter.ExecuteNonQuery(optionsSQL, sqliteCon);
    sqlitewriter.ExecuteNonQuery(variablesSQL, sqliteCon);
}
```

1. kódrész

SQLite áttekintés:

Azért esett a választásunk az SQLite-ra mert ingyenes, nyílt forráskódú és platform független, D. Richard Hipp tervezte és alkotta meg. Az SQLite napjainkban kezd elterjedni, több nagyobb cég is tárol információkat ilyen formában. Megvalósítja az SQL-92 szabvány nagy részét így kis mértékben támogatja a triggereket is. Érdekes egy adatbázis kezelőtől, hogy egy típus nem a tábla oszlopaihoz van hozzárendelve, hanem egyedi értékeket, tehát dinamikus típuskezelés használ. Emellett gyengén típusos adatkezelés van benne, ami nagyban hasonlít a Perl-höz ahol pl.: string típusú adat beilleszthető integer oszlopba, igaz ilyenkor az SQLite a stringet megpróbálja integerré konvertálni, persze ha az oszlop támogatott típusa integer. A szöveg típusú adatokat alapértelmezetten UTF-8-ban tárolja el.

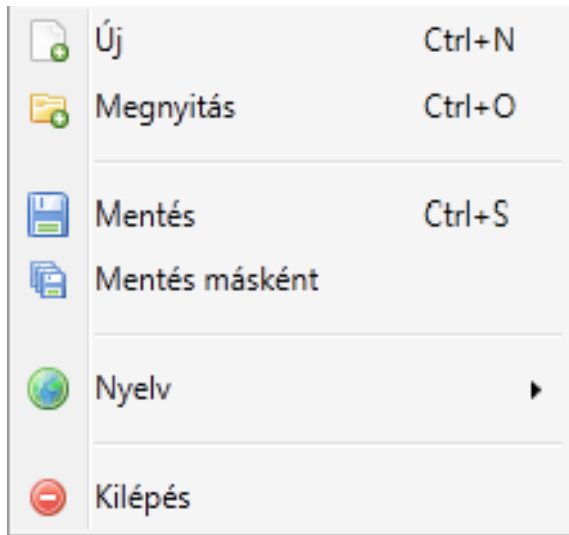
Egy adatbázist több szálon és több processzel lehet egyszerre használni. Az olvasási kérélmeket párhuzamosan elégíti ki. Az írási kérélmek végrehajtása akkor történik meg, ha már semmilyen más kérés kiszolgálása nincs folyamatban egyébként nem sikerült az írási folyamat és hibakóddal tér vissza.

Számunkra hatalmas előnyt jelentett az is, hogy egyetlen fájlban tárolódik az egész adatbázis így ezt fel tudtuk használni egy úgynevezett solution fájlban, ami tartalmazza a megírt programunkról az összes információt.

³ Cascading Style Sheets

Funkciók bemutatása:

1.File menü:



4. ábra

- Új projekt létrehozása
- Megnyitás
- Mentés
- Mentés másként
- Nyelvválasztás
- Kilépés

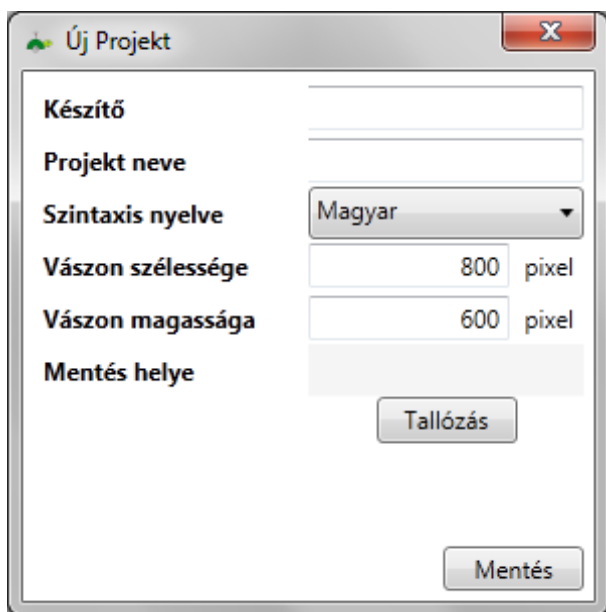
A file menü bemutatását a saját solution file-unk ismertetésével kezdeném, amit már említettem feljebb, hogy tulajdonképpen egy SQLite adatbázist használunk fel erre a célra. A solution fájlunk neve RoboLOGO Solution. Itt tárolunk minden információt a megírt szoftverünkről:

- Program kód
- Változók
- Eljárások
- Beállítások

Nyelv:

A RoboLOGO fejlesztői környezetben lehetőség van több nyelv közül is választani, jelen esetben a magyar és az angol van implementálva.

Új projekt létrehozása:



5. ábra

Az 5. ábra mutatja, hogy a programunk többnyelvű szintaxisra is képes. Jelenleg angol és magyar nyelv közül választhat a felhasználó. A szintaxis implementációja Resource Dictionary-vel van megoldva, ami nagyban elősegíti ezek egyszerű bővítését.

Itt adható meg a vászon szélessége és magassága is.



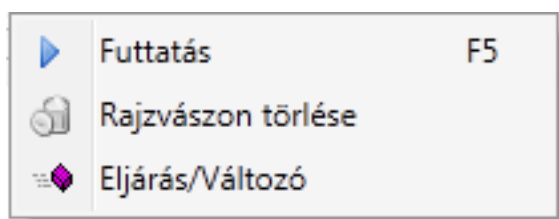
6. ábra

2. Szerkesztés:

- Vissza
- Előre
- Másolás
- Kivágás
- Beillesztés

Ez egy teljesen átlagos szerkesztés menüpont (6. ábra), a C# előre definiált függvényeit használja: undo, redo, copy, cut, paste.

3. Futattás:



7. ábra

- Kód futtatása
- Rajzvászon törlése
- Eljárás/változó deklarálása

Futtatás:

A futtatásra kattintva fordíthatjuk le a megírt kódunkat, ezzel a funkcióval csak a programban lévő teknős fog elindulni és rajzolni a vászonra (canvas). Ez egy szimulációt nyújt, ki tudjuk próbálni a programunkat, mielőtt átküldhetnénk a robotra. Ezen kívül még arra törekedtünk, hogy az ImagineLOGO-hoz hasonló, már a megszokotthoz hasonló felhasználói felületet biztosítsunk, ezzel arra törekedve, hogy ne kelljen egy teljesen új dolgot megtanulni, felhasználható a már ez előtt elsajátított LOGO programozási tudás. Itt volt nagy szükségünk arra, hogy .NET 4.0-ról áttérjünk .NET 4.5-re, erre épül a szimulátorunk motorja.

```
private void RunClick(object sender, RoutedEventArgs e)
{
    //..
    List<Robopreter.Command> com = rp.Run(menu.GetFullSource());
    for (int i = 0; i < com.Count; i++)
    {
        Draw(com[i]);
        int time = GetTime(com[i]);
        Wait(time);
    }
    //..
}
```

2. kódrész

Ezzel az volt a problémánk, hogy a futtatásra kattintva animáltan el kellett volna indulnia a teknősnek, de csak lefagyott a program és animáció nélkül kirajzolta alakzatot a képernyőre (2. kódrész).

Persze meg lehetett volna ezt oldani .Net 4.0-ban is úgy, ha több szálvá tesszük a programunkat, de ez lényegesen bonyolultabb lett volna. Példa, hogy ez hogy működik tökéletesen .Net 4.5-ben:

```
private async void RunClick(object sender, RoutedEventArgs e)
{
    //..
    List<Robopreter.Command> com = rp.Run(menu.GetFullSource());
    for (int i = 0; i < com.Count; i++)
    {
        Draw(com[i]);
        int time = GetTime(com[i]);
        await Task.Factory.StartNew(() => Wait(time));
    }
}
```

3. kódrész

Az aszinkron hívás létrehoz egy új állapotgépet, ennek a segítségével, szépen animáltan mozogva rajzolja ki a teknős a kívánt alakzatot (3. kódrész).

Ezzel bonyolult többszálvú programozástól kíméltük meg magunkat.

Program küldése robotra:

Ez funkció csak a toolbar-on kapott helyett, arra szolgál, hogy a megírt letesztelt programot egy kattintással átküldhetjük a robotra.

Rajzvászon törlése:

Ez a funkció arra szolgál, hogy a rajzvászonról (canvas) le tudjuk törölni a rosszul sikerült alakzatot. Tulajdonképpen alaphelyzetbe állítja a canvast és a teknőst a „haza” parancssal beállítja a vászon közepére.

Eljárás/változók:

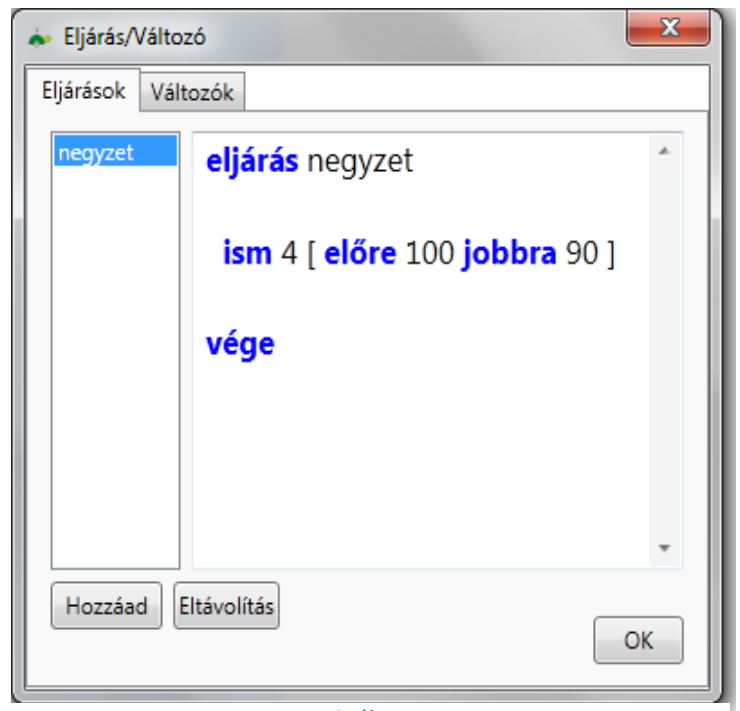
A szoftverünk képes tárolt eljárások és változók definiálására, ezzel is törekedve az ImagineLOGO kompatibilitásra.

Eljárások definiálása:

A 8. ábra mutatja, hogyan van lehetőségünk az eljárások létrehozására.

Az interpretert úgy írtuk meg, hogy rekurzióra is képes legyen.

Syntax Highlight nincs az ImagineLogóban, de a kényelem érdekében implementáltuk. A syntax highlighter segítségünkre van a program könnyű debugolásában, és ezzel a funkcióval sokkal jobban átlátható megírt program.

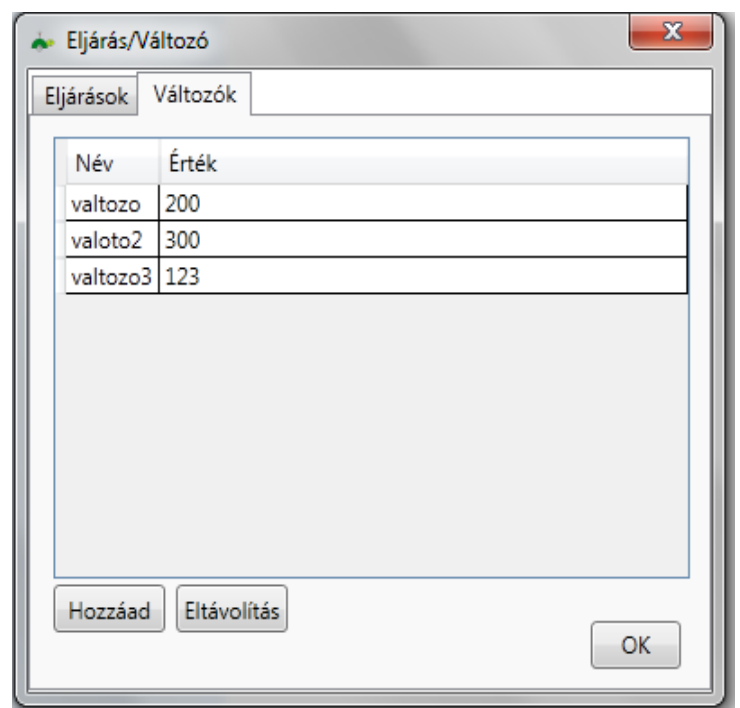


8. ábra

Változók definiálása:

A 9. ábra mutatja, a változók fület, itt adhatunk hozzá új változókat.

A RoboLOGO fejlesztői környezet DataGridView segítségével könnyíti meg a változók hozzáadását, módosítását.



9. ábra

DataGridView áttekintés:

DataGridView-t vezérlőelemet általában táblázatok kiírására használjuk fel. Az összetett tulajdonságokkal és metódusokkal ellátott vezérlővel adatbázisok lekérdező tábláit is megjeleníthetjük. A mi esetünkben azért volt szükség erre az elemre, hogy könnyedén módosíthatóak legyenek a változók nevei és értékei: a módosítás úgy történik, hogy kétszer kattint a módosítani kívánt attribútumra és egyszerűen átírja.

Robot felépítése

- ArduinoUNO mikrokontroller
- Bluetooth modul
- 2 léptető motor, 2 bolygókerék
- Motorvezérlő áramkör
- Szervó motor
- Lithium akkumulátorok

Arduino mikrokontroller:

Eredetileg Olaszországban gyártják, de ez nagyon drága, ezért mi az ugyan olyan jól működő kínai „utánzatot” használtuk.

Az Arduino egy az Atmel AVR mikrovezérlő családra épülő, szabad szoftveres elektronikai fejlesztőplatform, ezt arra tervezték, hogy különböző projektekben az elektronikai eszközök könnyebben hozzáférhetőek és kezelhetőek legyenek.

Szoftver:

Egy java nyelven írt keresztplatformos fejlesztői környezet, mely segítségével írhatjuk meg az arduino .ino kiterjesztésű programunkat. A programokat egy C/C++ nyelven írhatjuk, feltöltése USB-n keresztül történik a mikrokontrollerre. Az IDE-hez tartozik a „Wiring” nevezetű C/C++ programkönyvtár, amellyel egyszerűen végezhetjük el a leggyakoribb input/output műveleteket.

Hardver:

Az arduino boardok többféle változatban, méretben készülnek el. Ezek a változatok leginkább a belső memóriában, számítási kapacitásban és a be és kimenetek számában térnek el egymástól.

Léteznek olyan boardok amelyek beépített ethernet vagy bluetooth csatlakozóval rendelkeznek.

A teljesség igénye nélkül pár arduino board:

- Arduino UNO
- Arduino NANO
- Arduino Mega
- Arduino ADK

Bluetooth modul:

A mi arduino boardunknak nem volt beépített bluetooth csatlakozója ezért egy külső modult kellett használnunk, erre a robot és a megírt alkalmazások közötti kapcsolat miatt volt szükség.

Azért jó választás, mert kicsi az energia fogyasztása és a mai okostelefonok és számítógépek szinte mindegyike használ bluetooth-t.

Léptető motor:

A robot mozgását oldottuk meg ezzel:

- Előre
- Hátra
- Jobbra
- Balra

Bolygó kerekek:

A robot stabilitásáért és könnyebb fordulásáért felelősek.

Szervó motor:

Ezzel oldottuk meg a „tollatle”, „tollatfel” parancsokat a roboton

Lítium akkumulátorok:

A Lítium akkumulátorok létrehozásával először M. S. Wittingham foglalkozott az 1970-es években. Ezek még drágák és veszélyesek voltak a titán-szulfid-katód és fémes lítiumanód

miatt.

A fémes lítiumanód kisebb üzemzavar esetén is képes volt felforrósodni és szélsőséges esetben fel is robbanhatott. Később persze tökéletesítették ezt a hibát.

Előnyei:

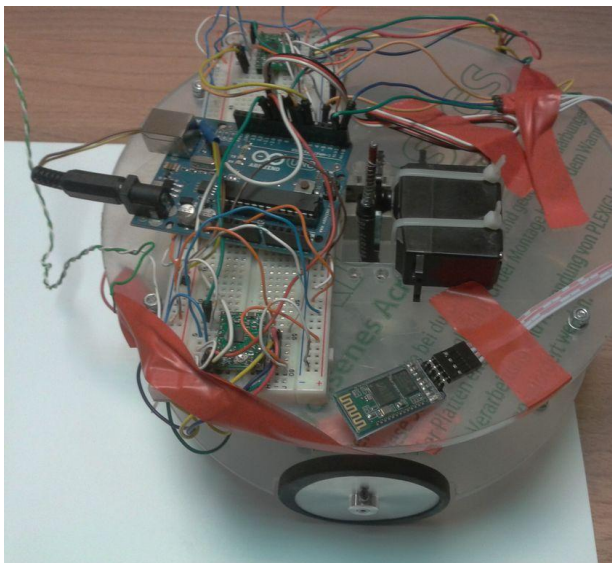
- A lítium a legkönnyebb fém így az ilyen anyagból készült akkumulátorok lényegesen könnyebben, mint a nikkel alapúak.
- Ni-MH technológiához hasonlóan nagyon kevés szennyező anyagot tartalmaznak, ezzel is védve a környezetet.
- Lassan veszíti el a töltöttségét

Hátrányai:

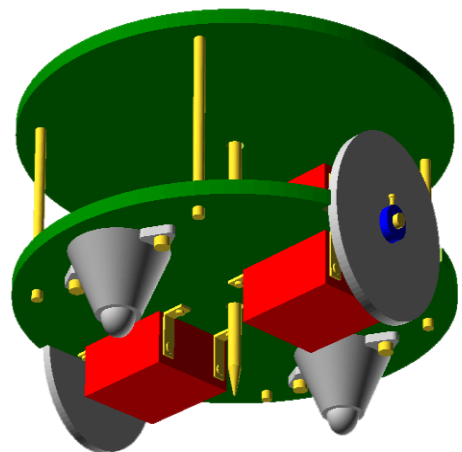
- Túltöltés vagy esetlen túlfeszültség esetén hő fejlődik, ami az akkumulátor felrobbanásához is vezethet
- Védőáramkörökre van szüksége
- Elhasználódási és öregedési probléma

Azért esett a választásunk a lítium akkumulátorokra, mert könnyűek (ez nagyon fontos tényező a robot szempontjából) és sokkal tartósabbak, mint a nikkel alapúak.

Robot jelenlegi kinézete:



10. ábra



11. ábra

Többnyelvűség – Multilanguage

Minden szoftvernek, mely tömeggyártásra kerül, lényeges részét képezi a többnyelvűség támogatása. Nem elég, ha csupán egy nyelven érhetőek el a program egyes funkciói, gondolnunk kell arra, ha más országokban is sikerül bevezetni. Ennek megfelelően legalább egy világnyelvet – jelen esetben az angolt – is támogatnia kell, a magyar mellett. Emellé nem lenne hátrány, ha német, illetve orosz nyelvet is beépítenénk, és persze ahhoz, hogy ezt megvalósítsuk, már csak az adott nyelvre való fordítás szükséges, mely már nem szoftvertervezési probléma.

Ez a lehetőség azért áll rendelkezésünkre, mert a szoftverünk dinamikus nyelv váltást támogat. A dinamika alapját a WPF⁴ alól elérhető Resource Dictionary komponens biztosítja. Ez a megoldás egyrészt lehetőséget biztosít, hogy dinamikus módon bővíthessük termékünket új nyelvekkel. Ez alatt azt értjük, hogy nem statikus módon, előre definiálva építünk bele nyelveket, hanem külön fájlba emelünk ki minden egyes nyelvet, majd ezeket a fájlokat használjuk a megfelelő rutinokkal, e módon új nyelv hozzáadása során csak egy új fájlal kell bővítenünk a rendszert. A másik lehetőség az előbbiekből következik, mert a nyelvek váltásakor csak a megfelelő nyelvet leíró fájlt kell használnunk, mely igen könnyen cserélhető.

Megoldásunk szükséges feltétele, hogy minden egyes szövegrészlet, melynek állítani kívánjuk a nyelvét, ki legyen emelve string-ként egy Resource Dictionary fájlba. Itt minden egyes objektumot egy kulcs és egy érték jelképez, a kulccsal hivatkozhatunk a szövegerőforrás értékére, magára a szövegre. Az alábbi XAML⁵ kódrészletben látható, hogy a különféle nyelvi erőforrásfájlok hogyan épülnek fel, és láthatjuk, hogy minden fájlban ugyanazon kulcsok találhatók meg, de más és más nyelvű szöveget rendelünk hozzá, ezáltal biztosítva a többnyelvűséget, és cserélhetőséget.

Magyar nyelvi erőforrás fájl részlete, 4. kódrész:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="newvarheader">Új változó</system:String>
    <system:String x:Key="name">Név</system:String>
    <system:String x:Key="add">Hozzáad</system:String>
    <system:String x:Key="newprojectheader">Új Projekt</system:String>
</ResourceDictionary>
```

4. kódrész

⁴ Windows Presentation Foundation

⁵ Extensible Application Markup Language

Angol nyelvi erőforrás fájl részlete, 5. kódrész:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="newvarheader">Variable</system:String>
    <system:String x:Key="name">Name</system:String>
    <system:String x:Key="add">Add</system:String>
    <system:String x:Key="newprojectheader">New Project</system:String>
</ResourceDictionary>
```

5. kódrész

Látható, hogy a nyelvi erőforrás XAML kódban íródik, mely itt nagyon praktikusan alkalmazható. Vörössel aláhúzva láthatjuk a szövegerőforrás azonosítóját, a kulcsot, az angol és magyar fájlrészletben is, illetve zölddel aláhúzva az erőforrás értékét, melyet a kulcs azonosít, és amely majd megjelenik a programunk menüiben, beállításában, stb.

Példaként az előző, 5. kódrészletben leírt erőforrás használatára, a szoftverünk *Új projekt létrehozása* ablakának leíró részéből vett kódrészletet mutatom be, 6. kódrész:

```
Title="{DynamicResource newprojectheader}" Height="300" Width="300">
```

6. kódrész

A Title attribútum itt az ablak címét hivatott beállítani, itt értéknek megadhatnánk egy egyszerű stringet is, azonban kapcsos zárójelbe zárva a DynamicResource kulcsszót használva, és a vörössel aláhúzott kulcsot megadva az erőforrásunkra hivatkozunk, ami dinamikusan változik.

Erőforrások – Resource Dictionary

Látható, hogy az erőforrások használata majdhogynem elengedhetetlen, nemcsak a többnyelvűség biztosításának céljából, de később meg fogjuk látni, hogy a szintaxis felismerésében is nagy szerepet játszik. Épp ezért az alábbiakban ennek a remek komponensnek a tulajdonságairól, és funkcionalitásáról ejtenék néhány szót.

Szoftverünk felépítése során erőforrásokat használunk. Erőforrások alatt itt azt értjük, hogy ha egyfajta komponenst, beállítást többször, illetve több helyen is használunk, akkor ezeket kiemelhetjük egy resource fájlba, majd ezt töltjük be és használjuk, ahelyett, hogy statikusan, a program minden pontján leírnánk ezeket. Ezzel egyrészt sok felesleges kódot spórolunk meg, másrészt dinamikussággal vértézzük fel az egyes funkciókat. Mivel az erőforrásokat egy helyen definiáltuk, ezért ha változtatni kell egy funkción, elég egy helyen átírni azt, és ez által mindenhol érvényesülni fog a változás.

Az erőforrás egy .xaml kiterjesztésű fájl, és XAML kódban íródik. Természetesen bármi lehet erőforrás, amit használni kívánunk. Jelen esetben ezek alatt string, azaz szöveg típust értünk, de lehetnek például stílusok, ami ugyancsak hasznos, ha bonyolult stílust szeretnénk leírni, ebben az esetben már csak azért is érdemes kiemelni erőforrásként, hogy a kódot szeparáljuk, ezzel növelve az átláthatóságot, és az egyszerűbb kódolhatóságot. Akár

összetett, komplex objektumokat is alkalmazhatunk erőforrásként. Minden erőforráshoz tartozik egy kulcs, amely azonosítja, és ami alapján elérhetjük azt, segítségével történik a betöltés, melyet az alábbi kódrészekon tekinthetünk meg:

Az erőforrásokat az App.xaml fájlban kell definiálni. Itt statikusan töltjük be az erőforrást, azaz a program indulásakor ezt használja alapértelmezettként, 7. kódrész:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Languages/Hungarian.xaml" />
            <ResourceDictionary Source="SyntaxKeywords/Hungarian.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

7. kódrész

Amint láthatjuk a MergedDictionaries tulajdonság kihasználásával egyszerre több erőforrást is betölthetünk alapértelmezettként. Egyértelmű, hogy a ResourceDictionary tag-et kell használnunk, illetve a Source attribútumnak az erőforrás fájl projekten belüli elérési útvonalát kell megadni.

Azonban ahhoz, hogy ezek a fájlok, erőforrások dinamikusan cserélhetőek legyenek, szükségünk van C# kódra. Példaképp az 8. kódrész mutatja a hozzárendelést:

```
switch (logolang)
{
    case "langhu":
        App.Current.Resources.MergedDictionaries[1].Source =
            new Uri("SyntaxKeywords/Hungarian.xaml", UriKind.Relative);
        break;
    case "langen":
        App.Current.Resources.MergedDictionaries[1].Source =
            new Uri("SyntaxKeywords/English.xaml", UriKind.Relative);
        break;
}
```

8. kódrész

Itt egy switch elágazás szerkezetben váltunk a nyelvek között, attól függően, hogy a felhasználó mit választott. A betöltés logikailag hasonló a XAML kódban való betöltéshez, a MergedDictionaries tömb az applikációnk erőforrásait tartalmazza. Kiválasztjuk a cserélni kívánt erőforrást a megfelelő indexszel, majd beállítjuk az elérési útvonalát egy Uri objektum segítségével, az elérési útvonalat megadva.

Szintaxis Kiemelés – Syntax Highlight

A fejlesztői nyelv, esetünkben a LOGO szintaxisának kiemelése kulcsfontosságú szerepet kap a szoftverben. Programozók lévén jól tudjuk, hogy nagyban megkönnyíti a kódolást a megfelelő és igényes fejlesztői környezet, ennek része, hogy a kód nem csak szemantikusan átlátható, de könnyen felismerhetőek benne a nyelv kulcsszavai. Ez által a program egyes részei jobban átláthatóak, könnyebben elkülöníthetőek, részben pedig úgy működik, mint egy szintaktikai elemző, mivel csak a helyesen begépelt kulcsszavakat emeli ki.

Fontos megjegyezni, hogy az eddigi LOGO szerkesztők nem rendelkeznek ilyen funkcióval, így sokkal több hibalehetőséget tartalmaznak, nehezebben átláthatóak az itt írt kódok.

Szükségesnek láttuk, hogy többféle nyelven képes legyen a LOGO szintaxisának felismerésére, így megoldásként itt is a Resource Dictionary komponenst alkalmaztuk, illetve a tényleges szövegformázás megvalósításához szükség volt még egy RichTextBox komponensre, mely a későbbiekben kerül bemutatásra. Mivel a szoftver magyar és angol nyelvet támogat jelenleg, a szintaxis kiemelés is ezt a két nyelvet ismeri fel, ez azonban dinamikusan bővíthető, akár csak a nyelvek. Ezen kívül opcionálisan biztosítja a case-sensitivity tulajdonságot, azaz kis- és nagybetű közti különbséget szükség esetén, illetve a személyre szabhatóság kedvéért a szövegkiemelés több színben is lehetséges.

A szintaxis kiemelés alatt valójában a kulcsszavak kiemelését értjük. Minden programozási nyelvben vannak előre definiált kulcsszavak, ezeket különítjük el a kód többi részletétől, például konstansoktól, különleges karakterektől, stb. Dinamikus megoldásunk miatt az elkészített osztályok bármilyen más programozási nyelv kulcsszavainak felismerésére képes. Ehhez használjuk a Resource Dictionary-t. Minden programozási nyelvhez, illetve esetünkben a LOGO nyelv angol és magyar kulcsszavaihoz tartozik egy-egy erőforrás fájl, melyben a nyelvekhez hasonlóan, kulcsokkal azonosítva adhatjuk meg a programozási nyelv kulcsszavait. Ezt a megvalósítást láthatjuk a 9. kódrészben.

```

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="kw_efore">előre</system:String>
    <system:String x:Key="kw_elorer">e</system:String>
    <system:String x:Key="kw_hatra">hátra</system:String>
    <system:String x:Key="kw_hatrar">h</system:String>
    <system:String x:Key="kw_jobbra">jobbra</system:String>
    <system:String x:Key="kw_jobbrar">j</system:String>
    <system:String x:Key="kw_balra">balra</system:String>
    <system:String x:Key="kw_balrar">b</system:String>
</ResourceDictionary>

```

9. kódrész

Látható, hogy az x:Key attribútum az erőforrás kulcsát tartalmazza, mellyel a nyitó és záró XAML tag közé zárt LOGO kulcsszóra hivatkozunk. Az angol LOGO szintaxis ugyanilyen formában adhatjuk meg, egy másik erőforrás fájlban.

Az erőforrásokban definiált szintaxis kulcsszavakat el kell juttatnunk az azt használó programrészekhez ezt már C# kód segítségével oldjuk meg. Mivel szoftverünk LOGO nyelvet használ, illetve azon belül többnyelvű felismerhetőséget is lehetővé tesz, szükségünk van egy string tömbre, mely tartalmazza azokat a kulcsokat, melyeket az előbbi kódrészletben bemutatott erőforrásfájl tartalmaz.

Ennek a string tömbnek egy részlete a következőképpen néz ki (10. kódrész), a kapcsolat átláthatósága érdekében vörössel aláhúzva tekinthető meg az a kulcs, melynek a párja a Resource Dictionary-ben aláhúzva látható:

```

string[] kw_keys = newstring[]
{
    "kw_efore",
    "kw_elorer",
    "kw_hatra",
    "kw_hatrar",
    "kw_jobbra",
    "kw_jobbrar",
    "kw_balra",
    "kw_balrar",
    ...
};

```

10. kódrész

Az itt definiált kulcsok betöltését az alábbi metódus végzi el (11. kódrész):

