Humán erőforrás menedzsment szoftver készítése az Ellerton Call Centers Kft. részére

Szakdolgozat

Készítette:
Név: Holics Gábor
Neptunkód: G4DGDN
Szak: Mérnök Informatikus Bsc
Infokommunikációs szakirány
<table>
<thead>
<tr>
<th>Tartalom</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Az Ellerton Call Centers Kft. elvárásai</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Igények felmerülésének okai – jelenlegi „rendszer”, működésének leírása</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Szoftverrel szemben támasztott igények</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Célok meghatározása</td>
<td>4</td>
</tr>
<tr>
<td>2 HR terület ismertetése</td>
<td>4</td>
</tr>
<tr>
<td>3 A Call centerek általában</td>
<td>5</td>
</tr>
<tr>
<td>4 Ellerton Call Centers Kft. ismertetése</td>
<td>6</td>
</tr>
<tr>
<td>5 Virtual Call Center szoftver ismertetése</td>
<td>6</td>
</tr>
<tr>
<td>5.1 Az adminisztrátori felület fontosabb eleme</td>
<td>7</td>
</tr>
<tr>
<td>5.1.1 Projektek kezelése</td>
<td>7</td>
</tr>
<tr>
<td>5.1.2 Statisztikák kezelése</td>
<td>7</td>
</tr>
<tr>
<td>5.1.3 Minőség-ellenőrzés</td>
<td>7</td>
</tr>
<tr>
<td>5.2 Az operátori felület</td>
<td>7</td>
</tr>
<tr>
<td>5.3 Adatbázis API</td>
<td>8</td>
</tr>
<tr>
<td>6 A fejlesztéshez rendelkezésre álló lehetősége</td>
<td>8</td>
</tr>
<tr>
<td>6.1 A Java és a C# összehasonlítása</td>
<td>8</td>
</tr>
<tr>
<td>6.1.1 Hasonlóságok a két nyelv között</td>
<td>9</td>
</tr>
<tr>
<td>6.1.2 Különbségek a két nyelv között</td>
<td>10</td>
</tr>
<tr>
<td>7 Fejlesztéshez használt eszközök, technikák, technológiák</td>
<td>12</td>
</tr>
<tr>
<td>7.1 A .NET [3] [4]</td>
<td>12</td>
</tr>
<tr>
<td>7.3 Visual Studio 2015</td>
<td>14</td>
</tr>
<tr>
<td>7.4 WPF (Windows Presentation Foundation) [5] [8]</td>
<td>14</td>
</tr>
<tr>
<td>7.6 A LINQ to SQL [3] [4]</td>
<td>17</td>
</tr>
<tr>
<td>7.7 A Windows Communication Foundation (WCF) [4]</td>
<td>18</td>
</tr>
<tr>
<td>7.9 MVVM [8] [9]</td>
<td>20</td>
</tr>
<tr>
<td>7.10 Repository minta [10]</td>
<td>22</td>
</tr>
<tr>
<td>7.11 VCC adatbázis API [12]</td>
<td>23</td>
</tr>
<tr>
<td>7.14 MetroChart</td>
<td>27</td>
</tr>
<tr>
<td>8 A fejlesztés leírása, program részeinek magyarázata, technikai megvalósítások</td>
<td>28</td>
</tr>
</tbody>
</table>
8.1 Tervezés ......................................................................................................................... 28
8.2 Modulok fontosabb részeinek bemutatása ...................................................................... 30
  8.2.1 Az adatbázis modul ............................................................................................... 30
  8.2.2 A modell modul ..................................................................................................... 33
  8.2.3 A logikai modul ..................................................................................................... 35
  8.2.4 A szolgáltatás modul ............................................................................................. 43
  8.2.5 A felhasználói felület modul ................................................................................ 44
9 Tervezett fejlesztések és funkciók .................................................................................... 51
10 Szükséges szoftverek és telepítési útmutatójuk ................................................................ 52
11 A HR szoftver használati útmutatója .......................................................................... 54
12 Köszönőnyilvánítás ....................................................................................................... 54
13 Irodalomjegyzék ............................................................................................................ 55
14 Összefoglalás ................................................................................................................ 56
15 Summary ........................................................................................................................ 57
16 Szójegyzék ...................................................................................................................... 58
17 Mellékletek ....................................................................................................................... 58
1 Az Ellerton Call Centers Kft. elvárásai

1.1 Igények felmerülésének okai – jelenlegi „rendszer”, működésének leírása


1.2 Szoftverrel szemben támasztott igények


Általános szoftver igények a gyors, hibamentes működés, korszerű megjelenés, logikus felépítés.
1.3 Célok meghatározása

Egy olyan szoftveres rendszer megalkotása, amely a HR osztály minden igényét kielégíti. Kellő mértékben átláthatóvá teszi a folyamataikat, gyors és dinamikus munkavégzést biztosít számukra.

Egyéni célom a szoftver elkészítésével a fent említettek mellett, hogy új szoftverfejlesztési technológiákat ismerjek meg és sajátítsak el, olyan területekkel foglalkozzak, amivel eddig még nem volt alkalmam. Fejlesztési gyakorlatot szerezzek komplexebb rendszerek felépítésében. Megismerkedjek különböző, már elkészített és szabadon felhasználható modulok saját rendszerbe integrálásával. Egy már meglévő professzionális rendszerrel API kapcsolaton keresztül kommunikáló szoftvert készítek.

2 HR terület ismertetése [14]

A HR (Humán Erőforrás Menedzsment) stratégiailag nagyon fontos a cégek számára. Figyelemmel kíséri a szervezet működését, annak emberi erőforrásra tett hatásait. Egy új munkakör kialakítása, vagy egy meglévő munkakör bővítése, újjáépítése folyamatos részét képezi a személyügyi tevékenységnek. A szervezet hatékonysága és teljesítménye függvényében szükségesessé teszi a vállalat szerkezetének vizsgálatát, az egyes feladatkörök módosítását.

A HR terület a munkavállalók adatainak tárolására, a változások nyomon követésére, adatok kezelésére és elemzésére, speciálisan HR területre fejlesztett szoftvereket használnak. A HR szoftverek a következő főbb modulokat tartalmazhatják, melyeket különböző módokon valósíthatnak meg:

- Toborzás és kiválasztás
- Munkaügyi nyilvántartás
- Beosztás tervezés
- Cafeteria
- Teljesítmény értékelés
- Elektronikus képzések
- Bérszámfelvétés
- Képzésmenedzsment
- Dolgozói elégedettség mérése
3 A Call centerek általában

A call centerek, olyan telefonközponti rendszerek, ahol a telefonos ügyfélszolgálati munkatársak, értékesítők nagy mennyiségű telefonhívást bonyolítanak le. Egyértelmű céljuk, hogy adott idő alatt a lehető legtöbb ügyfelet érjék el, ezáltal növelve a vállalat profitját vagy az ügyfél-elégedettséget. A rendszer alapja egy saját telefonközpont működtetése, de képességei és elvárásai miatt nagyban különbözik egy átlagos telefonközponttól, hiszen itt a hívásokat és az ügyintézőket különféle szempontok alapján mérni kell. Szigorú törvényi feltételeket kell teljesítenie egy call centernek. A hívásokat rögzíteni kell esetleges ügyfél-reklamáció esetére és az ügyintéző munkájának értékelésére.

További elvárás egy call centerrel szemben a saját IVR kialakításának lehetősége, amely egy automata hangbemondó rendszer menüpontokkal és a menüpontokhoz tartozó várakozási sorokkal. Az IVR humán erőforrás optimalizációjának egy megoldása. Hiszen az ügyfél saját maga választja ki, hogy milyen témában szeretne a vállalatot képviselő ügyintézővel beszélni. Ezen téma alapján az automatikus hívásosztó ki tudja választani a legmegfelelőbb, elérhető ügyintézőt, hogy csökkentse a probléma megoldásának idejét. Az IVR működtetése nagyvállalatoknál nagyon költséghatékonyságú lehet, de a rendelkezésre álló eszközök száma és minősége a szállítás minőségére és értékesítésére szolgáló munkanélküliségére is nagymértékben befolyásolja.

A call centerekre is nagy hatással volt az informatika dinamikus fejlődése és az ügyfelügyelt technológia fejlődése. Megjelentek új technológiák melyek új alapokra helyezték a call centereket. Így jöttek létre a contact centerek. A contact centerben a telefonos kommunikáció mellett megjelentek olyan alternatív kommunikációs csatornák, mint a fax, SMS, e-mail, Skype, közösségi média chat funkciói. Így az ügyfél saját maga döntheti el, hogy milyen formában a legkényelmesebb számára a kapcsolatfelvétel. A VoIP technológia megjelenése és az internetes sávszélesség növekedése további lehetőségeket hozott a contact centerekhez. A VoIP egy internetprotokoll feletti hangátvitel, így a kommunikáció nem a telefonhálózaton, hanem az interneten keresztül történik. Ez lehetőséget teremt a virtuális contact centerek kialakítására. Ennek köszönhetően megvalósítható az iparában a távmunka, otthoni munkavégzés, bár ez számos kockázatot hordoz magában. A VoIP-nak köszönhetően a contact centerek üzemeltetése jelentősen csökkent, nem kell drága kommunikációs eszközök beépíteni a telefonhálózatba, hanem elegendő, megfelelő sávszélességű és minőségű internetkapcsolat, jól felépített lokális hálózat és munkaállomások kialakítása az ügyintézők

3
Humán erőforrás menedzsment szoftver készítése

részére. Az ilyen szoftverrendszerek számos, a call center területen elvárt és alkalmazott funkciót implementálnak, hogy még tökéletesebbé tegyék a munkavégzést.

4 Ellerton Call Centers Kft. ismertetése

Az Ellerton Call Centers Kft. (későbbiekben csak Ellerton) 2008-ban alakult meg, olyan szakemberek részvételével, akik évtizedes nemzetközi tapasztalattal rendelkeztek már az alakuláskor is a contact centerek és a hozzá kapcsolódó szolgáltatások tervezésében, kialakításában és üzemeltetésében. Kitűzött célja, hogy partnereit tökéletesen szolgálja ki, ezzel hozzájárulva a magyarországi általános szolgáltatási kultúra folyamatos fejlődéséhez, továbbá olyan professzionális támogatás nyújtása az ügyfelek számára, amely segítségével javíthatják piaci pozíciójukat, növelhetik elégedettségüket és csökkenthetik a csomagolási költségeket.

Az Ellerton Call Centers Kft. (későbbiekben csak Ellerton) 2008-ban alakult meg, olyan szakemberek részvételével, akik évtizedes nemzetközi tapasztalattal rendelkeztek már az alakuláskor is a contact centerek és a hozzá kapcsolódó szolgáltatások tervezésében, kialakításában és üzemeltetésében. Kitűzött célja, hogy partnereit tökéletesen szolgálja ki, ezzel hozzájárulva a magyarországi általános szolgáltatási kultúra folyamatos fejlődéséhez, továbbá olyan professzionális támogatás nyújtása az ügyfelek számára, amely segítségével javíthatják piaci pozíciójukat, növelhetik elégedettségüket és csökkenthetik a csomagolási költségeket.

Ellerton on Call Center

Az Ellerton Call Centers Kft. (későbbiekben csak Ellerton) 2008-ban alakult meg, olyan szakemberek részvételével, akik évtizedes nemzetközi tapasztalattal rendelkeztek már az alakuláskor is a contact centerek és a hozzá kapcsolódó szolgáltatások tervezésében, kialakításában és üzemeltetésében. Kitűzött célja, hogy partnereit tökéletesen szolgálja ki, ezzel hozzájárulva a magyarországi általános szolgáltatási kultúra folyamatos fejlődéséhez, továbbá olyan professzionális támogatás nyújtása az ügyfelek számára, amely segítségével javíthatják piaci pozíciójukat, növelhetik elégedettségüket és csökkenthetik a csomagolási költségeket.

Kitűzött célja, hogy partnereit tökéletesen szolgálja ki, ezzel hozzájárulva a magyarországi általános szolgáltatási kultúra folyamatos fejlődéséhez, továbbá olyan professzionális támogatás nyújtása az ügyfelek számára, amely segítségével javíthatják piaci pozíciójukat, növelhetik elégedettségüket és csökkenthetik a csomagolási költségeket.

5 Virtual Call Center szoftver ismertetése

A Virtual Call Center szoftvert (későbbiekben VCC) a VCC Live Hungary Kft. fejleszti és értékesíti szolgáltatásként. Célja a call center feladatnak, elvárásainak teljes lefedése. A VCC egy komplex üzleti megoldásokat kínáló szoftver, amely beruházás nélkül biztosít professzionális call center hátteret vállalatok és intézmények számára Magyarországon is külföldön egyaránt. Nagy hangsúlyt fektet az adatbiztonságra, a rendelkezésre állásra és a könnyű kezelhetőségre. Adatbiztonságának igazolása végett sikeresen megszerezte a PSI DSS tanúsítványt, mely feljogosítja kiemelten érzékeny adatok tárolására is, olyanokra, mint például bankkártya adatok. A szoftvert felhő alapú technológia felhasználásával fejlesztették, melynek legnagyobb előnye, hogy a működtetéséhez szükséges technikai eszközökről, erőforrásról és távközlési kapacitásról a VCC Live Hungary Kft. gondoskodik, amit az ügyfelek saját igényeik szerint használhatnak.

TODO: eszközök?

