



**NEUMANN JÁNOS
INFORMATIKAI KAR**



SZAKDOLGOZAT

**OE-NIK
2012**

Hallgató neve:
Hallgató törzskönyvi száma:

**Szűcs Krisztián
T 348/FI38878/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertechnológia Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Szűcs Krisztián**
Törzskönyvi száma: **T 348/FI38878/N**

A dolgozat címe:

Informatikai ismeretek oktatását támogató stratégiai kvízzjáték fejlesztése

Developing strategic quiz game for IT skill training

Intézményi konzulens: **Erdélyi Krisztina**
Bíró: **Tarcsi Ádám, ELTE Informatikai Kar**

Beadási határidő: **2011. május 16.**

A záróvizsga tárgyai: **Architektúrák**
Szoftvertechnológia

A feladat

Fejlesszen stratégiai kvízzjátékot, amely az egyetemi évek alatt megszerezhető informatikai ismeretek elsajátítását segíti mérnök informatikus hallgatók számára. A kérdések feltöltésére a felhasználóknak is legyen lehetősége, mely elbírálás után bekerül a játékba. A játékosok egymás ellen játszanak, majd a helyes válaszok és az elfoglalt területek arányában növelik pontszámukat, ezzel rangsort felállítva. A játékosok legyenek képesek egymással kommunikálni.

Nézze meg, milyen adatbázis-elérési technológiákat támogat a .NET keretrendszer (DataReader, DataSet, LINQ to SQL, Entity Framework). Hasonlítsa össze a technológiákat az elkészítendő játék igényeinek megfelelően. Készítsen teszteseteket, amelyek az adatbázis-elérés gyorsaságát mérik. Válassza ki a legmegfelelőbbnek ítélt technológiát, választát indokolja.

Tervezze meg és valósítsa meg a szoftvert. Ismertesse a teszteredményeket. Foglalja össze a fejlesztés során szerzett tapasztalatait.

A dolgozatnak tartalmaznia kell:

- a .NET adatbázis-elérési technológiájának összehasonlító elemzését,
- az elkészítendő stratégiai játék rendszertervét,
- a szoftver prototípusát,
- a tesztelés eredményeit,
- a fejlesztés során szerzett tapasztalatok összefoglalását.

Ph.

.....
Dr. Tick József
intézetigazgató

A szakdolgozat elévülésének határideje: **2013. május 16.**
(OE TVSz 32.§ 8. pont szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens



HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Budapest, 2012. 12.20.

.....
hallgató aláírása



KONZULTÁCIÓS NAPLÓ

Hallgató neve:
Szűcs Krisztián

Neptun Kód:
JQUI4W

Tagozat:
Nappali

Telefon:
0620-258-6744

Levelezési cím:
szucs.krisztian09@gmail.com

Szakdolgozat címe magyarul:
Informatikai ismeretek oktatását támogató stratégiai kvízzjáték fejlesztése

Szakdolgozat címe angolul:
Developing strategic quiz game for IT skill training

Intézményi konzulens:
Erdélyi Krisztina

Külső konzulens:

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.			
2.			
3.			
4.			

A Konzultációs naplót összesen 4 alkalommal az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a „Szakdolgozat” tantárgy aláírási követelményét teljesítette.

.....
Intézményi konzulens

Budapest, 20.....



KONZULTÁCIÓS NAPLÓ

Hallgató neve:
Szűcs Krisztián

Neptun Kód:
JQUI4W

Tagozat:
Nappali

Telefon:
0620-258-6744

Levelezési cím:
szucs.krisztian09@gmail.com

Szakdolgozat címe magyarul:
Informatikai ismeretek oktatását támogató stratégiai kvízzjáték fejlesztése

Szakdolgozat címe angolul:
Developing strategic quiz game for IT skill training

Intézményi konzulens:
Erdélyi Krisztina

Külső konzulens:

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.			
2.			
3.			
4.			

A Konzultációs naplót összesen 4 alkalommal az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a „Szakdolgozat” tantárgy aláírási követelményét teljesítette.

.....
Intézményi konzulens

Budapest, 20.....

Tartalomjegyzék

1. Bevezetés	1
2. Hasonló rendszerek vizsgálata	2
2.1. Összehasonlítás.....	2
3. Adatbázis elérési módok a .NET-ben.....	3
3.1. A kapcsolat-alapú modell.....	3
3.1.1. A DataReader tulajdonságai	4
3.1.2. Mikor használjuk?.....	5
3.2. Kapcsolat-nélküli modell	5
3.2.1. A DataSet tulajdonságai	6
3.2.2. A DataSet adatszerkezete.....	7
3.2.3. Erősen típusos DataSet.....	7
3.2.4. Mikor használjuk?.....	8
3.3. A LINQ jelentősége	9
3.3.1. LINQ to SQL	11
3.3.2. Mikor használjuk?.....	12
3.4. Entity Framework	12
3.4.1. Felépítése	12
3.4.2. Használata	14
3.4.3. Mikor használjuk?.....	15
3.5. Tárolt eljárások	15
4. Adatbáziselérések tesztelése, kiértékelés.....	16
4.1. A teszt környezet	16
4.2. A tesztek	17
4.2.1. Első eset	17
4.2.2. Második eset	18
4.2.3. Harmadik eset	19
4.2.4. Negyedik eset	20
4.3. Összegzés, technológia kiválasztása	20
5. A rendszer specifikációja	22
5.1. Adminisztrátor szerepkör	22
5.2. Játékos szerepkör	24
5.3. A játékmenet.....	24
5.3.1. Belépés a játékba	24
5.3.2. Játéktér.....	24
5.3.3. Kérdés típusok.....	25
5.3.4. Területfelosztás	25
5.3.5. Területfoglalás.....	25
5.3.6. A játék hatása a használóira	25

6. Az elkészítendő stratégiai játék rendszerterve.....	26
6.1. Fejlesztési technológiák vizsgálata	26
6.1.1. Flash	26
6.1.2. PHP	26
6.1.3. ASP.NET	27
6.1.4. Összehasonlítás	27
6.2. A rendszer architektúrája	28
6.2.1. Felépítés	28
6.2.2. Adatbázis	28
6.2.3. DataAccessLayer	30
6.2.4. BusinessLogic.....	30
6.2.5. User Interface.....	31
6.2.6. IIS.....	33
6.3. Játék logika	34
6.3.1. Player osztály.....	34
6.3.2. RobotPlayer osztály	35
6.3.3. GameQuestion osztály	35
6.3.4. GameTipQuestion osztály	36
6.3.5. GameChooseQuestion osztály	36
6.3.6. GameButton osztály	37
6.3.7. GameLogic osztály.....	37
6.4. A játékmenet megjelenítése.....	40
6.4.1. Az aktuális adatok tárolása.....	40
6.4.2. Belépés a játékba	40
6.4.3. A területfelosztás játékmenete	42
6.4.4. A területfoglalás játékmenete	44
6.4.5. A játék vége	44
7. Implementáció	46
7.1. Programnyelv kiválasztása.....	46
7.1.1. Visual Basic .NET és C#	46
7.1.2. Visual Basic .NET.....	46
7.1.3. C#.....	46
7.1.4. Összehasonlítás	47
7.2. Konvenciók	47
7.2.1. Pascal casing	48
7.2.2. Camel casing.....	48
7.2.3. Kommentek	48
7.3. Fejlesztési döntések.....	48
7.3.1. Felhasználók kezelése	48
7.3.2. Megjelenítés	49
7.3.3. Újratervezés	49
7.4. Kompatibilitás	50
8. A rendszer tesztelése	51
8.1. Automatizált tesztek	51
8.1.1. Teszt környezet	51
8.1.2. A tesztelés célja	52

8.1.3. A tesztelés folyamata	52
8.1.4. Webes alkalmazás teljesítménytesztelése	52
8.1.5. Terheléses teszt.....	52
8.1.6. Automatizált tesztek kiértékelése	55
8.2. Felhasználói tesztek	55
8.2.1. Teszt környezet	55
8.2.2. A tesztelés célja	55
8.2.3. A tesztelés folyamata	56
8.2.4. Felhasználói tesztek.....	56
8.2.5. Hibák felderítése	57
8.2.6. Felhasználói tesztek értékelése	57
9. Az alkalmazás elemzése, továbbfejlesztési lehetőségek	58
9.1. A rendszer elemzése, alkalmazása	58
9.2. Továbbfejlesztési lehetőségek.....	58
10. Összefoglalás	59
11. Summary	60
Irodalomjegyzék	61

1. BEVEZETÉS

Az Óbudai Egyetemen töltött éveim során sokszor felmerült bennem a gondolat, hogy miért nincs egy olyan szoftver, amellyel könnyedebben, játékos formában lehetne elsajátítani az ismeretanyagot. Ez megkönnyítené a felkészülést az egyes zárthelyikre, vizsgákra mivel a diákoknak nem csak a végtelennek tűnő jegyzeteket kellene átnézniük, hanem amellet egy online interaktív kvízzel is felmérhetnék a tudásukat egymás ellen. A verseny ösztönözné őket a tananyag megjegyzésére, valamint a téves válaszok után a helyes megoldást jobban el tudnák sajátítani, és a valódi megmérettetéseken is könnyebben fel tudnák idézni ezeket a válaszokat. Ennek a programnak az ötletét a Honfoglaló nevű online stratégiai és kvízzjáték adta, amely külföldön ConQUIZtador néven vált széles körben elterjedt közösségépítő műveltségi internetes vetélkedővé. Mivel az én célom is az ismeretek széleskörű felmérése és bővítése, ezért vált számomra követendő példává.

Amit a legjobban hiányoltam a fent említett játékban az, hogy hiányoznak belőle az informatikai jellegű kérdések, ezért elhatároztam, hogy olyan szoftvert készítek, amely hasonlóképpen izgalmas és tanulásra csábító, mint az eredeti, viszont szakmai tartalommal töltöm meg, amelyet a NIK-en szerzett ismeretekből gyűjtök össze. Lehetőség nyílik majd az egyes tantárgyakra külön-külön készülni a hozzá tartozó kérdésekkel, valamint az összes témakörből merítve a teljes tudását is felmérheti a játékos. Több fejlesztési környezetet is megvizsgáltam egy ilyen weboldal elkészítéséhez, mint például a Flash, PHP és az ASP.NET. Ezek vizsgálata után úgy döntöttem, hogy az ASP.NET megfelelő lesz számomra, mivel nincs szükség túl bonyolult grafikára, amely rontja a teljesítményt és itt egyébként is a tartalmi háttér a jelentős.

Az eddigiekben a játékmenetről volt szó, amelyet a későbbiekben még részletesebben ki fogok fejteni, viszont most szeretnék áttérni a szakdolgozatom elméleti hátterére. Egy dinamikus interaktív weboldalnak az egyik legfontosabb követelménye a sebesség. Az adatbázisban tárolt adatokat a lehető leggyorsabban kell a játékosok számára közvetíteni, hogy a felhasználói élmény a legjobb legyen.

Az adatbázisok elérésének sok módja van. A .NET is többféle technikát nyújt ennek megoldására. A továbbiakban több módszert is meg fogok vizsgálni, mint például a DataReaderrel, a DataSettel, a LINQ to SQL-lel, tárolt eljárásokkal, valamint az Entity Framework-kel történő csatlakozást az adatbázishoz. A szakdolgozatomban ezeket fogom összehasonlítani, részletes tesztekkel lemérni a sebességüket, valamint a szakirodalom segítségével kiválasztani a leghatékosabb megoldást, melyet végül a webes alkalmazásomban meg fogok valósítani, így a legjobb teljesítményt és élményt nyújtva a játékosok számára, a rugalmas továbbfejleszthetőséget is figyelembe véve.

2. HASONLÓ RENDSZEREK VIZSGÁLATA

A szakdolgozatomban megvalósítandó webes alkalmazás elkészítéséhez több létező szoftvert is megvizsgáltam. Alapvetően egy interaktív online tanulást segítő játékot szeretnék készíteni, amelyhez a Honfoglaló nevű stratégiai kvízzjáték jó mintának tűnt. Ez egy népszerű magyar weboldal, melyet átlagosan egyidejűleg 5-10 ezer felhasználó használ.

A regisztrációt követően be tudunk lépni a játék fő ablakába, ahol statisztikát kaphatunk eddigi eredményeinkről, valamint a többi játékosról. Ezután elindíthatjuk a játékot, ahol egy új ablakban megjelenik egy felület, ahol chatelhetünk a többi felhasználóval, valamint kiválaszthatjuk, hogy melyik szobába szeretnénk belépni. Lehetőség van általános kérdésekben felmérni tudásunkat, valamint témakörök szerint is választhatunk.

Miután kezdeményezzük a belépést a szerver megvárja, hogy összegyűljön a három játékos, amennyiben ez nem történik meg robotokkal helyettesíti a hiányzókat. Először tippelős kérdésekkel felosztódnak Magyarország területei, majd pedig területeket foglalhatunk az ellenfelektől. A játék végén mindenki a jó válaszok és a megszerzett területek arányában szerez vagy veszít pontot.

