Budapesti Gazdasági Egyetem

Pénzügyi és Számviteli Kar

Nagy Alexander István

Gazdaságinformatikus

Egyedi alkalmazásfejlesztése, dokumentálása és bevezetése

2021

NYILATKOZAT

Alulírott NAGY ALEXANDER ISTVÁN büntetőjogi felelősségem tudatában nyilatkozom, hogy a szakdolgozatomban foglalt tények és adatok a valóságnak megfelelnek, és az abban leírtak a saját, önálló munkám eredményei.

A szakdolgozatban felhasznált adatokat a szerzői jogvédelem figyelembevételével alkalmaztam.

Ezen szakdolgozat semmilyen része nem került felhasználásra korábban oktatási intézmény más képzésén diplomaszerzés során.

Tudomásul veszem, hogy a szakdolgozatomat az intézmény plágiumellenőrzésnek veti alá.

Budapest, 2021 év 05 hónap 02 nap

.....

hallgató aláírása

BUDAPESTI GAZDASÁGI EGYETEM
ALKALMAZOTT TUDOMÁNYOK EGYETEME

PÉNZÜGYI ÉS SZÁMVITELI KAR

Nagy Alexander István

Gazdaságinformatikus

Beszámoló a szakmai gyakorlatról

2020

1149 Budapest, Buzogány utca 10-12.
Telefon: (+36-1) 469-6600
Fax: (+36-1) 469-6610
www.uni-bge.hu

1. Bevezetés

A szakmai gyakorlatom helyszíne kiválasztásakor próbáltam olyan helyet keresni, ahol leginkább a programozással tudok foglalkozni. Több hetes keresgélés, és több interjún való részvétel után megtaláltam a számomra ideális helyet, ahol az elvárások egy egyetemista szintjéhez igazodtak és nem egy már 5 éves tapasztalattal rendelkező gyakornokot kerestek. Így a választások a Vincotech Hungária Kft.-re esett, ahol az IT gyakornok pozíciót töltöm be. Amikor a HR-es megkeresett, hogy szeretnének behívni egy interjúra, már akkor sokkal kedvesebbek voltak, mint azok a helyek ahol előtte voltam. Az interjún jelen volt a Humánerőforrás-menedzsmenten kívül 3 programozó is, akiknek később a munkájukat kell segítenem. Egész idő alatt szinte csak ők beszéltek, bemutatták a céget, hogy pontosabban mivel is foglalkoznak, mi lesz a későbbiekben a feladatom és, hogy a jelenleg is folyamatban lévő projektek közül mi is lenne nekem az ideális így az első napokban. Az elején elmondtam, hogy mik azok a területek, amikben jártas vagyok. A végére már tudtam, hogy ez a hely nekem való, mind munkaügyileg, mind az ott lévő emberek kedvessége miatt.

2. Cég bemutatása

A Vincotech egy dinamikusan fejlődő nemzetközi vállalat, a Mitsubishi Electric Corporation tagja, németországi központtal, magyarországi fejlesztő és gyártó részleggel. Magas színvonalú teljesítmény-elektronikai áramköröket terveznek és gyártanak ipari hajtásokhoz és napenergiás inverterekhez. A vállalat erőteljes technológiai portfóliójának alapját egy erősen motivált és tapasztalt mérnöki csapat végzi, az R&D részleg. A cég világszerte kb. 800 embert foglalkoztat.

3. Adatbázis

Az adatbázis azonos tulajdonságú, többnyire strukturált adatok összessége, amelyet egy tárolásra, lekérdezésre és szerkesztésére alkalmas szoftver kezel. Célja az adatok hosszú távon való megbízható tárolása, és gyors visszakereshetőségének biztosítása. Az adatbázis-kezelővel nem szabad összekeverni, mivel ez egy eszköz,

amivel az adatbázist működtetjük. Jellemzően két fajátja van az adatbázisnak, logikai és fizikai.

3.1. Adatmodell

Adatmodellnek az adathalmaz és az adathalmaz elemei között fennálló kapcsolatok strukturált leírását nevezzük. Az adatok és az azok közötti összefüggések leírására szolgál. Olyan mesterséges rendszer, amely felépítésében megegyezik a vizsgált rendszerrel.

3.1.1.Hierarchikus adatmodell

Az adatokat fa struktúra szerint építjük fel. Az adatbázis több egymástól független fából is állhat. A fa csomópontjaiban és leveleiben helyezkednek el az adatok. Ez a szülő gyerek kapcsolatnak felel meg, így csak 1: n kapcsolatok hozhatók létre. Az 1: n kapcsolat az jelenti, hogy az adatszerkezet egyik típusú adata a hierarchiában alatta elhelyezkedő egy vagy több más adattal áll kapcsolatban

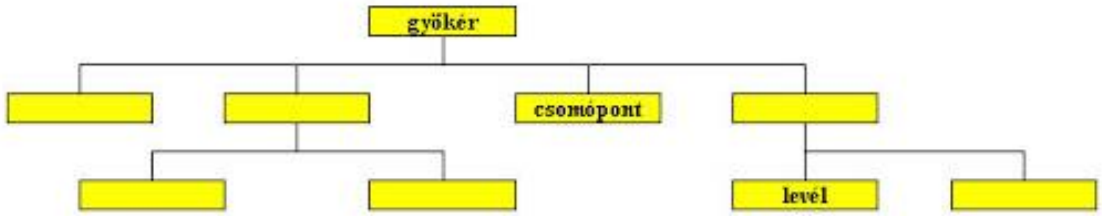

Table 1 *Hierarchikus adatmodell*

3.1.2.Hálós adatmodell

A fa modellhez képest itt az adatok összefonódhatnak, így a fa helyett inkább gráfot alkotnak. Ez esetben az azonos vagy különböző adatok között a kapcsolat egy

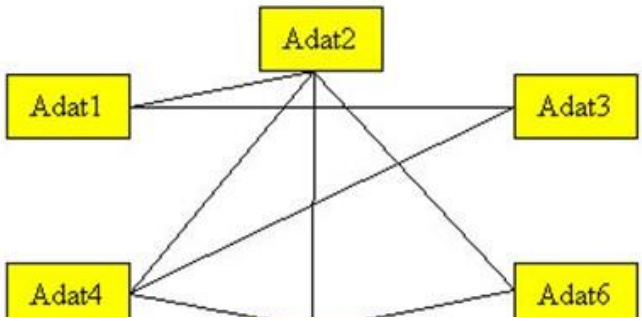

Table 2 *Hálós adatmodell*

gráffal írható le. A csomópontok között akkor van kapcsolat, ha őket él köti össze. Ebben a modellben n:m típusú kapcsolat is leírható.

3.1.3.Relációs adatmodell

A modell kialakítását az indokolta, hogy az adatbázis megtervezésekor az adatokat a felhasználó számára áttekinthető, és egyszerűen kezelhető táblázatokban lehessen elhelyezni, hasonlóan, mint a hagyományos adatszervezés rekordstruktúrája. Képes egyidejűleg több, összefüggő tábla feldolgozására. Az adatok tárolása táblázatos formában, a tábla soraiban és oszlopaiban valósul meg.

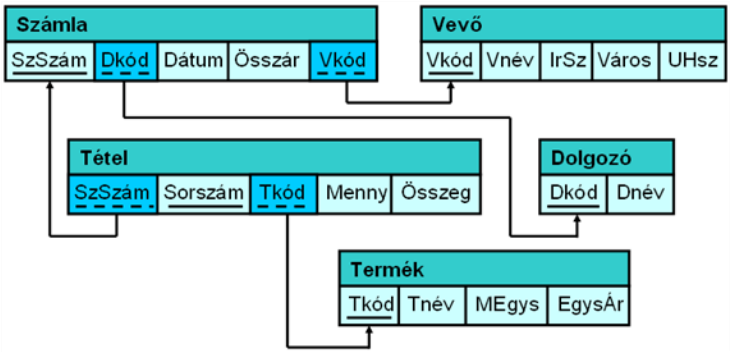

Table 3 *Relációs adatmodell*

3.2. Adatbázis tervezés, kialakítás

A gyakorlatom végzése alatt történő adatbázis használat során, általában a relációs adatmodellt használtam. Azt vettem észre, hogy a megtervezés során több olyan dologra is kell figyelni, mint például a kulcsok szerepe a táblában. Egy-két tábla esetén egyszerű lehet a kapcsolatok létrehozása. Viszont mikor három, akár négy fő tábla van, és ezekhez kell rengeteg kisebb táblát csatolni, bajban lehet vele az ember.

Az első olyan projektet, ami adatbázist is igényel, a gyakorlatom viszonylag elején megkaptam. Ez egy nagyon egyszerű program, amit ütemezni kell. Lényege, hogy egy automatikusan generált Excel fájl-t, amit egy másik program készít, fel kell tölteni időnként adatbázisba.