5.1 Az adminisztrátori felület fontosabb elemei

5.1.1 Projektek kezelése

- Minden projekthez saját adatbázis tartozik, melyekbe a kezelt ügyfelek listáját lehet betölteni a kívánt adatokkal és lehet kiexportálni onnan.
- Saját telefonszámat, telefonszámokat rendelhetünk a projektekhez, amelyen a hívásokat fogják lebonyolítani.
- Beállítható, hogy adott bejövő hívást mely operátorok kezelhetik, prioritásokat lehet meghatározni az ügyintézők között.
- Meghatározhatjuk, hogy mely ügyintézők léphetnek be a projektbe és ott milyen műveleteket végezhetnek.
- Egy javascript alapú grafikus szerkesztővel adatlapokat hozhatunk létre, amelyen az ügyintézők kezelni fogják a betöltött ügyféladatokat.
- Statisztikai adatot kérhetünk le az ügyintézők munkájáról.

5.1.2 Statisztikák kezelése

- A VCC-ben minden esemény regisztrálásra kerül így nagyon sokrétű statisztikai adathalmaz kérdezhető le.
- Külön statisztikák vannak kimenő, bejövő és összesített hívásirányokra.
- Szünetidőkre és egyéb nem híváshoz köthető státuszokra.

5.1.3 Minőség-ellenőrzés

- Az összes hívás rögzítésre kerül a rendszerben és azok vissza is hallgathatóak.
- Ezekhez minőség-ellenőrző lapokat lehet definiálni, melyek alapján adott hívások kiértékelhetőek.

5.2 Az operátori felület

- az ügyintézők által használt felület
- lehetőségük van projektek közt váltani
- hívásokat indíthatni és fogadni, amennyiben adott projektben van hozzá jogosultságu
- ütemezhetik, hogy adott ügyfelet mikor kell visszahívniuk
• ügyféladatokat tudnak kezelni

A VCC a vállalati folyamatok automatizálása érdekében több integrálási lehetőséggel is rendelkezik, így egyszerűen összeköthető vállalatirányítási és egyéb rendszerekkel. Ezen lehetőségek közül az adatbázis API-t használtam fel az általam készített szoftver megvalósításához.

5.3 Adatbázis API

- Megvalósítása JSON és REST alapú, mely az egyik legnépszerűbb az API technológiákban. A technológia segítségével könnyedén HTTP GET, PUT, POST és DELETE üzenetekkel lehet kezelni a VCC-ben lévő adatbázist. HTTP protokollon keresztül történik a kommunikáció, mely keretein belül egy meghatározott címre történő üzenet küldésével lehet elérni az adatbázist.
- Mivel ezen a csatornán keresztül sok üzletileg érzékeny adat is elérhető, ezért elengedhetetlen a megfelelő hitelesítés. Ennek első lépéseként a VCC szoftverben meg kell adni azt az IP címet, amelyről az adatbázis el szeretnénk érni és generáltatni kell egy jelszót.
- A hitelesítés HTTP Basic Authentication szabvány alapján történik és ezt minden egyes kérésnél meg kell adni. Mivel a HTTP BA nem biztosít titkosítási védelmet a továbbított adatoknak, csupán Base64-ben kódolva továbbítja, ezért választhatók a fejlesztők a HTTPS protokollt a kommunikációra.
- Mikor az adatbázison műveleteket szeretnénk elvégezni, akkor a HTTP kérés fej részének Authorization mezőjébe mindig be kell állítani a BA-ban a felhasználó nevet és jelszót, különben a szerver visszautasítja a kérést.
- A szerver válaszként egy JSON object-et küld a kért adatokkal. Ezt könnyedén fel lehet dolgozni, mivel szabványos.

6 A fejlesztéshez rendelkezésre álló lehetőségek

6.1 A Java és a C# összehasonlítása [1][2][3][4][7]

Ma már alapvetően számos programozási nyelv közül lehet választani, de véleményem szerint e feladat megoldására az objektum orientált nyelvek a legmegfelelőbbek. Ezen belül a két legnépszerűbb nyelv közül választottam, ami alkalmas a desktopos alkalmazások fejlesztésére, a Java és a C#.
6.1.1 Hasonlóságok a két nyelv között

- Mind a két nyelv objektum orientált. Létre lehet hozni osztályokat, az osztályok példányosíthatóak, a példányok elérésekor mind a két nyelvben referencia típuson keresztül történik. A „.” operátorot használva érjük el a tulajdonságok értékét és hivhatjuk meg a különböző metódusokat. Ezen példányok direkt törlését egyik nyelv sem támogatja, mivel mind a kettőnek van saját személyüjtő mechanizmusa.
- Az automatikus személyüjtés mind a két nyelv előnyét képezi. Ez a megoldás kiküszöböli az olyan nyelvek „hibáit”, mint például a C++, ahol a programozónak kell gondoskodni a memória felszabadításáról. Ez a fajta memória menedzsment segíti a programozót és az alkalmazás optimális működését. Figyeli az objektumokra hivatkozásokat, és ha már nincs érvényben lévő hivatkozás, akkor megszünteti őket és felszabadítja a lefoglalt erőforrásokat. Hasznos előnyök, amiket az automatikus személyüjtés biztosít:
  - Nincs memóriaveszteség! Ez egy nyilvánvaló előnye. A programozónak nem kell veszödnie amemória felszabadításával, így nincs elfelejtett felszabadítás, ennek köszönhetően az alkalmazás nem foglal felesleges memóriát.
  - Ösztönzi a fejlesztőket, hogy írjanak minél több objektum orientált kódot. Ennek köszönhetően sokkal átláthatóbb és olvashatóbb kód születik.
  - A személyüjtés könnyebbé teszi az adatok megosztását. Kiküszöböli az olyan helyzeteket, amikor egy objektumot töröl a felhasználó (C++), majd később adatot akar kinyerni belőle. Ez a helyzet sem Java sem C# használata esetén nem fordul elő.
  - A programok segítségével hiba mentesebbé és jobbá válnak. A programozó előtérbe helyezheti a program logikáját és nem kell egyáltalán foglalkoznia a memória menedzselésével.
- Mind a két nyelv típus biztos (Type-Safe). Egyetlen implicit konverzió létezik csak a biztonságos konverzió. Ennek értelmében például egészek tágabb intervallumba konvertálása vagy leszármazott osztályok ősosztályban konvertálása engedélyezett. Nincs implicit konverzió például egészek és logikai típusok között. Ezt már fordításiidőben ellenőrizik a fordítók és kivételt dobok, ha ez a kritérium nem teljesül. A típusbiztoságnak előnye, hogy a fejlesztőt arra készteti, hogy helyes kódot írjon, és egyben segíti a létrehozott szoftver biztonságosabbá válását.
A Java és a C# is úgynevezett „tiszta” objektum orientált nyelv. Minden létrehozott osztály ösosztályára az object osztály. Azt, hogy mitől „tiszta” objektum orientált egy nyelv igen szubjektíven lehet megítélni.

- Véleményem szerint az alábbiaktól:
  - Lehetővé teszi osztályok létrehozását
  - Létezik az örökloidés és az interface implementálás
  - Minden osztály közös őse ugyan az, az object
  - Az örökült metódusok felülsorolására lehetséges
  - Lehetővé teszi a hozzáférések szabályozását
  - Kezeli a túlterhelést operátorokat
  - Kellően típus biztos
  - Jól kezelhetőek benne a kivételek


Beépített szál és szinkronizáció támogatás. Mind a két nyelv támogatja e funkció megvalósítását a saját nyelvi megkötnései között. Ez a megoldás lehetővé teszi a fejlesztők számára, hogy több szálas alkalmazásokat hozzanak létre. Ezzel a megoldással egy olyan munkautasítás, mely hosszú időt venne igénybe egy háttérszálban megtalálható, ez által a felhasználók nem feltétlenül kell megvárnia a folyamat végén. Illetve a működés is gyorsítható vele.

Unicode támogatása. Ennek fontossága a nemzetközi nyelvek támogatása. Nem kötelezi a fejlesztőt arra, hogy csak angolul írhat a forráskódba. Ennek egyértelműk az előnyei, de tartalmaz hátrányokat is, hiszen ha egy nyelv specifikus forráskódban (például magyar) kell módosítást végeznie egy olyan fejlesztőnek, aki nem ismeri a nyelvet, akkor ez a tulajdonság negatív hatású lesz.

6.1.2 Különbségek a két nyelv között

Bár nagyon sok mindenben hasonló a két nyelv, hiszen az egyik alapját képezte a C# megalkotásakor a Java nyelv, de vannak különbségek is.

A Java platform független. Kétség kívül ez az egyik fő erénye. A legnépszerűbb operációs rendszerek mindegyikén fut (Windows, Unix, Linux, MacOSX). Igaz léteznek fordítók, melyek segítségével a Windowsra írt C# kódot is le lehet fordítani más operációs rendszerre, ilyen például a Mono C# fordítója, aminek segítségével Linuxra fordíthatjuk kódunkat, de ez a megoldás messze nem tőkéletes. Az alkalmazás nem lesz olyan stabil és biztos, mint Windowson ugyan ez a kód. Ráadásul a .NET-ben vannak olyan OS specifikus csomagok, melyeknek nincs megfelelője Linuxos környezetben, így ezek a csomagok nem használhatóak abban az alkalmazásban, amit monoval szeretnénk fordítani.

Kulcsszavak különbsége. Az alábbi kulcsszavak értelmezettek C#-ban, de Java-ban nem: + as: Az „as” bináris kaszt operátor. Ez egy előzetes vizsgálati lehetőség kasztoláskor. Futásidőben el tudja dönteni, hogy adott típus kompatibilis-e a másikkal. Inkompatibilitás esetén null értékkel tér vissza. Pl.: TestClass tc2 = tc1 as TestClass;
+ decimal: Adattípus, mely meghatároz egy 128 bites számot.
+ delegate: Lényegében egy metódusreferencia. Úgy kell létrehoznunk, hogy a visszatérési értéke és a paraméter szignatúrája megegyezzen a metóduséval, amelyre mutatni fog. Például: public delegate int DelegateOp(int a, int b); Ez a delegate bármilyen metódusra mutathat, melynek egész a visszatérési értéke és két egész paramétert vár. Amikor a fordító, ilyen metódusreferenciát dolgoz fel, létrehoz egy lezárt (saled) osztályt a System.MulticastDelegate osztályból. Ez nyújtja a szükséges „infrastruktúrát” a metódusreferenciának, a később hívható metódusok listájának tárolásához.
+ event: Biztosítja, hogy a fejlesztő saját eseményeket hozzon létre.
visszatérés előtt. Tehát a hívótól kapott paraméter értékét felülírhatja a hívóban. Természetesen van még különbség a felsoroltakon kívül is, de igyekezem a leglényegesebbeket megemlíteni.

7 Fejlesztéshez használt eszközök, technikák, technológiák

7.1 A .NET [3] [4]
A Microsoft a Javat kibővítette Windows specifikus függvényeivel és szolgáltatásaival, viszont ezért a Java jogait birtokló Sun Mikrosystem beperelte, mert a platform függetlenségi alapelvét sértette. Ezért a Microsoft saját keretrendszer fejlesztésébe kezdett és ez a keretrendszer a .NET. A keretrendszer hivatalos nyelveként adták ki a C# első verzióját. Létrehozásakor nagy hatással volt rá a Java, C++, Object Pascal. Mindent tud, amit egy objektum orientált nyelvtől el lehet várni. A .NET-nek három fő építőeleme van:

- CLR - Közös nyelvi futtatórendszer: Felfogható úgy, mint futtató réteg.
  - Feladata:
    - kezelje a .NET típusokat
    - memória kezelése
    - alkalmazástartományok létrehozása
    - végrehajtási szálak kezelése
- CLS - Közös nyelvi specifikáció: Definiálja azoknak a közös típusoknak és programozási szerkezeteknek a részhalmazát, amelyeket minden. NET programozási nyelv elfogad.

A .NET alaposztály könyvtárai definiálnak számos típust, amelyek megkönnyítik az adatbázisok manipulációját, felhasználói felületek kialakítását, fájl műveleteket, szálkezelést, stb.
7.2 A C# [3] [4]

A C# alapvető szintaxisa nagyon hasonlít a Java szintaxisára, de nem tekinthetjük a Java egyik „leszármazottjának”. A C# támogatja a tulajdonságok létrehozását, lehetővé teszi az operátorok túlterhelését, visszahívható függvények létrehozását metódus referenciáik révén. A C# meghatározó tulajdonságai a teljesség igénye nélkül:

- Nincs szükség mutatókra.
- Automatikus memória kezelés a személyjegyző segítségével. Az objektumok nem szabadithatóak fel közvetlen módon, csak a személyjegyző teheti ezt meg, amikor már nincs rájuk hivatkozás. Ennek a módszernek a segítségével elérhető, hogy ne legyen hivatkozás már nem létező objektumra.
- A C# nyelv kizárólag egyszeres öröklődést támogat, de egy osztály megvalósíthat több interfacet is.
- Generikus típusok és tagok. Használatukkal hatékony és típusbiztos forráskódot lehet létrehozni.
- Névtelen metódusok támogatása.
- Egyetlen típus több kódfájlon keresztüli definiálásának képessége.
- Különféle adatformátumokkal való kommunikációhoz szükséges lekérdezések támogatása (LINQ)
- Bővitő metódusok, melyekkel létező metódusok funkcionalítását bővíthetjük ki.
- Labda operátort bevezetése, amely leegyszerűsíti a .NET metódusreferencia-típussal végzendő programozást. (=>)
- Objektuminicializáló szintaxis, amely lehetővé teszi a tulajdonságok beállítását az objektum létrehozásakor (elég csak egy, paraméter nélküli konstruktor).
- Tulajdonságok (Properties) használatának támogatása.