```

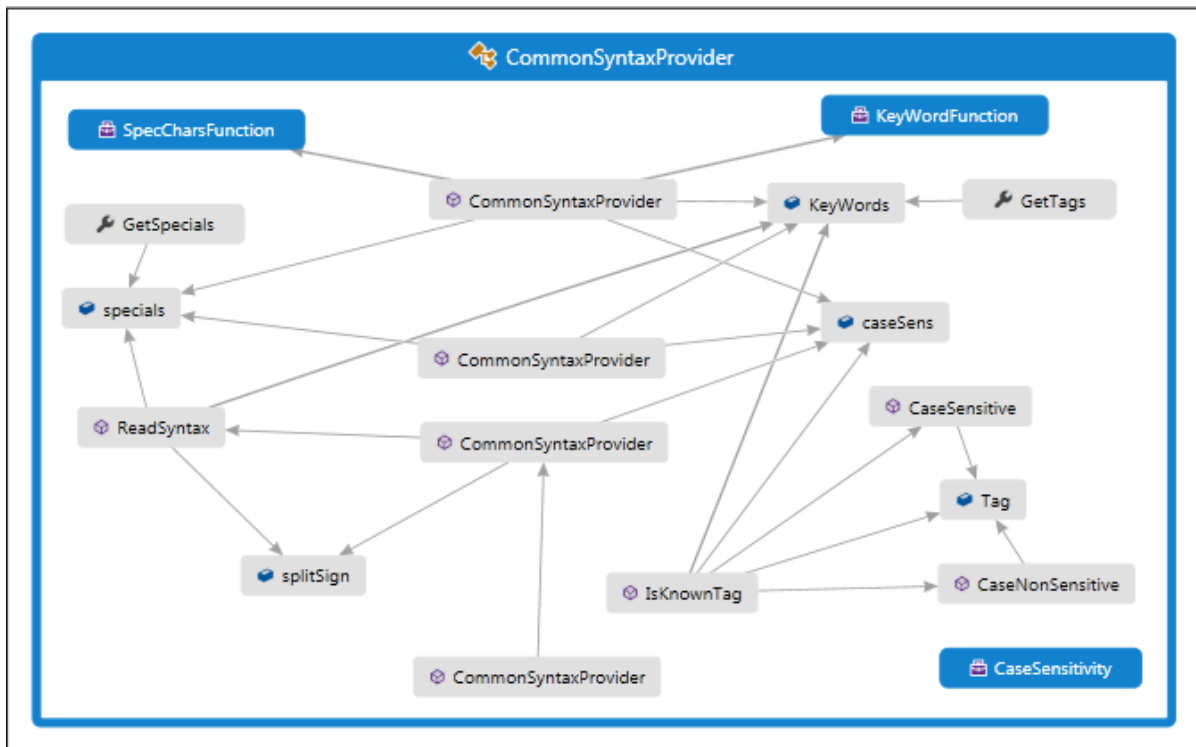
publicvoid UpdateKeywords()
{
    keyWords.Clear();
    for (int i = 0; i < kw_keys.Length; i++)
    {
        keyWords.Add(App.Current.TryFindResource(kw_keys[i]).ToString());
    }
}

```

11. kódrész

Valójában a metódus végigiterál a kulcsokon, és kinyeri az erőforrásban hozzájuk tartozó szintaxis kulcsszavakat, ezeket pedig odaadja annak az osztálynak, mely elvégzi a szintaxis felismerés rutinját. A továbbiakban ezeknek az osztályoknak a bemutatásáról lesz szó.

CommonSyntaxProvider osztály



12. ábra

Az osztály szerkezetét remekül szemlélteti a gráf szerkezettel felépített 12. ábra. Ez a funkció a Visual Studio 2012-es verziójában érhető el, nagyon hasznos, mivel megjeleníti az osztály referenciáit, a kapcsolatokat. Ahogyan a neve is mutatja, ez az osztály biztosítja majd a szintaxist, ez alatt pedig a kulcsszavakat értjük. Ennek tárolására egy *KeyWords* nevű string alapú listát használunk, ebbe kell beolvasnunk a kulcsszavakat, ezen kívül szükséges egy lista a speciális karakterek tárolására, ez a dolga a *specials*, karakter alapú listának.

Középen láthatóak az osztállyal azonos nevű konstruktorok, melyek többféle szintaxis beolvasási lehetőséget adnak. Erre azért van szükség, mert a megfelelő szintaxist beolvashatjuk egy egyszerű szövegfájlból is, azonban ez többnyire tesztelés céljából alkalmas, illetve kisebb projektek esetén célszerű. Ez esetben a szövegfájlból egyenként ki kell olvasnunk a kulcsszavakat, illetve elkülönítve a speciális karaktereket, és elhelyezni a megfelelő listákban, erre szolgál a *ReadSyntax* metódus. Lehetőség van arra, hogy rögtön listából kapjuk meg a kulcsszavakat, ez esetben nincs is szükség semmilyen konverzióra. Ezen belül is két lehetőség van, az egyik konstruktor már kész listát vár paraméterként, a másik, amelyet a programban is alkalmazunk megoldásként, hogy a konstruktor egy callback függvényt vár paraméterként, amely egy listát ad vissza. Ez azért hasznos, mert így a

kulcsszavakat beolvashatjuk Resource Dictionary-ből az osztályon kívül, ezt függvénybe zárhatjuk, és magát a függvényt adjuk paraméterként a konstruktornak. A konstruktor meghívja a függvényt, és felhasználja az eredményt. Így nem szemeteljük tele az osztályt olyan teendővel, ami nem a feladata, és dinamikus is marad a módszer.

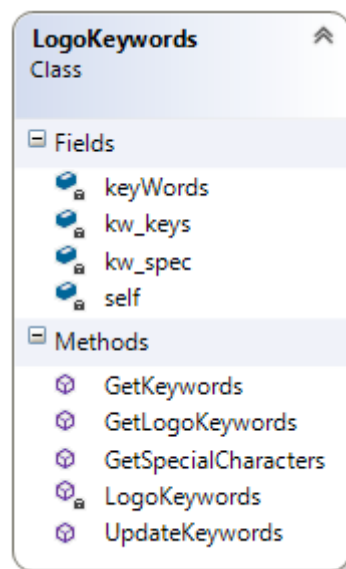
Az osztályban ezen kívül van két metódus, a *CaseSensitive* és *CaseNonSensitive*, amelyeket az *IsKnowTag* függvény használ, mely megmondja az ellenőrzött szóról, hogy kulcsszó-e vagy nem. Értelmszerűen, az egyik függvény figyelembe veszi a kis- és nagybetűk közötti különbséget, a másik nem.

LogoKeywords osztály

Szükség van egy olyan osztályra, mely a LOGO nyelv kulcsszavait kigyűjti az erőforrás fájlokból. Ezt a problémát a *LogoKeywords* osztály (13. ábra) oldja meg. Itt definiáljuk a már korábban bemutatott *kw_keys* tömböt, mely a LOGO szintaxis kulcsszavainak erőforrásbeli kulcsait tartalmazza stringek-ként.

Helyet kap az a két függvény, melyet a *CommonSyntaxProvider* osztály konstruktorának adunk oda, ezekben írjuk meg azt a rutint, mely az erőforrásokból a kulcsok alapján listába gyűjti ki a kulcsszavakat és a speciális karaktereket, így ez a két függvény egy string és egy karakter listát ad vissza.

Amikor a szoftverünkben átállítjuk a szintaxis kiemelésének nyelvét, szükség van a kulcsszavak frissítésére, ezt a feladatot végzi el az *UpdateKeywords* metódus. Az osztály maga is tárolja a kulcsszavakat egy listában.

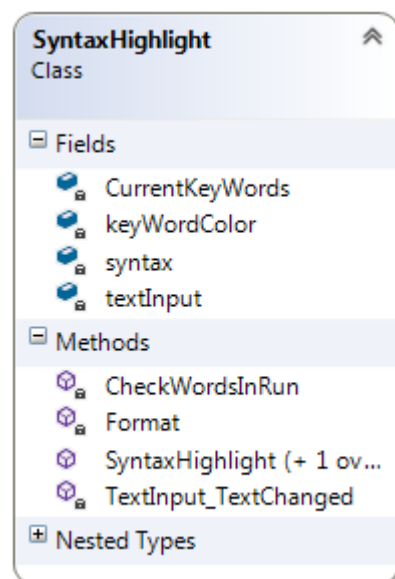


13. ábra

SyntaxHighlight osztály

Ahogy a nevéből is látszik, ez az osztály (14. ábra) építi fel és kezeli valójában a szintaxis felismerését, kiemelését, formázását. A kulcsszavak kiemelését végző folyamat a következő műveletekből áll:

Szükség van egy komponensre, melybe a kódot írhatjuk, ennek pedig rendelkeznie kell a szöveg szerkesztését, formázását biztosító tulajdonságokkal, függvényekkel. Esetünkben ez a *RichTextBox* komponens. A kulcsszavak felismerését természetesen minden egyes karakter beírása után ellenőrizni kell, mert ha a bevitt



14. ábra

karakter egy kulcsszó utolsó karaktere, a kulcsszót már is ki kell emelni. Tehát figyelni kell eseményként a karakterbevitelt, és minden változásra meghívni az ellenőrző rutint. Ekkor ki kell nyernünk a kulcsszavakat, elmentve a pozíciójukkal, majd a kulcsszavakon elvégezni a formázást a megfelelő paraméterekkel.

A *SyntaxHighlight* osztályunknak két konstruktort biztosítottunk, az egyik az alapérzelmezett formázási paramétereket tartalmazza, a másikban opcionálisan állíthatjuk a szövegkiemelés színét. A konstruktornak két kötelező paramétere van, az egyik egy RichTextBox példány, a másik pedig a *CommonSyntaxProvider* osztályunk egy példánya, osztályunk maga is tartalmaz egyet-egyét mindkét típusból, ezáltal aggregációs kapcsolatot biztosítva az osztályok között. Ily módon az osztályunknak lesz egy referenciája arra a komponensre, melyen el kell végezni a megfelelő műveleteket. A RichTextBox komponens TextChanged eseményére feliratkoztatjuk az általunk definiált *TextInput_TextChanged* függvényt, így a referencia által, ha az eredeti komponensnek megváltozik a szövege, rögtön meghívódik ez a függvény, és elvégzi a dolgát:

Üres szöveg esetén nincs teendője. Minden más esetben először meg kell szereznünk a beviteli parancsmező szövegét a RichTextBox példánytól. Ehhez szükségünk van a TextRange osztályra, mely egy bizonyos szövegtartomány tárolására képes. A szöveget ebben a környezetben TextPointer objektumokkal tudjuk kijelölni, méghozzá itt szükség van a tartalom kezdetét és végét jelző pontra, tehát a TextRange konstruktora ezt a két paramétert várva építi fel a szövegtartományt tároló objektumot.

A kinyert szövegen egy „navigátor”, TextPointer típusú objektummal navigálunk végig. Ez úgy történik, hogy ciklikusan megvizsgáljuk, hogy a navigátorunk elérte-e már a szöveg végét, ha igen, akkor megromázzuk a kinyert szavakat. Ha nem, akkor minden szövegrészből létrehozunk egy kontextust a TextPointerContext típus segítségével, úgy, hogy a navigátor által kiolvassuk a tőle visszafelé eső szövegrészt. Leellenőrizzük, hogy így egy számunkra megfelelő kontextust kapunk-e. Tulajdonképpen, ha az adott kontextus kompatibilis a Run típussal – amely egy formázott, vagy formázatlan szövegrész távot reprezentál – akkor erről a Run típussá kényszerített objektumból minden ciklusban kigyűjtjük a kulcsszavakat. Ahogy a kulcsszavak kigyűjtése megtörtént egy kontextusból, a navigátor felveszi a következő kontextus pozícióját.

A kulcsszavak kigyűjtését külön metódus végzi el, mely egy Run típusú objektumot vár paraméterként, mivel ennek az objektumnak már van text tulajdonsága, azaz ebből már lekérdezhető a szöveges tartalma is. Természetesen az objektumból kinyert szövegen kell végigiterálnunk, és ebben kell megkeresnünk a kulcsszavakat, amelyeknek így már tudni fogjuk az indexét is. A kulcsszavak keresése során sok fontos dologra kell odafigyelnünk. Fontos, hogy egy kulcsszót csak akkor emelhetünk ki, ha elszeparálva találjuk a többi kódrésztől, azaz ha előtte és utána vagy white space⁶ karakter áll, vagy speciális karakter. Itt nyer értelmet a *CommonSyntaxProvider* osztály által tartalmazott *specials*, karakter alapú lista, mely ezeket a speciális karaktereket tartalmazza, ide tartoznak például a pont, zárójelek, stb. karakterek. Erre azért van szükség, mert különféle programozási nyelvekben a

⁶ Szóköz, tabulátor, enter, stb.

kifejezéseket gyakran zárjuk zárójelek közé, illetve a pont karakter általában egy példány, vagy osztály tulajdonságainak, metódusainak elérésére szolgáló operátor, ezek a karakterek pedig közvetlenül állhatnak kulcsszavak előtt, illetve után. Így ha ilyen karaktereket találunk, az nem számít egybeolvadásnak a kulcsszóval.

Tehát, ha egy kulcsszót a megfelelő környezetben találtunk meg, fel kell vennünk egy listába. Ez úgy történik, hogy először a megfelelő indexek alapján kiszedjük az elemzés alatt álló szövegből (12. kódrész), ehhez nyilván szükség van a szó kezdő és vég indexére. Meg kell vizsgálnunk, hogy a talált szó valóban a nyelvnek megfelelő kulcsszó-e. Ha igen, beletehetjük a listába.

```
string word = text.Substring(sIndex, eIndex - sIndex + 1);
```

12. kódrész

A lista típusát először definiálnunk kell. Mivel ez az a lista, amely a formázni kívánt kulcsszavakat tárolja majd, szükségünk van egy olyan struktúrára, mely képes tárolni a kulcsszóhoz szükséges információkat. Erre a struct szerkezetet használjuk. Tehát definiálunk egy olyan structot, melynek mezői maga a kulcsszó szövegrésze, ez természetesen egy string mező, illetve szükség van két, TextPointer típusú mezőre, melyek a kulcsszó kezdő és végpontjait reprezentálják (13. kódrész).

```
struct Tag
{
    public TextPointer StartPosition;
    public TextPointer EndPosition;
    public string Word;
}
```

13. kódrész

A listában való elhelyezéshez minden kulcsszóhoz létrehozunk egy struktúrát, melynek kezdő- és végpontját a GetPositionAtOffset függvénnyel állapítjuk meg, a már ismert kezdő- és végpont indexek segítségével, illetve a kulcsszó struktúra szövegrésze az ellenőrzés alatt álló szöveg lesz. Ezután már csak hozzá kell adnunk a listához az újonnan létrehozott struktúrát (14. kódrész).

```
if (syntax.IsKnownTag(word))
{
    Tag t;
    t.StartPosition = run.ContentStart.GetPositionAtOffset(sIndex,
        LogicalDirection.Forward);
    t.EndPosition = run.ContentStart.GetPositionAtOffset(eIndex + 1,
        LogicalDirection.Backward);
    t.Word = word;
    CurrentKeyWords.Add(t);
}
```

14. kódrész

Arra is kell figyelniünk, hogy az ellenőrzött szövegrész utolsó szava is formázás alá essen, ha az kulcsszó, ezért a jelenlegi megoldás mellett ehhez szükség van egy külön rutinra, amely az utolsó szót ellenőrzi.

Eljutottunk oda, hogy készen áll a feltöltött lista a formázásra. Ezzel is külön függvény foglalkozik (15. kódrész):

```
void Format()
{
    textInput.TextChanged -= TextInput_TextChanged;

    TextRange r;
    for (int i = 0; i < CurrentKeyWords.Count; i++)
    {
        r = new TextRange(CurrentKeyWords[i].StartPosition,
            CurrentKeyWords[i].EndPosition);
        r.ApplyPropertyValue(TextElement.ForegroundProperty,
            new SolidColorBrush(keyWordColor));
        r.ApplyPropertyValue(TextElement.FontWeightProperty, FontWeights.Bold);
    }
    CurrentKeyWords.Clear();
    textInput.TextChanged += TextInput_TextChanged;
}
```

15. kódrész

Fontos megemlíteni, hogy amikor formázási műveletet végzünk el egy bizonyos RichTextBox által tartalmazott szövegrészen, az is kiváltja a szöveg megváltozása eseményt, így olyan lenne, mintha formázás közben beleírnánk valamit. Mivel az egész folyamat a kiváltott eseménytől függ, újra meghívódna ez egész folyamat, ez pedig hibát eredményezne. Tehát a formázás idejére leiratkoztatjuk az eseményfigyelést, majd a végén vissza.

A formázás műveletében a már korábban kigyűjtött kulcsszavak listáját iterálunk végig, és egyenként elvégezzük rajtuk a formázási műveleteket.

Ahhoz, hogy kapcsolatot létesítsünk a RichTextBox szövege és a listánk között, egy TextRange objektumra van szükségünk. Mivel a listába elmentett kulcsszavak struktúrája tartalmazza a kulcsszavak eredeti szövegben elfoglalt kezdő- és végpozícióit, ezért ha a TextRange objektum konstruktorát ezzel a két mutatóval paraméterezzük fel, akkor ez az objektum az eredeti szövegbeli kulcsszóra fog mutatni. Majd ezen az objektumon elvégezzük a formázási műveleteket. Ezt a TextRange objektumon keresztül elérhető ApplyPropertyValue függvény segítségével végezhetjük el. A formázást elvégző függvény két paramétert vár, az első paraméter a formázás tulajdonsága, melyeket a TextElement osztályon keresztül érhetünk el, itt állíthatjuk be, hogy például a betűk színét akarjuk állítani. A másik paraméter a formázás értéke, jelen esetben a kiválasztott szín. Szinte mindenféle formázási lehetőség adott, esetünkben a színen kívül félkövérre állítjuk a kulcsszavakat.

Ha az összes kulcsszót megformáztuk a listából, akkor kiürítjük a listát. A formázás végeztével a parancsbeviteli mezőt megvalósító RichTextBox objektum TextChanged eseményére újra feliratkoztatjuk a *TextInput_TextChanged* függvényünket, hogy ezután újra elinduljon a formázási folyamat, ha megváltozik a szöveg.

RichTextBox komponens WPF alapokon

Mindazon helyeken a szoftverben, ahová kódot írhatunk, a beviteli mezőt, illetve a parancssort RichTextBox komponens reprezentálja. Mivel azonban a programot WPF alapokon valósítottuk meg, azt a komponenst kellett használnunk, amit ez a technológia nyújt. Itt azonban rögtön nehézségekbe ütköztünk, mivel ez nagyban eltér a már jól ismert Windows Forms által nyújtott RichTextBox komponenstől.

A legnagyobb eltérés az, hogy az általunk használt kontroll tartalma nem csak egyszerű szöveg lehet, hanem lehetőséget nyújt paragrafusok, képek, táblázatok, természetesen szövegek és egyéb formátumok tárolására, illetve lehetővé teszi a formázásukat. Éppen ezért, mivel felépítése általános tartalom kezelésére képes, sokkal nehezebb is kezelni. Tulajdonképpen minden olyan objektumot képes kezelni, mely kompatibilis a FlowDocument típussal.

Elemeit többféleképpen is felsorolhatjuk, megadhatjuk. Tulajdonképpen definiálhatjuk statikusan, XAML kódban leírva, illetve, ami sokkal hatásosabb, dinamikus módon C# kódban. Programunkban az utóbbi megoldásra volt szükség.

A 16. kódrészben láthatjuk, ahogyan statikus módon, XAML nyelven elemeket adunk a RichTextBox kontrollhoz:

```
<RichTextBox Margin="10,10,0,13" Name="myRichTextBox" HorizontalAlignment="Left" VerticalAlignment="Top" Width="500" Height="300">
  <FlowDocument>
    <Paragraph>
      This is a paragraph, this can be edited.
      <Bold>Go ahead.</Bold>
    </Paragraph>

    <Paragraph Foreground="Blue">
      This paragraph's foreground color had been changed to blue.
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

16. kódrész

Látható, hogy már itt is beállíthatjuk az objektumok formázását, jelen esetben az egyik paragrafusban megjelenő szöveg félkövér lesz, a másikban a szöveg színe kék.