Az első teendő az adatbázis tervezésekor, a tárolni kívánt adatok feltérképezése. Ez úgy történt az én esetemben, hogy az Excel fájl-t megnyitva a rajta található Output sheet-en (munkalapon), az összes oszlop összes adatát megvizsgálva kell megállapítani, hogy az egyes oszlopokhoz tartozó adatok, milyen típusúak. Az Excelben található egy

Szűrő lehetőség, ahol az oszlopok fejlécére kattintva, megtekinthető, hogy abban az oszlopban milyen adatok találhatóak. Ezt követően egy jegyzettömbbe leírom az oszlop neveket, amelyek később az adatbázis táblájában is szerepelni fognak. Ilyenkor a könnyebb használhatóság miatt is, és a jobb átláthatóság miatt érdemes ugyanazokat az elnevezéseket használni. Az összes elnevezés mellé kigyűjtöm a hozzá tartozó adattípust. Ilyenkor figyelni kell arra, hogy az adatbázis-kezelő-ben más a megnevezése egy-egy típusnak. Néhány példa, ami a legtöbbször előfordulhat:

Típus megnevezés	**Adatbázisban**	**C# nyelven**
Szöveg	Nvarchar, varchar	String
Karakter	Char	Char
Egész szám	Int	Int
Tizedestört	Float,Decimal	Double, float, decimal
Dátum	Date,DateTime	DateTime
Logikai	Bit	Bool

Table 4 Adattípusok

A C#-os elnevezésnek a későbbiekben lesz jelnetősége, amikor a kódunkat fogjuk írni.

A tervezés befejeztével, két lehetőségünk van létrehozni a táblát. Az első lehetőség, hogy legeneráljuk kóddal, amit egy új query-t megnyitva a következő képpen tudjuk ezt megtenni SQL nyelven:
CREATE TABLE táblanév (megnevezés típus [egyéb opció], megnevezés2 típus2 [egyéb opció], ...) Az egyéb opciók között az alábbi feltételek jelenhetnek meg:

- PRIMARY KEY: elsődleges kulcs
- UNIQUE: kulcs
- REFERENCES tábla(oszlop): külső kulcs

A másik opció, hogy az adatbázis-kezelő nyújtotta funkciót kihasználva, grafikai megjelenéssel hozzuk létre a táblákat. Ez úgy néz ki, hogy az adatbázisban a táblákat tartalmazó könyvtáron jobb egér gombot nyomva, megjelenik egy olyan lehetőség, hogy tábla létrehozása. Ha ezt az opciót választjuk megjelenik egy táblázatos felület, ahol 3 oszlopot látunk: Oszlop neve, Típusa és hogy szerepelhet-e benne üres érték. Amikor létrehoztuk a táblát, és megfelelően szerepel benne minden név és adattípus, ki kell jelölni a kulcsot. A kulcs a tábla rekordjainak egyértelmű azonosítója, értéke egyedi. Olyan mezőt kell választanunk, amiben nincs, és nem is lehet a későbbiekben sem ismétlődés.

Ha az általunk létrehozott adatbázisban nincs ilyen, létre kell hoznunk plusz egy mezőt, ami ezt a célt szolgálja. Van egy olyan lehetőség is, hogy ha beállítjuk, akkor ez az azonosító automatikusan növekszik eggyel minden hozzá adott sor után.

Az adatbázis elkészültével, érdemes kézzel felvinni pár sor tesztadatot, hogy meggyőződhessünk arról, hogy minden rendben. Ezt elvégezve, jó ha töröljük a táblából az összes adatot, amit ha egy egyszerű törléssel oldunk meg, akkor az azonosító számozása, onnan fogja folytatni, ahol az abbamaradt. Ilyenkor ki kell adni azt az utasítást, hogy TRUNCATE TABLE táblanév. Ez az utasítás megmondja, hogy a tábla mostantól üres, és nem csak a tartalmát törli.

4. A program elkészítése

4.1. A Feladat leírása, megfogalmazása

A program elkészítésének első lépéseként, ki kell tűzni a feladatot és magunkban, akár leírva megfogalmazni, hogy hogyan is fog ez kinézni. Ez a leírás hétköznapi nyelven történik, ami lehet akár egy darab mondat is. Az én esetemben ez csak annyi volt: Írjunk egy olyan programot, amely egy Excel fájl adatait, bizonyos időközönként feltölt egy adatbázisba. Mivel a program ütemezve lesz, érdemes konzolos alkalmazást készíteni. A könnyebb tesztelés érdekében, először egy formos (ablakos) alkalmazást hoztam létre, amire, csak egy gombot helyeztem el. Erre rákattintva, elindul a feltöltés.

4.2. A feladat specifikációja

A feladat úgynevezett konyhanyelven való megfogalmazása után, a program készítője saját szemszögből fogalmazza meg a feladatot, ami már szakmai nyelven történik. Ilyenkor vizsgáljuk a bemenő, kimenő adatokat, azok típusát és hogy vannak-e ezek valamilyen feltételhez kötve. A bemenő adatokra elő-, a kimenőre utófeltételeket fogalmazunk meg. A specifikáció megírása történhet szöveges, matematikai és logikai szimbólumok használatával. A legtöbbször használt jelek a következők:

- $\wedge$: ÉS logikai művelet
- $\vee$: VAGY logikai művelet
- $\rightarrow$: AKKOR logikai művelet (implikáció)

- $\leftrightarrow$: AKKOR és CSAK AKKOR logikai művelet (ekvivalencia)
- $\forall$: „minden” logikai kvantor
- $\exists$: „létezik” logikai kvantor

Az általam készített specifikáció bemenő adatai között szerepel: szöveg, szám, dátum és logikai típus. A kimenő adatok megegyeznek a bejövőkkel, mivel módosítás nélkül kerülnek feltöltésre. Egy előfeltételt kell alkalmazni, hogy létezik-e a fájl, amit fel kell dolgozni.

4.3. Tervezés

A következő lépésben tervezés következik, ahol a megoldandó feladathoz, megfelelő adatszerkezeteket és algoritmusokat kell alkalmazni. Amint fentebb is említettem, egy konzolos alkalmazásra lesz majd szükségünk, de előtte egy ablakos alkalmazást hoztam létre a könnyebb tesztelés végett. Nagyjából megterveztem, hogy hogyan is fog kinézni a kódom, hogyan is fog felépülni.

4.3.1. Az algoritmus

Az algoritmus egy feladat megoldásának egyértelmű sorrendben való leírása sorrendben. Fontos, hogy sorrendben, mivel a végrehajtás lépésenként történik. Minden lépésnek egyértelműen végrehajthatónak kell lennie. Az algoritmus műveletekből és vezérlő szerkezetekből épül fel. A vezérlő szerkezet a feladat műveletre bontását és végrehajtásának sorrendjét írja le.

Vezérlő szerkezetek:

- Szekvencia(műveletsor),

- Szelekció(kiválasztás),
- Iteráció (ismétlés, ciklus)

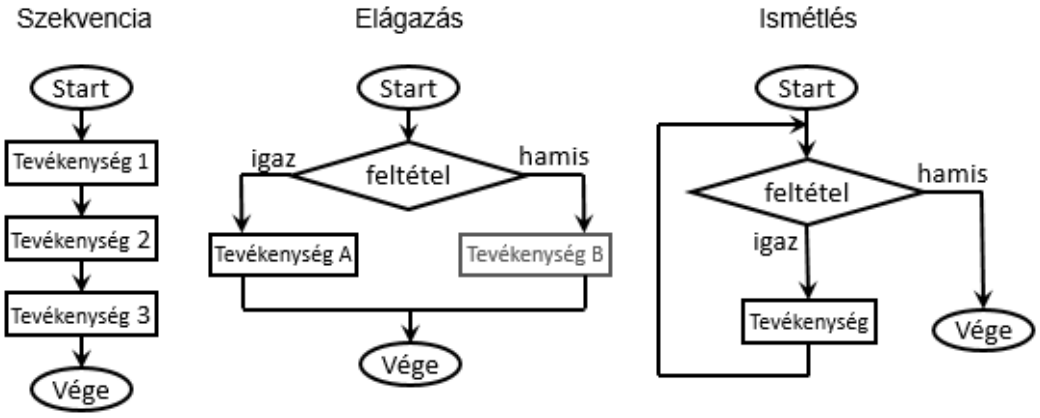

Table 5 *Vezérlő szerkezetek*

A program fejlesztése során hasznos dolog, ha érthető módon írjuk le az algoritmust.

Leírási módok:

- Mondatszerű leírás: Az algoritmus egymást követő lépéseit mondatokkal vagy mondat szerű szerkezettel próbáljuk meg leírni.

Grafikus ábrázolás:

- Folyamatábra: A feladatmegoldás lépéseinek sorrendjét, utasítástípusonként különböző geometriai alakzatok felhasználásával szemléltető ábra.
- Struktogram : Az egyetlen felhasználható elem a téglalap, ezekkel a téglalapokkal ábrázolják a strukturált alapszerkezeteket

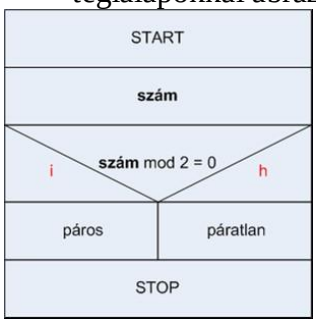

Table 6 *Struktogram*

- Jackson-ábra: Ez az eszköz az adat- és algoritmikus szerkezetek leírására **egységes**ábrakészletet definiál

4.4. Kódolás

A kész tervet felhasználva elkezdtem megírni a kódot. Az első dolgom az, hogy egy olyan módszert találjak ki, ami egyszerűen beolvassa az Excelt. Erre találtam az interneten egy egyszerű megoldást, amit egy kis módosítással fel tudtam használni.