A weboldal megjelenítése Flash technológián alapul, valamint javascriptet használ. Az üzemeltetőktől nem sikerült részletesebb információkat kapnom a hardver és szoftver háttérrel, csak a játékok során szerzett tapasztalatomra hagyatkozhatok. A betöltődése lassú, valamint a kérdések és a grafikai elemek megjelenítésekor többször teljesítmény problémákat fedeztem fel.

A játék ConQUIZtador néven Németországban, Oroszországban és Franciaországban is elterjedt. Ezek a verziók főképp az elfoglalandó területben és a kérdésekben térnek el, a felület és a játékmenet lényegében ugyanaz.

A többi interneten megtalálható kvízzjáték általában nem ad lehetőséget a játékosok egymás elleni versenyére, nem ennyire izgalmas, és a népszerűségük sem nagy.

2.1. Összehasonlítás

Az én megvalósításomban nagy hangsúlyt fektetek az informatikai ismeretek bővítésére, hogy a diákok felmérhessék és elmélyíthessék tudásukat a számonkérések előtt. Lehetőség lesz az egész egyetemi tananyagot átölelő versenyekben részt venni, valamint egy-egy tantárgy kiválasztásával az adott témakörből felkészülni. Az elfoglalandó terület a NIK épülete lesz.

A fejlesztéshez a Visual Studio 2010 ASP.NET fejlesztőkörnyezetét választottam, mivel ez rugalmas, biztonságos és gyors weboldalak készítését teszi lehetővé. A Honfoglalóval ellentétben nem fogok flasht használni, mivel a bonyolult grafikai elemek miatt lassabbá teszi a játékot, az ASP AJAX támogatásával pedig megoldható az egymástól független területek gyors frissítése és a Flash Player telepítését sem igényli.

3. ADATBÁZIS ELÉRÉSI MÓDOK A .NET-BEN

Ahhoz, hogy részletes és pontos információkhoz jussak az elemzésemhez átnéztem az irodalmi alkotásokat ebben a témakörben. Rátaláltam a Pro C# 2008 and the .NET 3.5 Platform [\[1\]](#) című könyvre, amely nagy segítségemre volt az ismeretek rendszerezésében és megértésében, valamint a Wikipedia [\[2\]](#) is hasznos információkat nyújtott. A .NET keretrendszer számos hozzáférési módot kínál a különféle adatbázisokhoz, ezeket összefoglalóan ADO.NET-nek (ActiveX Data Objects) hívjuk. Ez egy köztes réteget képez az alkalmazásunkban, amely elkülöníti az adatok fizikai elérését, a rajtuk végrehajtandó műveleteket, valamint a felhasználó számára történő megjelenítést.

A System.Data osztálykönyvtárban található típusok a CLR memória-kezelő protokollját használják ragaszkodva a többi .NET nyelv által is használható felépítéshez, így lehetővé téve hogy programozási nyelvtől függetlenül is azonos módon tudjuk elérni az adatbázisokat. Itt találhatóak a DataReader-hez, a DataSet-hez, a LINQToSQL-hez és az Entity Framework-höz szükséges típusok.

Ezen osztályok bemutatásával fogom megvizsgálni a különböző adatelérési módokat, majd a tulajdonságaik elemzésével döntöm majd el, hogy melyik a legjobb módszer a fejlesztendő webes alkalmazásom számára. Alapvetően kétféle modellt különböztetünk meg. A kapcsolat alapú modellben állandó kapcsolatra van szükség az adatbázissal, a kapcsolat nélküli modellben csak a változásokat küldjük át a hálózaton.

3.1. A kapcsolat-alapú modell

Ezt a módszert használva az alkalmazásunk folyamatos kapcsolatban lesz az adatbázissal a műveletek elvégzése alatt mindaddig, amíg le nem zárjuk az összeköttetést. Ez nagy hálózati adatforgalommal jár és akadályozza az újabb kapcsolatok létesítését. Az ADO.NET számos adatszolgáltatót támogat, melyekhez az elérést biztosító osztályok különböző változatait biztosítja, így a neki megfelelő paraméterezéssel és parancsokkal tudunk kommunikálni. Ezen adatszolgáltatókhoz tartozik az ODBC, az Oracle, a Microsoft SQL Server és az OLE DB.

Alapvetően három típust használunk az eléréshez: a Connection, Command és a DataReader osztály adatbázis-specifikus változatát. Tehát ha például egy SQL adatbázishoz szeretnénk csatlakozni, akkor az SqlConnection, SqlCommand, SqlDataReader osztály példányait kell igénybe vennünk. A továbbiakban ezen típusokra a szolgáltatótól függetlenül fogok hivatkozni, de a konkrét megoldásban értelemszerűen az adott változatot kell használni. Az 3.1. táblázat [\[1\]](#) megmutatja, hogy ezek a típusok mely interfészeket valósítják meg, és hogyan is működnek.

Objektum	Absztrakt osztály	Megvalósított interfészek	Működés
Connection	DbConnection	IDbConnection	Lehetőséget nyújt az adatbázishoz való csatlakozáshoz és a kapcsolat megszakításához
Command	DbCommand	IDbCommand	Egy SQL lekérdezést vagy egy tárolt eljárást reprezentál. A DataReader számára hozzáférést biztosít az adatokhoz
DataReader	DbDataReader	IDataReader, IDataRecord	Egy szerver-oldali kurzort használva csak olvasható és csak előre haladó hozzáférést biztosít az adatokhoz
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	A DataSet számára biztosít hozzáférést az adatokhoz. A Kapcsolat-nélküli modellhez tartozik, részletes működését ott fogom kifejtetni
Parameter	DbParameter	IDataParameter, IDbDataParameter	Egy paraméterezett lekérdezésben levő névvel ellátott paraméter
Transaction	DbTransaction	IDbTransaction	Egy adatbázis-tranzakciót foglal magába

3.1 táblázat Adatszolgáltatók elérésére használható osztályok

Ezen interfészeket implementálva akár saját adatbázis-elérő osztályokat is készíthetünk az egyéni igényeinknek megfelelően. Az adatbázishoz történő csatlakozáskor a Connection osztály számára aConnectionString-et érdemes a webszerveren levő web.config fájlban elhelyezni, mivel így csak egy helyen kell módosítani, hogyha megváltozna az elérés. A továbbiakban a kapcsolat-alapú modell adatok kinyerésére szolgáló osztályát fogom megvizsgálni: a DataReader osztályt.

3.1.1. A DataReader tulajdonságai

A kapcsolatteremtés lényegében három lépésből áll. Először a Connection osztállyal megnyitjuk az összeköttetést az adatbázissal, ezt követően a Command-dal létrehozuk a kívánt SQL parancsot, amelyet végre szeretnénk hajtani az adatokon. Amennyiben kiolvasni akarjuk az információkat a Command objektum ExecuteReader() metódusával létrehozunk egy DataReader példányt, amelynek Read() utasításával egyesével végiglépkedhetünk a rekordokon amíg a végére nem érünk. Ez a funkció csak előre haladva, olvasható módon teszi elérhetővé az adatokat, minden parancs azonnal továbbítódik az adatbázisszerverhez. Az egyes attribútumok értékét a típusuknak megfelelő metódussal olvashatjuk ki, melyben paraméterként meg kell adnunk az oszlop sorszámát. A műveletek végeztével mindig le kell zárni a kapcsolatot. Az alábbi programkód szemlélteti a használatát kiolvasva az adatbázisból a játékosok nevét.

```

SqlConnection conn = new SqlConnection(System.Web.Configuration.WebConfigurationManager.
    ConnectionStrings["NikfoglaloDB"].ConnectionString);
SqlCommand command = new SqlCommand("SELECT * FROM Players", conn);
conn.Open();
SqlDataReader data = command.ExecuteReader();
while (data.Read())
{
    Console.WriteLine("Név: " + data.GetString(data.GetOrdinal("PlayerName")));
}
data.Dispose();
conn.Close();

```

3.2 ábra DataReader használata

3.1.2. Mikor használjuk?

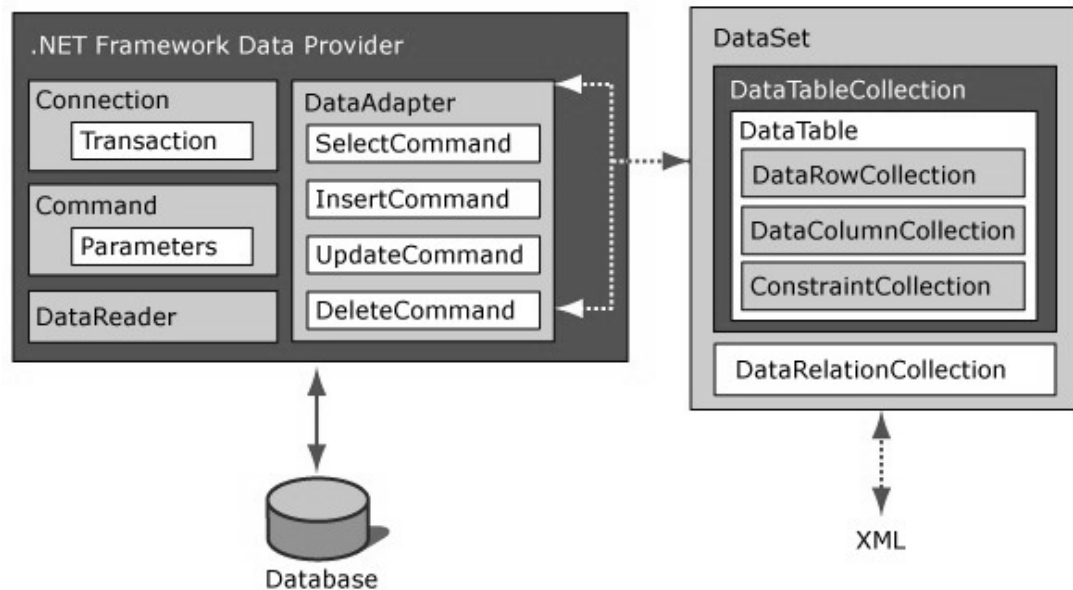
A DataReader egyszerűsége miatt a leggyorsabb hozzáférést biztosítja az adatokhoz a .NET-es technológiák közül, viszont állandó adatbázis-kapcsolatra van szüksége a műveletek végrehajtása alatt. Olyan alkalmazásoknál érdemes használni, amelyek előrehaladó és csak olvasható hozzáférést kívánnak. Minél gyorsabban ki tudjuk olvasni az adatokat a DataReader-ből és lezárjuk a kapcsolatot, annál jobb lesz a szoftverünk teljesítménye. A másik eset, amikor érdemes használni, hogyha az információnak mindig a legfrissebbnek kell lennie vagy kis mennyiségű adatot olvasunk be sokszor. Hátránya, hogy nehézkes a fejlesztés vele, SQL utasításokat kell készíteni, így romlik a program átláthatósága, nehezebb a továbbfejlesztése, mert ha valami változás történik az adatbázisban, akkor az összes parancsot végig kell nézni, és át kell írni.

3.2. Kapcsolat-nélküli modell

Az ADO.NET ezen megoldása az adatbázis elérésre lehetővé teszi, hogy ne kelljen folyamatos kapcsolatot fenntartani a kiszolgálóval, hanem a lokális memóriában az adatok másolatán végezhetünk műveleteket, ezzel mégis azt az érzést keltve, mintha állandó lenne az összeköttetés. Ennek a módszernek a működési elve, hogy először betöltjük a szükséges adatokat a memóriába, itt az adatok között létrehozhatunk kapcsolatokat, megszorításokat, nézeteket, kulcsokat. Ezután kapcsolat nélkül módosíthatjuk az adatokat, amikor pedig véglegesíteni kívánunk, újra csatlakozik, és elküldi a változásokat az adatbázisnak. Erre kínál megoldást a DataSet használata.

3.2.1. A DataSet tulajdonságai

A DataSet-ek mellett, hogy nem igényelnek állandó kapcsolatot, kiválóan alkalmasak XML-ek kezelésére, mellyel a hálózaton gyorsan továbbítható a tartalmuk. Itt már lehet előre és hátra is haladni a kiolvasással, valamint módosítani az adatokat. A DataSet nem függ egyetlen adatszolgáltatótól sem, a specifikus DataAdapter feltölti a háttérben az adott típusú DataReaderrel. A DataSet működése a 3.3. ábrán látható.



3.3. ábra Kapcsolat-nélküli modell

A kapcsolat-alapú modellel ellentétben a Connection objektumot nem kell csatlakozásnál megnyitnunk és bezárni, valamint a Command objektumokat sem kell mindig megírni, hanem rendelkezésünkre áll a DataAdapter osztály, amely ezeket automatikusan kezeli, a lehető legrövidebb ideig tartja nyitva a kapcsolatot, ezzel növelve a rendszer skálázhatóságát. Az SqlCommandBuilderrel pedig a DataAdapter Command-jait automatikusan elkészíttethetjük, így nem kell bajlódni az SQL parancsok megírásával.