A legtöbb esetben azonban szükséges, hogy e tulajdonságokat dinamikus módon változtathassuk, erre láthatunk példát az alábbi C# nyelvű, 17. kódrészben:

```

private void CreateAndLoadRichTextBox()
{
    FlowDocument mcFlowDoc = new FlowDocument();

    Paragraph para = new Paragraph();
    para.Inlines.Add(new Run("This is a paragraph, this can be edited."));
    para.Inlines.Add(new Bold(new Run("Go ahead.")));

    mcFlowDoc.Blocks.Add(para);

    RichTextBox mcRTB = new RichTextBox();
    mcRTB.Width = 560;
    mcRTB.Height = 280;

    mcRTB.Document = mcFlowDoc;

    ContainerPanel.Children.Add(mcRTB);
}

```

17. kódrész

Először is létrehozuk a saját FlowDocument objektumunkat. Majd létrehozunk egy Paragraph objektumot, és szöveget adunk neki. Ezt egy Run objektumként kapja meg, ennek az osztálynak van text paramétere, mely a szöveget képes tárolni. A szöveget tároló objektumot becsomagolhatjuk egy formázó objektumba, mely esetünkben a félkövérséget állító osztály. A FlowDocument objektumnak van egy, a paragrafusokat tartalmazó Blocks, listát megvalósító tulajdonsága, melyhez hozzáadjuk a létrehozott és szöveggel ellátott paragrafust.

Szükségünk van egy RichTextBox objektumra, melynek a tartalma az előbb létrehozott objektum lesz. Természetesen a létrehozott RichTextBox objektum tulajdonságait itt is beállíthatjuk, például a szélességet és a hosszúságot. Ezen objektum Documents tulajdonsága hivatott tárolni a komponens tartalmát, így odaadjuk ennek a FlowDocument objektumot, ezzel beállítva a szöveget tartalomként. Végül felvesszük a létrehozott RichTextBox kontrollt a programunk megfelelő pontján.

Mivel esetünkben az a legfontosabb, hogy a kód szövegét formázzuk, a RichTextBox objektumunkat képessé kell tenni szöveg bevitelére, kiolvasására, formázására. Az előbbi betöltési forma túl általános, többféle objektum is kezelhető, ezért csak szöveg betöltésére létezik egyszerűbben kezelhető megoldás is. Ebből egyik megoldásként ismertetném azt az esetet, amikor egyszerű szöveget helyezünk el tartalomként, illetve még egy hasznos esetet, amikor egy szövegfájlból töltjük be a szöveget.

Az előbbi esetet a 18. kódrészben láthatjuk. Itt egy System.IO névtérben található MemoryStream objektumot hozunk létre először, átadva neki a megfelelő kódolással ellátott szöveget, majd a RichTextBox komponens már képes fogadni ezt az objektumot:

```

public void SetString(string text, RichTextBox rtb)
{
    MemoryStream stream = new MemoryStream(ASCIIEncoding.UTF8.GetBytes(text));
    rtb.Selection.Load(stream, DataFormats.Text);
}

```

18. kódrész

A másik megoldásban (19. kódrész) az elérési útvonallal megadott fájlból tölthetjük be a szöveget. Ehhez szükség van, az ugyancsak System.IO névtérben helyet kapó FileStream osztályra, mely a szöveg kiolvasását végzi el. Hibakezelési okok miatt leellenőrizzük, hogy az elérési útvonalon megadott fájl valóban létezik-e. Egy TextRange objektum fogja képviselni a szöveg helyét és tartalmát a RichTextBox komponensben. A FileStream objektum által beolvashatjuk a fájlt, a TextRange objektum megfelelő metódusa pedig képes fogadni ezt a streamet⁷, és képes betölteni az így kapott szöveget. Az adatfolyam kapcsolatát le kell zárunk a művelet végén.

```

private void LoadTextDocument(string fileName)
{
    TextRange range;
    FileStream fStream;
    if (File.Exists(fileName))
    {
        range = new TextRange(RichTextBox1.Document.ContentStart,
RichTextBox1.Document.ContentEnd);
        fStream = new FileStream(fileName, FileMode.OpenOrCreate);
        range.Load(fStream, System.Windows.DataFormats.Text );
        fStream.Close();
    }
}

```

19. kódrész

Nem csak a RichTextBox kontrol tartalmát kell tudnunk módosítani, írni bele, képesnek kell lennünk annak kiolvasására is. Sajnos a komponens általános működése miatt nincs beépített metódus, mely vissza tudja adni a szöveget, ezért ezt nekünk kell megkonstruálnunk. Ehhez nem kell mást tennünk mint egy TextRange objektumba betölteni a tartalmat, annak kezdő és végpontja alapján, ezután ez az objektum az aktuális szövegre fog mutatni. A TextRange objektum pedig már rendelkezik string típusú Text tulajdonsággal, melyből így már egyszerűen kinyerhető a szöveg. Ezt a folyamatot írja le a 20. kódrész:

```

string ConvertRichTextBoxContentsToString(RichTextBox rtb)
{
    TextRange textRange = new TextRange(rtb.Document.ContentStart,
rtb.Document.ContentEnd);
    return textRange.Text;
}

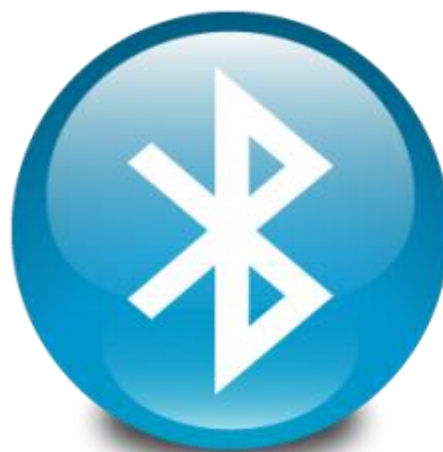
```

20. kódrész

⁷ Egy meghatározott adatfolyam

Kapcsolatteremtés – Bluetooth

Szoftverünket, és a parancsokat végrehajtó robotot össze kellett kötnünk. Evidens volt, hogy vezeték nélküli technológiát kell alkalmaznunk, a könnyebb irányíthatóság, a pontosság és az egyszerűbb kezelhetőség érdekében. Erre a legmegfelelőbbnek a Bluetooth technológiát tartottuk. Felmerült még az infra, illetve a WiFi lehetősége, azonban az előbbi megoldásnak határokat szab a távolság és a közvetlen rálátás szükségessége, az utóbbi kiépítése pedig anyagiasabb megoldás lenne.



15. ábra

Kiválasztáskor mérlegelnünk kellett a távolságot, mely legalább 2-3 méter, a megbízhatóságot, az energiaellátást, az interferenciát, az adatátviteli sebességet, illetve az összeköttetés kiépítésének költségét. Mindezen szempontokat figyelembe véve, egyértelmű volt, hogy a Bluetooth technológia lesz a legjobb megoldás a kapcsolat megteremtésére.

A Bluetooth egy rádiófrekvenciával működő vezeték nélküli, eszközök összekötését, közöttük zajló adatátvitelt és kommunikációt, kialakítható hálózatot (PAN⁸) megvalósító technológia. Full-duplex tulajdonságú, azaz a kommunikáció mindkét irányban, és egy időben történhet, pont-pont, illetve pont-multipont típusú összeköttetést is támogatva. A technológia a 2.402GHz és 2.480GHz közötti rádiófrekvenciás tartományban 79 vivőfrekvenciát használ 1MHz-es csatornaosztással, és frekvenciaugraltatásos módszert alkalmaz, 1600-as másodpercenkénti frekvenciaváltással. Ez azt jelenti, hogy kommunikáció közben egy algoritmus pszeudovéletlen módon frekvenciát váltva, a használható 79 csatorna között váltogatva kommunikál, és mivel egy-egy csatornát csak nagyon rövid ideig használ, így ha eközben olyan csatornára is vált, amelyen már zajlik kommunikáció, az interferencia elhanyagolható mértékű és hatású.

Szabványtól függően változik a rendszer által áthidalható kommunikációs távolság, a leggyakoribb szabvány a Class 2, kb. 10 méter hatótávig használható, más szabványban erősítéssel akár 100 méterig is lehetséges a jelátvitel. Ennek távolsága és erőssége függ a közegtől is. Maximális átviteli sebessége 24 Mbit/s, és igencsak energiagazdaságos.

A kommunikáció felépítése két Bluetooth technológiát alkalmazó eszköz között párosítással jön létre. Mielőtt megkezdjük a kommunikációt két eszköz között, párosítanunk kell őket.

⁸ Personal Area Network

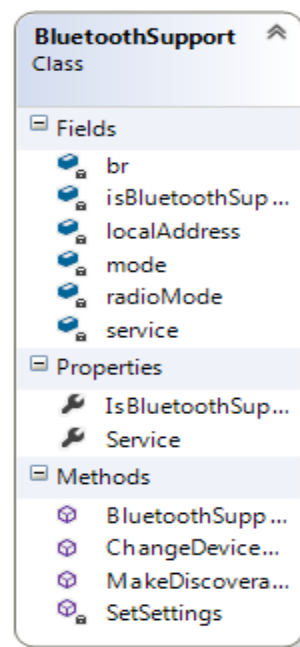
Ahhoz, hogy a szoftverünk Bluetooth-on, keresztül tudjon kommunikálni az Arduino egységgel, szükség van egy olyan API⁹-ra, mely képes kezelni ezt a technológiát, méghozzá .NET-es környezetben. Szerencsére 32feet.NET projekt támogatja a Bluetooth kapcsolat felépítését, a kommunikációt, és megfelelő interfészt biztosít a kezelésére. Ehhez nyilván szükség van egy-egy bluetooth adapterre is, az adó illetve a vevő félnél is. Szükségünk van két segédosztályra, melyek a kapcsolat kezelésére szolgálnak.

BluetoothSupport osztály

Az 16. ábra mutatja ezen osztálynak a szerkezetét, mely arra szolgál, hogy általa a saját eszközünk Bluetooth beállításait számon tartsuk, illetve módosítsuk azt.

Osztályunk tartalmaz egy BluetoothRadio példányt, mely a Bluetooth adaptert azonosítja, amelyen keresztül kommunikálunk. A RadioMode példány segítségével állíthatjuk be, hogy az eszközünk éppen milyen állapotban van. Megkülönböztetünk felfedezhető és kikapcsolt módot, illetve alapállapotban a kapcsolódni képes mód, mely a már párosított eszközökhöz való kapcsolódást teszi lehetővé.

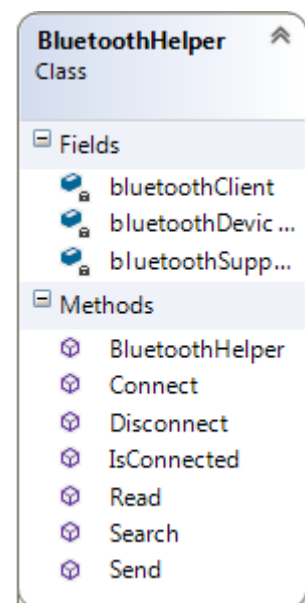
Ha egy eszköz még nincs párosítva, az csak akkor tud csatlakozni hozzánk, ha az állapot felfedezhető módban van. Párosítás után már nem szükséges a felfedezhető mód a kommunikációhoz. Ezen kívül string-ként tároljuk a BluetoothRadio példány címét is, a *localAddress* mezőben, a könnyebb elérhetőség érdekében. Szükség van arra, hogy tudjuk, az eszközünk támogatja-e a Bluetooth technológiát, a logikai típusú *isBluetoothSupported* mezővel ezt jelezzük. A különféle állapotok közötti váltásra a *ChangeDeviceMode* metódus szolgál.