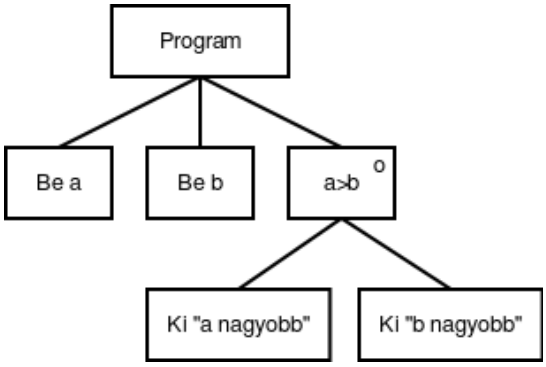

Table 7 *Jackson ábra*

Lényegében ez egy olyan metódus, ami a fájl útvonalának és nevének megadásával, beolvassa egy DataTable-be azt. A DataTable egy Excelhez hasonló objektum, ami tartalmaz DataSet-eket, ami az Excel munkalapjának felel meg. Ezen belül találhatóak meg a sorok (DataRow), és az oszlopok (DataColumn). Az adatok beolvasása után szúrópróbaszerűen leellenőriztem, hogy került-e adat a DataTable-be, mindezt debug módban. A debug módban az alkalmazásnak olyan kiegészítő információit nézhetjük meg, amelyek megkönnyítik a hiba keresését. Én most nem hibát kerestem, hanem csak ellenőriztem, hogy a beolvasott DataTable-ben szerepelnek-e az adatok. Ez azért jó, mert nincs szükség a grafikus megjelenítésre, hanem egyszerűen debug módban futtatjuk és megnézzük azokat az információkat, amelyekre szükségünk van. Ahhoz, hogy a programunk feltöltse adatbázisba a beolvasott DataTable-t, szükséges csatlakozni az adatbázishoz. Ehhez is találtam egy sokkal egyszerűbb módszert, amit a DataTable-hez könnyen tudtam alakítani. Megadtam egy változóban a Connection-hoz minden információt. Ilyenkor meg kell adni a Data Source-t, ami az adatbázisunk elérési útvonala, Inicializálni kell a Catalog-ot, ami az adatbázisunk neve, valamint szükséges egy felhasználónév és jelszó megadása is. Ezt követően egy targetTable változóban megadtam a céltáblát, ahová az adatokat szeretném feltölteni, majd a kapcsolat megnyitásával feltöltésre kerülnek az adatok. Fontos, hogy a feltöltés után zárjuk le a

kapcsolatot, különben hibára futhat a program. Az adatbázisba mentés metódusnak, két paramétert adtam meg, az egyik a DataTable, a másik a targetTable.

4.5. Tesztelés

Amikor kész a feltöltés kódja, ezt is tesztelni kell. Ez úgy történik, hogy megnyitom az adatbázist, és a programban megadom, hogy melyik DataTable-t, melyik táblába töltse fel. Ezt követően lefuttatom a programot, amely ha mindent jól csináltam fel kell töltse az összes Excelben szereplő adatot az adatbázisba. A teszt eredményeképpen a következőt kaptam: a program hiba nélkül lefutott, viszont nem töltött fel semmit az adatbázisba. Emiatt következett a debugolás ismét, viszont most ténylegesen hibát kerestem. Az első töréspontot a beolvasáshoz raktam, ilyenkor a program itt megáll és belép debug módba. F10-et nyomkodva lépésenként haladhatunk a program sorai között. A beolvasáson lépésenként mentem végig, ami meg is történt, tehát nem itt volt a hiba. Következett a feltöltés, itt szintén lépésenként haladtam, viszont itt a kapcsolat megnyitása után, amikor próbálná feltölteni a táblát, észrevettem, hogy nem adtam meg a WriteToServer()-nek, hogy mit is töltsön fel. Ezt gyorsan orvosoltam is, úgy, hogy a () -ek közé beírtam a DataTable nevét, majd újrafuttattam a programot. Ráfrissítettem az adatbázisra, mostmár szerepelt benne az összes adat, ami az Excelben.

4.6. Hibajavitás

A hibajavítás során igazából nem is hibákat javítottam, hanem a program működésén csiszoltam. Beleraktam egy olyan funkciót, ami a beolvasás előtt megnézi, hogy az adott mappa létezik-e, és hogy hány darab fájl van benne. Ha talált benne fájlokat, megkeresi azt, amelyikre nekem szükségem van, és ha létezik, akkor beolvassa azt. Lefuttattam újra a programot, és találtam is egy hibát. Az adatbázisban megkétszereződött a sorok száma, ami nem jelent jót, mivel újra feltöltötte az összes adatot. Nekünk viszont csak arra van szükségünk, ami az Excelben is szerepel. Ezt azzal oldottam meg, hogy visszatértem az adatbázishoz és létrehoztam egy indexet.

Az index a táblákban való keresés és sorba rendezés gyorsítására alkalmas eszköz. Az indexet egy vagy több mező értékei alapján hozzuk létre. Nem volt egyszerű kiválasztani az index-et, mivel nem volt olyan mező, ami egyedi lett volna, nekem viszont erre volt szükségem, ahhoz, hogy csak egyszer szerepeljen egy sor. Hosszas keresgélés

után találtam két olyan mezőt, amiből összerakhattam az indexet. Ezt úgy állítottam be, hogy nem lehetnek azonosak.

Ezt követően a már ismert módon töröltem a táblát és újra lefuttattam a programot. Majd egy gyors ellenőrzés után, hogy feltöltötte-e az adatokat ismét lefuttattam, ezúttal viszont nem voltak ismétlődések. Kipróbáltam, hogy mi van akkor, ha módosítás történik. Az első sorban átírtam pár adatot, majd lefuttattam a programot ismét, az adatbázisban sikeresen módosultak az adatok az első sorban.

Mostmár működőképes volt a program, viszont még mindig csak a formos alkalmazás volt kész. Létrehoztam, egy új parancssoros alkalmazást, ahová a kész könyvtárat beimportáltam, ami tartalmazza a beolvasást és a feltöltést. A fő metódusban pedig meghívtam őket, majd ismét megnéztem, hogy így is működik-e a beolvasás és feltöltés. Mindennel elkészülve írtam egy email-t, hogy használható a program, és be lehet üzemeltetni.

4.7. Dokumentáció

Mivel ez egy végtelenül egyszerű program a dokumentációval nem töltöttem sok időt. A dokumentálás nagyon fontos, mind a fejlesztői, mind a felhasználói részről. A fejlesztői dokumentáció lényege, hogy ha évekkel később elő kell vennünk, vagy valaki másnak elő kell vennie az alkalmazást, és módosítania kell, vagy csak szeretné megismerni a működését, nagyban megkönnyítse a dolgát. A felhasználói dokumentáció, mint a nevében is benne van, a felhasználókhoz szól. Ebben leírásra kerül a program használatának helyes módja. Viszont az elkészült program nem felhasználói szinten lesz használva, ezért ez most elmaradt.

5. Szoftverek, fejlesztői környezet

A végére hagytam az általam használt szoftverek bemutatását, mivel az eddig leírtak bármilyen más fejlesztői környezetben, bármilyen erre alkalmas szoftverrel megvalósíthatóak. Kezdjük azzal, hogy mi is a szoftver? A szoftver a számítógép kézzel nem fogható részeit jelenti. Két nagy fajtája van: a programok, és a programok működéséhez szükséges adatok. Egy szoftver készítése szellemi munka, ezért a terjesztéséhez, használatához különböző jogi szabályok kapcsolódnak.

Nézzünk egy csoportosítást jogi szempontból:

- Szabad szoftver
- Freeware
- Shareware
- Demó
- Kereskedelmi
- Adware
- Donationware
- Mailware

A szabad szoftver teljesen ingyen használható, forráskódja változtatható és szabadon továbbadható. A freeware nagyon hasonló az előzőhöz, viszont ezeknek a programoknak a forráskódja nem módosítható.

A shareware egy olyan szoftver, amit ingyen beszerezhetünk és használhatunk, de csak egy bizonyos időkorláton belül. Ezt a korlátot átlépve fizetnünk kell érte. A Demó egy olyan szoftver, ami a teljes programnak egy funkciókorlátozott változata. Teljesen ingyen beszerezhető és ha ez tetszik, akkor megvásárolhatjuk a teljes programot. A kereskedelmi szoftver az, amit a boltban, interneten megvásárolhatunk teljes értékű, korlátozás nélküli program.

Az adware, donationware és mailware egy egyéb kategóriai besorolást kapott. Ezek a szoftverek ingyenesen beszerezhetőek, viszont egy adware alkalmazás használata közben reklámok fognak megjelenni. Ezek az alkalmazások jellemzően mobiltelefonra készülnek. A donationware alkalmazás készítőinek önkéntes alapon adományt küldhetünk. A mailware alkalmazást készítők viszont csak egy köszönet e-mailt várnak cserébe.