A C# forrásfájlok fordítás után bináris fájlok keletkeznek (.dll vagy .exe). Ezek platformfüggő utasítások helyett platform független köztes nyelvet és típusmetaadatokat tartalmaznak. A metaadatokat a fordító generálja, amelyek általános adatokat tartalmaznak az adott osztályról.
7.3 Visual Studio 2015

A Visual Studio a Microsoft vezető integrált fejlesztői környezete. 1997-benadták ki az első verzióját. 2002-ben mutatták be a 7.0-ás verzióját, ez azért különleges, mert ezzel egyidejüleg debütált a .NET Framework és a C# 1.0 is.


A szoftver felépítését tekintve olyan, mint a már megszokott fejlesztői környezetek.

7.4 WPF (Windows Presentation Foundation) [5] [8]

A WPF megalkotásának egyik fő célja az volt, hogy a már meglévő Windows Forms-ot és a hozzá kapcsolódó API-k sokaságát leváltsa. Windows Forms segítségével vezérlőkkel ellátott űrlap felületek készíthetőek alapvetően. Amennyiben 2D grafikát szeretnénk már a GDI+(System.Drawing.dll) szükséges. A 3D grafikai támogatás a DirectX API-kon keresztül érhető le. Ezen technológiák teljesen más-más gondolkodásmódot igényelnek, még akkor is, ha szintaktikai szempontból hasonlóaknak tűnnek. Tehát a WPF egyesített objektummodellben ötvözi ezen egymáshoz szorosan nem kapcsolódó API-kat.

A WPF egyik legnagyobb előnye, hogy a programozás logikája teljesen különböztetheto a megjelenéstől. Ennek alapját az XAML nyelv biztosítja, mely XML alapokon nyugszik. Másik nagy előnye még, hogy a megjelenítés szolgáltatásait DirectX motor kezeli, amely sokkal hatékonyabb és gyorsabb renderelést biztosít, mert hardveres támogatással rendelkezik. További hasznos tulajdonságai felsorolás szinten:

- Különféle elrendezés kezelők (sokkal több, mint elődjének a Windows Forms-nak), melyek segítségével rugalmas vezérlést biztosítanak.
- Fejlett adatkötési lehetőségek, mellyel az adatok különféle módon köthetőek a vezérlőelemekhez. Ennek két fő módja az esemény alapú (evets) és a kötés alapú (Binding).
• Rendselkezik stílusmotorral, segítségével saját témákat definiálhatunk WPF alapú alkalmazásainkhoz. Ezeket rugalmasan lehet kezelni és újrafelhasználni.
• Alkalmazza a vektorgrafikát, mely segítségével a képernyő méretéhez és felbontásához automatikusan igazítható a megjelenítendő tartalom.
• 2D, 3D grafika támogatása, videó lejátszás és hanglejátszás támogatása
• Együttműködés korábbi GUI modellekkel, mint például Windwos Forms

A WPF API segítségével többféle felhasználói felület készíthető:

• Hagyományos végrehajtó szerelvény, mely a helyi gépen fut: Ez lehet egyszerű szövegszerkesztő, adatbázis kezelő, célszoftver, multimédiás alkalmazás, stb. Ezek az alkalmazások fordítás után *.exe formátumú fájlok formájában jönnek létre és a megszokott módon telepíthetők.
• WPF alkalmazást létrehozhatunk navigáció alapú szerkezettel is: Ezzel egy hagyományos asztali alkalmazást egy webböngésző-alkalmazás funkcióit is végrehajtható. Ez a módszerrel létrehozott alkalmazásban a felhasználó „előre” és „vissza” gombok segítségével lépkedhet a különböző felhasználói felületek között. Ebben a szerkezetről egy adott felhasználói felület (page) nevezzünk. Ilyenkor az alkalmazás nyilvántartja az oldalakat, biztosítja az infrastruktúráját és az oldalak közötti kommunikációát, adatátvitelt, valamint kezeli és karbantartja az előzménylistát. Természetesen ettől még ez nem lesz egy webes alkalmazás. Ez a szerkezet mindenki számára ismerős, aki használ Windows operációs rendszert. Ezen szemléleten alapul a Windows 7 intézője is, mint azt az 1. ábra mutatja.
1. ábra

- **XBAP (XAML Browser Application)** avagy webböngészőben hosztolható alkalmazás: Ezen modell használata esetén, ha a felhasználó a hosztolt alkalmazás URL-jére érkezik, akkor egy XBAP-alkalmazás a háttérben letöltődik és települ a felhasználó számítógépére. Ezen alkalmazások fordítás után .xbap kiterjesztésű fájlra fordulnak. Ennek a felületnek nagy hátránya a kompatibilitás! Másik hátránya, hogy csak korlátozott módon férnek hozzá a rendszer erőforrásaihoz és nem minden .NET szerelvényt használhatunk fel benne. Ezekben kívül külön ablakok, akár párbeszéd panelek nyitására sincs lehetőség.

- **Silverlight alkalmazások**: különböző operációs rendszerekkel hosztolható böngésző alapú alkalmazások. Segítségével interaktív és sokoldalú webes alkalmazások készíthetőek. Ez a megoldás inkább egy kooperáció a WPF-fel mintsem a WPF része.

### 7.5 Az Entity Framework [3] [4]

Az Entity Framework a Microsoft megoldása az objektumrelációs leképzésre (ORM). Az alábbi előnyökkel jár az ORM megoldások használata (a felsoroltak közül nem mind Entity Framework specifikus):

- **Absztrakció**: Az adatainkat általában relációs modellekben tároljuk, melyek tárolásra és gyors visszakeresésre lettek optimalizálva, de ezt a formátumot sokszor nehéz leképezni objektumaink adatmodelljére. Az ORM megoldások fogalmi
megfeleltetései lehetővé teszik számunkra, hogy az entitásaink és azok tulajdonságai eltérjenek a fizikai tárolási módjuktól.

- **Kódgenerálás:** Az ORM megoldások hatalmas előnye, hogy biztosít a fejlesztő számára CRUD funkcionalitást. Így ezeket nem kell megirni minden objektum esetére! Mivel a kódot generálni fogjuk, így itt nem kell a hiba lehetőségével sem számolni, ezzel is csökkentve a fejlesztésre fordított időt. Ezen megoldás segíti az elnevezési konvenciók betartását is, ugyanis az osztályok és adattagok is automatikusan generálódnak. Egyetlen hátrányaként említhető, hogy az adathozzáférés többlet erőforrást igényel ORM megoldás használata esetén, mint ha közvetlenül az adatbázissal dolgoznánk! Viszont ez a minimális teljesítményveszteség csak nagyteljesítményű alkalmazásoknál jelentkezik. Általánosságban elmondható, hogy jóval több előnyt rendelkezik, mint hátránnyal.

- **Különböző adatbázisok támogatása:** Az Entity Framework nem csak a Microsoft által fejlesztett MsSQL-t támogatja, hanem például az Oracle, MySQL, SQLAnywhere, stb. adatbázisokkal is képes együttműködni. Olyan saját lekérdezőnyelvvel és mechanizmussal rendelkezik, amely képes együttműködni bármely SQL nyelvvel. Ezáltal biztosítja a fejlesztő számára, hogy nem kell minden különböző SQL nyelvet ismernie, hiszen az SQL nyelvet az Entity Framework „ismeri”.

- **Tervezésidejű támogatás:** Az Entity Framework hatékony és előnyös tervezésidejű támogatást biztosít. A Visual Studio-ban grafikus felületen engedi megtervezni az adattáblákat, a mezőket és a táblák közt kapcsolatokat. Ezt hatékonyan lehet használni így kétségtelen, hogy az egyik legelőnyösebb szolgáltatása az Entity Frameworknek.

- **A LINQ to Entities segítségével**, olyan összetett lekérdezéseket készíthetünk a .NET keretrendszerrel, amelyet a normál SQL nyelven egyáltalán nem, vagy csak nagy nehézségek árán lehetne létrehozni.

### 7.6 A LINQ to SQL [3] [4]

A LINQ to SQL egy API, ami lehetővé teszi jól megformázott LINQ lekérdezés-kifejezések alkalmazását relációs adatbázisokban lévő adatokra. Ez az API több olyan típust biztosít, amelyek megkönnyítik a kommunikációt a kód és az adatbázis-motor között. Fő célja, hogy konzisztenciát biztosítson a programozási logika és a relációs adatbázisok között. Ennek lényege, hogy egy adatbázis-lekérdezést ne hatalmas string-ek reprezentáljanak az
alkalmazásunkban, hanem erősen típusos LINQ lekérdezéseket használjunk. Előnyként kezelhető még az a tény, hogy a relációs adatbázisból származó rekordokat nem rekordokként, hanem szabványos objektumorientált programozási módszerekkel kezelhetjük. Mivel a LINQ to SQL segítségével az adatelléreést integrálhatjuk a C# forráskódból, így nem kell manuálisan létrehozni egyedi osztályok sokaságát. Ezen API használata esetén nem kell az ADO.NET-ből ismert SqlConnection, SqlCommand vagy az SqlDataAdapter típusokat használni. Egyszerűen a LINQ lekérdezés-kifejezésekkel, entitásosztályokkal és a DataContext típus segítségével megvalósíthatjuk az alap adatbázis-műveleteket (insert, update, delete), valamint definiálhatunk tranzakciós kontextusokat, létrehozhatunk adatbázis-entitásokat, akár adatbázisokat is, meghívhatunk tárolt lejárasokat, lekérdezhetünk előre megírt view-kat, stb. Ezekben felül a LINQ to SQL API fő feladata még, hogy megvalósítsa a standard ADO.NET adattípusok integrációját.

7.7 A Windows Communication Foundation (WCF) [4]

A Windows Communication Foundation API-t, avagy a WCF-et kifejezetten elosztott rendszerek készítéséhez fejlesztették. A WCF egyszerű és egységesített továbbá kiterjeszthető programozási objektummodellt kínál, mely sok korábbi már létező technológiával működik együtt. Ha például olyan alkalmazást szeretnénk fejleszteni, ahol minden kapcsolódó számítógép egy lokális hálózatban található és az összes gépen Microsoft Windows fut, akkor a WCF segítségével TCP protokoll alapú kommunikáció a leghatékonyabb. Viszont, ha olyan szolgáltatást szeretnénk, aminél nem számít, hogy a kapcsolódó kliensek helyileg hol találhatóak és hogy milyen operációs rendszer működteti őket továbbá, hogy a kapcsolódni kívánó alkalmazás milyen programozási nyelven íródott, akkor a WCF segítségével XML webszolgáltatás alapú protokollt is készíthetünk. Mivel a WCF lehetővé teszi az alkalmazásunknak megfelelő protokoll kiválasztását, nagyon könnyű az elosztott alkalmazás alapjául szolgáló összeköttetések beállítása.

A WCF-et jellemzi még az erősen típusos és típus nélkül üzenetek támogatása is. Ezen támogatás révén a .NET alkalmazások hatékonyan meg tudják osztani egyedi típusaikat. A más platformon készült szoftverekkel pedig a gyengén tipizált XML alapú adatokat. Többféle kötést támogat, így lehetővé teszi a legmegfelelőbb választását az üzenetek szállítására (pl.: HTTP, TCP, MSMQ és named pipe-ok). Teljesen integrálta biztonsági modellt is tartalmaz, mely magába foglalja a Win32/.NET natív biztonsági protokolljait és más semleges webszolgáltatási szabványra épülő biztonsági módszert is.
Támogatja a munkamenetszerű állapotkezelési módszereket és az egyirányú, állapot nélküli üzeneteket is. Ezen funkcionalitásokon felül nyomkövető, naplózó, teljesítménymérő, tranzakció támogató funkciókat és sok más hasznos lehetőséget is tartalmaz.

A WCF nagy előnye, hogy a szolgáltatási orientált architektúrá (SOA) által megalapozott tervezési elveken alapul. A SOA egy olyan módszer elosztott rendszerek építésére, ahol több autonóm szolgáltatás működik együtt az által, hogy üzeneteket küldenek egymásnak jól definiált interfészek segítségével, egy gépen futó különböző alkalmazások vagy akár hálózatba kötött gépek között is. Egy szolgáltatás interfész leírja a külső hívók által hívható tagok halmazát. A SOA által meghatározott fő tervezési elvek:

- **Explicit határok**: Ennek az elvnek a lényege, hogy a WCF szolgáltatás funkcionalitása jól definiált interfészeken keresztül jelenjen meg. A hívó egyedül az interfészen keresztül tud kommunikálni a szolgáltatással. Ez a szerkezet elrejti a szolgáltatás implementációs részleteit.

- **Minden szolgáltatás autonóm**: Minden WCF szolgáltatást, amennyire lehetséges egyedi entitásként kell kezelni. Teljesen függetlennek kell lennie mindentől. Ahhoz hogy ezen alapelve támogatás megvalósuljon az interfész alapú programozásnak egy pontos pontját nem szabad elfelejteni. Ha egy interfész már működésbe lépett, akkor nem szabad változtatni rajta. Ha egy WCF szolgáltatást szeretnénk kibővíteni, akkor egyszerűen új szolgáltatás kell definiálni, nem pedig a meglévő szerkezetét módosítani, mert ez a már meglévő szolgáltatáshoz kapcsolódó alkalmazások működését veszélyeztetné!

