



# Interfész megoldások imperatív és OOP nyelvek közötti kapcsolattartásra

## **Készítette**

Nagy-Tóth Bence

Szak: Programtervező informatikus

Specializáció: Szoftverfejlesztő informatikus

## **Témavezető**

Dr. Király Roland

egyetemi docens

EGER, 2022

# Tartalomjegyzék

<b>Bevezetés</b>	<b>1</b>
<b>1. Programozási nyelvekről általában</b>	<b>3</b>
1.1. A programozási nyelvek formális nyelvek? . . . . .	3
1.2. Jelenleg népszerű programozási nyelvek . . . . .	4
1.3. Milyen adatszerkezetek vannak? . . . . .	6
1.4. Használt adatszerkezetek . . . . .	6
1.5. Mi a helyzet az algoritmusokkal? . . . . .	7
<b>2. Marshalling</b>	<b>8</b>
2.1. Kommunikáció adatszerkezeteken keresztül . . . . .	8
2.2. Szerializáció . . . . .	8
2.3. Marshalling és serializáció . . . . .	8
2.4. Adatok konvertálása . . . . .	9
2.4.1. Egész és valós számok, logikai változók . . . . .	9
2.4.2. Szövegek . . . . .	9
2.4.3. Rekordok . . . . .	9
2.5. Managed és unmanaged kód közötti különbség . . . . .	9
2.6. Az ellenőrzött kódok előnyei és hátrányai . . . . .	10
2.7. PInvoke . . . . .	10
2.8. Memóriakezelés az InterOp marshaller segítségével . . . . .	10
<b>3. Elosztott rendszerek</b>	<b>12</b>
3.1. Gondolataim a streamalapú kommunikációról . . . . .	12
3.2. RESTful alkalmazások . . . . .	13
3.3. Google RPC és a Proto-nyelv . . . . .	13
<b>4. Szakdolgozati projektünkről</b>	<b>17</b>
4.1. Miről is szól a projektünk? . . . . .	17
4.2. Mit jelent az intelligens szoba? . . . . .	18
4.3. Delphi és C# programozási nyelvek összehasonlítása . . . . .	18
4.4. Mik azok, amelyeket a C# tud, de a Delphi nem? . . . . .	20

4.5.	Mik azok, amelyeket a Delphi tud, de a C# nem? . . . . .	20
4.6.	Alaphelyzet . . . . .	21
4.7.	DLL-függvények bemutatása . . . . .	21
4.7.1.	DLL megnyitása . . . . .	21
4.7.2.	Eszközök felmérése . . . . .	21
4.7.3.	Hibakódok . . . . .	21
4.7.4.	A WinForm bemutatása . . . . .	22
4.7.5.	DLL-ek üzenetküldése Win32-ben . . . . .	23
4.8.	Problémák és megoldások . . . . .	24
4.8.1.	Egyénileg definiált típusok és struktúrák . . . . .	24
4.8.2.	JSON-formátumra konvertáló függvények hívása .NET keret- rendszerből . . . . .	24
4.8.3.	Az SLDLL_Open-függvény paraméterezése . . . . .	25
4.8.4.	Stringek átadása Delphi DLL-ből C# részére . . . . .	25
4.9.	A relayDLL-ben eldobott kivételek nem kezelhetők . . . . .	27
4.9.1.	Megoldás . . . . .	27
4.10.	uzfeld metódus CSharp-os megfelelője . . . . .	27
4.10.1.	Megoldás . . . . .	27
4.11.	Eszközök azonosítóinak, típusainak átadása . . . . .	27
4.11.1.	Megoldás . . . . .	27
4.12.	A hangszóró egy egész hanglistát kezel - JSON . . . . .	28
4.12.1.	Megoldás . . . . .	28
4.13.	A hangszóró egy egész hanglistát kezel - kód . . . . .	28
4.13.1.	Megoldás . . . . .	28
4.14.	Két DLL működésének tesztelése . . . . .	29
4.15.	Teljesítmények összehasonlítása Delphi és C#-hívások esetében . . . .	29

## **Kivonat**

Szakdolgozatom arról szól, hogy...Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# Bevezetés

Kognitív képességeink fejlesztésére különösen oda kell figyelnünk életünk során. Ezen megállapításomat alátámasztja egy COVID-világjárvány utáni időszak, amely időszakban sorra jelennek meg olyan jelentések [18], amelyek azt támasztják alá, hogy a lezárások ideje alatt nőtt a különböző mentális betegségek kialakulásának kockázata. A *demencia* gyűjtőfogalomként alkalmas ezen betegségek együttes megnevezésére.<sup>1</sup>

A különböző digitális eszközök használata számtalan alkalommal hosszas órákon keresztül képes lekötni a figyelmünket, ezért bizonyos külső ingerekre egyre lassabban, kisebb amplitúdóval, azaz nagyobb közömbösséggel tudunk reagálni.

A demencia, azaz a kognitív képességek leépülésének tünetei közé tartoznak a különböző beszédzavarok, az ítélőképesség, a memorizálás és az elvonatkoztatás képességeinek romlása.



*Forrás: Kutatási jelentés a demencia és a demenciagondozás aktuális helyzetéről  
Kecskeméten, 2020 - Modus Alapítvány*

1. ábra. Valamely demenciával diagnosztizáltak számának emelkedése

Az értelmi hanyatlást különféle egészségkárosító szokások – például az alkoholfor-

<sup>1</sup> A demencia említésekor nem egyetlen betegsége, hanem több hasonló jellegű problémát csoportosító fogalomra, egy tünetegyüttesre gondolunk. Száznál is több típusát ismerik a demenciának, amelyek közül az Alzheimer-kór a leggyakoribb.

gyasztás – szervi károsodás, akár ezek együttese is okozhatja, a legjelentősebb kockázati tényező ilyen téren azonban az ember életkora.

Szakemberek véleménye szerint a megfelelő étrendek megalkotásával, a társas interakciók intenzitásának és tartalmi színvonalának növelésével, az emberek képzésével mind iskolai, mind munkahelyi szinten megelőzhetjük agyi teljesítőképességünk romlását. Kifejezetten preventív jelleggel hatnak a különféle társasjátékok, valamint a rejtvényfejtés.

Egy szó, mint száz: bármi, amivel munkára bírjuk agyunkat, alkalmas arra, hogy gátat vessen annak romlásának. Végül és abszolút nem utolsó sorban említhetjük a mozgás fontosságát, munkánkban a demencia megfékezésének ezen útját választottuk.

Szakedolgozatom ezen pontján azért érdemes erről a betegségről szót ejtenem, mivel alapvetően számomra már önmagában motivációul szolgál a tudat, hogy egy ilyen nagy volumenű elmélet megvalósításában vehetek részt, amely a demencia megelőzését, a mentális állapot folyamatos rongálódásának megfékezését célozza.

Az általunk készített szoftver támogatást nyújthat csoportfoglalkozásokon, tulajdonképpen egy tornaórát le tudunk vezényelni az eszközöknek köszönhetően. Természetesen az eszközök által kibocsájtott különböző fény- és hangjelzések önmagukban nem hordoznak semmiféle jelentést, a jelzések konkrét a tornaórákat vezénylő szakemberek dolga közölni a csoport számára.

# 1. fejezet

## Programozási nyelvekről általában

### 1.1. A programozási nyelvek formális nyelvek?

**1.1. Definíció.** Legyen  $\mathbb{A} = \{a_1, a_2, \dots, a_n\}$  véges, nemüres ( $\mathbb{A} \neq \emptyset$ ) halmaz, ezt a nyelv ábécéjének, elemeit betűknek vagy jeleknek nevezzük.  $\mathbb{A}$  halmaz elemeiből képezzük annak hatványait, ekkor

1.  $\mathbb{A}^0$  az üres szó ( $\epsilon$ ) egyelemű halmazát,
2.  $\mathbb{A}^1$  az egybetűs szavak ( $\mathbb{A}^1 \subseteq \mathbb{A} \wedge \mathbb{A} \subseteq \mathbb{A}^1 \iff \mathbb{A}^1 = \mathbb{A}$ ),
3.  $\mathbb{A}^2$  a kétbetűs szavak,
4.  $\mathbb{A}^n$  az  $n$  hosszú szavak halmazát jelenti és így tovább.

Jelölje  $A^* = A^0 \cup A^1 \cup A^2 \cup \dots \cup A^n$  az ábécé elemeiből képzett véges szavak vagy más néven jelsorozatok halmazát (ezt az  $\mathbb{A}$  ábécé feletti univerzumnak hívjuk). Ekkor  $\mathbb{A}$ -ból kirakható szavak  $\mathbb{A}^*$  halmazának egy részhalmazát **formális nyelvnek** nevezzük. Szokásos még az  $\mathbb{A}$  ábécé feletti formális nyelv megnevezés is. A hatványok a halmaz önmagával vett *Descartes-szorzatait* jelentik. [19]

Fő különbségek formális és természetes nyelvek között:

- A formális nyelveket egy dedikált célra hozzuk létre, ezeket általában nem használjuk interperszonális (emberek közötti) kommunikációra. Ezzel szemben egy természetes nyelv (például az angol) egy emberi közösség aktuális és a múltban használt jelkészletét rendszerezi.

A C++ programozási nyelv például azért jöhetett létre *Bjarne Stroustrup* dán szoftverfejlesztő jóvoltából, mert a C - procedurális paradigmájú nyelv lévén - nem tette lehetővé többek között a tisztább objektum-orientált programozást, a memóriacímek helyett a biztonságosabb referenciák használatát. [33]

- A formális nyelvek kulcsszavakból állnak. A természetes nyelvek több építőelem-ből tevődnek össze: fonémák (hangok, betűk), morféma (szótövek, toldalékok), szavak, mondatok, bekezdések, szövegek.
- A természetes nyelvek fejlődhetnek spontán, emberi generációról generációra va-lamint tudatos módon (például nyelvújítás) egyaránt. A formális nyelvek alakulá-sát egy tervezési fázis előzi meg, ekkor a nyelv szabályrendszerét lefektetik, tehát csak és kizárólag tudatos, mesterséges beavatkozással lehet megreformálni őket.
- Az ember által is beszélt természetes nyelvek esetén a használt szavak hangsú-lyozásának, hanglejtésének, valamint a beszéd hangerejének is jelentésmódosító ereje lehet, a formális nyelvek esetében hangsúlyról egyáltalán nem beszélhetünk.

[10] [9] Ezekből következően minden programozási nyelv formális nyelvnek számít.