A DataSet-tel érhetőek el a kinyert adatok. Ez több táblát is tartalmazhat, melyek sorokból és oszlopokból állnak, ez már hasonló a relációs adatbázisokban megszokott felépítéshez, emiatt könnyebben kezelhető. Relációkat is létrehozhatunk az egyes táblák között, ezzel tovább könnyítve a használatot. A memóriában lévő adatokon végezhetünk beszúrás, módosítás, törlés, listázás műveleteket, majd a véglegesítéskor a DataAdapter elküldi a változtatásokat az adatbázisnak, és betölti az utolsó frissítés óta módosult adatokat a memóriába. Ezáltal a felhasználó úgy érezheti, mintha folyamatos kapcsolatban lenne.

A DataSet valójában egy osztály, amely három féle gyűjteményt kezel. A DataTableCollection amely a táblákat tartalmazza, a DataRelationCollection foglalja magába a DataRelation objektumokat, melyek definiálják az eredeti kapcsolatokat, összeköttetéseket például külső kulcs megszorításokat, így tudunk lépkedni az összekapcsolt táblák között. A PropertyCollection-nel pedig létrehozhatunk kulcs-érték párokat, melyekkel eltárolhatunk egyedi információkat az adatbázissal kapcsolatban. A DataSet számos funkciót tesz elérhetővé számunkra. Másolhatjuk csak az adatok sémáját a Clone() metódussal, valamint az adatokat is a Copy()-val.

A GetChildRelations() és a GetParentRelations()-zel navigálhatunk a kapcsolatban álló táblák között. Az AcceptChanges()-zel elküldhetjük a legutóbbi elfogadás óta történt változásokat az adatbázisnak, a RejectChanges()-zel pedig visszavonhatjuk. A WriteXml(), WriteXmlSchema(), ReadXml(), ReadXmlSchema() metódusok lehetővé teszik a DataSet sémájának, vagy adatainak és struktúrájának elküldését a hálózaton XML formátumban, ezzel gyors és rugalmas adatközlést lehetővé téve.

3.2.2. A DataSet adatszerkezete

Táblákból áll, melyek oszlopokat és sorokat tartalmaznak a relációs adatbázisok felépítéséhez hasonlóan. Az oszlopok tulajdonságai magukban foglalják a bennük szereplő adatok típusát, felülírhatóságát, egyediségét. A sorok egy rekordot reprezentálnak. Módosíthatók, törölhetők, újabbakat adhatunk a táblához. A RowState tulajdonságuk megmutatja, hogy az adott sor milyen állapotban van, hogy most adták hozzá(Added), módosították(Modified), törölték(Deleted), létrehozták, de még nem adták hozzá a többihez(Detached), vagy még nem történt változás az utolsó jóváhagyás óta(Unchanged). Amikor meghívódik az AcceptChanges() metódus ezeket az állapotokat megvizsgálja, majd ezek alapján végrehatja a módosításokat az adatbázisban, majd mindegyiket Unchanged-re állítja. A DataSet-et a legtöbb adatköthető Control-hoz kapcsolhatjuk, így a felhasználó számára megtekinthetővé, és szerkeszthetővé tehetjük az adatbázist.

3.2.3. Erősen típusos DataSet

A Visual Studio lehetőséget nyújt számunkra, hogy automatikusan generáljunk DataSet-eket. Ha a projektünkhöz hozzáadjuk a DataSet elemet, akkor a Designer segítségével az adatbázisban található tábláinkat a Server Explorerből ráhúzhatjuk a tervezőfelületre, melynek hatására elkészül a séma fájl, mely tartalmazza a táblák és oszlopok eredeti felépítését és neveit. Ha ezt a DataSet-et példányosítjuk, akkor elérhetjük a táblákat és oszlopaikat a típusaiknak megfelelően, így az adatbevitelkor könnyebben elkerülhető a hibás adatok adatbázisba kerülése.

Az egyes tábláknak megfelelő DataAdapter-ek is létrejönnek, melyekkel az adott struktúrára illeszkedve tölthetjük át a mezőket. Ezzel a módszerrel típusosan tudjuk kezelni az adatbázis adatait, elkerülve az SQL parancsok megírását, mely sok hibalehetőséget rejt és nehézkes a fejlesztése és karbantartása. Az alábbi példa egy DataSet feltöltését mutatja be.

```
using (QuestionDataSet dataSet = new QuestionDataSet())
{
    using (DataAccessLayer.QuestionDataSetTableAdapters.TipQuestionTableTableAdapter adaptTip =
        new DataAccessLayer.QuestionDataSetTableAdapters.TipQuestionTableTableAdapter())
    {
        adaptTip.Fill(dataSet.TipQuestionTable);
    }
}
```

3.4. ábra DataSet feltöltése

3.2.4. Mikor használjuk?

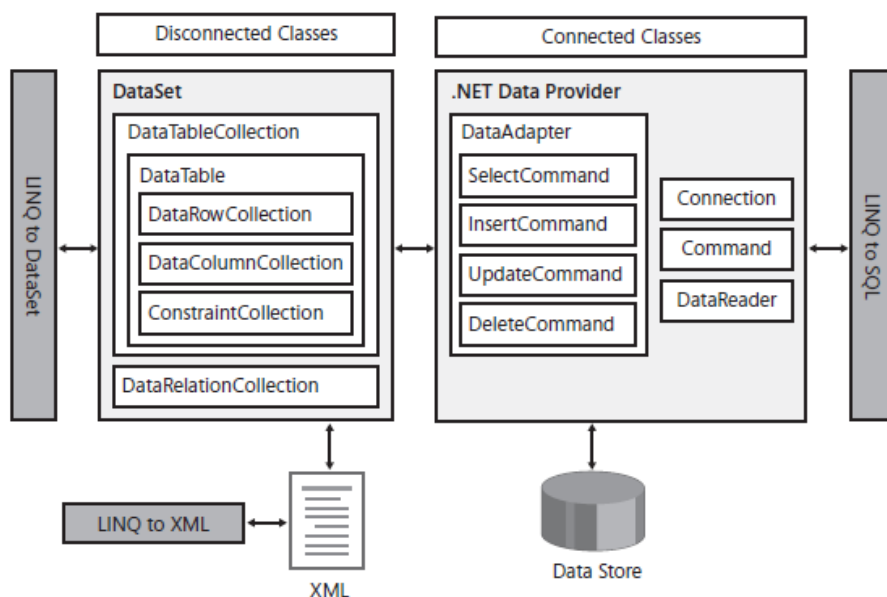
A DataSet-eket érdemes használni, ha az eredményeket nem kell azonnal aktualizálni az adatbázisban, valamint ha széleskörű műveleteket kell végrehajtani az üzleti logikából kifolyólag. Kinyeri az adatokat az adatbázisból, betölti a memóriába, majd megszünteti az adatbázis kapcsolatot, így más folyamatoknak több lehetősége lesz csatlakozni, míg a DataReader fenntartja az összeköttetést, amíg el nem végzi a műveleteket, ami egy nagy adatforgalom-igényű rendszernél hamar a kapcsolódási lehetőségek elfogyásához vezethet.

Jó választás akkor is, ha webszolgáltatásokat készítünk, mivel könnyedén sorosítható, így XML-ekben továbbíthatjuk őket. A nem túl gyakran változó adatbázisoknál optimális megoldás, mivel letárolhatjuk a webes alkalmazás Application memóriájában, vagy a Session adatok között, így nem kell mindig csatlakozni, ezzel növelve a teljesítményt. Ezt érdemes lesz fontolóra venni a webes alkalmazásomban. Jobb választás a DataReadernél, ha többször végig kell lépkednünk az adatokon. Vezérlők feltöltésénél is ugyanahhoz a DataSet-hez többet is hozzá lehet csatolni, nem kell mindegyikhez újra beolvasni.

Hátránya, hogy nagy tábláknál több időbe telik feltölteni vagy frissíteni, mintha csak egyszerű SQL utasításokat használnánk, valamint a memóriefoglalása is nagy. A gyorsan változó adatbázisoknál nem tudja naprakészen tartani az adatokat.

3.3. A LINQ jelentősége

A LINQ (Language Integrated Query) egy eszköz arra, hogy egységes módon kinyerjük és módosítsuk az adatokat a legkülönbözőbb forrásokból, mint például gyűjteményekből, XML-ekből, DataSet-ekből, objektumokból melyek megvalósítják az `IEnumerable<T>` interfészt, valamint relációs adatbázisokból. Az ehhez használt kifejezések formájukban hasonlítanak az SQL lekérdezésekhez, viszont a LINQ erősen típusos, így a fordító tud figyelmeztetni minket a hibákra. A 3.5. ábrán [3] jól láthatóak a LINQ felhasználási lehetőségei az eddig bemutatott modellben.



3.5. ábra A LINQ szerepe

A LINQ lekérdezésekben a `from`, `where`, `select` kulcsszavakat használjuk az adatok kiválogatására, valamint az SQL kifejezésekhez hasonlóan rendezhetünk az `orderby` kulcsszóval. Egy példa látható a 3.6. ábrán egy `int` tömbből a páros számok kiválogatására és rendezésére:

```
int[] Numbers = { 5, 10, 15, 4, 2, 20, 12 };  
IEnumerable<int> EvenNumber = from sz in Numbers where sz % 2 == 0 orderby sz select sz;  
//Az eredmény: 2,4,10,12,20
```

3.6. ábra LINQ használata

Lehetőségünk van a kapott eredményt a 3.7. ábrán látható módon tetszőleges gyűjtemény típusra konvertálni, mint például tömbre (`ToArray()`), listára (`ToList()`), dictionaryre (`ToDictionary(a=>a.id)`).

```
List<int> EvenNumberList = EvenNumber.ToList<int>();  
int[] EvenNumberArray = EvenNumber.ToArray<int>();
```

3.7. ábra Konvertálás LINQ-val

Mivel a LINQ késői kiértékelést használ, így a lekérdezéseket összekapcsolhatjuk, fokozatosan építhetjük fel, így sokkal átláthatóbb és könnyebb a fejlesztés vele. Az egymást követő szűrésekkel addig szűkíthetjük az eredményt, mígnem megkapjuk a kívánt értékeket.

A .NET 3.5-ben bevezetésre került a Lambda ($\Rightarrow$) operátor használata, amely jelentősen meggyorsítja a kód írását, és áttekinthetőbbé teszi azt. Az IEnumerable interfészt megvalósító gyűjtemények számára elérhetőek az Enumerable osztály nyújtotta statikus metódusok melyek többek között az alábbiak.

A szűrést segítő operátorok a Where() amely a megadott feltétel alapján csökkenti a tartományt, a Take() mely csak az első kívánt számú adatot tartja meg, a Skip() pedig átugorja a megadott mennyiségű elemet, és az azt követőeket adja vissza. A SkipWhile() és a TakeWhile() pedig a feltételt teljesítő adatokat hagyja ki illetve gyűjti össze. A Distinct() biztosítja, hogy az azonos adatok csak egyszer szerepeljenek. A kiindulási tartományt csoportokba gyűjthetjük a group kulcsszó után megadott mező szerint, és a by után szereplő feltétel alapján. A Count() az elemek számát adja vissza. A Sum(), Min(), Max(), Average() elemek összegét, legkisebb, legnagyobb elemet, valamint átlagot ad vissza. A Select($p \Rightarrow p.name$) metódussal az elem adott tulajdonságainak értékét kérhetjük le, nem kell az összeset feleslegesen kigyűjteni. Az ElementAt() metódussal a sorozat megadott helyen található tagját kapjuk meg.

Ezek közül több metódusban is használhatjuk a Lambda kifejezéseket, melyekkel csak meg kell adni, hogy az elem mely tulajdonságát vizsgáljuk milyen feltétel alapján, és nem kell leírni a lekérdezést a szokott SQL-ből ismert formában. A könnyebb érthetőség kedvéért az alábbi példa bemutatja a fenti lekérdezést Lambda operátorokkal.

```
IEnumerable<int> EvenNumbers = Numbers.Where(sz => (sz % 2 == 0)).OrderBy(sz => sz);
```

3.8. ábra Lambda operátorok használata

Jól látható, hogy ebben a formában sokkal olvashatóbb a lekérdezés, és könnyen hozzáfűzhetünk újabb metódusokat tovább szűkítve az eredményhalmazt.

3.3.1. LINQ to SQL

Az eddigiekben bemutatam a LINQ nyelvi elemet, mellyel számos adatforráshoz lehet csatlakozni. Ezek közül most az SQL adatbázisokhoz történő kapcsolódást fogom részletesebben megvizsgálni. A LINQ to SQL (DLINQ) egy olyan API, amely egy köztes réteget képez az adatbázis és a programozói felület közé elfedve a Connection, Command, DataAdapter osztályokat azáltal, hogy egyed osztályokat készít, amelyek a relációs adatbázis modelljét tükrözik, viszont szokványos .NET osztályok formájában. Létrehozhatjuk programkódból is ezeket, de gyorsabb és egyszerűbb, ha hozzáadjuk a LINQ to SQL elemet a projektünkhöz, majd a Server Explorerből a kiválasztott adatbázis tábláit ráhúzzuk a tervező nézetre. Ezzel létrejönnek a táblákat reprezentáló osztályok típusosan tárolva az adatbázis szerkezetét és az összeköttetéseket a DataContext osztályban, melyet példányosítva elérhetővé válnak gyűjtemények formájában. Az elkészült osztályok mező típusai az oszlopoknak felelnek meg. A külső kulcs megszorításokat úgy képezi le, hogy az osztály tartalmaz egy mezőt a kapcsolódó táblából készült osztály típusával. Ha példányosítunk, akkor egy objektumot kapunk, amely az adatbázis egy rekordját reprezentálja. Így SQL parancsok helyett típusos objektumokon keresztül végezhetünk LINQ lekérdezéseket az adatbázisból nyert adatokon.