A szoftver megvásárlása esetén csak korlátozott jogokat kapunk a használatra. Hogy melyek ezek a jogok, az a szoftverhez mellékelt licencszerződésben vannak leírva. Ez tartalmazza többek között azt, hogy hány gépen használható, továbbadható-e, és ha igen milyen formában, illetve készíthető-e másolat róla.

A szoftvereket funkciójuk szerint többféle csoportba sorolhatjuk.

- Rendszerszoftverek

- Rendszerközeli szoftverek
- Fejlesztői környezetek
- Alkalmazói szoftverek

5.1. Fejlesztői környezet

Ide tartoznak azok a szoftverek, amelyekkel más programokat lehet készíteni. A programot egy úgynevezett szerkesztő programban kell megírni. Ez lehet egy egyszerű szövegszerkesztő vagy egy Integrált fejlesztői környezet(IDE). Az IDE-k nem csak szövegszerkesztési lehetőséget nyújtanak, hanem minden olyan eszközt ami egy program fejlesztéséhez kell: tervezés, futtatás, hibakeresés. Az egyik legismertebb ilyen IDE a Visual Studio, amit én is használtam a gyakorlatom alatt, ugyanúgy, mint a középiskolában és az egyetemen is.

5.1.1. Visual Studio

A Visual Studio az a fejlesztői környezet, amit minden nap használtam. A Microsoft fejlesztette ki, és adta ki az elsőt 1997-ben. Azóta rengetek, 11 kiadása jött ki, jelenleg a Visual Studio 2019 a legfrissebb. Többek között a F#, C++, C# és Visual Basic programozási nyelveket, valamint az XML-t támogatja. Én ezzel közül a C#-pal foglalkoztam. A korszerű szoftvertechnológiát megvalósító fejlesztői eszközök lehetővé teszik, hogy egy-egy szerteágazó, vagy több önálló projektből álló alkalmazás kódjait is egységesen kezeljük. A parancssoros kódolás mellett lehetőségünk van ablakos alkalmazást is készíteni, amit kedvünkre szabhatunk. Ez is a Visual Studionak egy nagy előnye. Örültem mikor közölték, hogy ebben a környezetben kell majd programoznom, mert már ismertem annyira, hogy az alap feladatokat el tudjam végezni. A gyakorlatom alatt számos új és hasznos dolgot tanultam a használata során. Olyan funkciókat mutattak és fedeztem fel, amelyek nagyban elősegítik a kódolást és a dizájnolást.

5.2. Microsoft Office

A Microsoft Office úgy lett kialakítva, hogy segítsen a szervezeteknek a munkahelyi együttműködés digitális átalakításában. Az Office csomagban igencsak nagy a választék. Megtalálható benne az egyszerű szöveg szerkesztő, táblázatkezlő, prezentáció készítő, e-mailek kezelésére alkalmas szoftver, adatbázisokat tudunk

készíteni, videócsevegő alkalmazást is tartalmaz, mindazt, ami az alap és még az azon túli feladatokhoz szükséges. Egyaránt beszerezhető Windows rendszerekhez és Apple Macintosh rendszerhez is. Az általam legtöbbet használt programok a gyakorlatom alatt az Excel és az Outlook volt. Az előbbi a táblázatok kezelésére szolgál, az utóbbi pedig egy levelező rendszer. Az Excel használata azért volt indokolt, mert több olyan alkalmazást készítettem, vagy annak módosítását végeztem, amelyek Excel táblákat használtak, mint például a fentebb bemutatott program. Az Outlook-ot napi szinten használtam, hiszen a legtöbbször ezen keresztül vettem fel azzal a kapcsolatot, akinek éppen az adott feladatot teljesítettem. Az utóbbi időben a Teams alkalmazás az, amit elkezdtünk használni és home office-ból a napi meetingeket ezen keresztül tartottuk meg.

A teams egy Skype-hoz hasonló csevegő program. Viszont, olyan funkciókat is tartalmaz a Skype-pal szemben, ami jobban beilleszthető a cég által használt Office csomagba. Az Outlook naptárban létrehozva egy eseményt, megadhatunk egy teams-es meghívót, ami integrálódik az Outlookba és azon keresztül tudunk csatlakozni a meetinghez.

6. Összegzés

Mindaz, amit középiskolában és az egyetemen tanultam programozás és informatikai ismeretek egy remek alap volt az elhelyezkedéshez. Ez a 14 hetes gyakorlat volt az első alkalom arra, hogy komolyabban is kipróbáljam magam, valamint sok tapasztalatot is szereztem. Pozitívumként könyvelhettem el továbbá, hogy mindenki nagyon kedves és segítőkész volt. Az eddigi gyakorlataim közül ezt mondhatom a legeredményesebbnek is.

Azt vettem észre, hogy egy nagy cégnél nem szabad a saját fejünk után menni, mert az nem biztos, hogy jóra vezet. Ha kérdésünk van kérdezni kell, ha mondanivalónk, akkor beszélni kell. Csak így lehet csapatként együtt dolgozni. Nem szabad szégyellni, ha valamit nem tudunk, ugyan úgy nekik is kellhet segítség, azért vagyunk egy csapatban, hogy segítsük egymást. Minden ott dolgozó embernek megvan a saját munkaköre és mindenki ahhoz ért, amihez neki kell.

BUDAPESTI GAZDASÁGI EGYETEM
ALKALMAZOTT TUDOMÁNYOK EGYETEME

PÉNZÜGYI ÉS SZÁMVITELI KAR

Nagy Alexander István

Gazdaságinformatikus

Egyedi alkalmazásfejlesztése, dokumentálása és bevezetése

2021

1149 Budapest, Buzogány utca 10-12.
Telefon: (+36-1) 469-6600
Fax: (+36-1) 469-6610
www.uni-bge.hu

1. Bevezetés

A szakdolgozatom témájának kiválasztásánál fontos szempont volt, hogy olyan témát válasszak, ami kapcsolódik a felsőoktatási tanulmányaimhoz, illetve a gyakorlati helyemül választott Vincotech Hungária Kft. profiljához, és az ott betöltött pozíciómhoz illeszkedjen. Így hát a választásom, mint a címből is kiderült az egyedi alkalmazásfejlesztése, dokumentálása és annak bevezetésére esett.

A mai gyorsan fejlődő világban rengeteg olyan alkalmazás van, ami az emberek magánéletét könnyítik meg, esetlegesen már-már elképzelhetetlen a többség számára a nélkülük való élet. A munka világának minden területén szintén találkozhatunk különféle egyedi alkalmazásokkal, amelyek már kész formában használhatóak. Ezeknek a kész alkalmazásoknak a licenszét megvásárolhatjuk a készítő weboldalán, esetleg előfizethetünk rá. Ezeknek a programoknak egyik hátránya, hogy nem feltétlen találjuk meg a nekünk szükséges software-t. Ilyenkor az illetékes akinek szüksége lenne egy egyedi fejlesztésű alkalmazásra, felkeres egy programozót, vagy egy programozó csapatot (a project volumenétől függően) és az ő igényeit specifikálva megrendeli azt a programot.

Ebben a szakdolgozatban ezen alkalmazások fejlesztésének a menetét fogom bemutatni, egy kis elmélettel.

2. Programozási nyelvek

Kezdjük rögtön a programozási nyelvekkel, azok történetével és jellemzőivel.

2.1.C# programozási nyelv

2.1.1. C# Története

A C# programozási nyelv a Microsoft új fejlesztési környezetével, a 2002-ben megjelent Visual Studio.NET programcsomaggal, annak részeként jelent meg. A nyelv nem rendelkezik hosszú múlttal, elődjének mindenképp tekinthetjük a C++ nyelvet, a nyelv szintaktikáját, szerkezeti felépítését, A C, C++ nyelvekben készült alkalmazások elkészítéséhez gyakran hosszabb fejlesztési időre volt szükség, mint más nyelvekben, ezért olyan nyelvet kerestek, amelyik jobb produktivitást eredményez, ugyanakkor megtartja a C, C++ hatékonyságát. Erre a problémára az ideális megoldás a C# programozási nyelv. A C# egy modern objektumorientált nyelv, kényelmes és gyors lehetőséget biztosítva ahhoz, hogy .NET keretrendszer alá alkalmazásokat készítsünk, legyen az akár számolás, akár kommunikációs alkalmazás. A C# és a .NET keretrendszer alapja a *Common Language Infrastructure(CLI)*.