## 1.2. Jelenleg népszerű programozási nyelvek

2022-ben a legnépszerűbb programozási nyelveknek számítanak (a teljesség igénye nél-kül):[30][8]

### 1. JavaScript

- 1995, *Brendan Eich* fejlesztette a webböngészési funkcionalitások kibővítése végett.
- web-, játék-, valamint mobilfejlesztésre egyaránt használják
- webszerverként is tud funkcionálni (Node.js)

### 2. Python

- 1991, *Guido Van Rossum* tervezte annak érdekében, hogy olvashatóbb és nagyobb kifejezőerővel rendelkező kódok készülhessenek, a szintaktikai sza-bályok helyett a kód működésére tudjanak a programozók koncentrálni
- backend-fejlesztés
- automatizálás
- web scraping<sup>1</sup>
- Data Science<sup>2</sup>

---

<sup>1</sup> Információgyűjtés eszköze, amely lehetővé teszi, hogy automatizált módon (kód segítségével) bizonyos weboldalakról tetszőleges adatokat (például posztokat, közlő eseményeket) letölteni.

<sup>2</sup> Az informatika, a matematikai statisztika és az üzleti elemzés metszetében álló tudományág, amely adatok összegyűjtésével, ezek elemzésével foglalkozik annak érdekében, hogy a vállalatok jobb üzleti döntéseket tudjanak meghozni ezek segítségével. Forrás: Quora



### 3. HTML

- webdokumentumok kezelése: JSON, XML, SVG
- weboldalak statikus (állandó) részeinek fejlesztése

### 4. CSS

- weboldalak formatervét, kinézetét, stílusát alakítja ki
- HTML mellett hívják segítségül

### 5. Java

- 1995, Sun Microsystems fejlesztése, alapötlet: olyan eszközök vezérlése, amelyek elérnek egy kézben
- E-kereskedelem
- Financial Technology: pénzintézetekkel, tőzsdékkal, számlázással kapcsolatos szoftvereket jellemzően ezen a nyelven fejlesztik
- a megírt kódok futtathatóak különösebb átalakítás nélkül az elterjedtebb operációs rendszereken (a kód hordozható, platformfüggetlen)<sup>3</sup>

### 6. SQL

- 1972, *Donald D. Chamberlin* és *Raymond F. Boyce* az IBM alkalmazásában, adattáblák egyszerűbb kezelésének érdekében hozták létre
- adatbázisok kezelése, karbantartása
- Data Science

### 7. Go

- 2009, a Google fejlesztői alakították ki, hogy megoldják a hatalmas szoftverrendszerekkel kapcsolatos problémákat
- rendszerek, hálózatok programozása
- hang- és videószerkesztés
- Big Data<sup>4</sup>

### 8. C

---

<sup>3</sup> Ez azért lehetséges, mivel .exe fájl helyett egy átmeneti .class állomány (bytecode) készül, amit egy virtuális gép (Java Virtual Machine) tolmácsolja (interpretálja) gépi kódként a számítógépünknek Forrás

<sup>4</sup> Az informatika egyik tudományága, amely tömérdek mennyiségű, hagyományos számítógéppel nehezen kezelhető adatok tárolásával és feldolgozásával, ezek elemzésével foglalkozik. Forrás

- 1970-es években Ken Thompson és Dennis Ritchie jóvoltából, Assembly-nél magasabb szintű (természetes nyelvezethez közelebb álló) nyelv kialakítása volt a célja
- hardverelemek illesztőprogramjai, vezérlőkódjai
- operációs rendszerek fejlesztése
- 3D videók szerkesztése
- alacsonyabb szintű a fentebb felsoroltaknál, ezért könnyebb optimalizálni memória és futásidő szempontjából [20]

Összességében elmondható, hogy a programozási nyelvek eszközökként szolgálnak a fejlesztők kezében. Véleményem szerint minél több eszközt ismerünk, annál jobban meg tudjuk határozni, hogy az előttünk álló problémához az eszköztárunkból melyik lenne a legalkalmasabb, amellyel kényelmesen és hatékonyan tudunk dolgozni.

### 1.3. Milyen adatszerkezeteink vannak?

A programozási nyelvek szintaktikában ugyan eltérnek egymástól, amikor viszont adatok tárolásáról van szó, egy dologban egyetértenek: típusokra szükség van. Mit jelent az, hogy egy változót például `bool` típusúként definiálunk? Az adat típusa meghatározza, hogy

- mekkora memóriaterületet<sup>5</sup> kell számára lefoglalni
- a számítások folyamán hogyan kell őt értelmezni (például ha másik változónak értékül adjuk, hány bitet kell másolni)
- továbbá milyen műveletek végezhetőek vele (például egész típusú változókon értelmezhetjük a szorzás műveletét, szövegeknél ezt már nem tehetjük meg).[6]

### 1.4. Használt adatszerkezetek

Ahogy említettem, a programozási nyelvek döntő része típusos, ezenfelül kisebb-nagyobb különbséggel hasonló adatszerkezeteket értelmez.

- a) *elemi adattípusok*: Elemi/primitív típusokat nem tudunk további részekre bontani, csak egyben értelmezhetjük őket. Ilyen például a `double` lebegőpontos típus,

---

<sup>5</sup> mivel a byte számít a legkisebb megcímezhető memóriaegységnek, ezért ennek mértékegysége alapértelmezetten bájtban értendő

amely 8 bájt képes lebegőpontos számot ábrázolni. Igaz, hogy csak az első bájt-ra van mutatónk, de nincs értelme további bájtokra darabolni, és megnézni az értékeket, mivel egyben értelmezendő, a műveleteket 4 byte-on fogjuk tudni vele végezni. Primitív adattípusoknak is nevezzük őket.

Vegyük például a C# programozási nyelvet, milyen elemi típusai vannak?

	Előjeles típus	Előjel nélküli típus	Méret (bájt)
	sbyte	byte	1
a) egész:	short	ushort	2
	int	uint	4
	long	ulong	8

	Típus neve	Típus mérete (Bájt)
b) lebegőpontos:	float	4
	double	8

c) logikai: bool 1 Bájt

d) karakteres: char karakterkódolástól függ<sup>6</sup>

b) *összetett adattípusok*: Az összetett adattípusok elemi típusokra szedhetők szét, vagyis primitív és/vagy további összetett típusú változókból épülnek fel. Ami a adattagok vagy mezőnek nevezzük. Az összetett adattípusú változók tárolásához szükséges memóriaterület kiszámítható az adattagjainak összegével.

C#-ban a **struct** és a **class** kulcsszavakkal tudunk összetett típusokat definiálni.

## 1.5. Mi a helyzet az algoritmusokkal?

Adatokat tudunk tárolni, és ezt tesszük azért, mert tervünk van velük, azaz valamit szeretnénk velük kezdeni. Az adatszerkezeteken végzett véges számú elemi lépéssorozatot algoritmusnak nevezzük. Az adatszerkezetek algoritmusok nélkül lényegében olyanok, mint a matematikai műveletek operátorok (összeadás, kivonás, stb.) nélkül, végül – ha már említettem a természetes nyelveket – mint a főnevek igék nélkül.

## 2. fejezet

# Marshalling

### 2.1. Kommunikáció adatszerkezeteken keresztül

A *Marshalling* egy olyan folyamat, amely összetett típusok átalakítására szolgál, hogy egy nyelv adatszerkezetét a másikkal megértessük. Magyar fordításával nem találkozunk ennek a kifejezésnek, de leginkább talán az 'átalakítás' szóval tudjuk leírni, ami adatszerkezetek esetében annyit tesz, hogy más nyelv által is értelmezhetővé tesszük, kevésbé hagyatkozunk az adott nyelv különlegességeire.[32]

A kód további portolhatósága, újrahasznosíthatósága végett megéri szabványos formátumokon keresztül kommunikálni, mint azt tesszük API-k<sup>1</sup> vagy konfigurációs fájlok esetében <sup>2</sup>, ilyenek például az általam használt JSON- vagy XML-formátumok, hogy más felületekre való költöztetés esetén kompatibilitási probléma többé már ne merülhessen fel. A Marshalling primitív típusok esetén abszolút működőképes, összetett adatszerkezetekre viszont inkább a szabványos szöveges formátumokat érdemes alkalmazni tapasztalatom szerint. Ha ezt elfogadjuk, akkor érdemes megismerkedni a serializáció és deserializáció folyamatával.

### 2.2. Szerializáció

[4]

### 2.3. Marshalling és serializáció

[27]

---

<sup>1</sup> Az Application Programming Interface utasítások, szabványok és metódusok halmaza, amely leírja a kommunikációt más, külső szoftverek részére. Az API-k lehetőséget biztosítanak két vagy több szoftver közötti adatok cseréjére, valamint adatokon végzett műveletek végrehajtására úgy, hogy annak mikéntjéről az API írója gondoskodik. Másként fogalmazva: az API a szerver azon része, amely felelős a kérések feldolgozásáért és kiszolgálásáért.

<sup>2</sup> Például az *app.config* fájl a C#-ban XML-formátumban látható.

## 2.4. Adatok konvertálása

### 2.4.1. Egész és valós számok, logikai változók

### 2.4.2. Szövegek

[13]

### 2.4.3. Rekordok

## 2.5. Managed és unmanaged kód közötti különbség

Először is fontos tisztázni, hogy a C# szemszögéből nézve a *Unmanaged Code* (nem felügyelt kód) elnevezést a .NET-keretrendszer mindenféle külső komponens megbélyegzésére használja, amelyet nem a .NET futtatókörnyezete felügyel és vezérel. Ettől még ezek lehetnek jó kódok, csak szimplán a Common Language Runtimenek nincs közvetlen hatása ezek működésére: a memóriacímeket közvetlenül lehet kezelni, felülírni, így bizonyos esetekben olyan memóriaterületre is hivatkozhat egy változó, amelyet az operációs rendszer nem a program részére foglalt le, ezt **segfault**nak nevezzük<sup>3</sup> [14]. A Common Language Runtime használatával fejlesztett kódot kezelt vagy felügyelt (managed) kódnak nevezzük. Más szóval, ez az a kód, amely a .NET futtatókörnyezete által végrehajtásra kerül. A menedzselt kód futási környezete számos szolgáltatást nyújt a programozó számára: ilyen a kivételkezelés, típusok ellenőrzése, a memória lefoglalása és felszabadítása, így a Garbage Collection<sup>4</sup>, és így tovább. A fent említett szolgáltatások a fejlesztőket szolgálják, hogy biztonságosabb és optimálisabb kódot készíthessenek.