16. ábra

BluetoothHelper osztály

A BluetoothHelper osztály (17. ábra) szolgál az egész kapcsolat létrehozásának, fenntartásának, és a kommunikáció lebonyolítására. Alapvető jelentőségű a három mező, egy BluetoothSupport példány, mellyel saját eszközünk paramétereit állíthatjuk, például a láthatóságot, egy BluetoothClient példány, melyen keresztül zajlik az egész kommunikáció, és egy BluetoothDeviceInfo típusú tömb, mely a felderített eszközöket, és a hozzájuk tartozó fontos információkat fogja tárolni.



17. ábra

⁹ Application Programming Interface

A kapcsolat első lépése a felderítés. Erre való a *Search* függvény. Ez lényegében felfedezhetőre állítja az eszközünket, és a kliensen keresztül felderíti a közelben lévő Bluetooth-t támogató eszközöket, és felveszi a tömbbe.

Ahhoz, hogy kommunikálni tudjunk egy eszközzel, kapcsolódnunk kell hozzá. Ehhez kapcsolódni képes módban kell lennie az eszközünknek. A kapcsolódáshoz tudnunk kell annak az eszköznek a címét, amihez kapcsolódni akarunk, ezt egy *BluetoothAddress* objektum kezeli, és a kapcsolódni kívánt eszközt jellemző *BluetoothDeviceInfo* objektum *DeviceAddress* paraméterével olvasható ki, melyet a megfelelő konverzióval alakíthatunk címmé. A kapcsolódás a kliensen keresztül, a *Connect* metódus segítségével valósul meg. A kapcsolódást végző rutin a 21. kódrészleten látható:

```
public void Connect(BluetoothDeviceInfo btDevice)
{
    bluetoothSupport.ChangeDeviceMode("Connectable");
    BluetoothAddress btAddress =
    BluetoothAddress.Parse(btDevice.DeviceAddress.ToString());
    if (BluetoothSecurity.PairRequest(btAddress, "1234"))
    {
        try
        {
            bluetoothClient.Connect(new BluetoothEndPoint(btAddress,
            bluetoothSupport.Service));
        }
        catch { }
    }
    else
    {
        throw new Exception(App.Current.TryFindResource("notconnect").ToString());
    }
}
```

21. kódrész

Ahhoz, hogy a program által előállított utasításokat az Arduino egység megkapja, szükség van egy küldő rutinra is, mely egy adott szöveget képes kódolva elküldeni. Erre egy logikai visszatérésű függvény a legalkalmasabb, mert így azt is ellenőrizhetjük a hívás helyén, hogy sikerült-e elküldeni az adatot. Csak akkor tudunk adatot küldeni, ha a kliensünk be van állítva, és a kapcsolat is szükséges, különben nem lenne kinek küldeni az adatokat. Egy *Stream* objektum segítségével kapcsolatot teremtünk a fogadó és a szoftverünk közt. A szöveget szét kell darabolunk byte-okra, mely alatt azt értjük, hogy a szöveg minden karakterének ASCII kódját beletöltjük egy *Byte* tömbbe, és a létrehozott *Stream* segítségével ezt az adatstruktúrát tudjuk elküldeni a másik félnek. A 22. kódrészleten látható a küldés művelete:

```

public bool Send(string text)
{
    if (bluetoothClient != null && bluetoothClient.Connected)
    {
        Stream sendStream = bluetoothClient.GetStream();
        Byte[] buffer = Encoding.ASCII.GetBytes(text);
        sendStream.Write(buffer, 0, buffer.Length);
        return true;
    }
    return false;
}

```

22. kódrész

A kommunikáció megvalósulásához nem elég a küldés, mivel a robot küld vissza jeleket a végrehajtott műveletek sikerességéről illetve sikertelenségéről, ezeket tudnunk kell fogadni és feldolgozni. Ehhez egy string típust visszaadó függvényre lesz szükség, mely elvégzi a visszaküldött adatok kiolvasását és dekódolását. Egy string típusú változó kezdeti értékét hibaüzenetre állítjuk, így abban az esetben, ha a kiolvasás sikertelen, ezt a hibaüzenetet adja vissza a függvény, egyébként pedig a kiolvasott adatot, szöveg formában. Nyilván az olvasás is csak az adatküldés részben meghatározott feltételek teljesülése esetén működik, beállított kliens és kapcsolat szükséges. A két művelet nagyban hasonlít egymáshoz, olvasáshoz is egy Stream típusú objektum biztosítja a kapcsolatot, szükséges egy elégséges méretű Byte típusú tömb, mely fogadni képes a kiolvasott adatokat. A Stream objektumon keresztül beolvassuk az adatokat a tömbbe, majd visszaalakítjuk az ASCII kódjaik alapján stringekké. Ha sikeres a művelet, ezzel a szöveggel tér vissza a függvény. A 23. kódrészleten a beérkezett adatok kiolvasását megvalósító függvény látható:

```

public string Read()
{
    string s = "Read error";
    if (bluetoothClient != null && bluetoothClient.Connected)
    {
        Stream readStream = bluetoothClient.GetStream();
        Byte[] buffer = new Byte[1000];
        readStream.Read(buffer, 0, buffer.Length);
        s = Encoding.ASCII.GetString(buffer);
    }
    return s;
}

```

23. kódrész

A szimulátor

Ebben a fejezetben a projekt szerves részét képező RoboLOGO szimulátort fogjuk bemutatni, és az általa nyújtott lehetőségeket, azonban előtte részleteznénk, hogy mi is volt a cél megalkotásánál, illetve, hogy hogyan is épül fel. A legfontosabb szempont, melyet szem előtt tartottunk, a hasonló környezet kialakításának lehetősége volt, hiszen a fiatal generáció az Imagine Logo / Comenius Logo által nyújtott környezettel már megbarátkozott, pontosan tudja, hogyan működik, mire számíthat, ezzel egy ismerős felület tárul eléjük.

A szimulátor tervezésénél ez fontos támpont volt, hiszen ha a felhasználó egy olyan felülettel találkozik, amely számára megszokott és nem kell újratanulnia a parancsokat, a működést, a funkciók eredményeit, a menüpontok feltárását, akkor az nagyban elősegíti, hogy az általa megvalósítani kívánt projekt hamarabb elkészüljön. Pontosán ezen célból terveztük meg úgy az IDE-t, illetve a szimulátort, hogy ezt az érzést nyújtsa a felhasználó számára.

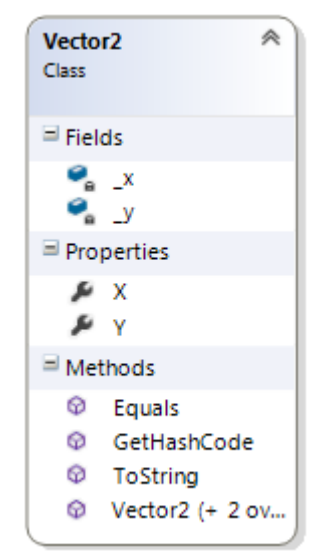
A szimulátor gyakorlati célja abban valósul meg, hogy mielőtt a megírt kódot a valóságban, a RoboLOGO-n alkalmaznánk, azelőtt kipróbáljuk, hogyan is fog ez megvalósulni az IDE keretein belül, ezzel egy olyan animációt kapva, mely szemlélteti számunkra, hogy az elkészített kód az elvárásainknak megfelel-e, szeretnénk-e rajta javítani, apró módosításokat elvégezni, kiegészíteni. Továbbá ez tökéletes lehetőséget nyújt a debugolásra is, hiszen kiszűrhetjük a szintaktikai, illetve szemantikai hibákat is, ha a program nem azt az eredményt produkálja, ami mi elvárunk tőle.

A szimulátor segítségével egy tesztelő környezetet kapunk magunk elé, így a programot csak akkor lesz érdemes elküldeni a robotnak, ha az már megfelelően működik, mindenféle anomáliától megszabadítva. Tőlünk pedig megszabadítja azt az időt, melyet a robottal való teszteléssel töltenénk, minden próbálkozás, minden kódmódosítás újabb és újabb feladatot jelentene a robotnak, mely lassítaná a munkánkat az előrehaladásban.

A szimulátort képező C# osztályok a Vector2, CanvasSize, Pen, Turtle és a TurtleImage. Ezen osztályok nyújtják az IDE számára azt a felületet és funkcionalitást, mellyel a teknős animálva lesz és a kiadott parancsokat végrehajtja.

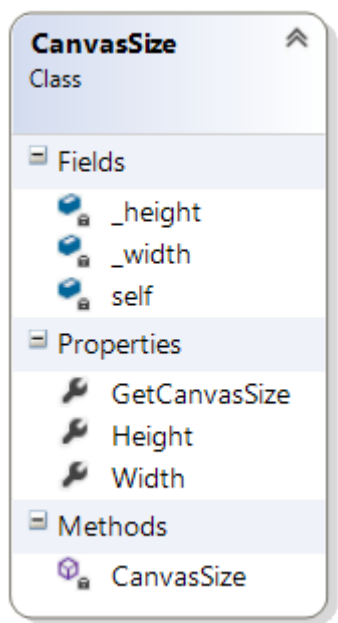
Először a Vector 2 osztályt szeretnénk részletesebben bemutatni, mely a szimuláció alapját képezi. Ez az osztály egy példányával

fogjuk meghatározni és nyomon követni, hogy a teknős jelenleg hol tartózkodik és az animációhoz is nélkülözhetetlen, ennek következtében szükségünk volt egy X és Y koordinátára, amely ezen értékeket eltárolja. Ezt a



18. ábra

2 mezőt property-ként (továbbiakban tulajdonság) tároljuk, így érhető el a többi felület számára. A példány létrehozásához 3 konstruktort definiáltunk, ahol az egyik esetben nincs paramétere, amit át kellene adnunk, ezzel egy teljesen üres példány jön létre, a második esetben a 2 koordinátát adhatjuk át paraméterként, a harmadik esetben pedig az osztály egy példányával inicializálhatjuk a pozíciót. Ennek következtében felülírtuk az Object osztálytól örökölt ToString() metódust, visszaadva a felhasználó számára az X és Y koordinátákat, továbbá az Equals(object x) metódust is, ahol már az imént említett koordinátákat hasonlítjuk össze az egyezés eldöntésének érdekében.



19. ábra

A CanvasSize osztály fogja a „játsszótér” szerepét betölteni, ugyanis a rajzvászon fogja tartalmazni a teknőst, itt fog a mozgatás, az animáció, az összes elküldött parancs végrehajtódni. Egy új projekt létrehozásakor a felhasználónak meg kell adnia a szükséges adatokat, hogy a rajzvászont létrehozzuk, mint például a szélessége, magassága, illetve figyelni kell, hogy a teknős ne léphesse át ezeket a határokat. Úgy kell a játsszótérre tekinteni, mintha fallal lenne körbekerítve. Ezáltal az osztály tartalmaz egy Height, mint magasság és Width, mint szélesség nevű tulajdonságot, s ezek értékeinél ellenőrizzük azt, hogy a megadott értékek pozitívak legyenek. Az osztály 1 konstruktossal rendelkezik, ahol ez előbb említett 2 értéket beállítjuk egy alapértelmezett méretre, ami a mi esetünkben 800*600. Ha a megadott méret meghaladja az IDE méretet, természetesen görgetősávot biztosítunk vízszintesen, illetve függőlegesen is attól függően, hogy melyikre van szükség.