Új elemeket úgy hozhatunk létre, ha a kívánt táblának megfelelő osztályból létrehozunk egy példányt, és feltöltjük a megfelelő mezőit, majd az InsertOnSubmit() metódussal hozzáadjuk. Az ismert LINQ lekérdezésekkel kigyűjthetjük és módosíthatjuk a keresett rekordokat, a DeleteOnSubmit()-tal pedig törölhetjük. Ezek a módosítások állandó kapcsolat nélkül elvégezhetők, de csak akkor kerülnek be az új adatok, hogyha meghívjuk a SubmitChanges() metódusát a DataContext() példánynak, ezzel automatikusan legenerálódnak a kívánt SQL parancsok, megnyitja a kapcsolatot, és végrehajtja őket az adatbázison. Az alábbi kódrészlet egy rekord kikeresését és egyik mezőjének értékmódosítását mutatja be LINQ to SQL-lel.

```
DataClassesDataContext context = new DataClassesDataContext();  
var john = (from p in context.PlayerTables  
            where p.PlayerName == "john"  
            select p).First();  
john.Points += 100;  
context.SubmitChanges();
```

3.9. ábra Lambda operátorok használata

3.3.2. Mikor használjuk?

A LINQ to SQL az erősen típusos objektum modelljének köszönhetően gyors, könnyű és formailag biztonságosabb adatkezelést tesz lehetővé. A fejlesztést rugalmassá és egyszerűvé teszi, mivel úgy kezelhetjük a táblákat, oszlopokat és rekordokat, mintha hagyományos objektumok lennének. A rajtuk végrehajtott LINQ lekérdezésekkel könnyen megtalálhatjuk a számunkra szükséges adatokat, mivel nem kell SQL parancsokat írunk, mint a DataReader esetében, valamint nem kell indexeléssel végiglépkednünk a sorokon, mint a DataSet-eknél. Egyik hátrányos tulajdonsága éppen az előnyéből ered, mivel az objektumok és a köztes réteg generálása erőforrás-igényes, ezért a gyorsasága messze elmarad a DataReader mögött, valamint a DataSet-től is gyengébb.

Összességében olyan alkalmazásoknál érdemes használni, ahol fontos szempont hogy az adatok típusait könnyen lehessen kezelni, a rendszer továbbfejleszthetőségét tekintve átlátható és rugalmasan átalakítható legyen. Az adatok kis részén történő manipuláció esetén előnyös, mivel nem tölti be az összes adatot a memóriába, mint a DataSet, hanem csak a lekérdezésben szereplő adatkört nyeri ki az adatbázisból. Ennek köszönhetően alkalmas adatköthető vezérlők feltöltésére, a lapozás is könnyen megoldható vele, mivel egyszerre csak annyi adatot mozgat meg, amennyi a vezérlő egy oldalának felel meg.

Nem ajánlott a használata olyan nagy teljesítményigényű rendszereknél, ahol gyors, nagy mennyiségű adatelérésre van szükség, ilyenkor ezek közül a leggyorsabb módszer az egyszerű DataReader-ek használata.

3.4. Entity Framework

Az Entity Framework (EF) egy viszonylag új tagja az ADO.NET családnak, 2008-ban adták ki az első verzióját a .NET 3.5 első szervizcsomagjával, de a 4.0-ás verziónál lett kiforrottabb a technológia. Az ehhez kapcsolódó információkat részben a [\[4\]](#)-es forrásból merítettem.

Az Entity Framework létrehozásának az volt a célja, hogy a relációs adatbázis adatait egy köztes objektum modellen keresztül leképezze az üzleti folyamatokban használt objektumokká. Így az oszlopokon és sorokon történő műveletvégzés helyett úgynevezett entitásokat használhatunk, melyek erősen típusos objektumok. Ezeken szintén végezhetünk LINQ lekérdezéseket a szokott formában.

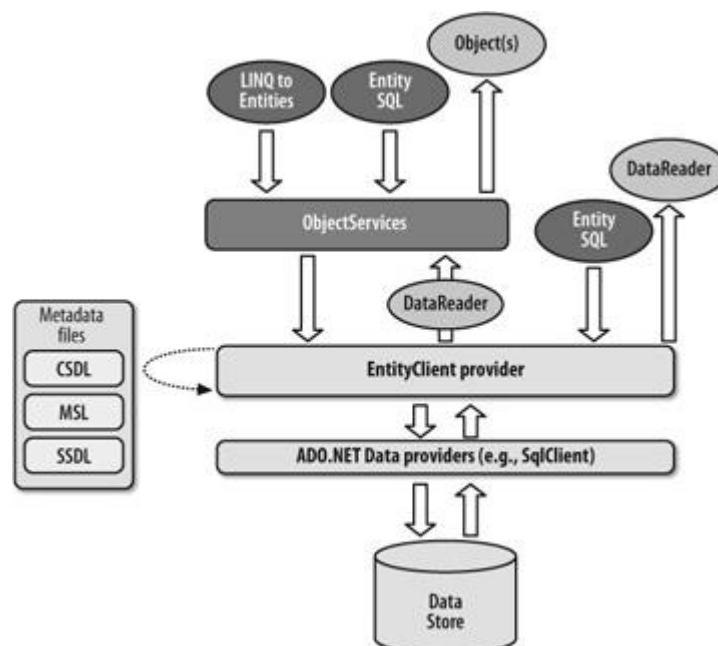
3.4.1. Felépítése

A kapcsolat alapú és a kapcsolat nélküli modellben mindig az adatbázis fizikai struktúráján hajtottuk végre a műveleteket. Ismernünk kellett a pontos elnevezéseket és az adatok felépítését ahhoz, hogy végre tudjuk rajtuk hajtani a szokásos kiválasztás, módosítás, törlés, beszúrás parancsokat. Amennyiben megváltozik az adatszolgáltató, vagy az adatbázis elrendezése, akkor át kell írunk a kódunkat. Az Entity Framework

lehetővé teszi, hogy SQL kódok írása nélkül, csupán objektumok állapotának elmentésével valósítsuk meg a feladatunkat. A leképezést a fizikai adatbázis modellre az EF runtime végzi, amely lehetővé teszi, hogy ne az adatbázisban használt elnevezéseket használjuk, hanem saját igényeinknek megfelelően nevezhessük el az entity osztályainkat, melyeket futásidőben hozzákapcsol a megfelelő struktúrához. Ezeket összefoglalóan Entity Data Modell-nek nevezzük.

Az Entity Framework működése két jelentős egységen alapul. Az egyik az object services, amely kezeli az egyedek közti kapcsolatokat, követi a rajtuk végrehajtott változtatásokat, valamint közvetíti ezeket az adatbázis felé. A másik az entity client, amely csatlakozik az adatbázishoz, elvégzi az átalakításokat a fizikai modell és az egyedek között, valamint elkészíti a LINQ lekérdezésekből az SQL utasításokat, melyeket már el lehet végezni a valós adatokon. Itt találhatók azok az osztályok, melyek a közvetlen elérést biztosítják, mint például a DataReader vagy a Connection. Itt képződnek le az entity-ken bekövetkezett változások a háttérben működő ADO.NET osztályokra.

Az adatmodell alapú adatbázis elérések három alapvető egységből állnak. A fogalmi modell meghatározza az entitásokat, és ezek kapcsolatait, melyet a logikai modell összeköt a fizikai modellben tárolt elnevezésekkel. A fizikai modell pedig létrehozza a táblák közti viszonyokat, struktúrákat, az indexelést, a kapcsolatot az adatszolgáltatóval, a valódi tábla és oszlopneveket. Ezen modellezési szintek összeköttetései az Entity Framework esetében egy XML alapú fájlban tárolódnak, melyek kiterjesztése edmx, ez biztosítja majd, hogy a designerrel tudjuk szerkeszteni a kapcsolatokat. Fordításkor mindegyik réteg számára egy-egy külön XML fájl jön létre(cSDL(fogalmi), msl(logikai), ssdl(fizikai)), melyek a hozzájuk tartozó beállításokat tartalmazzák. A rendszer összefoglaló nézete a 3.10. ábrán [5] látható.



3.10. ábra Entity Framework felépítése

A programozó számára ezt a komplex adatelérési rendszert azObjectContext osztály teszi elérhetővé. Az edmx-fájlban jön létre az entity osztályokkal együtt, és a példányával vehetjük igénybe az object service és az entity client nyújtotta lehetőségeket. Törölhetünk az adatbázisunkból objektumokat (DeleteObject()), tárolt eljárásokat hívhatunk meg (ExecuteFunction<T>()), megváltoztathatjuk a ConnectionStringet, valamint elküldhetjük a változásokat, hogy végrehajtsódjanak a valódi adatokon (SaveChanges()). Az ObjectContext tartalmazza az entitások gyűjteményeit ObjectSet<T> osztályok formájában, melyek generikus típusa az adatbázis tábláinak feleltethető meg. Ezzel a típussal hozhatunk létre új elemeket (CreateObject<T>()), az AddObject()-tel pedig a gyűjteményhez adhatjuk.

3.4.2. Használata

Az Entity Framework adatbázis elérési módszert legkönnyebben úgy integrálhatjuk a szoftverünkbe, ha a Visual Studio 2010-ben a projektünkhöz adunk egy ADO.NET Entity Data Modell elemet. A megjelenő varázslóval beállíthatjuk, hogy melyik ConnectionString-gel csatlakozzunk a kívánt adatbázishoz, majd kiválaszthatjuk a táblákat, melyeket kezelni szeretnénk. Ez után az edmx fájl elemeit az Entity designer segítségével beállíthatjuk, hogy milyen néven használjuk majd az adatbázis rekordjait. Az első fordítást követően létrejönnek a fogalmi, logikai, fizikai modelleket tartalmazó XML fájlok, melyek tartalmazzák a leképezéseket az entitások és az adatbázis között. A generált egyed osztályokat kiegészíthetjük saját kódunkkal partial osztályokon keresztül. A könnyebb kezelés érdekében LINQ lekérdezéseket is készíthetünk a DataContext osztály gyűjteményein. Erre példát a 3.11. ábrán láthatunk.

```
using (kszucs_dbEntities context = new kszucs_dbEntities())
{
    var player = (from p in context.PlayerTable
                  where p.UserId == 3000 select p).FirstOrDefault<PlayerTable>();
    player.Points+=200;
}
```

3.11. ábra Entity Framework elérése LINQ-val

Lehetőségünk van alacsony szintű lekérdezésre az Entity SQL módszer segítségével. Ilyenkor a kinyert adatokhoz még azelőtt hozzáférhetünk, hogy az entity objektumokhoz jutnának, így a leképezések létrehozása nem rontja tovább a teljesítményt. Az EntityClient használatával az EntityConnection, EntityCommand, EntityDataReader objektumokkal végezhetünk műveleteket a kapcsolat alapú modellnél megszokott módon.

3.4.3. Mikor használjuk?

Az Entity Framework előnyös megoldás, ha nagy rendszereket tervezünk, melyeknél fontos a rugalmas továbbfejleszthetőség, valamint az adatbázis struktúrája gyakran változik, viszont az üzleti folyamatokat nem akarjuk ennek megfelelően folyamatosan átvizsgálni. A többbéteges objektumleképezés viszont igen erőforrás igényes folyamat, ezért teljesítmény-kritikus alkalmazásoknál a problémás részeket tárolt eljárásokkal érdemes megoldani, vagy más, egyszerűbb technológiát választani, például a DataReader-t vagy DataSet-et.

3.5. Tárolt eljárások

A tárolt eljárások olyan műveletek, melyeket az adatbázisban helyezünk el. Ha egy feladatot sokszor végre kell hajtani, akkor azzal javíthatunk a rendszer teljesítményén, hogy nem az alkalmazásunkból hajtjuk végre mindig az SQL lekérdezéseket, hanem készítünk egy eljárást az adatbázison, amely elvégzi a kellő számításokat és már csak az eredményt küldi vissza a hálózaton keresztül a programunknak. Ezzel a módszerrel csökkenthetjük az adatforgalmat és az erőforrásaink terheltségét. A fent említett technikák mind lehetővé teszik tárolt eljárások meghívását, ezért ez hasznos kiegészítő megoldást jelenthet a számítás-igényesebb feladatoknál.

4. ADATBÁZISELÉRÉSEK TESZTELÉSE, KIÉRTÉKELÉS

Az eddigiekben bemutattam a .NET nyújtotta adatbázis elérési módok tulajdonságait, működését, használatát, valamint előnyeit és hátrányait. A döntés meghozatalához, hogy melyik technológiát válasszam a webes kvíz játékomban fejlesztéséhez teszteteket készítettem, melyek tükrözik az alkalmazásomban gyakran előforduló lekérdezéseket. A teszt objektívebb értékeléséhez röviden jellemzem a rendelkezésemre álló hardvereket és szoftvereket.