A felügyelet nélküli kód ennek pontosan az ellentéte: pointereket kezelhetünk a kódban, ezeket tetszésünk szerint lefoglalhatjuk, felszabadíthatjuk, megváltoztathatjuk a pointer értékét (azaz más területre mutathatunk), viszont ilyen típusú kódok növelhetik alkalmazásaink teljesítményét, mivel az extra ellenőrzések ki vannak kapcsolva a kód lefutásának idejére (például egy tömb túlindexelését vizsgáló algoritmus). A nem biztonságos blokkokat tartalmazó kódot az **AllowUnsafeBlocks** fordítói opcióval kell lefordítani.

Amikor eszközöket vezérlő kódrészleteket szeretnénk C#-on keresztül hívni, legyen ez C++ vagy éppen esetünkben Delphi nyelven készítve, egyszerűen megkerülhetetlen a felügyelet nélküli kódok futtatása, mivel a C#-nak ezekre ráhatásai nincsenek, nyelvi

---

<sup>3</sup> Másnéven *access violation* hibának is nevezik, a Delphi inkább ezzel a kifejezéssel él a hiba kijelzésekor.

<sup>4</sup> A Garbage Collection, azaz szemétygyűjtés egy automatikusan végbemenő optimalizálási folyamat, amely során futásidő közben felszabadulnak olyan memóriaterületek, amelyek a kód korábbi részén lefoglalásra kerültek, azonban a program későbbi szakaszban már egyáltalán nem történik hivatkozás rájuk.

korlátokba ütköznénk, ha lámpákat és nyilakat kéne vezérelnünk C#-forráskóddal. A .NET úgy áll ezekhez a kódokhoz, hogy „megengedett, de nem ajánlott” kategóriába sorolja őket, nekünk annyit kell tennünk az ügy érdekében, hogy minél kevesebbyszer hagyatkozzunk az unmanaged kódrészletekre.

## 2.6. Az ellenőrzött kódok előnyei és hátrányai

- A futtatott kód biztonságosabb, erről ellenőrző algoritmusok gondoskodnak (például ilyen az *array boundaries check*, ami a tömbök túlindexelését vizsgálja).
- A szemétygyűjtés automatikusan lezajlik, ezért nem is kell memóriaszivárgástól tartanunk.
- A dinamikus, azaz futásidejű típusellenőrzés biztosítva van.

Átokként és áldásként is felróható, hogy nem enged direkt ráhatást memóriacímekre: hogy a lefoglalások és felszabadítások mikor, a szekvenciának mely pontjain történnek meg, arról nem tudunk konkrét információkat, mivel a GC ezt elrejtí előlünk, a program bizonyos pontjain csak sürgetni tudjuk a Garbage Collectort, hogy végezzen optimalizálásokat.

Ami ténylegesen hátránként említhető, hogy nem enged hozzáférést alacsonyabb szintű részletekhez, így különböző mikrokontrollerek, hardverelemek vezérlésének szemszögéből kiindulva a programozás ezen módja nem tekinthető opciónak.

## 2.7. PInvoke

[28] [12]

## 2.8. Memóriakezelés az InterOp marshaller segítségével

Az Interop marshaller a .NET keretrendszer egyik komponense, amely a managed adat-típusok unmanaged típusokká történő konverziójáért felelős. A memóriakezelést illetően az Interop marshaller biztosítja, hogy a memória helyesen kerül lefoglalásra, és felszabadításra a konverzió folyamatában, különösen, ha nem felügyelt kóddal dolgozik. Amikor felügyelt objektumokat ad át nem felügyelt kódnak, az Interop az objektumot mély másolással<sup>5</sup> hozza létre, hogy a nem felügyelt kód a managed objektum befolyásolása nélkül módosíthassa az objektumot.

---

<sup>5</sup> A mély másolás (angoul deep copy) azt jelenti, hogy objektumonként a referencia szerint tárolt mezők is klónozódnak, úgyhogy két objektum között egyáltalán nem lehet közös referencia.

Amikor viszont nem felügyelt objektumokat adunk át a felügyelt kód részére, az Interop marshal egy sekély másolatot hoz létre, így a felügyelt kód hozzáférhet a memóriához. Ez azt jelenti, hogy írhatja és olvashatja egyaránt ezen változók értékeit.

Fontos megjegyezni, hogy a memóriakezelést az Interop Marshaller automatikusan elvégzi, így a fejlesztőknek nem kell aggódniuk memóriaszivárgás miatt. Bizonyos esetekben azonban szükség lehet a használt memória explicit módon történő felszabadítása. [17]

---

Ennek ellentéte a shallow copy, azaz a sekély klónozás, amelyben a tárolt objektumok memóriacíme egy az egyben átmásolásra kerül, de ekkor egy objektum ilyen típusú mezőjének változása a klónban ugyanazt a változást eredményezi.

## 3. fejezet

# Elosztott rendszerek

Az elosztott rendszerek lényege, hogy van egy alkalmazás, amivel különböző kliens-programok (ügyfelek) kommunikálhatnak, ez a szerveralkalmazás (másnéven host vagy kiszolgáló), amely legtöbb esetben egy másik számítógépen fut, amely számítógép ráadásul egy másik hálózat tagja, tehát a két gép közötti adatátvitel internetkapcsolat segítségével valósítható meg.

Az kliens és a szerver kommunikációja kétirányú: az ügyfelek többnyire kéréseket küldenek a kiszolgáló valamely erőforrásának (adatbázis) eléréséhez, a szerver is küld a kliensprogram részére üzeneteket, tájékoztatást hibaüzenetről, a végrehajtott művelet sikerességéről, annak eredményéről, és így tovább.

A szerveralkalmazás is fogadhat bizonyos adatokat a kliensektől, ezek lehetnek bejelentkezési adatok vagy éppen a felhasználó által használt kliensprogram verziószáma, elavultabb verziók esetén akár figyelmeztetheti is a felhasználót, hogy telepítse a program egy frissebb változatát. [2]

Kliens-szerver-kommunikáción alapuló alkalmazásokat különböző módokon építhetünk fel, amelyek a témám szempontjából csak annyiból fontosak, hogy például szolgáljanak programozási nyelvek közötti kapcsolatteremtés alkalmazási lehetőségeire.

### 3.1. Gondolataim a streamalapú kommunikációról

Ebben a megközelítésben a kliens és szerver kódjának nyelve megegyezik, a téma szempontjából már ez is egy tanulsággal szolgál: kell egy közös nyelv, egy szerződés a két fél között, amelytől egyik fél sem térhet el<sup>1</sup>: a streamalapú kommunikáció ezt az alábbi módon valósítja meg: szövegesen eljuttatja a szerver számára a végrehajtani kívánt metódus nevét és paramétereit.

Például egy kliensalkalmazásban elküldhetjük az alábbi üzenetet a streamen keresztül: *"ADD/2/3"*, ezt a szerver a várakozás állapotában megkapja, rá is illeszti az üzenet

---

<sup>1</sup> Amennyiben valamelyik fél eltér a meghatározott szabványtól, a kommunikáció nem fog létrejönni, a kliens és a szerver nem fogja érteni egymást.



első tagját ("ADD") az általa ismert parancsokra, és amennyiben ismeri az "ADD" parancsot, meghívja rá saját  $Add(int\ a, int\ b)$ -függvényét, majd egy hasonló stream üzenetet küld vissza a kliens számára: "OK/5". [21]

Láthatjuk, hogy hasonlóság a küldött üzenetek között, van a két fél között egy megállapodás, egy szabvány, ami leírja a kommunikáció formáját. Elméletem szerint ez már önmagában nevezhető egy közös nyelvnek a kommunikációban résztvevő két komponens között.

A szerver ismeri az  $\mathbb{A} = \{"ADD", "SUB", "MUL", "DIV", "EXIT"\}$  parancsokat, tehát  $\mathbb{A}$ -halmaz elemeit, ezen halmaz elemeit bátran nevezhetjük a közös nyelvet leíró kulcsszavaknak, amelyekkel a távoli számítógépen metódusokat hívhatunk meg), és a kliensnek erről feltétlenül tudnia kell, hogy ismerje ezen parancsok meghívási módját (kvázi az üzenetküldésre használt nyelv szintaxisát!), valamint ezeket meghívva milyen eredményt várhat.

Ha továbbgondoljuk, a nyelv szemantikája is megadható: tegyük fel, hogy az előbb felsorolt parancsok használatához szükség van bejelentkezésre. Bejelentkezik "LOGIN|username|password" üzenettel a szerverre, amely eldönti, hogy a bejelentkezési adatok passzolnak-e az általa eltárolt felhasználói rekordok bármelyikével, amennyiben igen, bejelentkezeti a felhasználót a szerverre. Ekkor már a többi parancs is megnyílik a számára, a "LOGIN"-parancs letiltásra kerül. Ha a bejelentkezést követően a felhasználó a "LOGIN|username|password"-parancsot ismételten kiadná, kapna a szervertől egy "ERROR|ALREADY\_LOGGED\_IN" üzenetet. Szintaktikailag<sup>2</sup> ugyan helyes parancsot adott ki a felhasználó – azaz az általa futtatott kliensprogram –, de már a parancs meghívása ebben a kontextusban (ebben a párhuzamban kontextusnak számíthatjuk a sessiont)<sup>3</sup> már egyszer megtörtént, így a megkapott hibaüzenetben tulajdonképpen egy *szemantikai hibáról*<sup>4</sup> kap tájékoztatást a felhasználó.

## 3.2. RESTful alkalmazások

## 3.3. Google RPC és a Proto-nyelv

A Google RPC, röviden gRPC a Google jóvoltából készült nyílt forráskódú RPC-keretrendszer. Remote Procedure Call-nak<sup>5</sup> nevezzük, amikor  $A$  processz meghívja  $B$  processz valamely meghirdetett szolgáltatását, így  $B$  a kért műveletet implementációja szerint elvégzi, majd  $A$  részére válaszüzenetben visszajuttatja az általa kért művelet

---

<sup>2</sup> Szintaktika

<sup>3</sup> Egy többfelhasználós rendszerben munkamenetnek (session) nevezzük azt az időszakot, ami eltelik a felhasználó be- és kijelentkezése között, ekkor az szabadon végezheti a dolgát, nem kell minden egyes funkció használatához bejelentkeznie.