- **A szolgáltatásoknak szerződéseken keresztül kell kommunikálni és nem implementáció keresztül**: Ez az elv is köthető az interfész alapú programozáshoz, mert a WCF szolgáltatást hívót nem érdeklik a szolgáltatás implementációs részletei. A hívók kizárólag a WCF szolgáltatás elérhető, nyilvános interfészein keresztül kommunikálnak a szolgáltatásokkal. Ezen kívül, ha a szolgáltatás interfésztagja egyedi összetett típust tesz elérhetővé, akkor ezeket egy adatszerződésben részletesen le kell írnia, hogy a hívó biztosan leképezhesse a tartalmat.

- **A szolgáltatás kompatibilitása házirenden alapuljon**: A CLR-interfészek erősen típusos szerződéseket tartalmaznak minden WCF ügyfélhez és a választott kötésen alapuló WSDL dokumentumot lehet generálni hozzá. Viszont az interfész és a WSDL együtt nem elég kifejező, hogy meghatározza a szolgáltatás határait. Ezért a SOA segítségével úgynevezett házirendeket definiálhatunk, melyek minősítik a
szolgáltatás szemantikáját. Ezekkel a házirendekkel elválasztjuk a szolgáltatás szemantikai leírását a hívás módjának a leírásától.

7.8 A JSON [11]

A JSON vagy JavaScript Object Notation emberek számára is olvasható formátumú struktúrált adattárolásra és adatközvetítésre alkalmazható szabvány. A JavaScriptből alakult ki, de ennek ellenére programozási nyelv független és nagyon sok nyelvhez van értelmezője. A formátumot az RFC4627 szabványban specifikálták először. A JSON fájlok kiterjesztése „.json”. A JSON-t a XML egyik alternatívájaként foghatjuk fel. Legtöbbször szerver és kliens számítógépek közötti adatátvitelre használják legtöbbször AJAX technológiával közösen. A JSON-nak két típusú felépítése lehetséges:

- Név – érték párok halmaza: megvalósításai az objektumok, rekordok, szótár, asszociatív tömbök lehetnek a különféle nyelvekben.
- Értékek rendezett listája: megvalósításai a tömbök, vektorok, listák, stb. A megvalósítási lehetőségek olyan általános adatszerkezetek, melyek szinte kivétel nélkül megtalálhatóak minden programozási nyelvben valamilyen formában. Ezért tökéletes választás lehet alkalmazások közötti kommunikációra!

7.9 MVVM [8] [9]

Az MVVM a Model-View-ViewModel rövidítése. Ez egy softver tervezési minta, mely nagyobb WPF megjelenítést használó alkalmazások esetében ajánlott. Azt alkalmazások fontos részét képezi a megjelenés, hiszen a felhasználó ezen keresztül végzi el feladatait. Egy modern, letisztult, de mégis egyszerű felhasználói felülettel rendelkező alkalmazással, sokkal szívesebben dolgozik mindenki. Az MVVM tervezés segít elválasztani az üzleti logikát a felhasználói felülettől és a felülethez kapcsolódó megjelenítési logikától. Ezen szétválasztás segítségével, sokkal áttekinthatóbb és megoldani számos fejlesztési és tervezési problémát és az alkalmazás könnyen ellenőrizhetőbbé, karbantarthatóbbá válik általa. Javítja a kód újrafelhasználhatóságát és a felület egyszerű cserérhetőségét. Amennyiben külön fejlesztői csoport dolgozik az alkalmazás logikán és a felhasználói felületen (UI), akkor e séma alkalmazásával a közös munka kialakítása és megosztása is sokkal könnyebb és rugalmasabb. Élesen szétválasztható általa a két terület munkája. Az alap mintában az alkalmazás három fő csoportra van osztva, de ez nem jelenti.
Humán erőforrás menedzsment szoftver készítése

azt, hogy ne lehetne további csoportokat beiktatni a fejlesztés során, mint például külön egység a WCF megvalósításra, vagy adatbázis kapcsolat és a hozzá kapcsolódó modellek, logikák, műveletek egysége. Ezen fő csoportok:

- **Model (Model):** Itt definiálhatjuk az alkalmazás logikáját. A felhasznált modelleket és munkaosztályokat. A modell osztály közvetlenül nincs kapcsolatban a nézetmodellel, nincs függőség, hogy a modell hogyan hajt végre egy feladatot.

- **Nézet (View):** ebben a logikai egységben a felületeinket valósítjuk meg. Itt találhatóak a nyomógombok, szövegdobozok, dátumválasztók definíciói. Alapvetően a tartalma a *.xaml fájlokra korlátozódik. Minden nézetnek kapcsolódia kell a neki létrehozott nézetmodellhez. Ezt a kapcsolatot a nézet DataContext tulajdonságának beállításával lehet létrehozni. A nézet határozza meg a nézetmodell viselkedését, például animációt indíthat, állapot változásokat közöl, adatokat frissít.

- **Nézetmodell (ViewModel):** ebben az egységben követhetjük nyomon, a felületen történő és a modellben történő változásokat. Tulajdonképpen a viewmodel kapcsolja össze a másik két modellt. Ez egy nem vizuális osztály, hanem egy alap C# osztály, mely nem származik semmilyen WPF osztályból. Megvalósítja a tulajdonságokat (properties) és parancsokat (commands), melyek a nézethez kötődnek (binding).

A 2. számú ábra szemlélteti a három fő csoportot és azok kölcsönhatásait:

![Diagram](https://i-msdn.sec.s-msft.com/dynimg/IC448690.png)

2. ábra Forrás: https://i-msdn.sec.s-msft.com/dynimg/IC448690.png (2017.03.26)

- **Notifications:** Az értesítések (Notifications) fontos részét képezik a tervezési mintának és ennek a programozási logikának. Az értesítések megvalósításához a viewmodelnek implementálnia kell az INotifyPropertyChanged interfészt. Az INotifyPropertyChanged interfészt megvalósító osztály segítségével nyomon követhetőek az állapotváltozások és a különböző események a felhasználói felületen
és a lentről jövő adatváltozások is. Ez által például, ha egy property értékét megváltoztatja az üzleti logika egy szála, akkor ez a változás azonnal megjelenik a felhasználói felületen is.

- **DataBinding:** Az adatkötés (DataBinding) nagyon fontos szerepet játszik az MVVM mintában. Úgy kell az osztályokat kialakítani, hogy támogassák az adatkötéseket, ez azt jelenti, hogy implementálniuk kell a megfelelő interfészeket (például az INotifyPropertyChanged tulajdonságok esetén vagy az INotifyCollectionChanged gyűjtemények, listák esetén). Az adatkötés több módú lehet! Egyirányú mikor például csak megjeleníteni szeretnénk valamit és többirányú, mikor a felhasználó módosíthatja a megjelenített értéket, vagy újat adhat meg.

- **Commands:** A felhasználó által a felületen kiadott parancsokkal adatokhoz való hozzáférést szabályozhatják, egy vagy több műveletet hajthatnak végre. Egy-egy parancs jellemzően egy vezérlőhöz van kötve (binding) például egy gomb megnyomása esetén elvégzendő művelet(ek)et definiálja. Az esemény alapú (events) programozás utódjaként tekinthetünk rá. Ez a módszer egy kényelmes megoldást biztosít a műveletek végrehajtásának szabályozásában és a végrehajtáshoz kötődő ellenőrzésekhez, például bizonyos mezők kitöltése kötelező és csak, ha ezen mezők töltöttek, akkor lehet a gombot megnyomni. A vezérlő eszközhez csak akkor köthető egy parancs, ha megvalósítja az ICommand interfészt. Ennek legegyszerűbb módja, ha egy olyan osztályt hozunk létre, mely implementálja az ICommand interfészt és ez a típus lesz a command property típusa. Ezt a property-t kötjük a vezérlőhöz és mikor el akarjuk kérni az értékét, akkor hívjuk a megfelelő metódust vagy metódusokat.

7.10 Repository minta [10]

Sok alkalmazásban az üzleti logika közvetlenül hozzáfér az adatbázishoz, direkt műveleteket végez rajta. Ez a megoldás az alábbi problémákhoz vezethet:

- Sokszor ismétlődő kódrészlet
- Nagyobb esély a programozási hibára
- Egy-egy adatbázis változás több helyen is módosítást igényel a kódban ezzel is növelve a további hibák lehetőségét
- Nehéz a logika izolált tesztelése az adatbázis függőségek miatt
Ezen hibák és kellemetlenségek elkerülése végett érdemes használni a repository mintát. Ennek segítségével egy különálló réteget hozunk létre az adatbázis és az üzleti logika között. Segítségével egységesíthetjük a hozzáférési szabályokat és logikát. Javíti a kód karbantartást és a kód olvashatóságát. A megoldás lényege, hogy itt összpontosul az adathozzáférés logika és ez a réteg biztosítja az alkalmazás rugalmasságát. A megvalósítás úgynevezett fekete dobozos terv, hiszen az üzleti logika nem tud és nem is kell tudnia arról, hogy milyen adatmodellek léteznek az adatbázisban. A repository mindig az üzleti logika igénye szerinti modellt ad vissza, ezzel is teljesen elválasztja a két réteget egymástól. A 3. számú ábra jól szemlélteti ezeket.


7.11 VCC adatbázis API [12]

A Virtual Call Center szoftver által biztosított API felület a rendszer adatbázisának lekérdezésére és manipulációjára. A megvalósítása REST és JSON alapú, melyek népszerű API technológiának számítanak. Ezen technológiák segítségével könnyen lehet a VCC adatbázisával kommunikálni HTTP GET, PUT, POST és DELETE üzenetekkel. A HTTP kérések testében JSON típusú struktúrában küldjük és fogadjuk az adatokat. Az adatbázis API erőforrás három részből áll:

- Protokoll meghatározás: https://
- Domain: [ügyfél].asp.virtual-call-center.eu
- Paraméterek megadás: /v2/[használni kívánt erőforrás]

Az ügyfelet a VCC szerződőskötéskor megadja. Biztonság szempontjából kiváló megoldást hoztak létre az adatok védelmének érdekében. Csak előre a VCC szoftverben megadott IP címről fogad HTTP üzeneteket az API és csak az ott megadott jogosultságoknak
megfelelőeket. Ennek folyamata, hogy a megfelelő menüpontban generálni kell egy tokent. Ez a token fogja tartalmazni az IP címet és azon erőforrások listáját, amihez az adott IP-ről hozzá lehet férni. Ezen felül generál egy jelszót is a tokenhez, ezt kell használni a kapcsolódás során. A hitelesítés alapja a HTTP Basic Authentication. Erre a hitelesítési formára a C# több megoldást is biztosít. Minden adatbázis API kérés egy egyedi erőforrás és a HTTP metódusnak tartalmaznia kell a 4 módszerből egyet (GET, PUT, POST, DELETE). Minden kérésnek az alább módon kell felépülnie:

- HTTP metódus meghatározása
- Erőforrás meghatározása
- Opcionális paraméterek megadás
- HTTP body-ban megadott JSON objektum (PUT és POST metódus esetén, egyéb esetben üres)

Válasz üzenetben a HTTP body-ban egy response objektum szerepel, melyben egy JSON objektumban kapjuk meg a kért adatokat.

7.12 MsSQL szerver 2016 [6]

A Microsoft SQL szerver egy relációs adatbázis-kezelő rendszer, amit a Microsoft fejlesztett ki. Első verziója 1989-ben jelet meg. A szerver csomag számos szolgáltatást foglal magába, melyek feladta az on-line tranzakciófeldolgozás, az on-line elemző folyamatok, az adatbányászat, az adatcsere, adatintegráció, jelentéskészítés és nem utolsó sorban az alkalmazásfejlesztés támogatása. Főbb szolgáltatásai a következők:

- Analysis Services – Multidimensional Data (Elemzési szolgáltatások – Többdimenziós adatok): Ez egy OLAP megoldás, mely lehetővé teszi a többdimenziós adatbázisok létrehozását és teljes körű kezelését, amelyek számos különböző adatforrásból származó aggregált adatokat tartalmaznak.
- Analysis Services - Data Mining (Elemzési szolgáltatások – adatbányászat): Támogatást nyújt az adatbányászati modellek alkalmazásában a különböző ipari szabványként ismert adatbányás algortmusok felhasználásával.
• Integration Services (Integrációs szolgáltatások): Az integrációt segítő funkció / platform. Az ETL funkciók segítségével támogatást nyújt az adatok átalakításában, támogatja az adatcsere a különböző feldolgozási igényeknek megfelelően.

• Replication (Replikáció): Olyan technológiák összessége, amely biztosítja az adatok és adatbázis objektumok másolását, támogatja a megosztást és szinkronizációt a konzisztencia megőrzésével különböző adatbázisok között. Az adatok megoszthatóak ezzel a szolgáltatással távoli felhasználókkal, lokálisan és interneten keresztül is.

• Reporting Services (Riport szolgáltatások): Weben keresztül elérhető jelentések készítését teszi lehetővé. A jelentések különböző formátumban készíthetőek és központosítottan menedzselhetőek.