2.1.2. C# jellemzői

A C# az új .NET keretrendszer bázisnyelve. Ehhez a keretrendszerhez tervezték, még a szabványosítási azonosítójuk is csak egy számmal tér el egymástól. A nyelv teljesen komponens orientált. A fejlesztők számára a C++ hatékonyságát, és a Visual Basic fejlesztés gyorsaságát, egyszerűségét ötvözték ebben az eszközben. Nagy volumenű programok írására is képes. Egy sorba több utasítás is írható. Az utasítások lezáró jele a pontosvessző. Minden változót deklarálni kell. Minden típus őse az object, így például egy egész típust csomagolhatunk objektumba illetve vissza. Függvények definíciói nem ágyazhatók egymásba, önmagát meghívhatja . Általános célú programozási nyelvként ez bárhol bevethető, de a Windows-alkalmazások illetve -szerverek terén szinte egyeduralma van. Az IOS-ra az Apple a saját nyelvén a Swiftben fejlesztenek alkalmazásokat. Abszolút befutó még a multi-platform mobilalkalmazás-fejlesztésnél. A legnépszerűbb játékok kb. harmada erre épül.

2.2. SQL lekérdezési nyelv

2.2.1. SQL Története

Az SQL kifejezés a Structured Query Language rövidítése, ami struktúrált lekérdező nyelvet jelent. A relációs adatbázis-kezelő rendszerekben tárolt adatok kezelésére fejlesztették ki. A relációs adatmodellt Edgar Frank Codd az IBM-nél dolgozta ki az 1970-es években. A System R projekt fejlesztéssel foglalkozó csoport létrehozta a SEQUL(Structured English Query Language) nevű, adatbázisok kezelésére használható nyelvet. Az English szó arra utal, hogy a nyelv szavai értelmes, adatkezelő műveletek jelentését angol nyelven fejezi ki. Az SQL nyelvet 1992-ben szabványosították.

2.2.2. SQL Jellemzői

Az SQL egy halmazorientált nyelv, mely a relációkon dolgozik. A halmazorientáltság azt jelenti, hogy a feladat nem eljárásszerű megfogalmazását kell megadni, amely a relációk kiválasztott sorain hajtódnak végre. Két felhasználási lehetőségi van: beágyazott SQL, önálló SQL. Az önálló felhasználása esetén a nyelv utasítási állnak rendelkezésre, a beágyazott SQL esetén harmadik generációs algoritmikus nyelvbe ágyazva alkalmazzuk. Az SQL nyelv további részekre bontható: adatdefiníciós nyelv, adatmanipulációs nyelv, lekérdező nyelv és adatvezérlő nyelvre.

Az SQL - ben a következő utasításcsoportok léteznek:
Adatdefiníciós utasítás: CREATE(létrehoz), ALTER(módosít), DROP(töröl)
Adatkezelési utasítás: SELECT(keres),INSERT(beszúr), UPDATE (módosít), DELETE(töröl)
Adatbiztonsági utasítás: GRANT(jog adás), REVOKE(jog visszavonás)

3. Programozási elvek

3.1. Stratégia elv

A stratégia elv az ókori latin kultúrából maradt ránk ami az oszd meg és uralkodj elve alapján fogalmazható meg. E sokféleképpen alkalmazható elv lényege, oszd részekre, majd a részek egymástól független való megoldásával az egész feladat könnyebben oldható meg. Így a feladatot könnyen kézben tarthatod, vagyis uralkodhatsz felette. A részfeladatokra bontás nem mást, mint, hogy pontosan ki kell jelölni az adott részművelet milyen adatokat kezel, milyeneket állít elő, és ezeket hogyan rendeli egymáshoz. Ilyenkor, kizárólag azt vizsgáljuk milyen adat jön be és mi lesz az eredmény, nem fontos a hogyan. Ha két részfeladatot azonos szinten definiálunk, biztosítani kell a harmóniát közöttük úgy, hogy a végrehajtásban az először végrehajtódó után szolgáltassa az adatait a következőnek. Ez a módszer a TOP-DOWN programozás, azaz a program felülről lefelé való kifejtése, amely a dekomponáláson, problémaanalizáláson, részekre osztáson alapul. Létezik egy másik módszer amikor a programot alulról felfelé építjük fel, ez a BOTTOM-UP programozás.

3.2. Taktikai elvek

Ez az elv úgymond a tanácsok elve, ami segít a kódunkat „finomabbá”, jobbá tenni.

- A párhuzamos finomítás elve
- A döntés elhalasztásának elve
- Vissza az ősökhöz elv
- A „nyílt rendszer” felépítés elve
- A döntések kinyilvánításának elve
- Az adatok elszigetelésének elve
- A párhuzamos ágak függetlenségének elve
- A szintenként teljes kifejtés elve

3.3. Technológiai elvek

A technológiai elvek az algoritmus és a kód írására és annak szabályaira vonatkoznak.

- Értelmes sorokra tördelés – világos tagolás
- Bekezdéses leírás
- Összetett struktúrák zárójelezése
- A „beszédes” azonosítók elve

3.4. Technikai elvek

Ezek az elvek technikai jellegűek, a program használhatóságáról szólnak.

- Barátságosság, udvariasság
- Biztonságosság
- Jól olvasható program
- A (jó) dokumentált program

3.5. Esztétikai-ergonómia elvek

A most következő elvek a program felhasználóbarátibbá tételéről szólnak. A már ismert udvariasság és „bolondbiztosság” finomítása.

- Lapkezelési technika
- Menütechnika
- Ikontechnika
- Tördelés
- Következetesség
- A hibajelzés követelményei
- Naplózás
- Ablaktechnika

4. Algoritmusok

Az algoritmusnak azt nevezzük, ami pontosan megmondja, hogy egy feladat megoldásakor, milyen műveletek hajtódnak végre, milyen sorrendben. Az elnevezés Muhammad ibn Músza l-Hvárizmi perzsa-arab tudós nevének latinos formájából alakult ki (Algorithmi).

Az algoritmus műveletekből és vezérlő szerkezetekből épül fel. A vezérlő szerkezet a feladat műveletre bontását és végrehajtásának sorrendjét írja le. Vezérlő szerkezetek:

- Szekvencia(műveletsor),
  - Egymás után végrehajtandó cselekmény sorozata
- Szelekció(kiválasztás),
  - Lépések közötti választás
- Iteráció(ismétlés, ciklus)
  - Valamelyik tevékenység sorozat ismételt végrehajtása

A program fejlesztése során hasznos dolog ha érthető módon írjuk le az algoritmust. Leírási módok:

- Mondatszerű leírás
- Grafikus ábrázolás:
  - Folyamatábra
  - Struktogram
  - Jackson-ábra

A mindennapokban is rengeteg algoritmust használunk a tudatunkon kívül. Egy egyszerű hétköznapi példa erre: Éhesek vagyunk és szeretnénk enni egy halat. Első lépésként megnézzük, hogy van-e otthon hal. Ha van(és minden más feltétel is adott), elővesszük, elkészítjük, megesszük és jól lakunk. Különben, ha nincs otthon elmegyünk a legközelebbi boltba és megnézzük, hogy ott van-e hal. Ha van megvesszük, hazamegyünk és a fentebb leírtak végrehajtódnak. Tegyük fel, nem volt a boltban hal, de

mi minden áron szeretnénk enni, mit teszünk? Elmegyünk a következő boltba és megnézzük ott. Ezt már nevezhetjük egy egyfajta ciklikusságnak. A ciklusoknak is van többféle típusa. A mi esetünkben egy hátultesztelős ciklust alkalmazva próbálunk meg halat szerezni. Ez azt jelenti, addig megyünk a következő boltba, amíg nem találunk halat. Pár boltot bejárva a 4. boltban találunk halat, örülünk neki, a feltétel igaz lett, hogy van hal. Megvesszük haza megyünk vele, ezzel az algoritmusunk elején vizsgált feltételt újra vizsgáljuk, hogy van-e hal otthon? Ebben az esetbán már igaz lesz ez az állitás, tehát azokat a műveleteket hajtjuk végre, amik akkor teljesülnek, ha a van otthon hal feltétel igaz.

Ebben a példában szerepeltek az elágazásos algoritmusok, aminek két fajátja van:
HA feltétel AKKOR utasítás

HA feltétel AKKOR utasítás KÜLÖNBEN utasítás_2

A feltétel logikai eredményt ad vissza: Igaz, Hamis.

Ezen kívül szó volt a ciklusokról, pontosabban a hátultesztelős ciklusról. A ciklus nem más mint egy feladat többszöri elvégzése. Alapvetően két fajtája van, feltételes és számlálós. A feltételes ciklusok közé tartozik a már előbb használt hátultesztelős, a másik az előltsztelős. Utóbbinál a feltételt vizsgáljuk először ami, ha igaz, ismétlődés történik. A számlálós ciklusnál is egy egyfajta feltétel teljesül, de ez már teljesen más feltétel. Ezeknél mindig egy számot használunk, amivel azt érhetjük el, hogy addig fusson a ciklus amíg I<N(I=1, N=10), az I-t a ciklusban mindig amíg igaz a feltétel megnöveljük. Amikor már nem igaz, hogy I<N akkor érünk a ciklus végére. Létezik egy másik számlálós változat, ami I-től megy N-ig. Itt is növeljük az I-t, amikor az I = N, akkor vége a ciklusnak.

Kevés, de egyértelmű szabályt kell kialakítani az algoritmusok leírására. Ezek számunkra legyenek kényelmesek, de ne menjen az egyértelműség és a precizitás rovására.