<sup>4</sup> Szemantika

<sup>5</sup> magyarul: Távoli eljáráshívás

eredményét. Így *A*-nak nem kell törődnie a végrehajtott algoritmus részleteivel, ennek implementációja és futtatása *B* processzre delegálódik. Ez lehetőséget teremt arra, hogy bizonyos kódrészleteket egy másik számítógépre, egy szerverre helyezzünk át, ettől lesz a hívás 'távoli'. A gRPC-t számos programozási nyelv támogatja, a teljesség igénye nélkül ezek a Java, C#, C, C++, Kotlin, Python és PHP-nyelvek. Még Delphi-ben is létezik egy külső komponens, amit a projekthez hozzáadva használhatjuk a gRPC nyújtotta szolgáltatásokat. [23]

A keretrendszer működtetője az úgynevezett *Protocol Buffers*, amely nyelv- platformfüggetlen adatcserét biztosít. A gRPC-ben ennek a segítségével teremthetünk kapcsolatot különböző vagy azonos programozási nyelven készített alkalmazásaink között, egy szerveralkalmazáshoz különféle platformokra írt kliensprogram is kapcsolódhat, mivel a kérések és válaszok nyelvezete közös, szabványos.

Ami weboldalaknál a HTML és a JSON-formátum, az a gRPC esetén a Proto-nyelv, ami átjárást biztosít akár más nyelven írt programok részére is. Amely nyelv képes implementálni a gRPC-t, és a Proto-nyelvből le tudja gyártani maga forráskódját, és a forráskódjából generálható a Proto-formátum, az alkalmas ezen a nyelven keresztül összekapcsolódni más, ugyanezt a keretrendszert alkalmazó szerverprogrammal is. A Proto-nyelv választott témám szempontjából kruciális, mivel kifejezetten erre a célra lett kitalálva, hogy programozási nyelvek felett álljon, ezért úgy gondolom, érdemes tovább vizsgálnunk a nyelvben rejlő lehetőségeket.

Miért kell ehhez egy külön nyelv? Nem lehetne maradni a jól bevált JSON-nél? A Proto nyelv további előnyökkel jár, amelyeket sem a JSON, sem az XML, sem bármilyen más szerializált formátum nem tud biztosítani: az adatszerkezeteket szigorúan típusosan írja le.

Ezenkívül nemcsak adatok, hanem szolgáltatások leírására egyaránt alkalmas a Proto-nyelv, ezért is érdemes az RPC ezen változatát használni, mivel nagyobb rugalmasságot biztosít, mint egy átlagos szerializált formátum.

```
service Adopt {  
    rpc Login(User) returns (SessionID);  
    rpc Logout(SessionID) returns (Result);  
    // ...  
}  
  
//bejelentkezéskor user ezt kapja, ezt használjuk a felhasználó kiléptetésére  
message SessionID{  
    string id = 1;  
    string username = 2;  
}  
  
//CRUD-műveletek visszajelzése szöveges formában  
message Result{  
    string response = 1;  
}  
  
//bejelentkezéskor ezt küldi a kliens a szervernek lekérdezésre  
message User{  
    string name = 1;  
    string passwd = 2;  
    bool isAdmin = 3;  
}
```

3.1. ábra. Beadandó projektben készített .proto állomány (részlet)

Lényegében elég a fentebb látható .proto kiterjesztésű fájlt elkészítenünk, ennek környezetét, a kliens- és szerveroldali stubok forráskódjának vázát az adott nyelv szabályrendszere szerint képes lesz legyártani ebből. Ennek lehetősége a legtöbb RPC-keretrendszerénél nem áll fenn.

Mint azt említettem, Proto-nyelvben a változókat csak típussal együtt vehetünk fel<sup>6</sup>, osztályokat `message`, a meghívható metódusokat a kifejezőbb `rpc` kulcsszóval látja el. Mint láthatjuk, a mezők nevéhez 1-től kezdődően számokat rendelünk, ez a serializációhoz szükséges. Binárisan ez úgy fog kinézni, hogy a mező azonosítóját annak típusával kombinálják, ezzel lényegében egy mező leírásához pusztán 1 bájtnyi terület elegendő. Ha üzeneteinket úgy szervezzük, hogy azonosítókat 1-től maximum 15-ig terjedően oszthatunk ki (tehát legfeljebb 15 mezőt engedünk meg osztályonként), akkor ez a tulajdonság garantált.

Természetesen ez is a gRPC előnyeirez sorolható, hogy kevesebb adatot kerül átadásra szervertől kliensig, és fordítva. Az adatokat a lehető legtömörebb bináris formátumban képes leírni és szállítani a hálózaton, a JSON-formátum ezzel szemben egy az egyben szövegesen kerül átadásra, ami jóval több adatforgalmat tesz ki.

Emellett nyugodtan kijelenthetjük, hogy ez a megoldás típusbiztonsági szempontból is előnyösebb, mivel a Proto-nyelvet használva a mezőket eleve típusokkal kötjük, így nem az érkezés helyein kell validálni a megérkezett adatokat. Szöveges serializáció esetében minden alkalommal, amikor például JSON érkezik, elsősorban ellenőriznünk kell, hogy a fogadott JSON-stringben az értékek típusai rendre megegyeznek-e az osztályban definiált mezők típusaival. Ez a Proto-nyelv esetében automatikusan teljesül, mivel a típussal kapcsolatos hibák már a küldéskor kiütköznek.

Az adatokat a lehető legtömörebb bináris formátumban képes leírni és szállítani a hálózaton, a JSON-formátum ezzel szemben egy az egyben szövegesen kerül átadásra, ami jóval több adatforgalmat tesz ki. [7]

GRPC-vel négyféle metódushívást tudunk leírni

- *Unáris hívások*: Itt egyszerű kérdés-válasz-üzeneteket cserélünk. Ez Protóban úgy néz ki, hogy a szolgáltatások paramétere és visszatérési értéke is message típusú.
- *Kliens streamelés*: A kliens egy streamet, egy adatfolyamot (kvázi egy listát) küld paraméterben a szerver részére, a kiszolgáló pedig egy normál message-objektummal tér vissza.
- *Szerver streamelés*: A kliens elküld egy kérést, és a szervertől válaszként streamet fogad.
- *Kétirányú streamelés*: A kliens és a szerver is streamekben kommunikál egymással.

```

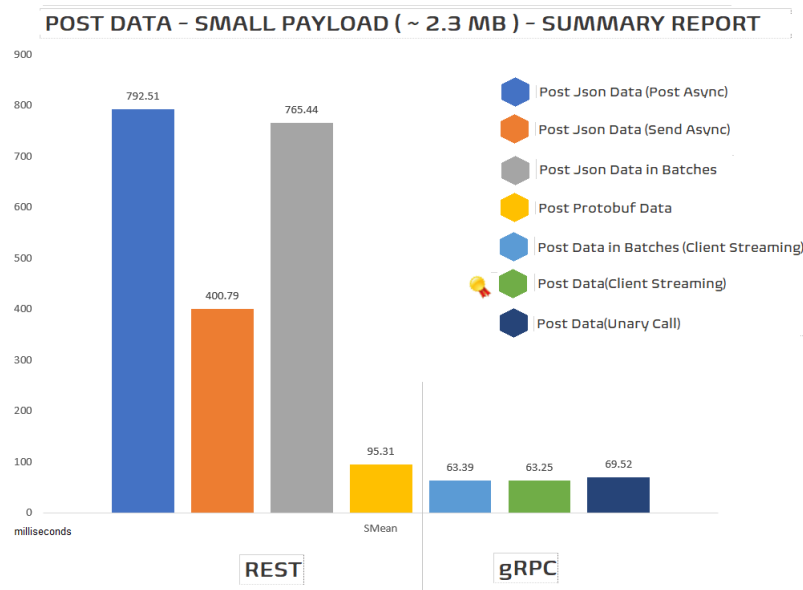
package dzM;

// The service definition.
service InventoryService {
    //Unary
    rpc SendProductList (InvProductList) returns (SuccessMessage);
    //Client Streaming
    rpc SendProductListAsStream (stream InvProductList) returns (SuccessMessage);
    //Server Streaming
    rpc GetProductListAsStream (MarkerMessage) returns (stream InventoryProduct);
    //Unary
    rpc GetProductList(MarkerMessage) returns (InvProductList);
    //Bi-Directional Streaming (Not In Scope)
    //rpc GetProductListAsBiStream (stream Ids) returns (stream InventoryProduct);
}

```

3.2. ábra. gRPC metódushívásainak típusai

Az alábbi ábrán látható, hogy a gRPC sokkal gyorsabb volt, mint az összes REST-hívás, a kísérlet győztese a `Client_Streaming` volt, ami 15-ször gyorsabb, mint a leglassabb REST-módszer, és legalább 5-6-szor gyorsabb, mint egy átlagos REST JSON-hívásokkal. A másik két gRPC módszer hasonlóan teljesített, és közel egy ezredmásodperccel lassabb. [1]



3.3. ábra. Szerializáció teljesítményének összehasonlítása JSON és Proto esetében

<sup>6</sup>Ezen tulajdonsága miatt a Proto szigorúan típusos nyelvnek számít.

## 4. fejezet

# Szakdolgozati projektünkről

Eme dokumentum mellé készített szakdolgozati szoftverprojektemet Sipos Levente hallgatótársam részére, munkájának megkönnyítésére készítem. Munkánk arról szól, hogy Keresztes Péter tanár úr által Delphiben implementált metódusokat hívunk meg .NET-es környezetből. A C# és Delphi programozási nyelvek összehangolása az én feladatom, az elkészült termék terveim szerint úgynevezett Helper-metódusok implementációit tartalmazó, lefordított<sup>1</sup> DLL-projekt lesz, amelyeket Levente az ő Windows Forms keretrendszerrel készített grafikus alkalmazásában tud meghívni. A projekt jelen állása szerint szükség volt továbbá egy úgynevezett relayDLL-re, amelyet én állítottam össze.

### 4.1. Miről is szól a projektünk?

Egy mentális egészségfejlesztésre használatos alkalmazást fejlesztésére vállalkoztunk 2022 szeptemberében, amely elméleti alapjait Somodi László futballedző munkásságának köszönhetjük, ezek gondolati vonulatairól titoktartási szerződésünk lévén csak nagyon érintőlegesen fogok beszámolni a későbbiekben.

A készített alkalmazásunk gyakorlatilag különböző fény- és hangjelzések kibocsátására alkalmas eszközök (lámpák, nyilak és hangszórók) vezérléséből áll, egy úgynevezett *intelligens szobában* az eredeti tervek szerint 8 eszköz (a készített program azonban tetszőleges,  $n$  darabszámú eszköz vezérlésére lett felkészítve) együttes vezérlését kell kezelünk megadott időközönként, amely időközöket egyszerűen ütemnek fogjuk nevezni. A módszer alkalmazása az intelligens szobával együtt működik teljességében.