• SQL Server Service Broker (SQL szerver szolgáltatás kezelő): A fejlesztőknek nyújt szolgáltatásokat biztonságos alkalmazások készítéséhez. Üzleti alapú kommunikációs platformja lehetővé teszi, hogy a független szoftverkomponensek egy egészként tudjanak együttműködni.

• Master Data Services (mester adatszolgáltatások): Ez a szolgáltatás a mester adatok központi támogatását valósítja meg. A mester adatok alatt egy vállalat alapvető adatait értjük. Ezek minden üzleti folyamatban, vállalati rendszerben és alkalmazásokban felhasználásra kerülnek.

• SharePoint Integration (SharePoint integráció): Feladata a SharePointos elemzések és jelentések integrálásának támogatása.

Az SQL szerver alatt egy szerverpéldányt értünk (instance), amely tartalmaz minden olyan folyamatot és szolgáltatást, ami a szerver futásához szükséges. Ezen kívül egy vagy több adatbázist, melyen a felhasználói és rendszeradatok tárolódnak. A felhasználói adatbázisok száma tetszőleges lehet, viszont minden szerverhez négy darab rendszeradatbázis tartozik.

• master: Ebben az adatbázisban tárolja az MSSQL a szerver szintű információkat
• model: Ez úgynevezett sablon adatbázis, melyet új adatbázisok létrehozásakor használunk. Ezen felül tartalmazza felhasználói adatbázisokban lévő rendszertáblák alapértelmezett beállításait.
• msdb: Az SQL Server Agent által használt adatbázis jobok és riasztások ütemezése
tempdb: A szerveren az összes ideiglenes adat ebben az adatbázisban lévő táblákban tárolódik. Ilyenek például a lekérdezések részereedményei.

Az SQL szerver sémai az adatbázison belül lévő névterek, amelyek az objektumok csoportjait tárolják. Az összes objektum az adatbázisokban sémakba van rendezve. A sémak felhasználóhoz történő rendelésével tudjuk korlátozni az elérhető objektumokat.

Adatbázis objektumok és jellemzőik:

- Tábla: Az adatok tárolására szolgál.
- Adattípus: A lehetséges adatértékeket határozzák meg.
- Constraint: A rekordok egyes tulajdonságainak értékére adhatunk meg szabályokat. Ezek a szabályok a constraint-ek.
- Default: Ezt az értéket veszi fel minden olyan mező, ahova nem adunk meg értéket.
- Index: Az adatok elérésének gyorsaságát segítő tárolási szerkezet.
- View: Segítségével fizikailag különböző helyen (különböző táblákban) lévő adatokat logikailag összekapcsolhatjuk.
- Stored procedure: SQL utasítások halmaza, rutin, amely meghívásakor a benne leírt utasításokat hajtja végre.
- Trigger: Speciális tárolt eljárás, amelyet nem kívülről hívunk, hanem egy esemény hatására fut le automatikusan.

7.13 MetroWindow [13]

A MetroWindow a mahapps csomag része. Az egész projektet Paul Jenkins indította 2011-ben, melynek célkitűzése az lett, hogy egy WPF alapú, letisztult, egyszerű, de mégis modern felhasználói felületet alakítson ki. Csak a stíluselemekre koncentrál a projekt, mely segítségével növelhető a felhasználói élmény. A projekt fejlesztői felfogás nem más, mint közösség a közösségért. Ennek lényege, hogy bárki fejleszthet rajta és bárki felhasználhatja. Kifejezetten a Microsoft Visual Studio környezet ajánlott a fejlesztéshez. További célként fogalmazódott meg, hogy ne csak modern és jól kinéző felület legyen, hanem használhatóság szempontjából is kimagaslót alkosson a közösség. Véleményem szerint ezeket a célkitűzéseket sikerült is elérni.
7.14 MetroChart


8 A fejlesztés leírása, program részeinek magyarázata, technikai megvalósítások

8.1 Tervezés

Minden logikai modul egy különálló C# projekt ez alól egy kivétel van az UI (User Interface), mert ebben két különálló logikai egység található a View és a ViewModel. A projekt indulás igen lassú és nehézkes volt. Többször is egyeztettem a HR osztályal, hogy minden általuk kért és kívánt funkció letisztult legyen. Az egyeztetések folyamán mindig szem előtt tartottam a megvalósíthatóságot és a használhatóságot. Mindenképp olyan rendszert akartam fejleszteni, amibe nem kell külön beletanulni, hanem minden egyértelmű helyen található, megfelelően egyszerű, letisztult, de mégis modern. Ahhoz, hogy a fő felületet is el tudjam képzelni, sematikus képernyőképet készítettünk, amelyet az 5. számú ábra mutat.

5. ábra

Miután világossá váltak a megvalósítandó funkciók, összeírtuk a tárolandó adatokat, azoknak típusait és megkötéseit, következett az adatbázis tervezés. A már ismertette módszerrel rugalmasan, grafikus megjelenés mellet egyszerű volt az adatbázis elkészítése. A szoftver kialakítása közben végig követtem a különböző elnevezési konvenziókat,
Humán erőforrás menedzsment szoftver készítése

miszerint osztályt, tulajdonságot, konstruktor, metódust nagybetűvel kezdek, privát adattagot alulvonással kezdek. Minden kódbeli elnevezés angolul és minél beszédesebben nevezem el, hogy utaljon a funkcióra, amit megvalósít. A fejlesztés kezdeti időszakában az adatbázis kapcsolatot és logikát írtam meg. Ezután elkészítettem a kapcsolatot a VCC rendszerével és a hozzá kapcsolódó logika felépítést, amely az adatok szinkronizációjáért felelős.

Mikor elkészültem a VCC modullal következett egy átfogó teszt minden már elkészült funkcióra. Erre a célra létrehoztam egy teszt projektet, amelyből az adatbázis műveleteket és a VCC modul logikáját teszteltem. Természetesen a fejlesztés közben is folyamatos volt a tesztelés. Ezután következett a WCF szolgáltatás modul felépítése, a különböző funkciók szolgáltatásként való meghatározása és kifejtése. Amint ez is elkészült már logikai egységekre bontva folytattam a munkát. Amint egy egység elkészült, azt teszteltem és ezt a jelenlegi formájáig ilyen módon készítettem. A szoftverben minden logikai modul egy különálló C# projekt ez alól egy kivétel van az UI (User Interface), mert ebben két különálló logikai egység található a View és a ViewModel, melyek azonos projektben, de külön mappában szerepelnek. A teljes rendszer sematikus felépítését a 6. számú ábra mutatja, ami tartalmaz még meg nem valósított, de tervezett kapcsolatokat is.

6. ábra
8.2 Modulok fontosabb részeinek bemutatása

8.2.1 Az adatbázis modul


- EF Designer from database: Ebben az esetben már létező adatbázison alapuló modellt hoz létre EF Designer-ben. Leképezi az osztályokat az adatbázis objektumok alapján. Ez az adatbázis először és a model először szemlélet összevont megvalósítása.
• Empty EF Designer model: Ezzel az opcióval egy üres EF Designer-t hoz létre, ami kiindulópont az adatbázis modellünk vizuális tervezéséhez. Ez a modell először szemlélet megvalósítása.

• Empty Code First model: Ezzel nem hoz létre EF Designer-t, hanem csak egy osztályt, amelyben köddel készíthetjük el a modellünket. A kód először megvalósítása.

• Code First from database: Ez is kód először szemlélet, viszont itt már van egy elkészített adatbázisunk és ennek leképezésében nyújt segítséget. Viszont köddel ugyan úgy bővíthetjük, kiegészíthetjük az adatbázismodellünk. Fejlesztésem során én az EF Designer from database szemlélet alapján dolgoztam. Ez sokat gyorsít a munkán, mert egyrészt a varázsló segítségével felépíthető a kapcsolat az adatbázissal. Nem kell külön drivererekkel bajlódni, osztályokat írni, hanem egyszerűen felhasználónév és jelszó megadással és a kapcsolatra való hivatkozás elnevezésével azonnal tudjuk használni az adatbázist. Másrészt az EF Designer által biztosított grafikus tervezés nagyon leegyszerűsíti az adatbázis elkészítését. Ezt a megoldást és szemléletet más néven szokták „no SQL”-nek is hívni, mivel egy sor SQL kódot sem kell írni, mégis teljes értékű relációs adatbázis hozható létre. Ennek a módszernek a hátránya, hogy ha fejlesztés közben hozzá kell nyúlni az adatbázishoz, akkor a legenerált SQL kódot használva minden táblát eldob, ha már létezik. Ez fejlesztés közben is zavaró lehet, hogy mindig új adatokat kell feltölteni a táblát, viszont ha már használják a szoftvert, akkor ez további fejtőrést okozhat! A varázsló futtatása után a projektben létrejön egy „.edmx” kiterjesztésű file ez maga az EF Designer és az App.config-ba beíródnak a connectionString tulajdonságba a kapcsolódáshoz szükséges adatok. Ezt használva tudunk az adatbázishoz kapcsolódni.

A tervezéshez az alábbi eszközöket biztosítja számunkra a tervező:

• Entitás: Ez hivatott reprezentálni a táblát. Létrehozáskor minden entitás esetén lehetőségünk van öröklődést beállítani, meg kell adnunk azt az elnevezést, hogy adott entitásra hivatkozni szeretnénk majd. Én minden esetben „név”+”s” (többes szám) alakban határoztam meg, tehát az Employee entitáson esetén Employees. Ezen kívül meg kell adnunk, hogy generáljon-e kulcsot a leendő táblához, annak mi legyen a neve és mi az adattípusa.
- Kapcsolat: Ezzel adhatjuk meg az entitások kapcsolatát, hogy milyen relációban állnak egymással. Amennyiben egy entitást létrehoztunk, felvehetünk hozzá mezőket, statikusat és navigációs mezőt. Az egyes mezőkre beállíthatjuk az sql-nél megszokott tulajdonságokat, hogy milyen típusú mező, null értéket vehet-e fel, egyedinek kell lennie, stb. Miközben készítjük az újabb entitásokat, az EF elkészíti a modellből az entitást reprezentáló osztályokat. Ezt az osztályt az EF készítette:

```csharp
namespace HG.Optimus.Database
{
    using System;
    using System.Collections.Generic;

    public partial class DocumentLibrary
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public byte[] Document { get; set; }
        public Nullable<System.DateTime> UploadDate { get; set; }
        public double FileSize { get; set; }
        public string Extension { get; set; }

        public virtual Employee Employee { get; set; }
    }
}
```

Miután elkészülem a teljes adatbázis modellezésével, akkor az a Designer segítségével legeneráltam az SQL kódot. Ennek hatására elkészült a Model.edmx.sql file. Ahhoz hogy az adatbázist ténylegesen létrehozzam az SQL szerveren, az említett sql script filet kellett futatni. Ennek eredménye lett a kész adatbázis, melynek a felépítését a következő oldal mutatja!
8.2.2 A modell modul

Kérdésként merülhet fel, hogy ha már az adatbázis modulban az EF legenerálta nekem a modell osztályokat, akkor miért volt szükség létrehozni látszólag úgyan azt? Az EF által generált modellek csak arra szolgálnak, hogy az adatbázissal kommunikálják. Alapvetően sem szerencsés ugyan ezeket a modulokat használni a szoftver többi részében, mivel ezek tartalmazhatnak EF specifikus adatokat, melyekre az EF-nek van szüksége az SQL szerverrel történő kommunikáció során. A másik lényeges ok a kliens-szerver felépítés. Mint később majd kifejtem, a kliens-szerver közti kommunikáció XML alapon történik és az XML környezetben az objektum nem értelmezett. Továbbá, ha a program során további adattagokkal kell kiegészíteni a modellt a szoftver többi részére, akkor ezt nem tehetném meg, ha a generált modellt használom. Minden osztályt úgy alakítottam ki, hogy alkalmas legyen XLM szerializációra, tehát alkalmasak legyenek a kliens-szerver kommunikációra. Ennek lényege, hogy minden osztályban fel van használva a System.Runtime.Serialization névtér, ekkor elérhető és beállítható két attribútum, ami az XML szerializáció során szükséges, hogy a folyamat sikerrel végbemenjen. ADataContract attribútumot az osztálydefiníció előtt kell megadni, aDataMember-t pedig minden egyes adattag előtt.
using System;
using System.Runtime.Serialization;

namespace HG.Optimus.Model
{
    [DataContract]
    public class BonusDO
    {        
        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public double? Amount { get; set; }

        [DataMember]
        public DateTime? AmountFromDate { get; set; }

        [DataMember]
        public DateTime? AmountToDate { get; set; }

        [DataMember]
        public string Denomination { get; set; }

        [DataMember]
        public int EmployeeId { get; set; }
    }
}
8.2.3 A logikai modul

Ebben a modulban valósítottam meg az adatbázist kezelő logikákat és ide került a VCC szinkronizációs logikája is, ezen felül a JSON objektumok leképezése. Minden SQL táblát megvalósító osztályra el kellett készítenem az alapvető DML utasításokat. Mivel összesen 13 osztályra kellett megvalósítni így létrehoztam egy egyszerű interfészt, az IRepository-t.
using System.Collections.Generic;

namespace HG.Optimus.Logic
{
    interface IRepository<T>
    {
        int Insert(T entity);
        void Update(T entity);
        void Delete(object id);
        List<T> SelectAll();
    }
}