Nézzünk pár ilyet:

- Az adatokat beolvasó és kiíró algoritmus az ablak szerepét látja el a külvilág felől és a program felhasználója felé.
- A változóknak az értékadását az értékadó utasítások látják el.
- Feltételektől függő utasításokat a feltételes utasítások, elágazások végzik.
- A felhasznált és még hiányzó eljárások finomítása.
- Rendelkezni kell az adatok és típusok leírására szolgáló eszközökkel is.

5. Alkalmazás megvalósítása

5.1. Feladat specifikáció

A programozási folyamat első lépése a feladat pontos megfogalmazása. Ez a specifikáció jöhet a megrendelőtől, vagy akár a megrendelővel közösen is el lehet készíteni. Első körben úgynevezett konyhanyelven fogalmazzuk meg a feladatot/feladatokat, ami lehet akár egy mondat is. Ezt követően a program készítője a saját szemszögből fogalmazza meg a feladatot, ami már szakmai nyelven történik. A tervezésben a legfontosabb, hogy tisztába legyünk a bemenő és a kimenő adatokkal! A bemenő adatokra elő-, a kimenőre utófeltételeket fogalmazunk meg.

5.1.1. Egyszerű program

Egy egyszerű program leírása a legtöbb esetben egy mondatban megfogalmazható. Szakmai gyakorlat beszámolómban is említett specifikáció: „Írjunk egy olyan programot, amely egy Excel fájl adatait, bizonyos időközönként feltölt egy adatbázisba.” Ez egy igencsak egyszerű program, bár feltételek itt is vannak. Pl.: léteznie kell a fájlnak amit fel akarunk tölteni, ha nem változott semmi az előző feltöltés óta akkor ne töltse fel stb..

5.1.2. Terjedelmesebb program

Terjedelmesebb programok esetében nem elegendő egy mondattal leírni, hogy mit is szeretnénk, nem intézhetjük el annyival, hogy „Készítsünk egy programot a

szerszámok leltározásához.”. Egy ilyen feladat leírással, nem sok mindent lehet kezdeni. Szükségünk van több információra, mint pl.:

- Csak polcra rakás és onnan levétel-re van szükség? Esetleg több helyen való tárolásra?
- A szerszámok nyomon követése, hogy történjen? Bejelentkezési felület kell-e, hogy tudjuk ki hova mozgatta az adott szerszámot?
- Kell-e szerkesztői felület? Külön jogosultságok? Stb…

Ezeket a specifikációkat legtöbb esetben a megrendelő és a programozó együtt beszélik meg.

5.1.3. Specifikáció során használható szimbólumok

A specifikáció leírása történhet szöveges, matematikai és logikai szimbólumok használatával.

A legtöbbször használt jelek a következők:

- $\wedge$: ÉS logikai művelet
- $\vee$: VAGY logikai művelet
- $\rightarrow$: AKKOR logikai művelet (implikáció)
- $\leftrightarrow$: AKKOR és CSAK AKKOR logikai művelet (ekvivalencia)
- $\forall$: „minden” logikai kvantor
- $\exists$: „létezik” logikai kvantor

5.2. Tervezés

A programkészítés következő lépésében, a feladatokat már tudva, jöhet a tervezés. Fontos, hogy a megfogalmazott feladatok pontosak és érthetőek legyenek, különben nehéz dolgunk lesz a programozás, de már a tervezés során is. A tervezési fázisban a programozó végig gondolja, hogy a megadott specifikáció alapján milyen eszközök kellenek a megvalósításhoz. Annak ellenére, hogy a számítógép felhasználásával megoldott feladatok nagy része nem matematikai problémaként fogalmazódik meg, az eredmény előállításához legtöbbször matematikai és logikai

eljárások sorozatos alkalmazására van szükség. A megfogalmazott feladatokhoz ezért általában matematikai modellt kell keresni.

Minden olyan eszközt, módszert, amelyek lehetővé teszik a programunk specifikálását a programtervezési eszközök közé sorolhatunk.

- Adatleírást segítő eszközök:
    - Matematikai és logikai szimbólumok
    - Típusleíró eszközök
    - Adatmodell-leíró eszközök
- Algoritmusleíró eszközök
- OOP tervezést segítő eszközök

Szerencsére nagyon sok előre elkészített algoritmusok vannak amiket szabadon felhasználhatunk a programunkhoz, így nem kell ugyan azt amit valaki egyszer már megcsinált (vagy akár pont mi), újra és újra elkészíteni.

5.2.1. Részekre bontás

A feladatot, ha minél több kisebb, de egyenlő nehézségű feladatra bontunk és azokat külön készétjük el, kisebb az esélye annak, hogy az egész nagy feladatot el kell dobnunk. Egy kisebb rész, ha valami miatt megoldhatatlan egyszerűbb visszalépni egy előzőre és ez fontos. Ne féljünk visszalépni és kitörölni, amit már megcsináltunk és nem jó. Pont ezt mondja ki a „Vissza az ősökhöz” elv is.

5.3. Felhasználói felület tervezés

Egy átlagos felhasználó nem kíváncsi arra, hogyan működik a program, ahogyan egy átlagos autóvezetőt sem érdekel, hogyan működik az autója. A felhasználónak az a fontos, hogy tudja használni, átlátható legyen, kapjon visszajelzéseket stb. Talán ez a legnehezebb egy program készítése során, hogy minden felhasználó tudja használni a programot. Nem szabad olyan elemeket használni a felületen, ami megzavarhatja vagy nem egyértelmű a felhasználónak. A felületet úgy kell megterveznünk, hogy minden egyértelmű legyen, mi vezessük végig a felhasználót a folyamaton, ne legyen semmilyen

kétsége afelől, hogy melyik gombot nyomja meg. Mindig mutassuk az utat! Egy rosszul megtervezett felület azt éreztetheti a felhasználóval, hogy a program ami a munkája segítésére készült inkább gátolja őt.

A felhasználó felületet érdemes úgy megtervezni, hogy a jövőben könnyedén tudjuk bővíteni azt akár új funkcióval is.

5.3.1. Prototípus-készítés

Mielőtt nekiállnánk a végleges verziónak, csináljunk egy-két prototípust amit meg tudunk mutatni a megrendelőnek. Jobb esetben kiválasztja az egyik verziót, de előfordulhat az is, hogy több prototípusból szeretne összegyúrni egyet. Ha olyan fejlesztői környezetben programozunk amely támogatja a grafikus felületekkel való munkát, akár ott is a projektünk könyvtárában létrehozhatunk több formot. Amennyiben sikerült kiválasztani az egyiket, abban rögtön el is tudunk kezdeni kódolni.

5.4. Megvalósítás(kódolás)

Ha a terv elkészült, jöhet a terv megvalósítása. Az eddigi munkánk jóesetben „univerzális” azaz, ha valaki megnézni és tudja értelmezni, bármilyen programozási nyelvre le tudja fordítani. A kód megírása előtt, ha még nem tettük meg válasszunk egy programozási nyelvet. Vannak olyan szoftverek, amelyek a kész tervet, a struktogrammot lefordítja és lefuttatja.

Ha kiválasztottuk a programozási nyelvet, jöhet a kódolás.

5.4.1. Változók

Még egy egyszerű programnál is elengedhetetlen a változók használata. Hogy mi az a változó? A változó az információ ideglenes tárolására szolgál. Leggyakrabban használt változó típusok:

- bool – logikai (igaz/hamis),
- char – karakter(1db),
- string - szöveg,
- int – egész szám,

- double – tört szám.

Következőképpen deklaráljuk(létrehozzuk) őket: **típus név**; pl: int number; A létrehozott változónak értéket is kell adni, azaz definiálni. pl number = 5; Ezt megtehetjük akár egyben is pl: int number = 5;.

Szám típusú változónál simán a változónév után egy „=” jellel és az érték megadásával történik, míg a szöveges típusokat (string, char) „” -jelek közé kell írnunk. pl: string szo=”szöveg”;

Most, hogy van egy int típus változónk aminek az értéke 5, a matematikában megszokott műveleteket tudjuk vele végrehajtani.

Ami elsőre szokatlan lehet, hogy szöveget is össze tudunk adni (összefűzni).

```
string szo1 = „alma”;
string szo2 = „fa”;
string szo3 = szo1 + szo2;
```

A szo3-nak az eredménye így „almafa” lett. Ezt ki is kell íratnunk, ha azt akarjuk, hogy a felhasználó is lássa. Egy konzolos alkalmazásba nyilván a konzolra íratjuk ki a Console.WriteLine(szo3); - paranccsal. Egy ablakos alkalmazásban ezt rengeteg féle képen megtehetjük, pár példa: Label, TextBox, ListBox, de akár egy felugró kis ablakba is.