4.1. A teszt környezet

A tesztekkel az eltelt időket és a memóriahasználatot mértem.

Hardver:

- Notebook: Acer Aspire 5920G
- Processzor: Intel Core 2 Duo T7300 (2.0GHz, 800MHz FSB(Intel GTL+, busz szélesség: 64bit, valódi órajel: 250MHz, sávszélesség: 2000MB/mp), 4MB L2 cache(On-Die, ATC, Full-Speed))
- Memória: 2 GB DDR2, 667MHz
- Virtuális memória: 3,4 GB
- Videokártya: NVIDIA GeForce 8600M GS (256MB beépített memória)
- Merevlemez: Western Digital 320GB, 5400rpm, SATA, 8Mb cache
- Ethernet kártya: Broadcom NetLink Gigabit, melyen keresztül a Márton Áron Szakkollégium által szolgáltatott internetelérést használtam. A teszt idején a letöltési sebesség 4,53MB/s, a feltöltési 2,89MB/s volt.

Szoftver:

- Operációs rendszer: Microsoft Windows 7 Ultimate, 32 bit
- Keretrendszer: Microsoft .NET Framework 4.0
- Fejlesztőkörnyezet: Microsoft Visual Studio 2010
- Programnyelv: C#
- Alkalmazás típusa: Console Application

Adatbázis:

- Szerver: MS SQL 2008 R2 adatbázis szerver (Developer Edition)
- Környezet: Egy virtuális gépen fut, amelynek 1 processzora és 1 GB memóriája van és Windows Server 2008 R2 Enterprise Edition operációs rendszer működteti.
- 1 tábla, 6 oszlop(3 darab varchar(50), 2 int és 1 date típussal)
- 500 majd 20.000 rekord

A tesztekhez konzolos alkalmazásokat hoztam létre, melyeket a fejlesztőkörnyezetből futtattam és a következő adatbázis-elérési módokat valósították meg: DataReader, nem típusos DataSet, erősen típusos DataSet, LINQ to SQL, az Entity Framework-on belül LINQ to Entities, Entity SQL, Entity DataReader, valamint tárolt eljárásokat írtam az MSSQL adatbázisba, melyet DataReaderrel, valamint LINQ to SQL-el értem el.

4.2. A tesztek

Átgondoltam, hogy a játékprogramom során az egyes esetekben milyen adatokra lesz szükségem, melyeket kell majd módosítanom és milyen gyakorisággal fogom végezni a kéréseket. A lekérdezéseket egymás után 100-szor lefuttattam, mértem az eltelt időt, majd ezt 5-ször megismételtem. A futás közben feljegyeztem a memóriahasználat mértékét is.

4.2.1. Első eset

Az első és egyben legfontosabb eset, amelyet vizsgáltam a kérdések feltevése. Minden játék során körülbelül húsz kérdést fognak kapni a játékosok, valamint párhuzamosan sok játék futhat egyszerre, ezért a kérdéskiválasztás a rendszer legkritikusabb pontja. Tehát a tesztet hogy válasszunk ki egy rekordot ID alapján a sok közül - amely az első esetben még csak 500 - a lehető leggyorsabban. Mindegyik módszerrel implementáltam ezt a feladatot a technikák sajátosságait figyelembe véve úgy, hogy mindegyikből a lehető legjobb teljesítményt hozzam ki. A futtatáskor keletkezett eredményeket a 4.1. táblázat mutatja.

Egy rekord lekérdezése 500 elemen, 100-szor (ms)									
	Data Reader	DataSet	Strongly Typed DataSet	LINQ to SQL	LINQ to Entities	Entity SQL	Entity DataReader	Stored Procedure LINQ to SQL-el	Stored Procedure DataReaderrel
1.	652	12	12	990	1716	1225	1361	773	651
2.	559	13	13	833	889	738	723	720	565
3.	534	12	11	799	1045	731	658	709	580
4.	551	12	10	828	883	816	649	722	568
5.	690	14	10	813	930	698	655	702	579
Átlag	597,2	12,6	11,2	852,6	1092,6	841,6	809,2	725,2	588,6
Mem. foglalás (Kbyte)	6700	7440	7388	8492	16656	13812	13732	7664	5820

4.1. táblázat Első teszt

Az eltelt időket nézve a legjobb eredményeket a DataSet érte el, viszont ehhez hozzá kell tennem azt, hogy a DataSet feltöltését még a többszörös lekérdezések elé helyeztem, mivel egy ASP-s alkalmazáserver az indulását követően feltöltheti a DataSet-et, majd ezt eltárolhatja az Application memóriában. Ennek köszönhetően az egyes lekérdezésekhez már nem kell újra betölteni az adatokat, hanem az operatív tárból közvetlenül el tudja érni nagyon rövid idő alatt, ellentétben a többi technológiával, amelyeknek minden lekérdezéshez ki kell építeniük a kapcsolatot az adatbázissal, ami lassítja a rendszert. Mivel a kérdések az alkalmazásban ritkán frissülnek - csak

akkor, ha valaki újat adott hozzá, vagy az adminisztrátor törölt illetve módosított -, ezért megengedhető, hogy az adatbázissal csak a változások esetén szinkronizálódjon.

A DataReader az elvárásoknak megfelelően jól teljesített, a LINQ to SQL kicsit lemaradt tőle, viszont a LINQ to Entities rosszul teljesített, a lekérések és az egyedek generálása túl sok erőforrást igényel, ezért romlott az eredménye, valamint nagyon sok memóriát is elfoglal. Az Entity SQL és az Entity DataReader közel azonos értékeket mutatnak, bár a LINQ-s lekéréseknél gyorsabbak, de a többi technológiához képest rosszul szerepeltek, megvalósításuk nehézkes és még a memóriaigényük is nagy. A tárolt eljárásoktól nagyobb sebességet vártam, viszont mivel csak egy rekord lekérése volt a feladat, ezért a kapcsolat kiépítésével elvesztett idő miatt a DataSet ebben a használati esetben jobb eredményeket mutat a többi technológiánál, ezért alkalmas lehet a feladatra, viszont a döntéshez még további vizsgálatokat végeztem.

4.2.2. Második eset

A DataSet jó eredményeit látva kíváncsi voltam, hogy hogyan viselkedik, hogyha a gyengeségét vizsgálom meg, vagyis hogy mennyi memóriát foglal sok adat esetén. A webszervernél fontos tényező, hogy a memória elegendő legyen még nagyszámú felhasználó mellett is. Ezt a tesztet 20.000 rekordon végeztem el, amely úgy gondolom elég nagy szám, mivel szerintem büszke leszek, ha már ennyi kérdés lesz a rendszeremben. A végeredmény a 4.2. táblázatban látható.

Egy rekord lekérése 20.000 elemen, 100-szor (ms)									
	Data Reader	DataSet	Strongly Typed DataSet	LINQ to SQL	LINQ to Entities	Entity SQL	Entity DataReader	Stored Procedure LINQ to SQL-el	Stored Procedure DataReaderrel
1.	911	586	534	1161	2102	1632	1744	1058	933
2.	874	556	506	1046	1357	956	925	941	790
3.	929	568	503	1039	1211	982	927	916	801
4.	854	557	516	1051	1324	947	939	913	799
5.	800	554	540	1076	1173	996	965	956	812
Átlag	873,6	564,2	519,8	1074	1433,4	1102,6	1100	956,8	827
Mem. foglalás (Kbyte)	6868	22452	19292	8236	16672	13844	13800	7708	6216

4.2. táblázat Második teszt

Ahogy várni lehetett a DataSet memóriahasználata a legmagasabb, 20 megabájt körüli. Viszont ha jobban megnézzük ezt az értéket, napjaink számítógépei számára ez nem annyira megterhelő, mivel még egy személyi számítógép 2-4 GB-os memóriája is könnyen megbirkózik vele. Másrészt az Entity Framework implementációi is igencsak nagy területet használnak fel az operatív tárból, viszont a teljesítményük fele sincs a

DataSet-ének. A DataSet most is a legjobb eredményt érte el, szinte kétszer gyorsabb, mint a többi technológia. Figyelembe véve, hogy ez a 20.000 rekord a rendszer terhelésének viszonylag szélső értékét képezi, valamint ha már a játék túllépi ezt a számot a népszerűségének köszönhetően, akkor valószínűleg lesz elegendő anyagi háttér a memória bővítéséhez is. Ezeket összevetve továbbra is jó megoldásnak tűnik a kérdések betöltésére.

4.2.3. Harmadik eset

A továbbiakban megvizsgálom, hogy a játék egyéb területein milyen jellegű lekérdezésekre lehet szükség. A játékosok adatainak kigyűjtéséhez, elemzéséhez, például, hogy mennyi az átlagpontja az egyes korosztályoknak (évfolyamoknak) az adatbázis elemeinek nagy részén végig kell menni, ami nagy számítási és adatforgalom igényű feladat, viszont viszonylag ritkán van rá szükség. A következő tesztetben ismét 500 elemű táblát fogok vizsgálni, melyből minden második játékos pontjainak értékéből átlagot számítok. A kapott értékek a 4.3. táblázatban megtekinthetők.

Minden második rekord lekérdezése 500 elemen, 100-szor (ms)									
	Data Reader	DataSet	Strongly Typed DataSet	LINQ to SQL	LINQ to Entities	Entity SQL	Entity DataReader	Stored Procedure LINQ to SQL-el	Stored Procedure DataReaderrel
1.	1203	18	14	1503	2207	2182	2228	779	712
2.	1105	18	13	1402	2157	2174	2178	652	575
3.	1052	16	15	1361	2151	2160	2146	640	595
4.	1061	18	13	1462	2166	2160	2143	654	602
5.	1180	15	13	1374	2166	2159	2154	655	559
Átlag	1120,2	17	13,6	1420	2169	2167	2170,4	676	608,6
Mem. foglалás (Kbyte)	7316	7396	7360	8732	13496	11876	11748	7568	6740

4.3. táblázat Harmadik teszt

Jól látható, hogy ez a feladat még kis elemszámnál is sok időt igényel. A DataSet itt is a leggyorsabb, viszont számításba kell vennünk azt is, hogy ha minden táblát a memóriában tárolnánk, akkor a szerver a számos egyéb feladattal együtt túlságosan leterhelt lenne, ezért érdemes másik megoldást keresni. Olyan technikára lenne szükség, amely nem igényel folyamatos, nagymértékű memóriefoglalást, valamint tudja kezelni az adatok gyors változását. Erre válasz lehetne a DataReader, viszont a felhasználók adatainak kezeléséhez típusos adatkezelés megfelelőbb lenne, valamint ezekre a funkciókra sok helyen szükség lehet, ezért körülményes lenne mindezeket SQL parancsok megírásával megoldani. Itt felmerül az Entity Framework és a LINQ to SQL használata.

Az Entity Framework láthatóan nagyon lassan dolgozik ebben az esetben is, mindamelllett a memóriahasználata is túl nagy, ami szintén terhelne a szerveret. Ezeket összevetve a LINQ to SQL tűnik megfelelőnek, mivel nem sokkal marad el a teljesítménye és memóriahasználata a DataReader-től, valamint csak akkor olvassa be az adatokat, amikor szükség van rájuk. Ebben az esetben már a tárolt eljárások is jól teljesítenek, mivel nem kell az adatbázisról áttölteni az adatokat, hanem helyben elvégezhetők a számítások, és a hálózaton már csak a végeredménynt kell átküldeni.

4.2.4. Negyedik eset

Végül az adatok módosítását vettem figyelembe, amely a játékosok pontszámának frissítésekor, valamint a chatbeszélgetések logolásakor is előfordul. Ebben az esetben lekérdeztem egy adott játékost, majd pedig a módosított pontszámát elmentettem az adatbázisba. Az eredmények a 4.4. táblázatban láthatók.

Egy rekord lekérdezése, majd módosítása 500 elemen, 100szor (ms)									
	Data Reader	DataSet	Strongly Typed DataSet	LINQ to SQL	LINQ to Entities	Entity SQL	Entity DataReader	Stored Procedure LINQ to SQL-el	Stored Procedure DataReaderrel
1.	1319	726	866	2293	3028	2644	2504	800	758
2.	1172	723	860	2183	2305	1954	2013	713	633
3.	1158	755	847	2282	2155	2026	2033	728	629
4.	1239	717	831	2228	2142	1975	2025	722	654
5.	1432	752	865	2224	2128	1999	1943	749	637
Átlag	1264	734,6	853,8	2242	2351,6	2119,6	2103,6	742,4	662,2
Mem. foglalás (Kbyte)	6892	7452	7924	8200	13632	13872	14024	7224	6140

4.4. táblázat Negyedik teszt

Ennél a feladatnál a DataSet is gyengébb teljesítményt mutat, mivel a gyakori adatfrissítések miatt el kell mentenie a változásokat az adatbázisba. Itt a tárolt eljárások szerepeltek a legjobban, mivel az adatbázison belül gyorsan tudnak módosítani. Ezek gyakran használt funkciók, ezért érdemes lehet tárolt eljárást írni hozzájuk, melyet például a LINQ to SQL-lel meg lehet hívni.