Mint látható ez egy igen egyszerű interfész, melynek lényeges része, hogy generikus típussal van létrehozva. Erre azért van szükség, mert ez az interfész nem lenne használható minden repository esetében, ha nem generikus típus szerepelne a definíciójában. Mivel egyértelmű, hogy ha beszúrást szeretnénk végrehajtani az employee táblába, akkor Employee típusú objektumot, ha pedig a user táblába szeretnénk beszúrni, akkor User típus objektumot szeretnénk átadni. Ez a nagy előnye a generikus típusnak, hogy segítségével olyan metódusok hozhatóak létre, melyek bármilyen adattípuson működnek! Ezt az interfész minden repository-ban implementáltam. Ez nem jelenti azt, hogy a repository-k csak ezeket a tulajdonságokat hordozzák, csak annyit, hogy ezeket mindenféle megvalósítják. Itt szeretnék kitérni az AutoMapperHelper osztályra, ami az automapper konfigurációját végzi el. Az automapper egy hasznos tool, amit NuGet segítségével hozzá tudunk adni a projektünkhoz. Segítségével nagyon egységes módon tudunk objektumokat másik objektumba konvertálni és csak egyszer, egy helyen kell beállítani. Ezt megtehetném egy for ciklus segítségével is, hogy adattagonként átmásolom az adatokat a megfelelő helyre. Viszont ez egyrészt nem szép megoldás, másrészt nagyon sok munka van vele, 13 osztályra külön-külön elkészíteni, továbbá ne felejtsük el, hogy ha fejlesztés közben rájövünk, hogy az egyik osztályunkat módosítani kell, nekem ez előfordult, akkor a már megírt for ciklusos másolásokat mindet módosítani kell. Az AutoMapperHelper osztály statikus osztály. Itt történik a Mapper inicializálása. Ennek lényege, hogy meghatározom, milyen objektumtípusokkal kell dolgozni és azokkal milyen módon. Az alábbi két metódus a lényege a konfigurációnak:
private static bool s_isInitialized = false;
private static object s_syncRoot = new object();

public static void Configure()
{
    lock (s_syncRoot)
    {
        if (s_isInitialized)
        {
            return;
        }
        Mapper.Initialize(cfg =>
        {
            cfg CreateMap<IssuedDeviceDO, IssuedDevice>();
            cfg CreateMap<IssuedDevice, IssuedDeviceDO>();
            // Itt állnak a további objektumok konfigurációi
            cfgForMember("Employee", (x) => x.Ignore());
            s_isInitialized = true;
        });
    }
}

public static T MapTo<T>(this object source)
{
    return Mapper.Map<T>(source);
}

Elsőnek két statikus tagot hoztam létre, az s_isInitialized –et azért, hogy meg tudjam vizsgálni az adott objektum, amire meg lett hívva, inicializálva lett-e, hogy elkerüljem a kivétel keletkezését, ami akkor jönne létre, ha a mapper hozzá akarna férni egy olyan objektum adattagjához, ami még nem lett inicializálva (NullReferenceException). Az s_syncRoot tag azért fontos, mert épp egy objektum adatait akarom lemasolni egy másikba és szeretném, ha másolás közben véletlenül se férne hozzá másik végrehajtási szál az objektumhoz és változtatná meg az állapotát. Igazából a kapott objektumra kölcsönös kizárást valósítok meg a C# „lock” kulcsszavával. Ennek lényege, hogy amíg a lock blokkja fut, más szál nem férhet hozzá az objektumhoz. A Mapper.Initialize metódussal állítom be a lehetséges objektumtípus párokat. A példában az látszik, hogy ha IssuedDevice típusú objektumból IssuedDeviceDO típusú készül, akkor az Employee tartalmát nem másolja át. Ez azért van, mert a kommunikációhoz és egyéb funkciókhoz használt DO modelekben nem tárolom a modellehez kapcsolódó objektumokat, csak a megfelelő id-t. Így ha arra is szükség
lenne, akkor az azonosító alapján az lekérdezhető. Viszont fordított esetben nincs kihagyott adat. A MapTo metódus egy teljesen generikusan létrehozott metódus. Így minden típuson értelmezhető. Ennek eredményeként, minden típusra hívható metódust kaptunk, ami egy névtérben szerepel vele. Ez a megoldás megint segítségemre van, hogy ne kelljen külön foglalkozni, hogy éppen melyik típuson akarom hívni. A repository-kra visszatérve az alapfunkciók, amiket az IRepository interfész meghatároz, minden esetben ugyan azon logika mentén történik a megvalósítás.

- A beszúrás:

```csharp
public int Insert(BonusDO entity)
{
    using (ConnectionString context = new ConnectionString())
    {
        Bonus bonus = entity.MapTo<Bonus>();
        bonus.Employee = (
            from employee in context.Employees
            where employee.Id == entity.EmployeeId
            select employee).FirstOrDefault();
        context.Bonuses.Add(bonus);
        context.SaveChanges();
        return bonus.Id;
    }
}
```

Minden esetben a kapcsolat megnyitása az első lépés. Ezt egy using blokk segítségével valósítom meg, hogy a blokk fejrészében létrehozom a kapcsolati példányt. Fontos, hogy a using fejrészébe csak olyan objektum példányosítását lehet használni, amely megvalósítja az IDisposable interfész. Ez azért van, mert a using lefutása után, mikor a vezérlés kilép a blokkból, akkor lefut az objektumra Dispose() metódus, mely felszabadítja az összes lefoglalt erőforrást. Az általam használt, EF által generált ConnectionString osztály a DBContext osztály leszármazottja és a DBContext többek között megvalósítja az IDisposable interfész. Ezt a formulát használva biztos lehetek abban, hogy a módsz eredeti lefutása után az összes erőforrás felszabadul, tehát a nyitott kapcsolat az adatbázissal is. A using utáni első sorban létrehozok egy Bonus típusú objektumot, amelybe a MapTo segítségével létrehozom
a paraméterként kapott BonusDO alapján az SQL-be beszúrni kívánt adatsort. Ezután a EF-ben a reláció reprezentálása végett le kell kérdeznem azt az Employee-t, akihez tartozik az paraméterként kapott bónusz. Ezt a LINQ segítségével hajtom végre és a lekérdezés eredményét elmentem a bonus Employee adattagjába. Eddig még minden a memóriában történik, viszont a contexten hívott SaveChanges() metódus hatására az általam végzett módosítások mentésre kerülnek az adatbázisban. Ezek után visszatérek a beszúrt sor azonosítójával.

- A törlés:

```csharp
public void Delete(object id)
{
    using (ConnectionString context = new ConnectionString())
    {
        var task = context.Cdrs.SingleOrDefault((item) => item.Id == (int)id);
        if (task == null)
        {
            throw new ArgumentException();
        }

        context.Cdrs.Remove(task);
        context.SaveChanges();
    }
}
```

A törlés esetemben egy tényleges törlés nem csak egy update (flag állítás), de ez nincs minden táblára megvalósítva, csak amelyekre kérte a HR osztály. Ennek egyszerű magyarázata az ingyenes adatbázis kezelő használata. A Microsoft SQL szerver Express kiadása ingyenes, viszont több megkötés mellett van egy méretre vonatkozó megkötés is, így ha semmi nem kerülne tényleges törlésre, akkor előbb utóbb elfogynia a rendelkezésre álló hely. Törlés esetén csak a törlendő objektum azonosítóját kell megadnom, ezzel egyértelműen azonosítható a törlendő sor, így a kommunikáció a szerver és a kliens között gyorsítható, mert csak egy egészet kell átküldeni XML-ben. Az azonosító alapján kikerem a megfelelő sort LINQ–val (melyben egy lambda kifejezés szerepel - névtelen metódus) és azt Remove metódussal eltávolítom, majd mentem a változásokat az adatbázisban.
• Adott tábla összes elemének lekérdezése:

```csharp
public List<JobDO> SelectAll()
{
    using (ConnectionString context = new ConnectionString())
    {
        return context.Jobs.Select(x => x.MapTo<JobDO>()).ToList();
    }
}
```

A tábla összes elemének lekérdezése összesen egy sor, egy lambda kifejezés és egy átalakítást tartalmaz, mivel az adatbázisból érkező adatok Job osztály példányai, viszont visszaadni JobDO adok vissza.

Lényeges megjegyezni a végén hívott ToList() metódust, mert e nélkül csak egy System.Linq.IQueryable típusú objektumunk lenne. Tehát végeredményben csak egy SQL szerveren futtatható lekérdezés, nem pedig a lekérdezés eredménye. Így magát a lekérdezést kényszerítem ki (futtatjuk le a szerveren) a ToList() meghívásával.

Amikor bizonyos feltétel szerint kérdezek le az adatbázisból, akkor csak annyi változás van a kódban, hogy nem Select metódust használunk, hanem Where-t és a feltételt ugyan úgy lambda kifejezéssel adom meg és mappelést az eredményen végzem el Select-tel. Például:

```csharp
context.Jobs.Where(x => x.Employee.Id == employeeId).ToList().Select(x =>
    x.MapTo<JobDO>()).ToList();
```

8.2.3.1 A kapcsolat felépítése a VCC rendszerrel


List<VccUser> vccUsersInDb = vccUserRepository.SelectAll();

var users = activeVccUsers.Where(a => !vccUsersInDb.Any(b => a.UserName == b.VccUserName));

A cdr adatok szinkonizációját a SynchronizeVCCCdrs metódus végzi. Ennek bemenő paramétere egy dátum, mely meghatározza, hogy melyik napot kell szikronizálni. Ezt a metódust bonyolítja az a tény, hogy a VCC-ből API-n keresztül maximum 500 sor cdr adat
kérhető le. Amennyiben az egész nap adatsorát le akarjuk kérni, akkor azt 500 darabonként kell megtennünk külön kérésekben. Erre egy do – while ciklust hoztam létre:

do
{
    vccCdrsTemp = GetVCCCrds(dateTime, count);
    if (vccCdrsTemp != null)
    {
        vccCdrs.AddRange(vccCdrsTemp);
        count += vccCdrsTemp.Count;
    }
}
while (vccCdrsTemp!= null && vccCdrsTemp.Count != 0);

Ennek lényege, hogy deklaráltam egy count változót mellyel vizsgálok, hogy éppen hol tartok és egy temp listába (vccCdrsTemp) lekérem az első 500 (itt a számláló 0, ez jelenti a VCC-nek, hogy az első 500 sort kell elküldenie). Amennyiben érkezett adat, hozzáadom a listához, amibe a napi összes sort gyűjtöm (vccCdrs), majd a számláló értékét növelem a rekordszámmal, amennyit feldolgoztam. Tehát a következő lekérdezésnél már 500-on van a számláló, ez az adatok lekérdezésénél annyit tesz, hogy az 500-adik sortól kezdődő következő 500 sort kell átküldenie a VCC-nak. Mindezt addig megy, míg üres listát nem kapok vagy null értékü. Ez jelenti azt, hogy elfogytak a rekordok adott napra. Ezután a kapott adatokat betöltöm az adatbázisba, de mivel lehetséges nap közbeni szinkronizáció is, ezért ellenőriznem kell, hogy az adott sor nem szerepel-e már az adatbázisban. Ezt egy if feltétellel vizsgálok, ami sajnos nem a leg optimálisabb, mert ezt akár az adatbázisra is bizhatnám, ha cdr adat uuid-jára beállított volna adatbázis szinten egy unique tulajdonságot. Így csak az ott keletkező kivételt kéne kezelni. De a módszer annyira nem volt lassú, mint vártam, mert a tesztek alatt ~40.000 sor teljes szinkronja 1.5 perc alatt lefutott, ami szerintem annyira nem mondható rossznak.

Ebben a modulban található még a VccCdrSyncronScheduler osztály, mely a VCC adatok szinkronizációjának ütemezéséért felelős. Ebben érdekesség, hogy közvetlenül nem lehet példányt készíteni belőle, mert privát konstruktorral hoztam létre. Így csak a statikus metódusán keresztül érhető el a GetScheduler()-rel. Ez a metódus annyit csinál, hogy ha még nem létezik példány az osztályból, akkor készít egyet, ha pedig már lett példány készítve, akkor visszaadja azt. Így biztosan csak egy példány hozható létre az osztályból. Ez azért lényeg, mert így szabályozható, hogy két frissítést párhuzamosan ne lehessen elindítani. Ezt úgy oldottam, meg hogy előbb lekérdezem a Scheduler értékét, ha null, akkor futtatom a
frissítést, ha nem null akkor a hívó tudni fogja, hogy már fut a frissítés így megvárja a végét. A StartVccCdrSyncron() metódus lényegében végigiterál az adatbázisban lévő utolsó naptól a futtatás napjáig az összes napon. Az utolsó napot azért kérdezi le, mert lehet hogy volt már adott napon frissítés az alkalmazásból, de az nem a teljes nap adatsorát tartalmazza, ezért végignézi, hogy minden CDR sor bekerült-e az adatbázisba.

8.2.4 A szolgáltatás modul

10. ábra

nem rendelkezik az adatbázis felé és a VCC rendszere felé sem. A szolgáltatások megvalósítása nagyon egyszerű, mert csak a megfelelő hívásokat indítja el a logikai modul felé és a klienstől érkezett paramétereket adjá meg. Véleményem szerint ennél többet nem is szabad tartalmaznia, mert akkor elmosná az előre megszabott éles határokat a modulok között.