„Gyakori feladat, hogy egy osztályt úgy kell megírunk, hogy csak egy példány lehet belőle. Ez nem okoz gondot, ha jól ismerjük az objektum orientált programozás alapelveit. Tudjuk, hogy az osztályból példányt a konstruktorával készíthetünk. Ha van publikus konstruktor az osztályban, akkor akárhány példány készíthető belőle. Tehát publikus konstruktora nem lehet az egykének. De ha nincs konstruktor, akkor nincs példány, amin keresztül hívhatnánk a metódusait. A megoldás az osztály szintű metódusok. Ezeket akkor is lehet hívni, ha nincs példány. Az egykének van egy osztály szintű szerezPéldány (angolul: getInstance) metódusa, ami mindenkinek ugyanazt a példányt adja vissza. Természetesen ezt a példányt is létre kell hozni, de a privát konstruktort a szerezPéldány hívhatja, hiszen ő is az egyke osztály része” – Dr. Kusper Gábor

```

static CanvasSize self;

public static CanvasSize GetCanvasSize
{
    get
    {
        if (self == null) self = new CanvasSize();
        return self;
    }
}

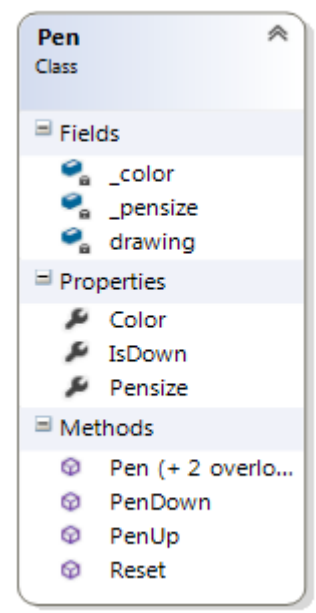
```

A következőkben a Pen osztályt (21. ábra) fogjuk bemutatni, mely azt az utat reprezentálja, amit a teknős a kiadott parancsok alapján már megtett. A Pen osztály létrehozása nélkül nem tudnánk követni és megfigyelni, hogy a teknőssel mit rajzoltunk, mi a programunk végeredménye.

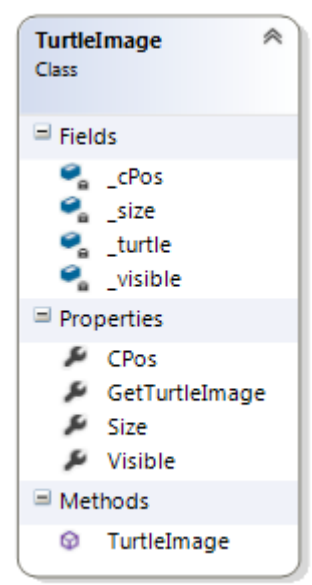
Az osztály 2 tulajdonsággal rendelkezik, melyek a Color, mint a toll színe, illetve a PenSize, mely a toll méretét hivatott biztosítani. Mivel ezen tulajdonságok rendelkeznek setterrel, ezért futás közben is tetszés szerint tudjuk módosítani akár a színt, akár a vastagságot, azonban az értékeket, csak úgy, mint a CanvasSize esetében, itt is ellenőrizzük, és csak a megfelelő értékeket fogadjuk el.

3 konstruktor adott, csak úgy, mint a Vector2 esetében, az első lehetőségnél egy üres példányt kapunk előre beállított alapértelmezett értékekkel, a második lehetőség, hogy mi magunk adjuk meg a toll színét, a harmadik lehetőség pedig, hogy a toll színét és vastagságát is rögtön beállítjuk.

Továbbá az objektum rendelkezik 3 metódussal is, melyeket most fogunk részletezni. A PenUp() látja el azon funkciót, hogy a teknős felemeli a tollat, így nem fogjuk reprezentálni, hogy milyen utat tett meg a közben kiadott utasítások során. A PenDown() az előbbi eljárás ellentéte, hiszen itt pontosan az a célunk, hogy megmutassuk, merre tart a teknős, milyen utat jár be, milyen alakzatot fogunk elkészíteni. A Reset() metódust hívja az összes konstruktor, itt kapják meg az alapértelmezett értékeket a tulajdonságok, ha azokat mi nem állítjuk be. Az osztály működését segíti egy logikai változó, melyet drawing névvel láttunk el, a felületen pedig IsDown tulajdonsággal hivatkozhatunk rá, ezen érték segít nekünk abban, hogy



20. ábra



21. ábra

meghatározzuk, a teknős tolla milyen állapotot vesz fel.

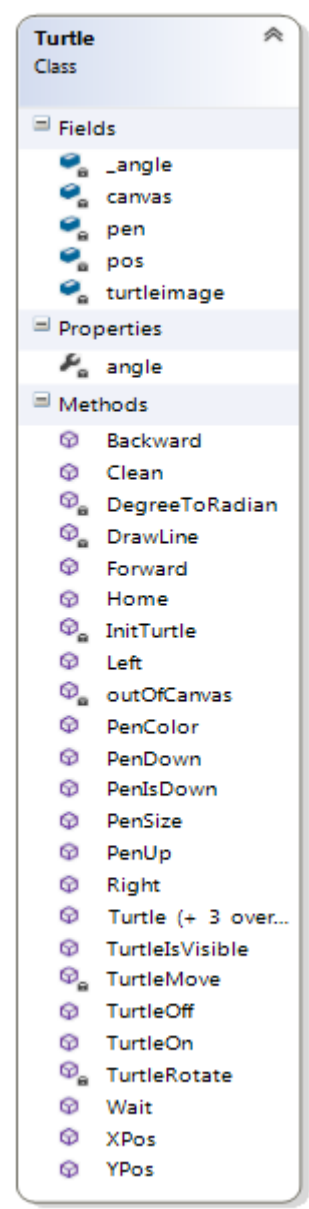
A 20. ábra mutatja a TurtleImage osztályt, mely a teknőst reprezentáló képet valósítja meg a következőképpen. A megjelenítéshez szükségünk van egy olyan tulajdonságokra, mint például a Size, melynek segítségével meghatározzuk a teknős méretét (szélességét és magasságát egyiránt) (alapértelmezett értéke 40), Visible, mely egy logikai változóval segíti azt eldönteni számunkra, hogy a teknőst megjelenítsük-e a játszótéren vagy sem, CPos, mely a Canvas pozíció rövidítése, itt tároljuk el, hogy a teknős a rajzvászon hányadik gyerekeleme, hiszen a későbbiekben erre szükségünk lesz, ha hivatkozni szeretnénk rá a kód bármely pontján.

Továbbá rendelkezik az osztály egy Image példánnyal is, amit a System.Windows.Media.Imaging névtér felhasználásával értünk el. Miután példányosítjuk az Image osztályt, a korábban említett méret (Size) tulajdonságot hozzárendeljük a kép szélesség és magasság tulajdonágaihoz, és a láthatóságot (Visible) igazra állítjuk, mindezt a konstruktorban hajtjuk végre.

Ezzel a konstruktorral alkalmazzuk a Turtle osztályban (22. ábra) az objektum összetételt, vagyis a TurtleImage osztály egy példányát mezőként tároljuk. Ahogyan a GOF2 is megfogalmazza: „Használj objektum összetételt öröklés helyett, ha csak lehet!”. Objektum összetétel esetén HAS-A kapcsolat alakul ki, vagyis egy osztály rendelkezik egy másik osztály egy példányával. Fontos megjegyezni, hogy az öröklődés mindig kiváltható objektum összetétel alkalmazásával.

Az objektum összetétel fajtái az aggregáció és a kompozíció. Egy egyszerű példán fogjuk ezt most szemléltetni, miszerint az előbbi esetben, ha meghal a gitáros, nem temetik vele a gitárját, utóbbi esetben, pedig ha meghal a gitáros, vele temetik a gitárját, ahol a gitár és a gitáros 1-1 objektumnak felel meg. Továbbá a Turtle osztály rendelkezik canvas mezővel, mely egy CanvasSize objektum, hiszen ide lesz felhelyezve a teknős, a Pen osztály egy példányával, amelyet korábban már bemutattunk, illetve egy angle mezővel, mely a teknős aktuális szögét hivatott reprezentálni.

4 konstruktor ad lehetőséget a példány létrehozására, első esetben csak a rajzvászont szolgáltatjuk paraméterként, a további értékek (toll színe, s annak vastagsága) alapértelmezettek, második esetben a játszótér és a toll színét is megadhatjuk, ekkora a vastagság kap alapértelmezett módon értéket, harmadik esetben játszótér és toll vastagság kombinációt választhatjuk, az utolsó esetben pedig a most említett mezők mindegyikét



22. ábra

kötelesek vagyunk megadni a paraméterlistában. Metódusok tekintetében bőséges a választási lehetőség, legelőször az InitTurtle()-t szeretnénk bemutatni, melyet a konstruktorok hívnak meg. Itt inicializáljuk a megkapott paraméter értékek által az

```
//Vonalat rajzol
private void DrawLine(int distance)
{
    Vector2 newpos = new Vector2();
    newpos.X = pos.X + (distance * Math.Cos(DegreeToRadian(angle)));
    newpos.Y = pos.Y + (distance * Math.Sin(DegreeToRadian(angle)));

    outOfCanvas(newpos);

    if (pen.IsDown)
    {
        Line line = new Line();
        line.Stroke = new SolidColorBrush(pen.Color);

        line.X1 = pos.X;
        line.X2 = newpos.X;
        line.Y1 = pos.Y;
        line.Y2 = newpos.Y;

        line.StrokeThickness = pen.Pensize;
        canvas.Children.Add(line);
    }

    pos = newpos;
    TurtleMove(pos);
}
```

23. ábra

objektumokat, helyezzük fel a teknőst a canvasra, és küldjük 'haza' a teknőst.

A 23. ábra mutatja a DrawLine() metódust, mely nevéből adódóan vonalat fog húzni, mégpedig a korábban bemutatott Pen osztály mezőinek segítségével, paraméterként megkapjuk azt a távolságot, melyet meg kell tenni, hogy elérjük az úti célt, és ennek segítségével számoljuk ki az új pozíciót is. Attól függően, hogy a teknős tolla le van-e eresztve vagy sem, vonalat húzunk vagy pedig csak a koordinátákat változtatjuk az újonnan kiszámított értékekre. Az eljárásban szerepel egy OutOfCanvas() metódushívás, mely ellenőrzi, hogy az új koordináták meghaladják-e a játszótér által szabott határokat, szélesség és magasság tekintetében, s ennek megfelelően nem engedjük, hogy a teknős letérjen a rajzvászorról.

A 24. ábra lévő TurtleMove() számára átadjuk az aktuális pozíciót, majd annak függvényében, hogy a teknős látható-e, a Canvas SetTop és SetLeft eljárásaival az új helyre mozgatjuk, ahol az első paraméter a teknősre való hivatkozást mutatja, a második pedig azt, hogy a középpontból kiindulva fogjuk elmozgatni.

```
//Teknőst mozgatja a Canvason
private void TurtleMove(Vector2 turtlePos)
{
    if (turtleimage.Visible)
    {
        Canvas.SetTop(canvas.Children[turtleimage.CPos], turtlePos.Y - turtleimage.Size / 2);
        Canvas.SetLeft(canvas.Children[turtleimage.CPos], turtlePos.X - turtleimage.Size / 2);
    }
}
```

24. ábra

```
//Teknőst forgatja a Canvason
private void TurtleRotate(int angle)
{
    if (turtleimage.Visible)
    {
        RotateTransform rotateTransform1 = new RotateTransform(angle - 90, turtleimage.Size / 2, turtleimage.Size / 2);
        canvas.Children[turtleimage.CPos].RenderTransform = rotateTransform1;
    }
}

//Balra fordítja a tekőst 'angle' fokkal
public void Left(int angle)
{
    this.angle -= angle;
    TurtleRotate(this.angle);
}

//Jobbra fordítja a tekőst 'angle' fokkal
public void Right(int angle)
{
    this.angle += angle;
    TurtleRotate(this.angle);
}
```

25. ábra

A 25. ábra mutatja a TurtleRotate() metódust, mely esetében hasonlóan járunk el, azonban itt a WPF által biztosított RotateTransform osztálynak egy példányát használjuk fel annak érdekében, hogy a teknőst elforgassuk a paraméterben megadott szögben, majd ezt a transzformációt hozzárendeljük a rajzvaszon megfelelő eleméhez, ami mi esetünkben maga a teknős. Az imént említett TurtleRotate()-et hívjuk megfelelően paraméterezve a Left és Right, vagyis a balra és jobbra forgatás megvalósításához azután, hogy a fordulás szögét frissítettük.