4.3. Összegzés, technológia kiválasztása

Az irodalmi háttér tanulmányozását követően, majd az egyes módszerek implementációit tesztelve a következő eredményre jutottam. A webes alkalmazásomban a kérdések kiválasztása a leggyakoribb, ezáltal a legkritikusabb pont az adatbázis elérést tekintve. Erre a feladatra az erősen típusos DataSet-et választottam, mivel ez a technológia betölti a szerver memóriájába azt a táblát, amelyre szükségem van, így nem

kell folyamatosan kapcsolatokat nyitnom, melyek a távoli adatbázis eléréséhez a hálózati adatforgalom miatt sok időt vesznek el.

A távoli eljárások is jó megoldást nyújtanának, ha az adattárolás lokálisan folyna, így nem merülne fel már egy rekord lekérdezésénél is az előbb említett probléma. A DataSet nagy memória igényét is megvizsgáltam, ami ebben a speciális esetben a rekordok mérete, és a lehetséges kérdések száma miatt nem jelent problémát. A kétféle DataSet megvalósítási módszer közül azért választottam az erősen típusost, mert a tesztekben nem mutatott jelentős különbséget a nem típusossal szemben, viszont a típusossága miatt rugalmasabb és átláthatóbb a fejlesztés vele.

Erre a konkrét problémára jó megoldás, viszont a teljes alkalmazásomban már gondot jelentene az összes tábla memóriában való tárolása, ezért egy hibrid megoldást választottam. A kritikus, gyakran ismétlődő részt erősen típusos DataSet-tel valósítom meg, de a ritkábban előforduló, az adatbázist módosító tevékenységekhez a LINQ to SQL-t fogom használni. Ez a módszer kevesebb memóriát foglal, amikor pedig adatbázis elérésére van szükség, akkor generál egy SQL kódot, amely végrehajtja az ad-hoc műveleteket.

A DataReader-rel szemben - amely egyszerűsége miatt igen gyorsan tud kommunikálni – azért választottam, mert számos esetben szükség van az adatok típusos tárolására, valamint a továbbfejlesztés és a kód átláthatósága szempontjából előnyösebb a DLINQ használata.

A típusos tárolás és az olvashatóság szempontjából az Entity Framework is felmerült megoldásként, viszont nagyfokú rugalmassága ellenére a teszteken igencsak rosszul szerepelt, ezért egy ilyen méretű projekthez elegendően flexibilisnek találtam a fent említett megoldást.

A LINQ to SQL egy rekord kiolvasásánál jól szerepelt, viszont amikor nagyszámú elemen kell végighaladni, vagy adatokat kell módosítani, akkor az adatbázison tárolt eljárások sokkal nagyobb hatékonysággal tudják helyben elvégezni a műveleteket, és csak a végeredményt elküldeni a hálózaton. Ezért a teljesítmény további növelése érdekében a gyakran végrehajtott adatmódosításokkor, vagy elemzésekkor tárolt eljárást fogok készíteni. Ennek eléréséhez két módszert vizsgáltam, melyek között egy apró különbség volt a DataReader javára, viszont figyelembe véve a kód továbbfejleszthetőségét és átláthatóságát, a már egyébként is használt LINQ to SQL-lel fogom betölteni a tárolt eljárások eredményét.

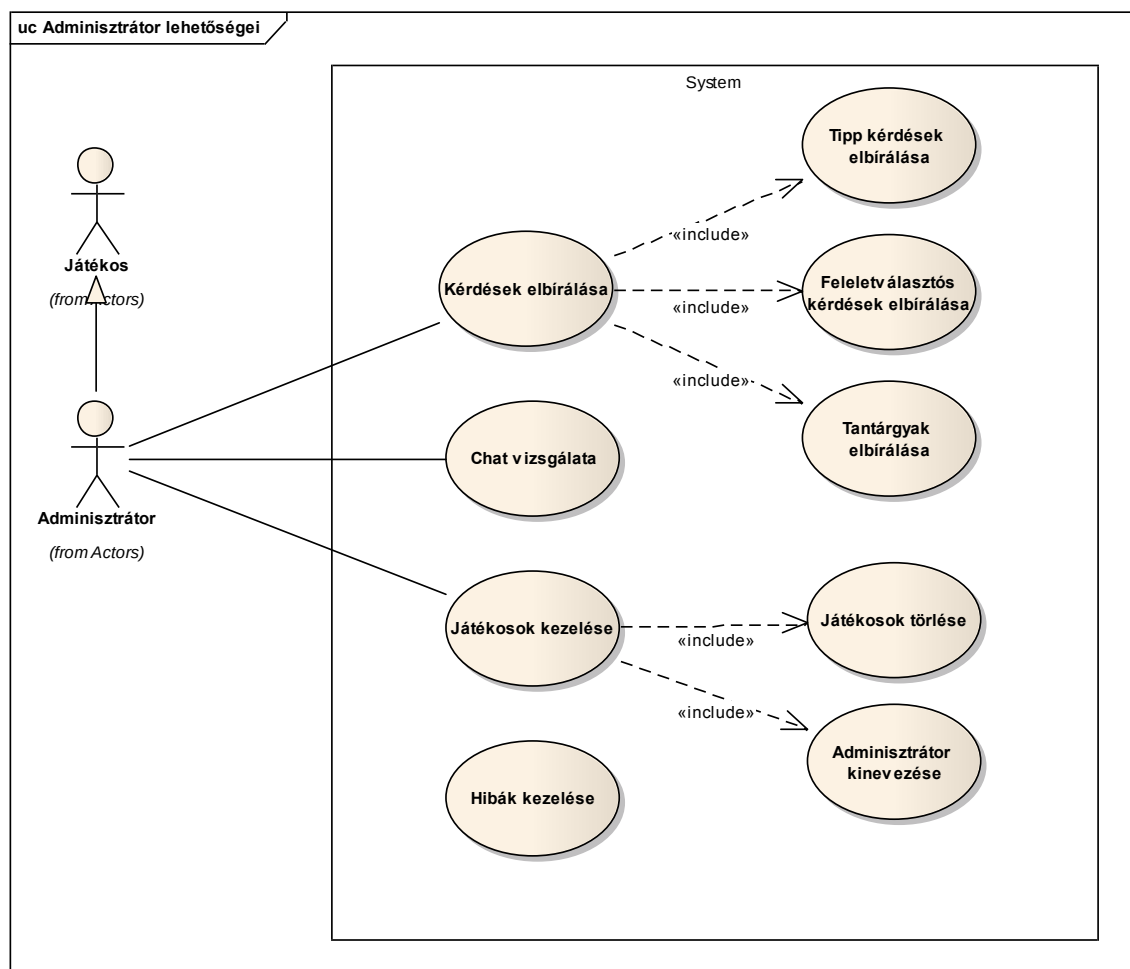
Irodalomkutatásom és tesztjeim végeredményeképpen az adatbázis elérést a kritikus területen erősen típusos DataSet-tel, a ritkábban használt műveleteknél LINQ to SQL-lel, valamint a sok lekérdezést vagy módosítást igénylő feladatokhoz az adatbázison tárolt eljárással fogom megvalósítani.

5. A RENDSZER SPECIFIKÁCIÓJA

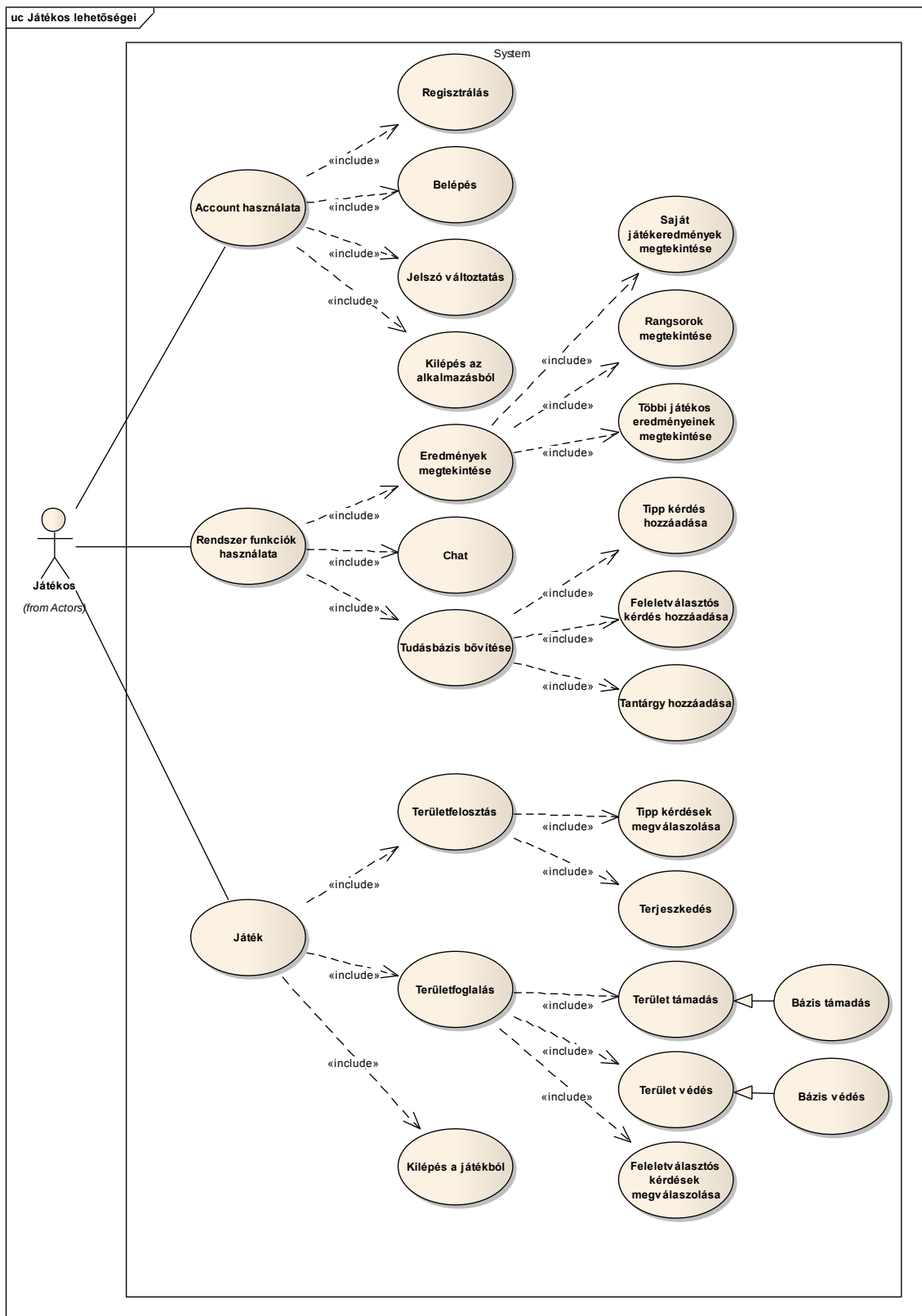
A szakdolgozatomban egy online stratégiai kvízzjátékot készítek, amely a mérnök informatikus hallgatók felkészülését segíti a zárthelyikre és a vizsgákra játékos formában. Alapvetően kétféle szerepkört különítek el, a játékost és az adminisztrátort.

5.1. Adminisztrátor szerepkör

Az adminisztrátor a játékosok számára elérhető funkciókon túl lehetőséget kap az adatbázisba feltöltött kérdések áttekintésére és a játék során feltett kérdések közé sorolására. Ez azért fontos, hogy ne kerülhessenek be hibás, ellenőrizetlen kérdések az éles rendszerbe. Ezen túl a chat beszélgetések naplózását is megfigyelheti, hogy kiszűrhesse az olyan játékosokat, akik zavarják a többiek nyugalmát, esetleg nem odaillő kifejezéseket használnak. A rendszerben levő játékosok adatait is megtekintheti és a problémás személyeket törölheti, illetve kinevezhet másik felhasználót is adminisztrátornak, aki ezt követően elérheti ugyanezeket a funkciókat és segítheti az üzemeltetési feladatokban. Végül a rendszerben történt hibák naplóbejegyzéseit is megvizsgálhatja, hogy felderítse a probléma helyét és kijavíthassa azokat. Az adminisztrátor szerepkör use case diagramja az 5.1. ábrán látható.



5.1. ábra Adminisztrátor használati esetei



5.2. ábra Játékos használati esetei

5.2. Játékos szerepkör

A játékos szerepkörben levő felhasználók valamint az adminisztrátorok használhatják a játék szolgáltatásait. A rendszerbe egy egyszerű regisztrációt követően tudnak majd belépni, miután meg tudják tekinteni eddigi eredményeiket, az egyes játékokban elért tipp és feleletválasztós válaszaik helyességének arányát, az elfoglalt terület nagyságát, valamint a szerzett illetve veszített pontokat. Megnézhetik még a többi játékos pontszámait és a különböző szempontok szerint felállított rangsorokat a lejátszott játszmáik alapján.

Lehetőség van a jelszavuk megváltoztatására, valamint új kérdések és tantárgyak létrehozására. Ezzel a funkcióval a játék dinamikusan bővíthető, így gyorsan tud nőni az elérhető tudásmennyiség. Mindezen túl a rendszert használók chatelhetnek is egymással. A játékos szerepkör use case diagramja az 5.2. ábrán látható.