8.2.5 A felhasználói felület modul

11. ábra

Ez a modul a legbonyolultabb mind közül. Ezt tükrözi a ClassDiagram.cd automatikusan generált osztálydiagram fájl is, melyet Visual Studio-val lehet megnyitni. Négy fő részre bontható:

- A szolgáltatás kliens (ServiceClient): Ez egy referencia a szolgáltatásokra. Ezt nagyon egyszerűen hozzá lehet kapcsolni a kliens megvalósításhoz, esetemben a HG.Optimus.UI WPF projekthez a Visual Studio segítségével. A szolgáltatás referencia hozzáadása nevű varázsló segít nekünk. Itt megadjuk a szolgáltatás címet és már hozzá is lehet adni a referenciát. Van egy másik megoldás is, ami hosszabb és a varázsló is ezt a műveletet végzi el a háttérben, hogy a projekt App.config fájljába, ami egy XML alapú dokumentum, írnám bele a <system.serviceModel> tag-ét. Ezen belülre pedig meghatároznék egy klienst a clinet tag-gel és ezen belül pedig létrehoznék egy végpontot.

```xml
<system.serviceModel>
  <client>
    <endpoint address="http://localhost:50987/OptimusService.svc"
      binding="basicHttpBinding"
      bindingConfiguration="BasicHttpBinding_IOptimusService"
  </endpoint>
</client>
```
Két fontos fájl tartozik még logikailag a kliens megvalósításhoz és a kapcsolat kiépítéshez a Service.cs és a ClientChannel.cs. A Service osztály feladata a kapcsolat kiépítése a szolgáltatással és a kapcsolat bontása. Ezt úgy oldottam meg, hogy létrehozom a ChannelFactory osztály egy példányát, mely osztály feladata, hogy létrehozzon különböző típusú csatornákat, amit a végpont (kliens) tud használni üzenetek küldésére. A GetChannel metódus arra szolgál, hogy létrehozzon egy konkrét csatornát, kapcsolatot a WCF szolgáltatással és ezt visszaadja a hívónak. A Close függvénye csak a kapcsolat lezárásáért felelős. A ClientChannel osztályt csak azért hoztam létre, hogy megvalósítsa az IDisposable interfész, mert így a kapcsolat felépítést és a kommunikációt egy using direktiván belül tudom használni, ezáltal mindig biztosan le lesz zárva a kapcsolat.

- A nézet modellek (ViewModel)
Az összes ViewModel osztály a ViewModelBase osztály leszármazottja! Ez azért fontos, mert a View-kal való kommunikációhoz minden osztálynak meg kéne valósítani az INotifyPropertyChanged interfészt. Így viszont elég, ha csak az össosztály valósítja meg, a többi pedig örököli ezt a tulajdonságot. Ennek az interfésznek egyetlen eseményt kell definiálnia, mégpedig a PropertyChanged-et melynek a PropertyChangedEventHandler metódusreferenciával kell együttműködnie. Ez fogja azt az eseményt kiváltani, amely a felhasználói felület frissítését végzi a modell egy tulajdonságának megváltozása esetén. Paraméterként a tulajdonság nevét kapja. Fontos és szinte minden ViewModel esetén használt osztály még a RelayCommand osztály, mely az ICommand interfészt valósítja meg. Ez szükségessé teszi a vezérlőként történő akcióval értesülő ViewModel felé. Ehhez az osztályhoz három konstruktort készítettem, amely túlterhelt. Az első csak egy metódust vár paraméterként, amelyet végrehajt a kiváltó esemény hatására, de a futtathatóságot nem vizsgálja, mivel a paraméter típusa akció. A második konstruktor egy végrehajtható metódust és predikátum típusú metódust vár paraméterként. Ennek lényege, hogy a predikátum típusú canExecute paraméter igazolja tér vissza, meghívja az akció típusú execute paraméterként kapott metódust. Ennek akkor van nagy szerepe, ha például egy
gombot csak akkor nyomatunk meg, ha előtte egy kijelölt legördülő menüből kiválasztottunk egy értéket. Ezt dinamikusan figyeli és folyamatosan vizsgálja, hogy teljesül-e a kritérium, azaz kiválasztottunk-e egy elemet. Amint ez megtörtén azonnal elérhetővé válik a gomb és meg lehet nyomni. A harmadik konstruktor annyival egészíti ki az előzőeket, hogy a meghivandó metódusnak tudunk átadni paramétert. A ViewModel-jeim két fő típusra oszthatóak a reprezentált modell megjelenítését megvalósító és a modell listát megvalósító. A listát megvalósítók fő eleme az ObservableCollection típusú ViewModel lista. Azért használtam ezt a típust, mert ez a generikus lista megvalósítja az INotifyPropertyChanged és az INotifyCollectionChanged interfészét. Ez azért fontos, mert ha a lista tartalma változik, akkor az XAML alapú felületen csak akkor jelenik meg a változás, ha ezekez az interfészeket megvalósító listát használunk. Ezeket a listákat az osztály konstruktornak meghívásakor inicializálóm, például:

```csharp
public void Init(int employeeId)
{
    List<DocumentLibraryDO> documentLibraryList;

    using (var channel = Service.GetChannel())
    {
        documentLibraryList = 
        channel.Client.SelectEmployeeAllDocumentLibrary(employeeId).ToList();
    }

    DocumentLibraryList = new 
    ObservableCollection<DocumentLibraryViewModel>(
        documentLibraryList.Select(x=> new 
        DocumentLibraryViewModel(x)));
}
```

Minden esetben létrehozok egy lokális listát, amibe lekérdezem a modellnek megfelelő adatokat. A példa esetében ez az adott dolgozóhoz tartozó dokumentumok listája. A Service statikus osztály GetChannel() metódusának hívásával létrehozok egy kommunikációs csatornát a szerver és a kliens között és a szerver SelectEmployeeAllDocumentLibrary szolgáltatását meghívom és megadom neki a dolgozó egyedi azonosítóját. Itt a visszakapott DocumentLibraryDO[] tömböt a toList() segítségével átalakítom listává. Majd létrehozom a felhasználó felé megjelenített listám példányait egyesével, a Select linq kifejezés használatával. Így már fel is töltöttem a listám. A példánál maradva a DocumentLibraryViewModel osztály példányosításakor megkap paraméterként egy DocumentLibraryDO típusú
objektumot, melyet beállít a saját privát adattagjába. Ennek beállításakor értéket kapnak a tulajdonságok is, melyeket a felhasználónak szeretnénk megjeleníteni:

```csharp
private DocumentLibraryDO _documentLibrary;

public string Name
{
    get { return _documentLibrary.Name; }
    set
    {
        _documentLibrary.Name = value;
        OnPropertyChanged("Name");
    }
}
```

Látható, hogy amikor egy tulajdonsághoz beállításra kerül egy új érték, akkor a beállítás után azonnal meghívom az OnPropertyChanged metódust, hogy a felhasználói felületen is megjelenjen a változás. A nézetmodellek között különleges szerepet ellátó modell a DocumentLibraryGenerateViewModel. Ez az osztály valósítja meg az automatikusan generált felhasználói dokumentumokat word sablonok alapján. Úgy írtam meg, hogy a konstruktorában négy paramétert kér, amelyek által le tudja generálni a kért dokumentumot. Példányosítása után hívható a GenerateDocument metódusa, amely regexek segítségével cseréli ki a beállított munkavállaló adataira a jelölt szavakat. Ilyen például a #Név, ez a munkavállaló nevével helyettesíti. Természetesen bármilyen word dokumentumra alkalmazható. A word fájl olvasásához a standard StreamReader-t alapul, írásához pedig StreamWriter-t használtam. A legnagyobb osztály az EmployeViewModel lett, mivel a munkavállalók kezelésénél jelenik meg a ViewModel-ek nagy része. Ezen modell nézete magába foglalja a munkavállalóhoz tartozó összes modellt kivétel a statisztikai adatok. Ez a felület kialakítás miatt van így, mert egy adott munkavállalóhoz tartozó adatok tabitem segítségével vannak megjelenítve. Így ablakváltás nélkül tekinthető át az összes adat és ezen felül az adatok módosítása, új adatok hozzáadás, adatok törlése, dokumentumok feltöltése is innen van vezérelve. Ezért ebben a modellben nagyon sok vezérlő megvalósítás található. Például a beszélt nyelvekhez történő új nyelv hozzáadása:
private RelayCommand _saveNewLanguageCommand;

public RelayCommand SaveNewLanguageCommand
{
    get
    {
        if (_saveNewLanguageCommand == null)
        {
            _saveNewLanguageCommand = new RelayCommand(
                SaveNewLanguage, CanExecuteSaveLanguage);
        }
        return _saveNewLanguageCommand;
    }
}

private bool CanExecuteSaveLanguage(object obj)
{
    if (LanguageViewModel != null)
    {
        if (string.IsNullOrEmpty(LanguageViewModel.Name) ||
            string.IsNullOrEmpty(LanguageViewModel.Level))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
    return false;
}

private void SaveNewLanguage(object obj)
{
    LanguageViewModel.InsertOrUpdateLanguage();
    VisibilityLanguage = Visibility.Collapsed;
    LanguagesListViewModel.Init(_employee.Id);
}

Minden vezérlőnek létrehoztam egy saját RelayCommand típusú privát adattagot. Az adattagokhoz egy speciális tulajdonságot rendeltem, melynek csak getter-e van, setter-e nincs. Ezeket a tulajdonságokat kötöttem hozzá a kívánt vezérlőhöz. A vezérlő aktiválásakor, például egy gombra kattintáskor, kérdezi le a felület a tulajdonság értékét. Az adattag értékének elérése esetén nem csak az érték visszaadása történik, hanem előtte egy vizsgálat fut le. Amint a példában
látható megvizsgálom, hogy az `_saveNewLanguageCommand` adattagot próbáltam-e már elérni. Ha még nem akkor az értéke null és ebben az esetben létrehozok egy RelayCommand objektumot és két paramétert adok át. Az első a `SaveNewLanguage` metódus, ami akkor hívódik meg mikor például a gombra kattintunk, fontos hogy a visszatérési értéke void, tehát nincs. A második paraméter a `CanExecuteSaveLanguage` nevű metódus boolean visszatérési értékké, melynek érdekes a, hogy a felület vezérlő rövid időnként kiértékeli. Ebben olyan rövid logikákat érdemes megírni, melyek irányítják, hogy melyik vezérlő mikor érhető el. Amig a kiértékelt hamsis értékkel tér vissza a vezérlő inaktív, látszik a felületen de nem használható, viszont ha igaz értékkel tér vissza a vezérlő azonnal aktívvá válik. A példában azt láthatjuk, hogy amíg a `LanguageViewModel` értéke null biztosan nem használható az új nyelv mentése gomb. Amint kapott értéket, már tudjuk vizsgálni, hogy a nyelv neve és szintje üres-e. Ha ez a két mező kitöltésre kerül, már menthetővé válik az adattartalom. Azért van két külön ha függvény meghatározva, mert amíg a `LanguageViewModel`-nak nincs értéke, addig a modell, név tulajdonságának lekérdezésére tett kísérlet, kivételt váltana ki. Ennek a kivétel keletkezésének megelőzése miatt lett kialakítva ez a szerkezet.


```xml
```
Erre azért van szükség, hogy a fordító tudja, hogy hol keresse a kódban meghivatkozott vezérlőket. Ezek után a view xaml fájlt kellett módosítanom, hozzáadni a néveret és egy általam megfelelőnek vélt névvel ellátni, amellyel hivatkozhatok majd rá, én Controls-nak nevezem el:

```xml
```


- A stílusok (Styles): Ebben vannak meghatározva a vezérlőelemek stílusjegyei. Hasonlóan kell elképzelni, mint a HTML és CSS kapcsolatát. Mivel a stílusokat külön fájlból helyeztem el, nem pedig az ablak xaml kódjában, mert így sokkal áttekinthetőbb és értelmezhetőbb maradt a kód, ezért a fordítóval tudatni kell a forrást, ahol az xaml kódban meghivatkozott stílust megtalálja. Ezt az App.xaml fájlból tudjuk megtenni nagyon egyszerű módon:

```xml
<ResourceDictionary Source="Styles/ButtonStyles.xaml"/>
```

A stílusok meghatározása semmi eltérőt nem tartalmaz az xaml-ben megszokottól. Viszont a Metro stílus alkalmazása miatt volt egy problémám, ami a fejlesztés közben fejtőrész okozott. A felület fejlesztését sajnos nem a legoptimálisabban kezdem el, mert a stílusfájlok kialakítását utólag készítem. Először meg minden stílus tulajdonság a nézet xaml fájljába volt. A kód tisztítása során ezeket a beállításokat helyeztem el vezérlőtípusonként külön fájlba, viszont itt felmerült egy probléma.
Amint egy stílus meghatározást áthelyeztem a végleges helyére a vezérlő kinézete visszaállt az alapértelmezetre a Metro helyett. Ezt a problémát többször is körbejárva, interneten is segítséget keresve, végül megoldottam. A megoldást a BasedOn tulajdonság beállítása jelentette a Metro + vezérlőtípusa formában.

```xml
<Style x:Key="BaseDataGridStyle" TargetType="DataGrid"
    BasedOn="{StaticResource MetroDataGrid}"
    <Setter Property="AutoGenerateColumns" Value="False"/>
    <Setter Property="SelectionMode" Value="Single"/>
</Style>
```

9 Tervezett fejlesztések és funkciók

A fejlesztés közben folyton jutottak eszembe új lehetséges funkciók ebből kettőt jelöltem is az alkalmazást bemutató ábrámon. Az egyik a toborzáskezelés, melynek megvalósításához kapcsolatot szeretném kiépíteni a cég honlapját üzemeltető szerverrel. A honlapon létre fogok hozni egy ürlapot, ahova az álláskeresők a főbb adataikat meg tudják adni és dokumentumot tudnak felölteni, mint például az önletrajzuk. A külön mezőknek a kereshetőség miatt lenne fontos szerepe. A másik ötlet a cafeteria rész újragondolása. Egy belső üzemeltetésű, a már működő IIS által hosztolt weblapot készítetek, amin a munkavállalók be tudják adni a cafeteria igényüket. Ez a weblap figyelni fogja a különböző kikötéseket, mit például az egy elemből kérhető maximális összeg. Jelezni fog a HR osztálynak, hogy kitöltött egy cafeteria igényt (ez a jelzés lehet a szerverben is, vagy akár email útján) és az igény csak akkor kerül be a munkavállaló adatai közé, ha azt a HR vezető jóváhagyta.