Előfordulhat, hogy két különböző típussal kell műveletet végrehajtani, például maradékos osztást kell csinálni. Ha két int típust osztunk egymással pl.: 5/10 ennek az eredménye 0 lesz, mert az int nem kezeli a tört számokat. Ilyenkor a következőket tehetjük : castolás vagy konvertálás.

```
int szam1 = 5;
int szam2 = 10;
double eredmeny =  (double)szam1/szam2;  //castolás
double eredmeny2 = Convert.ToDouble(szam1) / szam2; // konvertálás
```

5.4.2. Elágazások

Most, hogy tisztába vagyunk a típusokkal jöhet az „igazi” programozás. Nagyon sok esetben használunk elágazásokat(szelekciót), aminek minimum egy vagy több ága is lehet. Több ág esetén összetett elágazásról beszélünk.

```
if(feltétel){
        //utasítás
} elseif(feltétel2) {
        //utasítás
} else{
        //utasítás
}
```

A zölddel kijelölt rész nem szükséges az elágazás működéséhez, de több elseif ág is használható.

A feltételhez relációs és egyenlőségi operátorok írhatunk:

- egyenlő == (dupla =)
- kisebb < , kisebb egyenlő <=
- nagyobb >, nagyobb egyenlő >=
- nem egyenlő !=

Amikor egy feltételt vizsgálunk az if()-ben, igaz vagy hamis értéket kapunk. Igaz esetén az if ágban lévő kód fut le, ha nem igaz a feltétel, azaz hamis értéket ad, az else ágban lévő kód fut le. Több feltétel esetén az a kód fog lefutni amelyik igaz, feltéve, ha egyik sem, akkor az else fog lefutni.

if(szam<10){
Console.WriteLine(„A bekért szám kisebb mint 10!”);
} else{
	if(szam<10){
	Console.WriteLine(„A bekért szám kisebb mint 10!”);
	} elseif(10<szam<20){

Console.WriteLine(„A bekért szám nagyobb mint 10”);
}
	Console.WriteLine(„A bekért szám10 és 20 között van”);
	}else{
	Console.WriteLine(„A bekért szám nagyobb mint 20”);
	}

Ebben a két példában jól látszik, hogy mi a történik a több elágazás esetén.

Elágazásokat egymásba is lehet építeni. Ez abban az esetben jó, ha például meg akarjuk vizsgálni egy számról, hogy az kétjegyű-e, ha igen az páros vagy páratlan?

```
if(szam<100){
        if(szam % 2){
        Console.WriteLine(„A bekért szám kétjegyű és páros!”);
        }else{
        Consolne.WriteLine(„A bekért szám kétjegyű és páratlan!”)
} else{
Console.WriteLine(„A bekért szám nem kétjegyű!”);
}
```

5.4.3. Ciklusok

A változók és elágazások után fontos és szinte nélkülözhetetlen a programozásban az iterációk (ciklusok). Három típus van:

- Feltételes
  - Elöltesztelő
  - Hátultesztelő
- Számláló
- Iteráló

A feltételes ciklus egyfajta feltételhez kötjük az ismétlődésének számát. Ezek közül is kettőt különböztetünk meg, elől- és hátul-tesztelős.

Előltesztelős	Hátultesztelős

while(feltétel){
utasítások
}
	do{
	utasítások
	}while(feltétel)

A számláló ciklusnál is feltételhez van kötve az ismétlődése, de itt konkrétan meg van határozva, hogy hányszor fusson le.

```
for(i=0; i < n; i++){
utasítások
}
```

Végül az iteráló ciklus, ami mindenképpen végig megy az adott tárolón, ami lehet egy tömb, lista stb.

```
foreach(item in tömb){
utasítások
}
```

5.4.4. Tömbök, Listák

Még nem volt szó a tömbökről és a listákról. Ezek egy adott típusú változók tárolására alkalmas.

Kezdjük a tömbbel, amit úgy kell elképzelni, mint egy polc rendszert, ami fel van számozva és a polcokon vannak pl. gyümölcsök. Létrehozásánál úgy, mint a változóknál is, adunk neki egy típust, nevet viszont akár több értéke (azaz ebben az esetben már elemről beszélünk) lehet. string[] polc = new string() {„alma”, „banán”,”körte”};

Hivatkozni egy elemre annak a helyével tudunk. A „polc” nevű tömbünknek jelen esetben 3 eleme van, és szeretnénk a banánt kiíratni.

```
Console.WriteLine(polc[2]); // helytelen
```

```
Console.WritLine(polcs[1]) // helyes;
```

Miért helytelen a polc[2]? Azért, mert a tömbök indexelése, a polcok száma 0-tól kezdődik. Tehát a banán helye igaz, hogy a 2. helyen van, de az indexe 1.

Meg szeretnénk jeleníteni a polc összes elemét, amit egy számláló ciklussal meg is tehetünk hacsak nem akarjuk egyesével leírni, ami egy 1000 elemű tömbnél kicsit fárasztó lehet.

```
for(i=0; i< 3; i++ ){
Console.WriteLine(i+". elem:"+polc[i]);
}
```

A lista megismerése után, gyakorlatilag a tömböt el is felejthetjük. A tömbbel szemben, dinamikusan változtatja a méretét, tehát nem kell megadnunk előre, hogy 10 elemet szeretnénk benne tárolni. A lista egy generikus típus, tehát bármilyen típusból létrehozhatunk saját listát (int,string,osztály stb…).

```
List<T> lista = new list<T>() // T mint típus
```

Elemet hozzáadni az Add() metódussal lehet. lista.Add(1);lista.Add(2)… Kiíratni mind a for mind a foreach ciklussal lehetséges.

```
for(i=0; i < lista.Count; i++){
Console.WriteLine(lista[i]);
}
```

```
foreach(item in lista){
Console.WriteLine(item);
}
```

A listának több metódusa is van ami plusz egy pont a tömbbel szemben. Pár példa:

- List.Insert(mit,hova)
- List.Remove(mit), List.RemoveAt(honnan)
- List.Clear()
- List.Sort()

5.4.5. OOP

Mivel C#-ról van szó ami igazán az OOP-ról szól azaz a objektum orientáltságról, így ez sem maradhat ki.

Az OOP egy olyan módszer, ami igazából a valós világ modellezésén alapul. Az összetartozó adatokat és azokkal műveleteket végző eljárások, függvényeket egy egységbe, ún. osztályba szervezünk. Egy osztályt nevezhetünk mintának, melyből példányokat tudunk készíteni, ez lesz egy objektum, amit az adott osztály alapján példányosítunk. Egy objektum változóit mezőnek vagy tulajdonságnak hívjuk. Nézzünk egy példát:

```
Class Ember{
        public string Nev;
        public int Eletkor;
        public date SzuletesiIdo;
        public string Lakhely;
}
```

Példányt létrehozni a következő képen lehet, akár többet is:

```
Ember e = new Ember();
e.Nev = „Példa János”;
e.Eletkor = 33;
e.SzuletesiIdo = 1976 .01.12;
e.Lakhely = „Budapest”;

Ember e2 = new Ember();
e2.Nev = „Példa Béla”;
```

Az objektum osztályok továbbfejlesztési lehetősége az öröklés. Minden osztály amit létrehozunk már egy származtatott osztály, őse az object. Az eredeti osztályt ősosztálynak(szülő) nevezzük. Az új, továbbfejlesztett osztályt származtatott osztálynak(gyerek). Öröklés lényege, hogy a meglévő osztályunkat felhasználva létre hozunk egy másikat, ami megkapja az összes tulajdonságát, metódusát, amelyeket ki lehet egészíteni újabbakkal.

```
Class Ősosztály{
}
Class Gyermekosztály : Ősosztály{
}
```

Minden származtatott osztálynak egy szülője lehet, de egy szülőnek több gyereke is.

5.5. Üzembe helyezés, tesztelés

A kódolást követi a tesztelés, hiba keresés mivel a program első változata általában nem hibátlan. Első lépésben mi teszteljünk a programot, hogy minden funkciója működőképes és az helyesen is működik. A nagyobb hibák elsőre szemet szoktak szúrni, de megeshet egy hosszas keresgélés után sem vesszük észre a hibát. Tesztelés során előkerülő hibákat vagy azonnal javítjuk vagy több hibával együtt. Olyan metódusokban, amelyek egymásra épülnek, hibát találunk, célszerű végig nézni az összes hozzá kapcsolódó metódust.

Két féle tesztelési technikáról beszélhetünk:

- Feketedobozos: specifikáció alapján tesztelünk
- Fehérdobozos: forráskód alapján tesztelünk

A fekete dobozostesztelés esetén a specifikáció alapján teszteljük a programot. Ilyenkor tudjuk a bemenő és kimenő adatokat is, így futtatjuk le a programot. Az így kapott eredményeket össze tudjuk hasonlítani és ellenőrizni az adott funkció helyességét.

A fehérdobozos tesztelés esetén magát a struktúrát teszteljük. Általában ezeket:

- kódsorok
- elágazások
- metódusok
- osztályok
- funkciók

Amikor egy olyan állapotához ér a program, hogy a megrendelő által is tesztelhető, adjuk ki neki tesztelésre. A legtöbb kis apró hibát a program használója fogja

megtalálni. Ez azért van, mert a program készítője jellemzően úgy teszteli a programot, ahogy annak működnie kell. Ezt a verziót érdemes kiadni, az éles üzembe helyezés előtt, ugyanis nem csak logikai hibák lehetnek a programban, lehet, hogy magát a kezelő felületet is át kell alakítanunk kényelmesebb használat érdekében.