5.3. A játékmenet

5.3.1. Belépés a játékba

A legfontosabb része a rendszernek maga a játék. A felhasználók tudnak választani a tantárgyak közül, hogy melyikre szeretnének felkészülni, így csak abból a témakörből fognak kérdéseket kapni. Választhatják azt is, hogy vegyesen, bármilyen jellegű kérdéseket kapjanak. Ezt követően bekerülnek egy várakozási listába, amely az általuk választott témakörben várakozó játékosokat tartalmazza.

Ha legalább hárman várakoznak, akkor egymás ellen játszhatnak, ha sokan jelentkeztek, akkor véletlenszerűen kerülnek kiválasztásra az ellenfelek. Ha háromnál kevesebben várnak, akkor bizonyos idő után robot játékos jön létre, mely automatikusan generált válaszokat ad. Így abban az esetben is gyakorolhat a felhasználó, hogyha éppen kevesen használják a webes alkalmazást, illetve ha az adott tantárgy iránt alacsony az érdeklődés.

5.3.2. Játéktér

Bizonyos idő elteltével megjelenik a játéktér, amelyen látható a Neumann kar épülete rajta az elfoglalható területekkel, melyekből minden játékos kap egyet-egyet, amely majd a bázisául szolgál. Mindegyikük eltérő színnel rendelkezik, hogy jól meghatározható legyen, hogy melyik terület kihez tartozik. A játékosok eddig elért pontszáma is megjelenítésre kerül, hogy mindenki tudhassa, milyen erős ellenfelekkel áll szemben. Az adott játékban elért pontszám is látható, amely az elfoglalt területek arányában változik. Rövid idő elteltével megjelenik az első kérdés.

5.3.3 Kérdés típusok

A kérdéseket alapvetően két csoportra bontottam, tippelős és feleletválasztós típusú válaszokat kell adni rájuk. A tipp kérdéseknél egy számot kell beírni, amelynek minél közelebb kell lennie a helyes megoldáshoz. Ebben az esetben az nyer, aki a legjobban megközelíti a jó választ, vagy egyenlő válaszok esetén a leghamarabb írta be a megoldást. A feleletválasztáskor négy lehetőség közül lehet kiválasztani a leginkább megfelelőt.

5.3.4. Területfelosztás

A játékmenet két részre osztható. Az elején a tipp kérdések megválaszolása alapján kapnak a játékosok területet jutalmul a helyes válaszáért. Ez mindaddig folytatódik, amíg az összes terület a játékosok fennhatósága alá kerül.

5.3.5. Területfoglalás

Ezután következik a második szakasz, amikor már egymás területeit támadhatják meg és foglalhatják el. Véletlen sorrendben kapnak lehetőséget a kiválasztott ellenfél területének megjelölésére, majd egy feleletválasztós kérdés megválaszolását követően siker esetén a támadó megkapja a területet, rossz válasz esetén a védőé marad, döntetlen esetén pedig egy tippelős kérdés dönt a csata kimeneteléről. Számos harcot követően véget ér a játék, mikor mindenki láthatja az eredményeit és a módosult pontszámát, mely a válaszok helyessége és az elfoglalt területek arányában változik.

5.3.6. A játék hatása a használóra

Ez a játékmódszer elősegíti, hogy a játékot használó diákok nemcsak pontosan, hanem gyorsan is tudják feleleveníteni az órákon megszerzett tudást, ezzel időt nyerve a vizsgákon. A rosszul megválaszolt kérdések helyes megoldásai pedig könnyebben eltárolódnak a memóriájukban, mintha csak az írott jegyzetekből készülnének fel.

6. AZ ELKÉSZÍTENDŐ STRATÉGIAI JÁTÉK RENDSZERTERVE

A szakdolgozatom célja egy olyan online alkalmazás elkészítése, mely a mérnök informatikus hallgatók játékos formában történő felkészítését segíti az egyetemi megpróbáltatásokra. Erre a feladatra a Honfoglaló nevű stratégiai kvízzjátékot találtam alkalmas mintának, mivel könnyed formában teszi lehetővé a tanulást és a játékmenete miatt nagy népszerűségnek örvend. Az én megvalósításomban a játék logikai felépítését kisebb módosításokkal átveszem, viszont az eredeti weboldal teljesítmény problémái miatt a technikai hátteret teljesen átformálom.

Az adatbázis eléréshez az előző fejezetekben már megvizsgáltam és kiválasztottam a feladat szempontjából a leghatékonyabbakat. A játékban saját algoritmusokat fogok használni, a továbbiakban pedig kiválasztom a megfelelő web fejlesztési technológiát, amellyel elkészíthetem a webes kiszolgálót, amely kiszolgálja majd a felhasználók kéréseit.

6.1. Fejlesztési technológiák vizsgálata

6.1.1. Flash

A Honfoglaló weboldalainak nagy része a Flash technológiát használja, ezért elsőként ezt vizsgálom meg. Ezzel a technikával szép animációkat lehet készíteni, ez a website egyes részein hasznos lehet, viszont hogyha minden ebben készül el, akkor azok a funkciók, amelyeket szerver oldalon kellene megoldani áttolódnak egy lassú, script alapú kliens oldali rendszerbe. Ezeket a teljesítménybeli hiányosságokat gyakran tapasztaltam is a játékok során. Sokszor csak fekete képernyőt láttam, a grafikai felület teljesen összeomlott, és ki kellett lépnem, hogy folytatni tudjam a játékot, ezzel pontokat veszítve.

A Flash egy gyengén típusos script nyelv, futási időben határozza meg a változók típusát, ezért könnyen megbújhatnak benne olyan hibák, melyek egy magasabb szintű objektumorientált nyelvben már a fordítás előtt felfedezhetőek. Az alkalmazás nyomkövetése, a hibák felderítése is nehézkes. Ezzel ellentétben a .NET-ben például a futó webes alkalmazás minden objektuma külön-külön megfigyelhető és nyomon követhető.

6.1.2. PHP

A Honfoglaló weboldalainak fennmaradó része PHP nyelven íródott, így például a bejelentkezés és a felhasználó által bevitt adatok validálása. A PHP egy gyengén típusos script nyelv, ezért könnyen előfordulhatnak az adatok feldolgozása közben konverziós hibák. Emiatt a felhasználótól megkapott felhasználói név és jelszó is nehézkesebben ellenőrizhető, mivel ezek tartalmazhatnak olyan szöveget, melyek html (XSS) vagy adatbázis parancsokként (SQL injection) értelmezhetőek. Ha az ilyen hibás vagy rosszindulatú adatokat nem szűrjük ki a bevitt követően, akkor, ha bekerülnek a

rendszerbe komoly biztonsági rést okozhatnak, melyen keresztül könnyen hozzáférhetnek fontos adatokhoz ezzel veszélyeztetve a felhasználókat.

Az ASP.NET az AJAX segítségével egyszerű kontrollokkal biztosítja, hogy csak a megfelelő formátumú adatokat lehessen bevinni, és ezek helyességéről felhasználóbarát módon tájékoztatja szoftvert használót.

A PHP nyílt forráskódú programnyelv, ez nagy előnye az ASP.NET-tel és a Microsoftos technológiákkal szemben, viszont mivel számos, esetleg kevésbé hozzáértő ember is fejlesztette, ezért könnyebben előfordulhatnak biztonsági rések, harmadik féltől származó komponensek könnyen tartalmazhatnak veszélyes kódrészleteket.

6.1.3. ASP.NET

Az egyetemi évek során részletes ismereteket szereztem a .NET fejlesztőkörnyezetről és a C# programnyelvről, viszont a .NET által nyújtott webfejlesztési lehetőségekről keveset tanultunk, ezért felkeltette a kíváncsiságomat az ASP.NET, amely webes alkalmazások készítését segítő keretrendszer. Mivel a .NET-es követelményeket megvalósítja és Visual Studio fejlesztőkörnyezettel lehet fejleszteni, ezért számos eszköz áll rendelkezésre, hogy megbízható és jól nyomon követhető webes alkalmazásokat hozzunk létre.

A .NET platformmal lehet a leginkább felfedezni a hibák forrását, mivel például a reflexió segítségével futásidőben minden elemről el tudjuk dönteni, hogy hova tartozik. A debuggolása nagyon részletes és pontos, mivel a szerverrel kapcsolatban álló összes kliens adatai lekérdezhetőek.

A C# erősen típusos nyelv, ezért még fordítás előtt kiderülnek a nagyobb problémák, és a hibakezelési rendszere is nagyon fejlett. Mindezek mellett az ASP.NET az oldal elemeit XHTML tag-ekké alakítja, és így küldi a böngészőnek, nem pedig az egész weboldal egy nagy Flash animáció. Így az egyes elemek is jobban elkülönülnek, ezért megfelel a www szabványainak.

6.1.4. Összehasonlítás

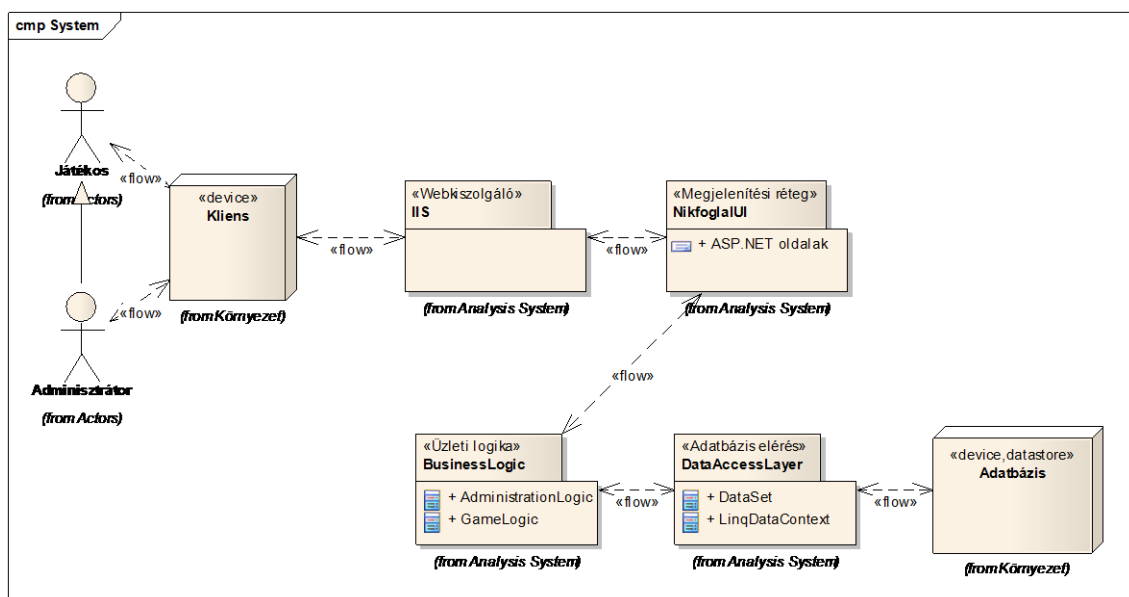
Az összegyűjtött információk alapján a Flash által nyújtott grafikai lehetőségekkel szemben előtérbe helyezem az ASP.NET teljesítménybeli erősségeit, valamint a nyílt forráskódú PHP-vel szemben előnyben részesítem a megbízhatóbb, biztonságosabb validálást biztosító Microsoftos megoldásokat. Tehát a gyengén típusos script nyelvekkel szemben az objektumorientált, jól nyomon követhető, stabil ASP.NET-et választom. Mindezek mellett az ASP.NET kihívást jelent számomra, mivel az egyetemi évek alatt nem volt alkalmam megtanulni ezt a technológiát, ezért a szakdolgozatom megírása folyamán szeretnék ezen népszerű, és az üzleti életben is elterjedt webfejlesztési keretrendszerrel megismerkedni. A továbbiakban az ehhez szükséges környezet kiépítését fogom megtervezni.

6.2. A rendszer architektúrája

6.2.1. Felépítés

A webes alkalmazás elkészítéséhez először részletesen megtervezem az egyes rétegeket, melyekből a rendszer fel fog épülni. Először is szükség van egy adatbázisra az adatok tárolására, majd a szoftvernek csatlakoznia kell ehhez az adatelérési rétegen keresztül, hogy az üzleti logika a megfelelő módon feldolgozhassa és továbbíthassa az információt a felhasználói felület felé.

A webes felületet az ASP.NET keretrendszerrel fogom megvalósítani, amelyhez IIS webkiszolgálóra lesz szükség. A rendszer logikai felépítésének terve a 6.1. ábrán látható.



6.1. ábra A rendszer felépítése

6.2.2. Adatbázis

A rendszerben szükség lesz a felhasználók adatainak tárolására. Ennek a megvalósítására két lehetőség állt előttem. Az egyik, hogy létrehozom az adatbázisban a szükséges táblákat, a felhasználói felületen létrehozom a szükséges kontrollokat, biztosítom az adatok validálását, biztonságos tárolását figyelembe véve az esetleges támadásokat. Viszont rájöttem, hogy ennél sokkal célravezetőbb megoldás, ha ezen a területen igénybe veszek már kész komponenseket, melyek alaposan teszteltek, és megbízhatóak. Ezért úgy döntöttem, hogy az ASP.NET által nyújtott membership és role provider funkciókat fogom használni, melyek működéséről részletesebben majd az üzleti logika alfejezetben írok.