Tovább lehetőség még a „HR-es” lekérdezések, amelyek lényegében olyan lekérdezéseket takarnának, amik a HR osztálynak fontosak. Ilyen például az egészségügyi alkalmassági vizsgálat vagy a munkavédelmi oktatás kiknek esedékes. A most még hiányzó VCC-s statisztikai lekérdezések megvalósítása. VCC adatok alapján automatikus jelenlét készítése, ami alapja lehetne egy szabadság és betegség nyilvántartó modulnak. A szoftver tesztelése közben merült fel a hr-es kollégákban, hogy örölnének, ha a munkavállaló pozíciójának változásait, történetiségét is nyomon tudnák követni a rendszerben. A telepítés leírása közben merült fel, hogy ezt is fejleszteni szeretném, hogy sokkal egyszerűbb legyen, és ne kelljen ennyi manuális beállítást elvégezni, hanem egy telepítő modul csinálja meg. Ezek a felid vigindig van, amikor a kéddel foglalkozom, meglátok olyan dolgokat, amiket már
lehet nem is úgy valósítanám meg, mint ahogy elsőre elkészítettem. Véleményem szerint egy szoftver soha nincs kész, mindig van mit fejleszteni és javítani rajta.

10 Szükséges szoftverek és telepítési útmutatójuk

- Microsoft Windows 7 operációs rendszer, vagy újabb
- .NET 4.5 vagy újabb
- IIS express telepítése és beállítása
  - A Microsoft hivatalos oldaláról ingyenesen letölthető.
  - Telepítés után a Windows szolgáltatások ki- bekapcsolásának intézőjében szükséges lehet a szolgáltatás bekapcsolása.
  - Ezután az ASP.NET-et regisztrálni kell az IIS-ben. Adminisztrátori móddal kell elindítani a cmd-t és az alábbi mappába navigálni: C:\WINDOWS\Microsoft.NET\Framework64\v4.0.xxxx. az x-ek a megfelelően legmagasabb számmal helyettesítiőek, esetemben a v4.0.30319 volt a verzió.
  - Itt az alábbi parancs kiadása szükséges: „aspnet_regiis.exe –i” (registry-ket is frissíti)
  - Ezek után a futtatás menübe be kell írni az alábbit: inetmgr. Ez indítja el az IIS menedzerét.
  - Itt a kapcsolatoknál a helyek mappán jobbgomb, Webhely hozzáadása menüpont.
  - A hely neve tetszőleges lehet.
  - Az alkalmazás készletet fontos átállítani „ASP.NET v4.0” –ra
  - Fizikai elérési út menüpontnál a mellékelt „HG.Optimus.Service” mappát kell beállítani.
  - Kötés típusa „http”, IP címek a hoszt gép IP-jét megadni, portnak javaslom az 51200-at.
  - Az állomásnév kitöltésére nincs szükség.
  - Ezután a létrehozott új helyre kell pozicionálni és a szolgáltatások nézetben a hitelesítés pontot választani. Itt a „névtelen hitelesítés” ponton jobb egérgomb és ott a „szerkesztés” választása és „Alkalmazáskészlet identitása” beállítása.
  - Tartalom nézet kiválasztása, majd egy szabad helyen jobb egérgomb és „engedélyek szerkesztése…” menüpont. Itt a mappa biztonságánál be kell állítani két szereplőt, aki a mappához férhet, az IIS_USER és a HÁLÓZATI SZOLGÁLTATÁSOK.
  - Utolsó lépésként a választott portot (51200) a tűzfalon engedélyezni.
  - A beállítások sikerességének a teszteléséhez az IIS menedzer tartalom nézetén belül az „OptimusService.svc” fájlon jobb egérgomb használata és a menükből a tallózás kiválasztása. Ekkor az alapértelmezett bőngészőben megnyílik a szolgáltatás, amelynek eredményét a 12. számú ábra mutatja.
12. ábra

- Az MsSQL telepítése és beállítása
  - A telepítés a szokásos
  - Javaslom a „Microsoft SQL Management Studio” telepítését is a könnyebb kezelés végett.
  - Létre kell hozni egy adatbázis „HrSoftware” néven, „Hungarian_CI_AS” karakterkészlettel.
  - A mellékelt „HG.Optimus.Database” nevű mappában található „Model.edmx.sql” nevű SQL server query típusú fájlt kell futtatni a létrehozott adatbázisra. Ez a script fájl létrehozza a táblákat és a teljes szerkezetet.
  - Célszerű egy automatikus mentést ütemezni éjszakára.

- A HR szoftver telepítése

  A mellékelt „\HG.Optimus.UI\bin\Release” mappa tartalmát kell a kiválasztott helyre felmásolni. Első indítás előtt a „HG.Optimus.UI.exe.config” alkalmazás-konfigurációt meg kell nyitni akár egy szövegszerkesztővel és a szolgáltatás címét az IIS-ben beállított címről cserélni:
endpoint address=\texttt{http://192.168.0.154:51200/OptimusService.svc}

A alkalmazás a „HG.Optimus.UI.exe” fájllal indítható el.

11 A HR szoftver használati útmutatója

Ebben a részben csak a statisztikai rész használatát ábrázolom, mert a szoftver többi része nem szorul magyarázatra, mint azt a HR osztály is megerősítette. A statisztikai modul kezelése:

![Diagram](image)

13. ábra

12 Köszönetnyilvánítás

Ezúton szeretném köszönetemet és tiszteltemet kifejezni mindenkinnek, aki a szakdolgozatom elkészítéséhez hozzájárult. Elsőként nagybátyámnak, Lippai Tamásnak, akire minden téren számíthattam, legyen az szakmai, anyagi vagy lelki támogatás! Szűcs Miklós mesteroktató, témavezető tanár úrnak a hozzáállásáért, segítőkészségéért, szakmai észrevételeiéért és a mérhetetlen rugalmasságáért! Továbbá tanárainknak, akik az egyetemi éveim alatt tudásukkal és felkészültségükkel bevezettek az informatika mélyebb világába. Édesanyámnak, aki mindig támogatott, ahogy csak erejéből futotta. Végül, de nem utolsó sorban szeretném kifejezni köszönetemet és szívből jövő szeretetemet a családomnak, feleségemnek és lányaimmak, akik szeretetükkel és megértésükkel mindvégig támaszt nyújtottak nekem és igyekeztek ideális körülményeket biztosítani a szakdolgozatom készítése alatt.
13 Irodalomjegyzék


14 Összefoglalás

Dolgozatomban egy humánerőforrás szoftver programozása volt a cél, amelyet az Ellerton Call Centers Kft. megbízása alapján készítettem el. Az alkalmazás felé támasztott elvárásokat a cég HR osztálya határozta meg, így tervezés során velük egyeztettem. Mivel már több mint öt éve dolgozom call centeres területen, így a teljesítménymérés témakörében széles tapasztalatokkal rendelkezem, ezért a szoftver e részének megértése nem okozott gondot. Fontos kritérium volt, hogy az alkalmazást egyszerre több munkatárs is tudja használni, ezért döntöttem a kliens-szerver struktúra kialakítása mellett. A szoftver elkészítése során, a már korábban megszerzett programozói tudásom és tapasztalatom bővítésére volt szükség. Ilyen nagyságú rendszert ezelőtt még nem készítettem, így ez a feladat nagyon jó kihívást jelentetett számomra. Sikerült felépítenem adatkapcsolatot egy fejlesztőcég által készített professzionális szoftverrel. Ennek keretén beül új, eddig nem használt adatstruktúrával és annak feldolgozásával ismerkedtem meg, ami a JSON. Megismerkedtem az MVVM mintával, amely segítségével logikailag teljesen szétválaszthatóak a szoftver egyes részei, ezáltal segíti a kód megértését, karbantartását és továbbfejlesztését. Meglepõdve tapasztaltam a WCF technológia egyszerűségét és megvalósíthatóságát, amelyet véleménym szerint a Microsoft nagyon jól alakított ki. Az adatbázis kapcsolat során használt Entity Framework megoldását mát többször használtam és véleménym szerint sokkal hatékonyabb és gyorsabb, mint az SQL kódokkal megalkotni a relációs adatbázist. Ennek ellenére úgy gondolom, hogy ha nagyon nagy adatbázist kell építeni, amiben akár több száz tábla is van, akkor ez a módszer már nem a legoptimálisabb választás. Végül hasznost tapasztalatokat szereztem már meglévõ modulok újrafelhasználásában, mint például a MetroWindow vagy a MetroChart. Programozás közben mindig szem előtt tartottam az objektum orientáltságot és az egyszerűséget. Végeredményben szerintem sikerült egy olyan programot megvalósítani, amely nagyon jó alapot biztosíthat, akár egy magasabb szintű szoftver megalkotásához is, amit nem csak a megrendelô az Ellerton Kft., hanem más call centerek HR osztálya tudna hasznosíttani. Ehhez viszont még mindig nagyon sok munka kell, mert minden alkalommal, mikor a kódon változtatok észrevettem valamit, amit lehet jobban csinálni, vagy ki lehetne egészíteni, hogy jobb legyen. Ez alátámasztja filozófiámat az alkalmazáskészítésről, hogy „egy szoftver fejlesztését soha nem lehet befejezni”, véleménym szerint nincs kész szoftver, mert mindig van mit fejleszteni, javítani rajta.
15 Summary

In my dissertation, I was programming a human resource software, which was commissioned by Ellerton Call Centers Ltd. The expectations of the application were determined by the HR department of the company, so during the planning I agreed with them. Since I've been working in Call Center areas for over five years, I have extensive experience in the field of performance measurement, so understanding this part of the software was no problem. An important criterion was that the application could be used simultaneously by more than one HR employee, so I have decided to create a client-server structure. During software development, I needed to expand my previously acquired programming knowledge and experience. I did not make such a size system before, so this job was very good challenging task for me. I managed to build a data connection with a professional software developed by a developer establishment. Within this work, I got to know new and unused data structure and its processing, which is JSON. I got acquainted with the MVVM pattern, which allows logically separating parts of the software, helping to understand, service, and improvement of the code. Surprisingly, I found the simplicity and feasibility of WCF technology, which I think Microsoft has made very well. I used the Entity Framework solution used in the database connection several times and in my opinion it is much more efficient and faster than creating a relational database with SQL codes. Nevertheless, I think that if you have to build a very large database with more than hundred of tables, this method is no longer the most optimal choice. Finally, I have gained experience in reusing existing modules such as MetroWindow or MetroChart. During programming, I always kept the object orientation and simplicity in mind. In the end, I think it has been possible to implement a program that can provide a very good foundation for creating a higher level of software that could be utilized not only by the customer but also by the HR department of other call centers. For this, there is still a lot of work to do, because every time I change the code I notice something that can be done better or can be added to make it better. This underprop my philosophy of making the application that "software development can never be completed", in my opinion there is no ready software, because there is always something to develop and improve on it.
16 Szójegyzék

1. CRUD : Created, Read, Update, Delete
2. OLAP - On-Line Analytical Processing
3. ETL – Extract, Transform, Load
4. VCC – Virtual Call Center

17 Mellékletek

1. Eredetiségi nyilatkozat
2. HR szoftver képernyő fotói
3. CD melléklet
1. számú melléklet

EREDETISÉGI NYILATKOZAT

Alulírott …………………………………………………………………; Neptun-kód:…………………
a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős …………………… szakos hallgatója
ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom,
ho gy
………………………………………………………………………………………………………………
című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom
felhasználása a forráskezelés szabályai szerint történt.
Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:
  • szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
  • tartalmi idézet hivatkozás megjelölése nélkül;
  • más publikált gondolatainak saját gondolatként való feltüntetése.
Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium
esetén szakdolgozatot visszautasításra kerül.

Miskolc,..............év .................hó ........nap

………………………………………………
Hallgató
2. számú melléklet

CD melléklet:

- Humán erőforrás menedzsment szoftver készítése:
  - Humán erőforrás menedzsment szoftver készítése.doc
  - Humán erőforrás menedzsment szoftver készítése.pdf
  - 16_Automatika_BSc_szakdolgozat_HolicsGábor.doc
  - Magyar nyelvű összefoglaló.doc
  - Magyar nyelvű összefoglaló.pdf
  - English summary.doc
  - English summary.pdf
- HG.Optimus – forráskód
- HG.Optimus – futtatható kód
- HR szoftver - képernyő fotók