---

<sup>1</sup> Lévén a C# fordított nyelvnek számít, ezért az alkalmazások futtatásához/használatához az elkészült forráskódokat első lépésben futtatható állományra (gépi kódra) kell fordítani egy fordítóprogram segítségével. Ezt a folyamatot szakszóval *compilingnak* vagy *buildelésnek* is nevezzük.

## 4.2. Mit jelent az intelligens szoba?

Az intelligens szoba elnevezés olyan helységet takar, amelynek mind a négy falán jeladókat helyeztünk el. Ezek különböző típusú eszközök: fények, nyilak és hangok. Ezek különállóan vagy együttesen jeleket küldenek, amely jelekre speciális mozdulatokat kell a foglalkoztatottnak végrehajtani. Az eltérő színek más feladatokat írnak le, mást kell tenni piros, és szintén mást zöld szín felvillanása esetén, ez emlékeztetheti az embert a forgalmi jelzőlámpák működésére is: minden színhez más jelentést rendelhetünk. A nyilak felvillanásának és a hangszóróból érkező különféle hangok észlelése esetén pedig irányváltásokat kell végezni.

Egy feladatsor több egymást követő ütemből áll, mely a programban azt fogja jelenti, hogy  $x$  másodperc késleltetéssel a felmért eszköz tömb elemeinek tulajdonságait (mezőit) a feladatsor aktuális ütemének megfelelően módosítjuk. Mivel minden eszköz együttesen vezérlünk, a program ütemenként az összes eszköz állapotát felül fogja írni, ezért szükségünk van egy olyan állapotra is, ami azt közli az adott eszközzel, hogy éppen semmit ne csináljon (azaz várakozzon). Ez a fényeszközök esetében egyszerűen  $(0,0,0)$  RGB-színkód<sup>2</sup> közlését, míg hangeszköz esetén egy 0 dB hangerősségű tetszőleges frekvenciájú hangjelzés kiküldését fogja jelenteni. Somodi László edzővel való együttműködésünk Dr. Király Roland tanár úr jóvoltából jöhetett létre, akinek az alkalmazással kapcsolatos múltbéli tapasztalatait és ötleteit folyamatos egyeztetések, konzultációk útján tudtuk segítségül hívni.

## 4.3. Delphi és C# programozási nyelvek összehasonlítása

Először is érdemes leszögeznünk, hogy a továbbiakban Delphi alatt nem a fejlesztői környezetet, amely az Object Pascal nyelvvel dolgozott együtt, hanem inkább a programozási nyelvet értjük, jelenleg az Object Pascal megnevezés úgynevezett „umbrella term” formájában él tovább.<sup>3</sup> [29] A Delphi nyelv (megjelenési éve: 1986, a Borland nevű cég jóvoltából) idősebb nyelvnek számít a C#-hoz (megjelenési éve: 2001, a Microsoft jóvoltából) viszonyítva, ebből fakadóan egy stabilabb, kiforrottabb, időtállóbb eszköznek számít a programozók kezében.

Mindkét nyelv **objektumorientált**, ami azt jelenti, hogy bizonyos, logikailag összetartozó adatokat (mezőket), valamint a rajtuk végezhető műveleteket (metódusokat) egy egységbe zárunk, ezt az egységet a továbbiakban osztálynak nevezzük. Az osztály

---

<sup>2</sup> Az RGB-színkódolás egy szín leírását három komponens, vörös (**R**ed), zöld (**G**reen) és kék (**B**lue) alapszínek arányától teszi függővé.

<sup>3</sup> Az umbrella term (magyarul gyűjtőfogalom) olyan kifejezés, amely több fogalmat rendel ön-maga alá, így már fogalmak egy csoportját, kategóriáját jelenti összefoglalóan.

mezőit és metódusait különböző láthatósági szintekkel vétezzhetjük fel, ezzel tudjuk védeni az osztályunk kritikus részeit más osztályokkal szemben. Az osztályok között akár öröklődéssel akár objektum-összetétellel viszonyokat alakíthatunk ki.

A két nyelv **erősen típusosnak** számít, mivel egy változó definiálásakor meg kell mondanunk azt is, hogy milyen típusú értékeket szeretnénk abban tárolni, a típusok már a forráskódban explicit módon megjelennek, így az adott nyelv fordítóprogramja fel van készítve a változó típusaira, amely változó hatókörén<sup>4</sup> belül nem is változhat meg. Ahogy azt már korábban is említettem, a változó típusa meghatározza, hogy az értéket hogyan kell értelmezni a memóriában, a megadott típusnak mely műveletei vannak értelmezve, így például `string` típuson nem értelmezhetünk logikai ÉS (konjunkció)-műveletet.

Az elkészült kódok teljesítménye alapján is érdemes összehasonlítani a két nyelvet. Ehhez először is külön kell választanunk a fordításhoz és a futtatáshoz szükséges időt, lévén ezek fordított (nem interpretált) nyelvek, ezért ezek nem szimultán módon<sup>5</sup> történnek. A Delphi fordítóprogramja azonnal gépi kódot<sup>6</sup> készít, míg a C# esetében első lépésben egy köztes nyelvű<sup>7</sup> kód készül, amelyet a .NET virtuális gép képes futtatni. Fordítási időben a Delphi-kód nyertesnek számít, futásidőben azonban közel ekvivalens a két nyelv révén gyártott kód.

Ha ennél is tovább megyünk, akkor a C#-nak nagy előnye származik abból a Delphivel való összevetésben, hogy aszinkron<sup>8</sup> kódolási lehetőséget is biztosít, amely felgyorsítja a végrehajtást, jobban ki tudja használni a rendelkezésre álló CPU teljesítményét. A LINQ<sup>9</sup> használata `yield` kulcsszóval az iterációkban biztosítja, hogy csak akkor van végrehajtva a kód, amikor ténylegesen szükség van rá (lusta kiértékelés). A `delegate`-ek használata szintén növelheti a C#-kódok teljesítményét, bár a Delphi is rendelkezik ehhez hasonló funkciókkal. [22]

---

<sup>4</sup> Hatókör, illetve *scope* alatt a program azon részét, kontextusát értjük, amely magába foglalja az adott változót. Ezen kontextusban vizsgálva a változó "életben van", tehát a memóriában hely van lefoglalva számára, nevére vagy memóriacímére hivatkozva értékét felülírhatjuk, kiolvashatjuk.

<sup>5</sup> nem egyidőben, nem egyszerre

<sup>6</sup> A gépi kódban már az utasításokat is számok jelzik, ezen nyelv utasításkészlete már a számítógépben működő processzor típusától is erősen függ. Gépi kód az esetek nagy részében fordítóprogram eredménye, a hardverközeli vezérlőprogramok elkészítéséhez is inkább a magasabb szinten álló Assembly nyelvet használják.

<sup>7</sup> A(z) (Common) Intermediate Language a .NET-keretrendszerben a magasabb absztrakciós szintű C# és a legalacsonyabb szintű gépi kód között helyezkedik el, ami még processzortól és operációs rendszertől független. A .NET Runtime futtatókörnyezete képes futtatni.

<sup>8</sup> Az aszinkron programozás lehetővé teszi, hogy az alkalmazás egy időigényes folyamat futtatását háttérbe helyezze, így a programot futtató szál, lévén nem várakozik a válaszra, addig ugyanúgy képes a felhasználói interakciókat kiszolgálni. [11]

<sup>9</sup> A **Language Integrated Query** (magyarul: nyelvbe ágyazott lekérdezés) egy gyűjtőfogalom a C# nyelvbe épített szintaktikai elemekre, amely elemek lehetővé teszik, hogy akár lambda kifejezésként, akár SQL-szintaxishoz hasonló módon meg tudjunk fogalmazni lekérdezéseket bizonyos - iterálható, vagyis bejárható, az `IEnumerable`-interfészt megvalósító - szerkezetekre. Ilyen szerkezetnek minősül például a `List` is.

A fejlesztés folyamán elkészített **modulokat** a különböző programozási nyelvek eltérő megnevezéseket használnak.<sup>10</sup> A vizsgált két nyelv tekintetében is ez áll fenn: a C# `namespace`, míg a Delphi `unit` kifejezéssel illeti, a lényegük ugyanaz lesz:

1. Mivel nagy projekteknél előfordulhat, hogy két, funkciójában eltérő osztálynak ugyanazt a nevet kellene adnunk, nem tehetnénk meg, mivel a fordítóprogram, így a futó program sem tudná eldönteni, hogy a két változat közül éppen melyiket kívánjuk használni. Ezt a problémát orvosolják például, hogy külön modulokban tudjunk azonos elnevezésű osztályokat kezelni.
2. Fontos továbbá, hogy mivel nem az nagyobb átláthatóság végett nem egy fájlban dolgozunk, és esetleg több kollégával fejlesztünk
3. Egy lefejlesztett DLL tekintetében is szükség van egyetlen összefoglaló névre, amit a futtatható állomány

[5]

Ami közös még a két nyelvben, hogy a fejlesztés moduljaiból Win32-szabványnak megfelelő DLL-ek készülhetnek a segítségükkel, ezek szintén lefordított, gépi kódú állományok, amelyek más szoftver forráskódjában felhasználhatóak, lényegében így válnak futtatható állománnyá.

#### 4.4. Mik azok, amelyeket a C# tud, de a Delphi nem?

- automatikus memóriakezelés - Garbage Collector
- lambdafüggvények és LINQ<sup>11</sup>
- szerializációt csak explicit, kódolt módon tudunk végezni

#### 4.5. Mik azok, amelyeket a Delphi tud, de a C# nem?

- Assembly nyelv
- case insensitive

[31]

---

<sup>10</sup> Az említett példákon kívül a Java-nyelvben *package*-ként, míg a Pythonban *module*-ként hivatkoznak az osztályokat összegyűjtő egységre.

<sup>11</sup> LINQ



## 4.6. Alaphelyzet

Amikor a munkát megkezdtem, rendelkezésemre állt Dr. Király Roland tanár úr által fejlesztett Delphis asztali alkalmazás, amely – mint kiderült – a mai napig teljes mértékben használható. Ezen alkalmazás forráskódjában követtem végig az hardvereket működtető függvények hívási sorrendjét (szekvenciáját), ezek módját és eredményeit, az esetlegesen előforduló hibaüzeneteket. Ezen alkalmazás C#-nyelvű megfelelőjének elkészítését és továbbgondolását kaptuk feladatként Leventével.