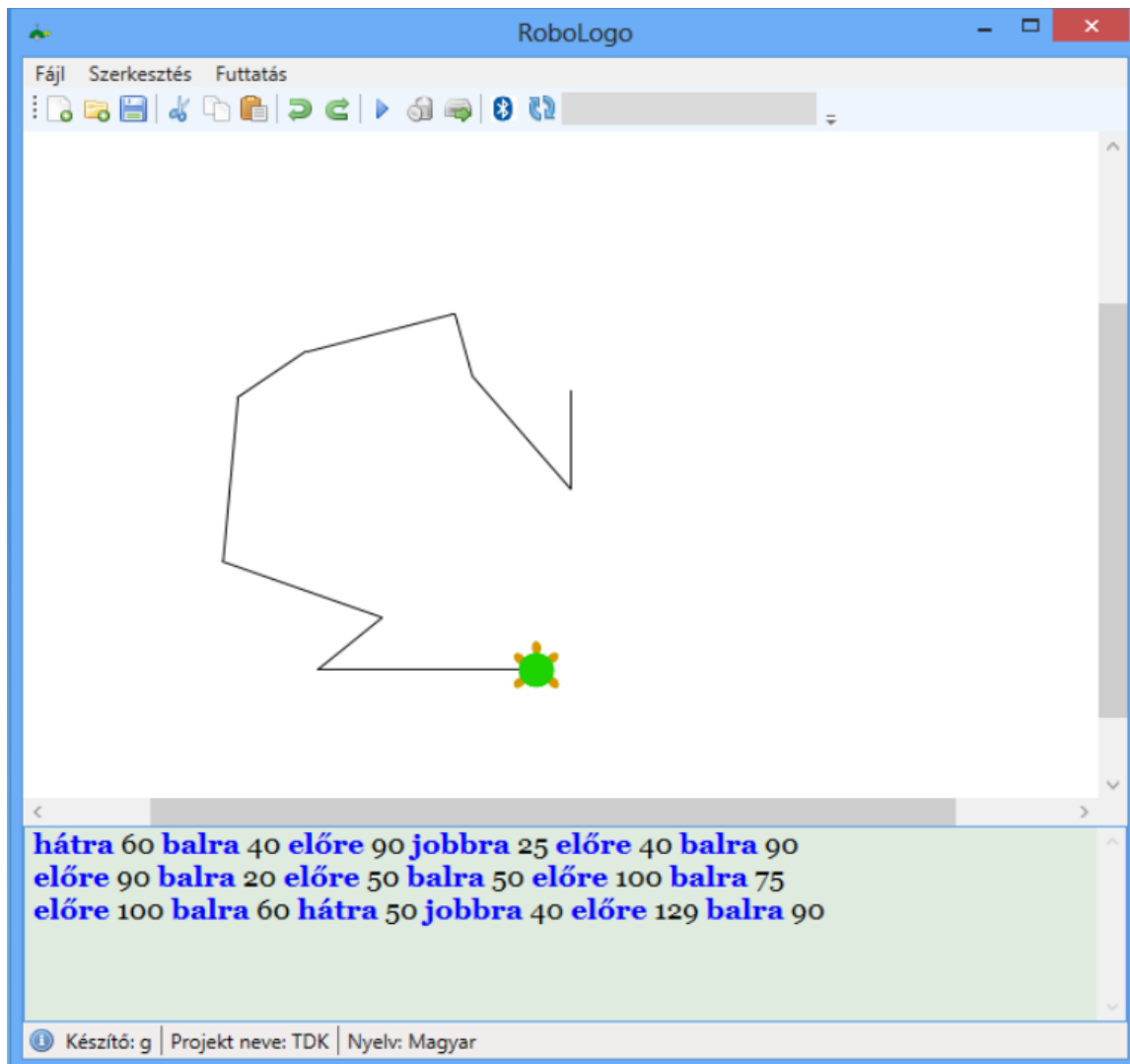
A következő módszer a `Clean()`, mely törli a rajzvászon összes gyermekét, hozzáadja a teknőst, s azt alaphelyzetbe állítja a `Home()` meghívásával, ahol az alapértelmezett értékeket rendeljük az X és Y koordinátákhoz, felhasználva a teknős szélességét és magasságát, és a már korábban részletezett mozgás és forgatás eljárásokat meghívjuk, hogy a 'hazatérés'-t befejezzük. A továbbiakban felsorolt módszerek kiegészítő funkcionálisitást látnak el. Ilyen például az `XPos()`, mely visszatér az aktuális pozíció X koordinátájával, az `YPos()`, mely ugyanezt hajtja végre az Y koordinátára, a `DegreeToRadian()`, mely egy fokból radiánt készít, a `PenUp()` és `PenDown()`, `PenIsDown()`, melyeket a toll osztálynál már ismertettünk. A `TurtleOff()`, `TurtleOn()`, `TurtleIsVisible()`, ahol a teknős láthatóságát kezelhetjük és kérdezhetjük le, illetve a `Wait()`, ahol adott ezredmásodpercet várakozhatunk, ez szerves részét képezi az animáció megvalósításának a későbbiekben.

A 26. ábra mutatja a `Draw()` módszert, melynek már a főprogramban van a helye, láthatjuk, hogy a `Command` osztály egy példányát kapja paraméterként, s ennek a példánynak a `comm` mezőjének értékétől függően döntjük el, - mely egy enum, azaz felsorolás - hogy a teknős melyik irányba haladjon tovább, vagy, hogy letöröljük-e az eddigi alkotásunkat, esetleg fel vagy leeresztjük a tollat.

```
private void Draw(Command com)
{
    switch (com.comm)
    {
        case Commands.elore: turtle.Forward((int)com.value); break;
        case Commands.hatra: turtle.Backward((int)com.value); break;
        case Commands.jobbra: turtle.Right((int)com.value); break;
        case Commands.balra: turtle.Left((int)com.value); break;
        case Commands.haza: turtle.Home(); break;
        case Commands.torol: turtle.Clean(); break;
        case Commands.tollatle: turtle.PenDown(); break;
        case Commands.tollatfel: turtle.PenUp(); break;
    }
}
```

26. ábra

A 27. ábra mutatja az animáció befejeztével az eredményt.



27. ábra

A 28. ábra mutatja azt a forráskódot, mely a szimulátor legfontosabb részét reprezentálja, az animációt, mely a .NET 4.5 és a C# 5.0 által biztosított aszinkron metódus hívás segítségével lett megvalósítva.

A modern alkalmazások fejlesztésénél egy bizonyos ponton mindenképpen elérjük a szinkron módon futó kód korlátait – ekkor kerülünk szembe az aszinkron és eseményvezérelt programozás nehézségeivel. A megszokottól eltérő módszereket kell használnunk a koordinációra, a kivételkezelésre, és felvetődik a megszakíthatóság, valamint a párhuzamosan futó aszinkron feladatok szinkronizációjának problémája is.

```

//futtatás
private async void RunClick(object sender, RoutedEventArgs e)
{
    runButton.IsEnabled = false;
    menu.Save(rtbhelper.GetString(commandLine));
    turtle.Clean();
    turtle.PenDown();
    RoboPreter rp = new RoboPreter();
    try
    {
        List<Robopreter.Command> com = rp.Run(menu.GetFullSource());
        for (int i = 0; i < com.Count; i++)
        {
            Draw(com[i]);
            int time = GetTime(com[i]);
            await Task.Factory.StartNew(() => Wait(time));
        }
    }
    catch (RPException rpe)
    {
        MessageBox.Show(rpe.NewMessage);
    }
    catch
    {
        MessageBox.Show(App.Current.TryFindResource("error").ToString());
    }
    runButton.IsEnabled = true;
}

```

28. ábra

Az async kulcsszót csakis olyan metódusok/függvények esetében használhatjuk, ahol a visszatérési érték típusa void, Task vagy Task<T> (néhány kivétellel, például a Main-en nem lehet). Az async meglepte nem kötelez minket az await használatára, ez fordítva viszont nem igaz. Az await utasítást használhatjuk minden olyan függvényen, amelynek visszatérési típusa Task vagy Task<T>, illetve ha megvalósítja a függvényt tartalmazó osztály a GetAwaiter metódust. A .NET 4.5 beépítve tartalmaz olyan műveleteket, amelyek használhatóak ezzel az utasítással, például a különböző Stream osztályok több metódusa is ilyen. Jellemzően Async utótagot kapnak, illetve az IntelliSense a függvény/metódus paraméterlistája mellett megjeleníti az awaitable szót is.

A metódus szignatúrája elé helyezett async szó azt jelzi a fordítónak, hogy a blokkjában valahol használni szeretnénk az await utasítást. Amikor ahhoz a bizonyos helyhez ér a program futása, az történik, hogy abban a pillanatban a metódus futása megszakad, és háttérbe vonul, amíg az adott művelet véget nem ér.

Azután felébred és befejezi a feladatát.

További tervek, megvalósításra váró funkciók

Autocompletion lehetőség – Automatikus kódkiegészítés

A könnyebb és kényelmesebb kódolhatóság érdekében felmerült az automatikus kódkiegészítés funkció beépítésének lehetősége, azaz a kód gépelése közben a szoftver megjeleníti, illetve opcionálisan kiegészíti a lehetséges kulcsszavakat, illetve szükség szerint a deklarált változókat, objektumokat, függvényeket, stb.

Ez a funkció igen hasznos, mivel nagymértékben képes meggyorsítani a kód írásának folyamatát. Az ötletet főleg a Microsoft Visual Studio kódkiegészítő funkciója, az IntelliSense ihlette, mivel ezt a fejlesztői környezetet használjuk a szoftver készítése során, tapasztalat alapján igen hasznos lenne egy hasonló funkciót beépíteni programunkba. Tulajdonképpen a *CommonSyntaxProvider* osztályunk már fel van készítve erre a lehetőségre.

Szintaktikai elemző

Programunk tulajdonképpen rendelkezik szintaxisellenőrző rutinnal, azonban ez csak fordításkor jelenik meg, és a szintaktikai hibáról csak az interpreter ad vissza üzenetet. Nagyban segítené a kódolást, ha aszintaktikai hibás kódrészeket egy kódolás közben is működő szintaktikai elemző ellenőrizné, lehetőség szerint aláhúzná a hibát, meghatározva ezzel a hiba helyét, illetve információt is szolgáltatna a szintaktikai hibáról.

Multiplatform

Szoftverünk jelenleg csak Windows-os környezetben üzemel, ami egyelőre nem gond, mivel az iskolák nagy része Windows operációs rendszert használ, azonban gondolnunk kell azokra a helyekre, ahol esetleg Linux/UNIX operációs rendszert használnak. Erre jelent megoldást a Mono. Ez egy ingyenes és nyílt forráskódú projekt, mely kompatibilis a .NET Keretrendszer komponenseivel, így a CLR¹⁰-el, és a C#-al is, illetve tartalmazza a szükséges eszközöket, mint például a C# fordító.

Mivel a Mono platformfüggetlen, működik a leggyakoribb operációs rendszereken. Ezáltal képesek leszünk platformfüggetlen alkalmazásokat írni .NET-es környezetben, illetve ami talán még hasznosabb, más, Windows-tól különböző operációs rendszeren és fejleszthetünk .NET-es szoftvert. Így a Linux rendszert preferáló fejlesztők is képesek C# programot fejleszteni, illetve futtatni azt.

¹⁰ Common Language Runtime

Irodalomjegyzék

<http://www.c-sharpcorner.com/UploadFile/mahesh/wpf-richtextbox/>

<http://msdn.microsoft.com/en-us/library/>

<http://msdn.microsoft.com/en-us/library/system.windows.documents.run.aspx>

<http://msdn.microsoft.com/en-us/library/aa970909.aspx>

<http://msdn.microsoft.com/en-us/library/system.windows.controls.richtextbox.aspx>

www.quantumr.hu/plantronics/MiisazaBluetooth.doc

http://www.inf.u-szeged.hu/projectdirs/bohusoktat/regi/szganyagok/esszek/szasz_zita.pdf

<http://32feet.codeplex.com/>

Reiter István – C# lépésről lépésre

Petró Emil - Aszinkron programozás magasfokon

Dr. Radványi Tibor, Dr. Kusper Gábor - Jegyzet a projekt labor című tárgyhöz

<http://msdn.microsoft.com>

<http://nyelvek.inf.elte.hu>

<https://devportal.hu/groups/wpf/pages/wpf-alapkonceptci-243-k.aspx>

<https://devportal.hu/groups/wpf/blog/archive/2008/10/08/tervez-233-s.aspx>

<http://msdn.microsoft.com/en-us/library/system.windows.forms.datagridview.aspx>

<http://www.zmgzeg.sulinet.hu/bemutakozunk/tankonyv/progism/Kiegeszitesek.pdf>

<http://itszotar.hu/?q=664>