5.5.1. Hiba keresése, javítás

A programban kétféle hiba fordulhat elő: szintaktikai(formai) és szemantikai(logikai) hiba.

A szintaktikai hiba megtalálásában segít a fejlesztői környezet (már ha támogatja ezt), tippeket is ad a javításában.

Ennél sokkal nehezebb a logikai hibák feltárása, amit a már előbb leírt módon tehetünk meg. Lépésről lépésre végig megyünk a program funkcióin és meggyőződünk arról, hogy működnek. Először a megfelelő bemenő adatokat beírva végrehajtjuk, majd elemezzük az eredményt. Amennyiben valamelyik eredmény nem helyes, fel kell kutatni a logikai hiba helyét és javítani azt.

Egy olyan beviteli mezőbe, ahova egy egész számot várunk, egyéb karaktert beírva hibára futhat a programunk. Szerencsére a legtöbb programozási nyelvben vannak hibakezelési módszerek. Az eljárás lényege, hogy a kódunk köré egy úgynevezett figyelő rendszert állítunk, ami könnyedén lekezeli(elkapja) az olyan hibákat, melyek a programunk futása közben keletkezik.

Ennek a működése a következő képen néz ki:

```
try {
                    // Futó program rész
} catch (error) {
                    // Hiba esetén lefuttatott rész
} finally {
                    // Minden esetben lefutó program rész
}
```

Ez annyit tesz, hogy ha a programban hiba generálódik, akkor megszakítja és kilép és a kialakult hibáról egy üzenetet ad vissza, amiben a hiba okait írja le. Régebben ezek az üzenetek elég nyersek voltak, de ma már igencsak érthetőbb formában jelennek meg. Persze csak abban a kód részletben kezeli a hibát, amit a

try blokkba írunk. A program futás közben sorban hajtja végre az utasításokat, ha hibához érkezik a program, kivétel keletkezik, átugrik a catch blokkba. Itt kezeljük a hiba előfordulását, tehát hiba esetén nem a try-ba tér vissza, hanem a catch-ben lévő kód fog lefutni.

Több kivételt is tudunk kezelni, előfordul, hogy nem számítunk mindenre. Ekkor lép be az „exception”, segítségével minden olyan hibát ellenőrizhetünk, ami előfordulhat a programban. Legegyszerűbb példa, egy beviteli mezőbe számot várunk, de egy betűt írnak be, a try-catch elkapja a hibát és vissza tudunk jelezni a felhasználónak, hogy hibás adatot adott meg.

```
try{
Console.WriteLine(„Írj be egy számot!”);
var n = Convert.ToInt32(Console.Readline());
}
catch(Exception ex){
Console.WriteLine(ex);
}
```

Itt az „ex” változóval íratjuk ki a hibát, de akár saját hiba üzenetet is meg tudunk adni.

A try-catch-nek van egy harmadik ága, ami a „finally”. Ezt nem kötelező használni, enélkül is működik a hibakezelés. Ide olyan kódot írunk amint minden esetben szeretnénk futtatni.

Hibák javítása során szükségünk lehet új specifikációra, de akár új tervre is.

5.6. Hatékonyságvizsgálat

A hatékonyságvizsgálatot egy szintre tenném a teszteléssel, akár a programunk tesztelése közben is megtehetjük ezt a lépést. Személy szerint szeretem a már majdnem kész programot vizsgálni hatékonyság szempontjából, így tudjuk tesztelni egyszerre az összes funkciót, azoknak gyorsaságát, bonyolultságát. Szintén a hatékonyság fő mérőszáma a gyorsaságon és a bonyolultságon kívül a tárhelyhasználat.

Amennyiben lassúnak bizonyult a programunk működése, nézzük át a kódunkat és optimalizáljuk azt. Keressünk olyan kódrészeket, amiket nem is használ a programunk, vagy meg tudjuk oldani a feladatot egyszerűbben, rövidebben. A felesleges változók használata is lassít a futási időn. Készítsünk eljárásokat azoknak a kódrészleteknek, amiket többször is felhasználunk.

5.7. Dokumentáció

A dokumentálás nagyon fontos, mind a fejlesztői, mind a felhasználói részről.

5.7.1. Dokumentáció fajtái

A dokumentálásnak négy fajtáját különböztetjük meg:

- Fejlesztői
- Felhasználói
- Programismertető
- Telepítési útmutató

A fejlesztői dokumentáció lényege, hogy ha évekkel később elő kell vennünk, vagy valaki másnak elő kell vennie az alkalmazást, és módosítania kell, vagy csak szeretné megismerni a működését, nagyban megkönnyítse a dolgát. Amit érdemes lehet beleírni:

- specifikáció,
- futási környezet, (perifériák használata)
- fejlesztési környezet, programozási nyelv(ek)
- algoritmusok, típusok, osztályok, függvények
- tesztesetek
- fejlesztési lehetőségek
- a készítő adatai

A felhasználói dokumentáció, mint a nevében is benne van, a felhasználókhoz szól. Ebben leírásra kerül a program használatának helyes módja. Következőkre lehet szükség:

- összefoglaló a működésről
- futási környezet
- bemenő adatok
- hibaüzenetek és jelentései

A programismertető célja a leendő felhasználóknak szól, hogy a program megfele-e az igényeinek, reklám jellegű.

Telepítési útmutató a nagyobb programok esetében szükségesebb. Itt szerepel minden olyan fontos információ lépésekre bontva, amik segítenek a program telepítésében.

5.7.2. Dokumentáció tulajdonságai

Három fontos tulajdonságra kell figyelnünk a dokumentáció megírása közbe.

- Szerkezet
  - ne legyen túl hosszú
  - ne legyen túl rövid
  - legyen érthető és jól tagolt
  - legyen tömör
  - legyen pontos
- Forma
  - legyen tartalomjegyzék
  - fejezetekre bontás
- Stílus

A stílus függ, a dokumentáció fajátjától.

- Programismertető
  - dicsérni kell a programot, kiemelni a jó tulajdonságait
- Felhasználói dokumentáció
  - részletes szöveges leírás
- Fejlesztői dokumentáció
  - legfontosabb a pontosság
- Telepítési útmutató
  - utasítások, teendők, válaszok

5.8. Karbantartás

Egy program üzembehelyezésével még nincs vége a dalnak. A programunk kiadása után, a tesztelések és hibajavítások ellenére előfordulhat, hogy előjön egy-két hiba amit ki kell javítanunk. Ilyenkor jöhet egy újabb hibakeresés, javítás majd tesztelés.

6. Alkalmazás bevezetése

Az elkészült programunk eljutott abba a szakaszba, hogy bevezessük élsebe. Azonben milőta használandó számítógépre feltelepítenénk, érdemes megvizsgálni, hogy minden olyan periféria adott, ami kellhet és megfelelően működnek. Nem elhanyagolható a számítógép teljesítménye sem, ugyanis egy rossz hardverrel rendelkező számítógépen, ha nem fut a programunk anélkül, hogy lefagyna vagy akadozna, ront a felhasználói élményen. Ez ahoz vezethet, hogy a felhasználók nem tudják és nem is akarják használni a programunkat.

Amit még jó ha megnézünk, hogy a program által használt framework telepítve van-e a gépre. Illetve a regionális beállításknál nézzük meg az elválasztó karaktert, hogy mire van beállítva, ha szükséges állítsuk át. Ez általában „ . ” vagy „ , ”.

Mint már említettem, egy program bevezetésével még nincs vége az ezzel kapcsolatos munkánknak. Karban kell tartanunk, frissítéseket kiadni, ha van rá szükség.

7. Összegzés

A szakdolgozatom írása közben, körülbelül félúton, rájöttem arra, hogy nem egy konkrét alkalmazás fejlesztésének a folyamatát akarom bemutatni. Inkább egy olyan általános leírást készítettem, ami az alkalmazások elkészülésének a menetéről szól egy kis programozási alapokkal. Próbáltam úgy összeállítani, hogy egy olyan személy, aki érdeklődik az alkalmazások fejlesztése iránt, el tudja képzelni, hogy miből is áll egy fejlesztési folyamat. Ezen folyamatok lépései, nehézségei akár szépségei alapján, ha már egy kicsit is közelebb került ahhoz a döntéshez, ami alapján elhatározza, hogy fejest ugrik a programozás világába, már megérte megírni.

Szerencsémre már középiskolától a C# és az SQL nyelveket tanulhattam. Persze több programozási nyelvet is kipróbáltam közbe, de ez a kettő az, ami végig megmaradt.

Érdemes kipróbálni több nyelvet is, mert nem biztos, hogy elsőre megtaláljuk azt amelyik tetszik.

8. Irodalomjegyzék, forrásmegjelölés

C#

http://compalg.inf.elte.hu/~tony/Informatikai-Konyvtar/09-Programozas%20C-sharp%20nyelven/Programozas-Csharp-nyelven-Konyv.pdf

SQL

http://www.agr.unideb.hu/~agocs/informatics/05_h_ecdl/ECDLcd/CD%202/modul5/10_05_01.htm

Programkészítési elvek

http://progalap.elte.hu/downloads/seged/eTananyag/lecke24_lap1.html