## 4.7. DLL-függvények bemutatása

A következőkben a Keresztes Péter tanár úr által Delphiben implementált metódusokat, ezek kezelésének lehetőségeit fogom részletezni. Általánosságban elmondható, hogy minden függvény egész típusú értékkel tér vissza, amely érték tájékoztat a lefutás eredményességéről: amennyiben a hívott metódus sikeresen (hiba, kivétel nélkül) lefutott, 0-val tér vissza, ahogy ezt egyébként az operációs rendszerek processzeinél is megszokhattuk. Ettől eltérő értékek az egyes hibatípusokat hivatottak meghatározni a Win32-szabvány<sup>12</sup> keretein belül. Ezen hibakódok projektünkre vonatkozó részét az alábbi táblázatban 4.1a összegyűjtöttem. [15]

A program azzal nyit, hogy felméri az USB-porton csatlakoztatott, egymással RJ11-csatlakozókkal sorba kötött eszközöket, őket a típusának megfelelő azonosítóval látja el. Az azonosító meghatározza, hogy egy eszköz milyen típusú.

### 4.7.1. DLL megnyitása

A program indulásakor elsőként lefutó `SLDLL_Open` függvényt hívva elkezdhetjük az `SLDLL` további metódusainak használatát.

### 4.7.2. Eszközök felmérése

`SLDLL_Felmeres`

### 4.7.3. Hibakódok

Az alábbi táblázatban látható hibakódokat C#-ban a következő megfelelő, egyénileg definiált kivételekkel, és sokkal kifejezőbb üzenetekkel váltom fel:

- `Dev485Exception`: Eszközök tömbjére vonatkozó hibaüzenetek
- `SLDLLException`: A DLL működésével kapcsolatos hibaüzenetek

---

<sup>12</sup> A Win32-es hibakódok szabványa szerint minden hibakódnak a `0x0000` (decimálisan: 0) és `0xFFFF` (decimálisan: 16 777 215) közötti tartományban kell lennie.

- `USBDisconnectedException`: Az USB-porton nem észlelhető eszköz hibaüzenete.

Hibakód	Hiba címe	Hiba jelentése (SLDLL esetén)	Gyakorlati példa
0	NO_ERROR / ERROR_SUCCESS	A függvény sikeresen lefutott.	Program indításakor nem érkezik hibaüzenet (nincs kivétel)
13	ERROR_INVALID_DATA	Az azonosító típusjelölő bitpárosa hibás, vagy nincs ilyen eszköz.	SLDLL_SetLampa függvényt hangszóró típusú eszközre akartuk meghívni.
24	ERROR_BAD_LENGTH	Hanghossz nem [1;16] intervallumból kap egész értéket.	SLDLL_SetHang függvény rosszul lett felparaméterezve.
71	ERROR_REQ_NOT_ACCEP	Jelenleg fut már egy függvény végrehajtása.	SLDLL_Felmeres-t hívja meg, miközben az SLDLL_SetLista fut.
85	ERROR_ALREADY_ASSIGNED	Az új azonosító más panelt azonosít, egyedinek kell lenniük.	Nem találkoztam még ezzel a hibával
110	ERROR_OPEN_FAILED	A megadott fájlt nem sikerült megnyitni.	Nem találkoztam még ezzel a hibával
126	ERROR_MOD_NOT_FOUND	A megadott fájl nem a firmware-re vonatkozó frissítési adatokat tartalmaz.	Nem találkoztam még ezzel a hibával
1114	ERROR_DLL_INIT_FAILED	Az SLDLL_Open még nem lett meghívva.	Bármely SLDLL-függvényt úgy hívjuk meg, hogy előtte az SLDLL_Open nem futott le
1247	ERROR_ALREADY_INITIALIZED	A függvény meghívása már megtörtént	SLDLL_Open függvény egymás után 2x való meghívása
1626	ERROR_FUNCTION_NOT_CALLED	Nincs csatlakoztatott USB-eszköz.	SLDLL_Open függvény hívásakor nincsenek csatlakoztatva az eszközök
egyéb	Windows műveleti hibakódok		

4.1. ábra. Win32-hibakódok magyarázata

Saját szerkesztés

#### 4.7.4. A WinForm bemutatása

A *WinForm*, teljes nevén a *Windows Forms* a .NET GUI-fejlesztést<sup>13</sup> támogató keret-rendszere, amelynek segítségével egy asztali alkalmazást egyszerűbb módon el tudunk készíteni.

Előkészített, a HTML-nyelvből már jól ismert, valamint ezek tárházát bővítő vezérlőelemekkel (gombok, legördülő listák, adattáblázatok, és így tovább) gondoskodik azok újrahasznosíthatóságáról. A Visual Studio fejlesztői környezet egy Designer-felülettel is rendelkezik, amellyel gyorsan és különösebb képzelőerő nélkül elkészíthetjük grafikus alkalmazásaink vázát.

Egy Windows Forms alkalmazásban a *Form* (továbbiakban űrlapnak is fogom nevezni) egy vizuális felület, amely információkat jelenít meg a felhasználó számára. Egy Windows Forms alkalmazás általában úgy készíthető el, hogy egy Formhoz vezérlőelemeket (Control) adunk, a felhasználó által végrehajtott műveletekre, például egérkattintásokra és billentyűleütésekre adott válaszokat implementálunk. A *vezérlőelem* különálló GUI-elemek gyűjtőfogalma, amely adatokat jelenít meg vagy adatokat fogad bevitelre.

Amikor a felhasználó műveletet hajt végre egy űrlapon vagy annak vezérlőelemein, ez a művelet eseményeket generál. Az alkalmazás kódban reagálhat ezekre az eseményekre, amennyiben azok bekövetkeznek, de figyelmen kívül is hagyhatja azokat. [16]

<sup>13</sup> A GUI a Graphical User Interface (grafikus felhasználói interfész) kifejezés rövidítése, asztali alkalmazásokat értünk alatta, amely átlagfelhasználók számára kényelmesebb megközelítés a konzolos alkalmazásokkal szemben

### 4.7.5. DLL-ek üzenetküldése Win32-ben

Van egy üzenetlista<sup>14</sup>, amin keresztül kommunikál az operációs rendszer a futó programmal, ezt projektünk szempontjából Levente ablakos alkalmazása fogja jelenteni. Az operációs rendszer ráteszi a lenyomott gomb által kiváltott üzenetet erre a listára, tehát például amikor az egér bal gombjával kattintunk, akkor azt ténylegesen nem a futó program fogja észlelni, mivel az operációs rendszer a központ, ahova az I/O-kérések befutnak.<sup>15</sup> Az alábbi lépéssorozat fog lejátsszódni:

1. Az operációs rendszer ráteszi a `WM_LBUTTONDOWN`-üzenetet az üzenetsorra.
2. A programunk meghívja a `GetMessage`-függvényt.
3. A `GetMessage` leveszi a `WM_LBUTTONDOWN`-üzenetet az üzenetsorról, és az érkező információkból feltölti a Message-adatszerkezetet.
4. A programunk meghívja a `TranslateMessage`- és `DispatchMessage`-függvényeket, utóbbiban az operációs rendszer meghívja az asztali alkalmazás `WndProc`-függvényét. Ez minden esetben lezajlik, attól függetlenül, hogy az ki van-e fejtve vagy sem.
5. Az ablakos alkalmazásban válaszolunk az I/O-kérésre (például egy gombra kattintva újabb ablakot nyitunk meg) vagy éppen figyelmen kívül is hagyhatjuk, ekkor a felhasználó belátja, hogy lényegében nem is történt semmi.

Már az is Win32-üzenetet vált ki, ha szimplán mozgatjuk az egerünket, ekkor az egér új pozíciója is az üzenetben tárolásra kerül, innen és a Form előre meghatározott tulajdonságaiból (ablak pozíciója, szélessége és magassága) tudjuk detektálni például, hogy az egér az ablak területére érkezett<sup>16</sup>. Ha erre a felhasználói bemenetre fel van készítve a programunk által üzenetküldésre használt metódus<sup>17</sup>, akkor az érzékeli, hogy erre az eseményre reagálnia kell, így egy másik állapotba lép. Természetesen a készített programban lehetőségünk van arra is, hogy egyszerűen ignoráljuk az operációs rendszer felől érkező üzeneteket.

A felhasználó ebből az egész folyamatból csak annyit érzékelhet, hogy a lenyomott gomb hatására valami esemény történt a programban, így ő azt gondolhatja, hogy közvetlenül a program érzékelt az interakciót. Lényegében ez is történik, csak az operációs rendszer végzi az I/O-eszközökről érkező jelek feldolgozását, és erről egy Win32-üzenet formájában tájékoztatja az éppen futó asztali alkalmazást is.

---

<sup>14</sup> A Message Queue (üzenetsor) egy sor (queue) adatszerkezetben, érkezési sorrendben tárolja az üzeneteket, majd ezek ugyanezt a sorrendet megtartva kerülnek le róla.

<sup>15</sup> Az operációs rendszer eleve azért felelős, hogy elossa az erőforrásokat és kezelje a kimeneti-bemeneti perifériákat, így az egerünk által kiadott jel is az operációs rendszerhez érkezik be.

<sup>16</sup> Erre vonatkozó esemény a `MouseEnter`-event Windows Forms esetében.

<sup>17</sup> Egy-egy Win32-üzenet feldolgozását Windows Form asztali alkalmazás esetében a `WndProc`-metódus szolgálja.

## 4.8. Problémák és megoldások

Ebben az alfejezetben a következő, munkánkat időnként meg-megakasztó, kutatómunkát igénylő tényezőkről kívánok szót ejteni.

### 4.8.1. Egyénileg definiált típusok és struktúrák

A Delphis projekt erősen függ az SLDLL-ben megkívánt típusoktól, ezért a teljes kódot nem tudjuk C#-ra átültetni.

#### Megoldás

Erre a problémára megoldást nyújthat akár a marshalling, akár a szerializáció, mi a projektben utóbbi módszert választottuk az előző problémából levont következtetés miatt: ha stringet át tudunk adni, akkor bármilyen típust képesek vagyunk leírni szöveges formában, ezt átadva újból fel tudjuk építeni a másik nyelvben. A C#-os objektumok példányai egy erre dedikált forrásból (JSON-formátumú szövegből) fognak értékeket kapni. A lényege a megoldásnak, hogy szerializálással küszöböljük ki a két nyelv közötti kompatibilitási hiányosságokat, ezért csak olyan adatokkal kommunikálunk, amelyek ismertek, feldolgozhatóak mindkét nyelv számára. Ilyenek a primitívek, az egész, lebegőpontos, logikai és szöveges típusok).

```
[
  { "azonos" : 49156 },
  { "azonos" : 32772 },
  { "azonos" : 16388 }
]
```

4.2. ábra. Eszközök azonosítóinak átadása JSON-formátumon keresztül

### 4.8.2. JSON-formátumra konvertáló függvények hívása .NET keretrendszerből

#### Megoldás:

A korábbi SLDLL-hez fűzzük őket, vagy tegyük egy új DLL-állományba őket, és ezeket is meg kell hirdetnünk a C#-os futtatókörnyezet számára is. A választott megoldás az új DLL-re esett, egy úgynevezett **relayDLL** nevezetű állomány forráskódját megírtam Delphiben, amelyen keresztül az SLDLL-függvényeket meg tudjuk hívni.

Ennek legfőbb indoka az volt, ami eldöntötte, hogy érdemes a hívó (.NETben készült **FormHelperDLL**) és hívott (SLDLL) felek között egy köztes réteget elkészíteni, hogy Delphiben különböző globális változók kapnak értéket, amelyeket nem tudunk máshol definiálni, csakis Delphiben, tehát a hívás környezetét így tudjuk megteremteni, hogy az

értékadások ténylegesen működjenek, ne fussunk `NullReferenceException`-hibákra. A `relayDLL` használatával ellenőrzöttebb módon tudom meghívni az `SLDLL`-ben meghirdetett függvényeket, ebből nem is kell mindent meghirdetni, csak annyit, amennyit ténylegesen a vezérléshez használnunk szükséges. Ezenkívül több, általam definiált hibakódot is vissza tudok adni, amelyek kiváltó okáról `C#`-ban megfelelő kivételek segítségével tájékoztatni tudjuk a felhasználókat.

### 4.8.3. Az `SLDLL_Open`-függvény paraméterezése

#### A probléma leírása:

Az `SLDLL_Open`-függvény várja az ablakos alkalmazás handle-mezőjét, amely az üzenetküldést engedélyezi az `SLDLL`-függvények részéről.

#### Megoldás:

`C#`-ban a `Form`-nak (a függvényeket meghívó félnek) létezik egy `Handle` nevezetű pointer, amit átadhatunk az `SLDLL`-nek.

### 4.8.4. Stringek átadása Delphi DLL-ből `C#` részére

Korábban szakdolgozati témánkat előkészítendő egy példaprojekt keretein belül kísérleteztem bizonyos Delphiből érkező típusok `C#`-nak való átadásával. Ez természetesen kiterjedt a karakterláncokkal való kommunikációra is, mivel ha szöveget képesek vagyunk átadni, akkor lényegében segítségükkel bármilyen más típus formátuma is – így a rekordoké is – leírható és feldolgozható.

Az alábbi példában vizsgáltam a referencia szerinti paraméterátadás működőképességét is, ez abban nyilvánult meg, hogy mind Delphi, mind `C#` oldalon meghirdettem egy kimenő (out) paramétert, ami azt jelzi a fordító számára, hogy ez a változó az eljárás futtatásakor fog beállítódni, ennek értékét – mivel memóriacímet adtunk át – a függvényt hívó környezetből is el tudjuk érni. Ez különösen akkor hasznos, amikor egy függvényhívással több változó értékét is viszont szeretnénk látni.

Alternatív megoldásként mindenképp érdemes megemlítenünk, hogy a visszatérési értékeket egyetlen, valami egységes karakterrel (például pontosvesszővel) elválasztva stringbe is fűzhetjük, vagy éppen egy rekordba csomagolhatjuk, ezekben az esetekben elérhetjük, hogy a ténylegesen egy – összetett – értékkel térhessen vissza.

Kényelmesebb azonban a paraméterekben megadott referenciákba tölteni a visszatérési értékeket, amely referenciákat a hívás helyéről könnyen el tudunk érni a függvényhívás után (mivel eleve onnan kerültek átadásra)<sup>18</sup>.

---

<sup>18</sup> A kimenő paraméterezés ebből a megközelítésből szintaktikai cukorkának (syntactic sugar) számíthat, mivel használata megkönnyíti a fejlesztők munkáját, nem feltétlenül szükséges.

Példának okáért elkészült függvényünk a lefutás sikerességet jelző kóddal tér vissza, emellett beállít egy string változót is, amely értékét szintén szeretnénk a hívás kontextusában elérni. A következő Delphi-eljárás kínákozott működő megoldásként: A C#-ban

```

procedure Welcome(resultStr:PChar); stdcall;
var
  greetings: string;
  name: string;
  school: string[50];
begin
  greetings := 'Hello ';

  write('Please enter your Name: ');
  readln(name);

  write('Please enter the name of your school: ');
  readln(school);
  writeln(greetings, name, ' from ', school);
  resultStr := 'Have a nice day!';
end;

```

4.3. ábra. Delphiben írt eljárás, amely paraméterében egy kimenő string paramétert vár.

ily módon végeztem el az eljárás deklarációját, ebben a példaprogramban ez le is futott, azonban a szakdolgozati projektünkhöz ez a megoldás nem volt megfelelő.

```

[DllImport("MyDLLplus.dll", CallingConvention = CallingConvention.StdCall, EntryPoint = "Welcome")]
extern public static void StringHandling([MarshalAs(UnmanagedType.AnsiBStr)] out string result);

```

4.4. ábra. Az eljárás hibás szignatúrája C#-ban

## Megoldás

```

//converting dev485 to JSON-format - JSON-serializing
function ConvertDEV485ToJSON(out outputStr: WideString): byte; stdcall; external RELAY_PATH;
//converting dev485 to XML-format - XML-serializing
function ConvertDEV485ToXML(var outputPath: WideString): byte; stdcall; external RELAY_PATH;

```

4.5. ábra. A függvény helyes Delphi-deklarációja

Stringek formájában alapértelmezetten a C# marshalling komponense Delphiből PWideChar típusú változókat vár. Amennyiben Delphiben a metódus WideStringet kér paraméterben, akkor azt a C#-oldali marshallerrel a [MarshalAs(UnmanagedType.BStr)] attribútum segítségével tudjuk közölni.

[24]

```
[DllImport(DLLPATH, CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode, EntryPoint = "ConvertDEV485ToXML")]
extern private static byte ConvertDeviceListToXML([MarshalAs(UnmanagedType.BStr)][In] ref string outputStr);

[DllImport(DLLPATH, CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode, EntryPoint = "ConvertDEV485ToJSON")]
extern private static byte ConvertDeviceListToJSON([MarshalAs(UnmanagedType.BStr)][Out] out string outputStr);
```

4.6. ábra. A függvény helyes szignatúrája C#-ban

## 4.9. A relayDLL-ben eldobott kivételek nem kezelhetők

### 4.9.1. Megoldás

Hibakódokkal jelezzük a C# felé, ha esetleg hibára futott valamelyik függvény, ebből a hibakódból a C# tudni fogja, hogy milyen típusú kivételt dobjon az asztali alkalmazás számára, amelyek már lehetnek egyénileg elkészített Exception-objektumok.

## 4.10. uzfeld metódus CSharp-os megfelelője

### 4.10.1. Megoldás

Elméletem szerint létezik ennek egy szabványos megfelelője, C#-ban ez `WndProc`-metódus, ami szintén Message-típusú változót vár referencia szerint átadva, ezzel tud az SLDLL futásidőben kommunikálni. Innentől kezdve:

1. `SLDLL_Open` kivált egy üzenetet (Win32 Message).
2. Ezen az üzeneten keresztül az operációs rendszer meghívja a Form `WndProc` metódusát<sup>19</sup>.
3. Amennyiben a felmérés sikeresen lezajlott, a `WndProc` meghívja a `dev485` és `drb485` változókat beállító `SLDLL_ListElem`-függvényt, mielőtt az `SLDLL_Felmeres`-re kerülne a sor. A megfelelő hívási szekvencia így automatikusan biztosított.

## 4.11. Eszközök azonosítóinak, típusainak átadása

### 4.11.1. Megoldás

A Delphiben tárolt tömb (`dev485`) eszközeinek legfontosabb mezőjét (ami az eszközt azonosítója, tehát az azonosítót XML-fájlba kimentjük vagy JSON-formátumú szöveggé alakítjuk). Az azonosító önmagában meghatározza az eszköz típusát, úgyhogy elég

<sup>19</sup> ezt azért teheti meg, mivel az `SLDLL_Open` hívásakor átadott Form Handle-n keresztül ismeri, hogy merre továbbítsa a rendszerüzeneteket.

azt átadni. [25] Az eszközök típusát a következő módon tudjuk meghatározni kizárólag beállított azonosítóikon keresztül:

- SLLELO: \$4000 -> lámpa azonosítók \$4000 .. \$4fff (decimálisan: 16 384 .. 20 479) tartományon kaphatnak értéket.
- SLNELO: \$8000 -> nyíl azonosítók \$8000 .. \$8fff (decimálisan: 32 768 .. 38 683) tartományon kaphatnak értéket.
- SLHELO: \$c000 -> hangszóró azonosítók \$c000 .. \$cfff (decimálisan: 49 152 .. 53 257) tartományon kaphatnak értéket.

Az eszköz típusának meghatározásában nem játszik szerepet az utolsó 3 bitnégyes, tulajdonképpen a legfelső hex<sup>20</sup> azonosítja a típust. Szigorúan hexadecimálisan nézve: ha egy szám négyjegyű és C-számjeggyel kezdődik, akkor az azonosító hangszórót fog jelölni.

Ez az eljárás a gRPC megoldására hasonlít, amelyet a 3.3 alfejezetben ki is fejtettem, a C# és a Delphi között akár az XML, akár a JSON közös nyelvként (hídként) funkcionál.

## 4.12. A hangszóró egy egész hanglistát kezel - JSON

### 4.12.1. Megoldás

egy hanglistát több index|volume|length 3-as írással, a pipe, azaz '|' karakter mentén széttördelt szövegen 3-as lépésközzel iterálhatunk végig.

## 4.13. A hangszóró egy egész hanglistát kezel - kód

### 4.13.1. Megoldás

A Speaker-osztályban az eszköz viselkedését leíró 3 mezőt (index, volume és length) egy külön objektumba szervezzük ki (Sound), és a Speaker – azonosítóján kívül – kizárólag Sound-típusú objektumok listáját tartalmazza, a listát JSON-ben "*index/volume/length*" rendezett hármassok összefűzésével szerializálja, a listához kizárólag listakezelő függvényekkel lehet hozzáférni (törlés, módosítás, hozzáadás), majd az SLDLL\_Hangkuldes számára kiküldött H-tömböt (hanglistát) Delphiben egy ciklussal a JSON-nel átadott adatok szerint a megfelelő sorrendben feltöltjük, az előző probléma megoldásában ennek mikéntjét fejtettem ki.

---

<sup>20</sup> 4 bit azonosít egy hexadecimális számjegyet – 0-tól F-ig terjedően –, így a bitnégyest hexnek is szokás nevezni.



#### 4.14. Két DLL működésének tesztelése

[3]

#### 4.15. Teljesítmények összehasonlítása Delphi és C#- hívások esetében

[26]

# Összegzés és kitekintés

Két ismert programozási nyelv között megteremthető kommunikáció kiaknázásával elértem, hogy egy olyan szoftver készülhessen el, amelynek segítségével szakemberek képessé válhatnak emberek mentális egészségének fejlesztésére.

Örömmre szolgált, hogy Somodi László minket bízott meg elméleteit kivitelező szoftveres háttérrel. Szakdolgozatunk nem kizárólag arról szól, hogy szakmai tudásunkat gyarapítsuk és mutassuk bizonyítványként egyetemi oktatóink felé, hanem ezzel a projekttel sok ember számára tudunk további segítséget nyújtani magunk legjobb tudása szerint.

Kifejezetten tetszik, hogy munkánkat több tudományág szakértője segítette, így kutatásunk interdiszciplinárisként is jellemezhető, azaz több szakterület, jelesül a mozgáskoordináció, beágyazott rendszerek vezérlése, együttes munkájának az eredményének részesei lehettünk.

A szoftverfejlesztés elméleti és gyakorlati alapjait egyrészt az Egyetem oktatóitól, szakmai és technikai vonulatait Dr. Király Roland Tanár Úrral folytatott konzultációinkon szerzett információk biztosították.

Inspirációnkat, motivációnkat Somodi Lászlóval való beszélgetéseinkből, valamint „*Mozgáskoordináció- és gyorsaságfejlesztő gyakorlatok óvodától a felnőtt korig*” címet viselő könyvében leírt gondolatokból merítettük.

Mint minden szoftverre és emberi termékre jellemző, a megoldásaink természetesen nem mondhatóak tökéletesnek, programunk épp annyira időnként karbantartásra és további fejlesztésekre szorulhat. Érdekes volt felfedezni a gRPC és a Proto-nyelv kapcsán a 3.3 fejezetben, hogy ötször-hatszor jobb teljesítmény érhető el az adatok bináris módon történő szerializációjával a szöveges formátumokhoz képest – jelen állás szerint mégis az utóbbi megoldás számít elterjedtebbnek –, én azt gondolom, hogy mindenképp érdemes lenne munkánkat esetleg egy másik szakdolgozat keretein belül a gRPC irányába elmozdítani.

# Köszönetnyilvánítás

# Irodalomjegyzék

- [1] Srikanth Chakilam. Comprehensive performance bench-marking experiment to compare REST + JSON with gRPC +Protobuf. *LinkedIn*, 2020-03-10.  
<https://www.linkedin.com/pulse/comprehensive-performance-bench-marking-experime>
- [2] V2 Cloud. Client-server application.  
<https://v2cloud.com/glossary/client-server-application>, ismeretlen.
- [3] coolcake (username). How to debug a dll file in delphi.  
<https://stackoverflow.com/questions/594253/how-to-debug-a-dll-file-in-delphi>, 2009-02-27.
- [4] edward (username). What is serialization?  
<https://stackoverflow.com/questions/633402/what-is-serialization>, 2009.
- [5] ELTE. Delphi programozási nyelv. <http://nyelvek.inf.elte.hu/leirasok/Delphi/index.php>, utolsó módosítás: 2014-06-04.
- [6] Dr. Geda Gábor. *Adatszerkezetek és algoritmusok*. EKF, 2013. 88. oldal.
- [7] Jeremy Hillpot. grpc vs. rest: How does grpc compare with traditional rest apis? *DreamFactory*, 2022-11-11.  
<https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-tradition>
- [8] HP. Computer history: A timeline of computer programming languages. *HP*, 2018.  
<https://www.hp.com/us-en/shop/tech-takes/computer-history-programming-languages>
- [9] Inbenta. What is natural language?  
<https://www.youtube.com/watch?v=Ian4sk4VcnA>, 2018.
- [10] InfoLake. Simple explanation of natural language and vs formal language.  
<https://www.youtube.com/watch?v=f9oFvg1YRaI>, 2021.
- [11] Jonathan Johnson. Asynchronous programming: A beginner's guide.  
<https://www.bmc.com/blogs/asynchronous-programming/>, 2020.

- [12] Microsoft Learn. Interoperating between native code and managed code.  
[https://learn.microsoft.com/en-us/previous-versions/visualstudio/windows-sdk/ms717435\(v=vs.100\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/visualstudio/windows-sdk/ms717435(v=vs.100)?redirectedfrom=MSDN), 2013-02-04.
- [13] Microsoft Learn. Default marshalling for strings.  
<https://learn.microsoft.com/en-us/dotnet/framework/interop/default-marshalling-for-strings>, 2022-05-20.
- [14] ParTech Media. Managed and unmanaged code: Key differences. *ParTech*, 2022-09-17.  
<https://www.partech.nl/en/publications/2021/03/managed-and-unmanaged-code---key-differences#>.
- [15] Microsoft. Win32 error codes.  
[https://learn.microsoft.com/en-us/openspecs/windows\\_protocols/ms-erref/18d8f8e8-a967-4f1c-ae50-99ca8e491d2d](https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/18d8f8e8-a967-4f1c-ae50-99ca8e491d2d), 2021.
- [16] Microsoft. Desktop guide (windows forms .net). *Microsoft Learn*, 2022.  
<https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-6.0>.
- [17] Microsoft. Default marshalling behavior. *Microsoft Learn*, 2022-09-17.  
<https://learn.microsoft.com/en-us/dotnet/framework/interop/default-marshalling-behavior#memory-management-with-the-interop-marshaler>.
- [18] David Nield. The brains of teenagers look disturbingly different after lockdown. *Science Alert*, 2022.  
<https://www.sciencealert.com/the-brains-of-teenagers-look-disturbingly-different>.
- [19] Dr. Király Roland. Formális nyelvek és automaták jegyzet.  
[https://aries.ektf.hu/~serial/kiralyroland/download/formalis\\_nyelvek\\_es\\_automatak\\_KR\\_TAMOP\\_20121116.pdf](https://aries.ektf.hu/~serial/kiralyroland/download/formalis_nyelvek_es_automatak_KR_TAMOP_20121116.pdf), 2012.
- [20] Ravikiran A S. Use of c language: Everything you need to know. *SimpliLearn*, 2023.  
<https://www.simplilearn.com/tutorials/c-tutorial/use-of-c-language>.
- [21] Dr. Király Sándor. Szolgáltatásorientált programozás (régi név: Az internet eszközei) elektronikus tananyag. 2. Streamalapú kommunikáció,  
<https://elearning.uni-eszterhazy.hu/course/view.php?id=1128>.
- [22] Wim ten Brink. Which (in general) has better performance csharp or delphi?  
<https://www.quora.com/Which-in-general-has-better-performance-C-or-delphi>, 2019.

- [23] ultraware (username). Delphigrpc github repository, 2018-10-17.  
<https://github.com/ultraware/DelphiGrpc>.
- [24] user1635508 (username). Passing string from delphi to csharp returns null.  
<https://stackoverflow.com/questions/53529623/passing-string-from-delphi-to-c-sharp-returns-null-however-it-works-fine-when-i>, 2019.
- [25] Abdul Salam (username). how to create xml file in delphi.  
<https://stackoverflow.com/questions/8354658/how-to-create-xml-file-in-delphi>, 2012.
- [26] Hidden (username). How to calculate elapsed time of a function?  
<https://stackoverflow.com/questions/16984360/how-to-calculate-elapsed-time-of-a-function>, 2013-06-07.
- [27] Peter (username). What is the difference between serialization and marshaling?  
<https://stackoverflow.com/questions/770474/what-is-the-difference-between-serialization-and-marshaling/770509#770509>, 2009.
- [28] Ruurd (username). Pinvoke: beyond the magic. *devops.lol*, 2014-03-16.  
<https://www.devops.lol/pinvoke-beyond-the-magic/>.
- [29] Tracing (username). Object pascal vs delphi?  
<https://stackoverflow.com/questions/15699788/object-pascal-vs-delphi>, 2012.
- [30] Sruthi Veeraraghavan. 20 most popular programming languages to learn in 2023. *Simplilearn*, 2023.  
<https://www.simplilearn.com/best-programming-languages-start-learning-today-art>
- [31] FreePascal Wiki. Pascal for csharp users.  
[https://wiki.freepascal.org/Pascal\\_for\\_CSharp\\_users#Types\\_Comparison](https://wiki.freepascal.org/Pascal_for_CSharp_users#Types_Comparison), 2023-02-08.
- [32] Wikipedia. Marshalling (computer science).  
[https://en.wikipedia.org/wiki/Marshalling\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Marshalling_(computer_science)), utoljára módosítva: 2023-01-18.
- [33] Wikipedia. C++.  
<https://en.wikipedia.org/wiki/C%2B%2B>, utoljára módosítva: 2023-02-06.

# Ábrák jegyzéke

1.	Valamely demenciával diagnosztizáltak számának emelkedése . . . . .	1
3.1.	Beadandó projektben készített .proto állomány (részlet) . . . . .	14
3.2.	gRPC metódushívásainak típusai . . . . .	16
3.3.	Serializáció teljesítményének összehasonlítása JSON és Proto esetében	16
4.1.	Hibakódok és magyarázataik . . . . .	22
4.2.	Eszközök azonosítóinak átadása JSON-formátumon keresztül . . . . .	24
4.3.	Delphiben írt eljárás, amely paraméterében egy kimenő string paramé- tert vár. . . . .	26
4.4.	Az eljárás hibás szignatúrája C#-ban . . . . .	26
4.5.	A függvény helyes Delphi-deklarációja . . . . .	26
4.6.	A függvény helyes szignatúrája C#-ban . . . . .	27