Ezeknek a szolgáltatásoknak az adatbázis elérését átirányítottam a saját adatbázisomra, és az általuk generált táblákat hozzákötöttem a sajátjaimhoz, ezáltal biztosítva a felhasználói adatok biztonságos tárolását. Ezekben a táblákban vannak elmentve az adminisztrátor és játékos szerepkörök, melyek az egyes felhasználói rekordokhoz vannak csatolva. Az általam létrehozott adatok a felhasználók egyedi azonosítóin keresztül csatlakoznak ezekhez a táblákhoz.

Először tárolnom kell a játékosok pontjait, eddig elért eredményeit, játékszámát, az egyes játékokat, statisztikákat. A játékosok közti kommunikáció tárolásához el kell menteni a chatbeszélgetéseket. Szükség van még táblákra a tippelős és feleletválasztós kérdések, valamint a tantárgyak számára. A rendszerben megjelenő hibákat is menteni kell, hogy a rendszergazda könnyen megtalálja a problémák helyét és okát. Az adatbázis struktúrájának részletes terve a 6.2 ábrán látható.



6.2. ábra Az adatbázis terve

6.2.3 *DataAccessLayer*

A harmadik fejezetben részletesen megvizsgáltam a rendelkezésre álló szakirodalmat a .NET által nyújtott adatbázis elérési módszerekről. Ezt követően mindegyik lehetőséget tesztek alá vettem a rendszer számára fontos szempontok szerint. Ezek során megállapítottam, hogy a kérdések eléréséhez a leghatékosabb módszer, ha az alkalmazás indulásakor egy típusos DataSet-be töltöm a TipQuestion, ChooseQuestion, Subject táblák tartalmát. Ezt követően már a szerver memóriájából nagy sebességgel ki lehet olvasni az adatokat osztályok struktúrájában.

Az adminisztrátori tevékenységekhez, vagyis a kérdések, felhasználói adatok, üzenetek, hibák listázásához, módosításához, törléséhez az adatokat LINQ to SQL-lel típusos objektumokként érem el. Így a rekordok nem tárolódnak feleslegesen a memóriában, az adatmanipulációs műveletek pedig SQL parancsokká alakulva gyorsan lefutnak az adatbázison.

A gyakran használt mentési, illetve adatkinyerési műveletekhez, amelyek nem a kérdésekhez kapcsolódnak és nincs szükség a memóriabeli tárolásukra az adatbázisban tárolt eljárásokat használok. Ilyen funkció a játékos rangsorban elfoglalt helyének lekérdezése, valamint a játék végeredményének és statisztikai adatainak elmentése. Ezeket a feladatokat azért tárolt eljárásokkal oldottam meg, mivel a tesztek során ez volt a leggyorsabb módja az adatok módosításának. Ezeket a tárolt eljárásokat az adatbázisban hozom létre és a LINQ to SQL osztályokkal érem el metódusokként.

A típusos DataSet és a LINQ to SQL számára szükséges osztályok egy külön osztálykönyvtárban helyezkednek majd el, amely referenciaként könnyen importálható az üzleti logikába, ezáltal jól elkülönül a megjelenítéstől és a program logikai felépítésétől, így könnyen cserélhető lesz ez a komponens, ha a jövőben újabb technológiával kell majd csatlakozni az adatbázishoz.

6.2.4. *BusinessLogic*

Ebben az osztálykönyvtárban helyezkednek el azok az osztályok, melyek elérik az adatelérési réteget és elvégzik a rendszer logikai feladatait, mint például a számításokat, a játék ütemezését, a kérdések, a játékosok és a robot játékosok kezelését. Ez a réteg szolgáltatja a felhasználói felület számára a megjelenítendő adatokat.

A DataAccessLayerHelper osztály hivatott arra, hogy az osztályokká alakított adatbázis táblákat elérje a rendszer. Az alkalmazás ezen az osztályon keresztül tud kommunikálni az adatbázissal, az itt lévő metódusokkal tudja csak elérni, így egy helyen módosíthatóak változás esetén az eljárások. Ezek a statikus metódusok felelősek LINQ to SQL kapcsolat esetén mentéskor létrehozni egy új egyedet, majd feltölteni a megkapott adatokkal és ezután elküldeni az adatbázisnak a változásokat tartalmazó generált utasítást. Ilyen feladat például az új felhasználók, kérdések, üzenetek, hibák mentése. Adatok kinyerésekor egy feltételnek megfelelő rekordot keres ki a

DataContextból, valamint meghívja a megadott paraméterekkel a kívánt tárolt eljárást. Az utóbbira példa a játék eredményének elmentése valamint a játékos rangsorbeli helyének lekérdezése.

A kérdésekkel és tantárgyakkal feltöltött DataSet elérése is ebben az osztályban történik. A statikus metódusok meghívásával feltölthetjük a DataSet-eket, valamint véletlenszerűen kiválasztott kérdéseket választhatunk ki a megadott tantárgy alapján, hogy a témakörnek megfelelő kérdéseket kaphassanak a játékosok. Az algoritmus garantálja, hogy egy játékon belül ne forduljon elő még egyszer ugyanaz a kérdés.

A kiválasztható tantárgyak közé a rendszer csak azokat gyűjti ki, amelyekből tippelős és feleletválasztós kérdésekből is rendelkezésre áll megfelelő számú egy játék végigjátszásához.

A másik nagy egység a rétegen belül a játék logika. Ide tartoznak a játékosok, robot játékosok, tippelős és feleletválasztós kérdések adatait és funkcióit tartalmazó osztályok. Ezen témakör részletes működését nagy terjedelme és összetettsége miatt a Játék logika című alfejezetben fejtem ki bővebben.

6.2.5. User Interface

Mivel az ASP.NET keretrendszert választottam ki az alkalmazásom fejlesztésére, ezért a felhasználói felület fő építő elemei az úgynevezett web formok. Ezek a web oldalak aspx kiterjesztésű fájlokban tárolódnak, ezek általában statikus XHTML tageket tartalmaznak, melyekben elhelyezhetőek a webes vezérlők, valamint a felhasználók által létrehozott kontrolok. Ezek mellett elhelyezhetőek dinamikus tartalmak is, melyek a szerverről kérdeznak le adatokat. A vezérlőkkel adatokat és kéréseket küldhetünk a kiszolgáló felé, amely újra tölti az egész oldalt, vagy az egyes részeket.

Az oldalakhoz rendelkezésre állnak „code-behind” fájlok, amelyek a megjelenítéstől elkülönítve tartalmazzák a dinamikus programkódot. Itt lehet az oldalbetöltődés életciklusának folyamán beállítani a kontrolokat, valamint az általuk küldött eseményeket kezelni és meghívni az üzleti logikát, hogy megjeleníthessük a felhasználó számára a kívánt tartalmakat.

Az ASP.NET alkalmazások az állapotmentes http protokollon keresztül érhetőek el a web szerveren. Mivel az elkészítendő játékban folyamatosan szükség van az aktuális állapotok tárolására, ezért meg kellett találni a megfelelő megoldást erre. Az ASP.NET számos lehetőséget nyújt ezen probléma kiküszöbölésére.

A legáltalánosabb az Application state, amely az egész alkalmazás számára, bármely felhasználó által elérhető. Ezt a gyűjteményt kulcs-érték párok definiálásával lehet használni. Bármilyen objektum elhelyezhető benne egyedi névvel.

Amikor az alkalmazás első példánya létrejön, meghívódik az Application_OnStart esemény, amelyben kezdőértékeket adhatunk meg, amelyek az utolsó példány megszűnéséig elérhetőek. Jelen esetben a kérdéseket tartalmazó DataSet tárolására és lekérdezésére a rendszer indulásától kezdve szükség lehet. Ebből kifolyólag az Application_OnStart esemény lefutásakor elhelyezhetjük a kérdéseket az Application

memóriában, így a játék folyamán már nagy sebességgel elérhetőek lesznek, nem kell majd az adatbázis kapcsolatok kiépítésére várni és folyamatos lekérdezéseket küldeni.

A játékosok közti legfrissebb chat beszélgetések tárolása is történhet ebben a memóriában, mivel minden játékosnak látnia kell, és így nem kell minden üzenet után az adatbázisba menteni. Ha az üzenetek száma elért egy bizonyos határt, akkor a régi üzenetek egy részét már el lehet tárolni, így nem telítődik túl a memória.

Az állapotkezelés másik módja a Session state, amellyel az egyes felhasználók adatait külön tárolhatjuk. Az ebben tárolt változók a Session gyűjteményen keresztül érhetőek el, amely minden egyedi minden kapcsolat számára. Egy idő intervallumot előre meg lehet határozni, amelynek leteltével, ha a felhasználó inaktív volt, akkor megszüntetjük a számára letárolt Session példányt. A felhasználó csak a kapott session ID-jával érheti el az objektumot, amit a böngésző cookiejában tárolunk vagy az URL-ben. Ez az állapot-tárolási mód megfelelő a játékosok ideiglenes adatainak tárolására, például az eltelt várakozási idő vagy az aktuális állapot a játékon belül.

A View state egy oldalon belüli információk megjegyzésére alkalmas a különböző kérések között. Ezzel a módszerrel a kontrollok állapota egyszerűen tárolható az egész játékmenet alatt.

Az ASP.NET-ben létrehozhatóak úgynevezett MasterPage-ek, amelyekkel egy általános keretet adhatunk az oldalainknak, így egységes keretet teremtve, és elkerülve az elemek felesleges másolását. Az általános, az összes tartalmazott oldal számára szükséges logikákat is lefuttathatjuk a master oldal mögöttes kódjában. Az oldalak fordításakor a master és a tartalmazott oldal egy összefüggő oldallá fordul, az életciklus folyamán sorban végrehajtja az összes eseményt. Itt található a fejléc, valamint a oldalak kerete és a navigálásra szolgáló menü. A fejlécben van elhelyezve a játékosok be- és kijelentkezését kezdeményező link.

A megjelenítési rétegben találhatóak a belépéshez és az új felhasználó regisztrációjához szükséges oldalak. A többi oldal csak a belépést követően szerepkörtől függően érhető el. Ezen oldalak AJAX alapú kontrollokat használnak, melyek a Membership provider szolgáltatást használják és javascript függvényekkel validálják a bevitt adatokat, majd tárolt eljárásokkal elmentik az adatbázisba. Ez a módszer nem enged a rendszert sértő adatokat bevinni, és az információkat is titkosítva küldi és tárolja, így a felhasználók adatai védve vannak az alapvető támadásokkal szemben.

Az adminisztrátori oldal csak adminisztrátor szerepkörrel rendelkező felhasználók számára lehetséges. Ezen az oldalon lehet a játékosok által feltöltött kérdéseket és tantárgyakat elbírálni, hogy valós adatokat tartalmaznak-e, megfelelnek-e a kritériumoknak és a rendszerbe lehet-e bocsájtani. Csak az adminisztrátor által jóváhagyott kérdések kerülhetnek be a játékba. Joga van még a játékosok törléséhez illetve másik felhasználót adminisztrátornak nevezni ki, hogy segíthesse a munkáját.

Lehetőség nyílik az üzenetek áttekintésére, hogy vannak-e olyan felhasználók, akik magatartásukkal zavarhatják a többi játékost, és emiatt figyelmeztetést vagy kizárást érdemelnek. A rendszerben felmerült hibák naplóbejegyzéseit is itt tekintheti

meg az adminisztrátor és ez alapján intézkedhet a javításokkal kapcsolatban. Hibák esetén egy oldal jelenik meg, amely tájékoztatja a felhasználót a hibáról. Ez az üzenet mindig a hibák mentése után jelenik meg.

A játékosnak lehetősége van új kérdések és tantárgyak felvitelére a rendszerben. Az ehhez szükséges oldalon kiválaszthatja, hogy milyen típusú feltöltést szeretne végrehajtani, és a megfelelő validálást követően beviheti az adatbázisba. Játékba csak az adminisztrátori ellenőrzést követően kerül be.

A felhasználók a ranglistákat tartalmazó oldalon megtekinthetik az eddigi eredményeiket, valamint a különböző szempontok szerint elért helyezéseiket. Megtekinthető a szerzett pontszámok szerinti ranglista, valamint a tippelős, feleletválasztós kérdésekre adott helyes válaszok aránya alapján számított sorrend, a szerzett területek aránya, a játékok során elért átlagos pontszerzés, valamint a játékokban elért helyezések alapján felállított rangsor. A játékos maga is kiválaszthatja, hogy milyen kategória szerint szeretné megnézni a helyezéseket, de ha nem választ, akkor a rendszer automatikusan, időzítve megmutatja egymás után mindet.

A bejelentkezést követően a felhasználó a főoldalra kerül. Innen tovább navigálhat a ranglistákat és a kérdésfeltöltést tartalmazó oldalakra, valamint adminisztrátori szerepkörrel elérheti az adminisztrátori funkciókat. A lap bal szélén megtalálható a chat ablak, ahol üzenetet küldhet a többi játékosnak. A beírt üzenetek a nevével és a küldés időpontjával együtt jelennek meg.

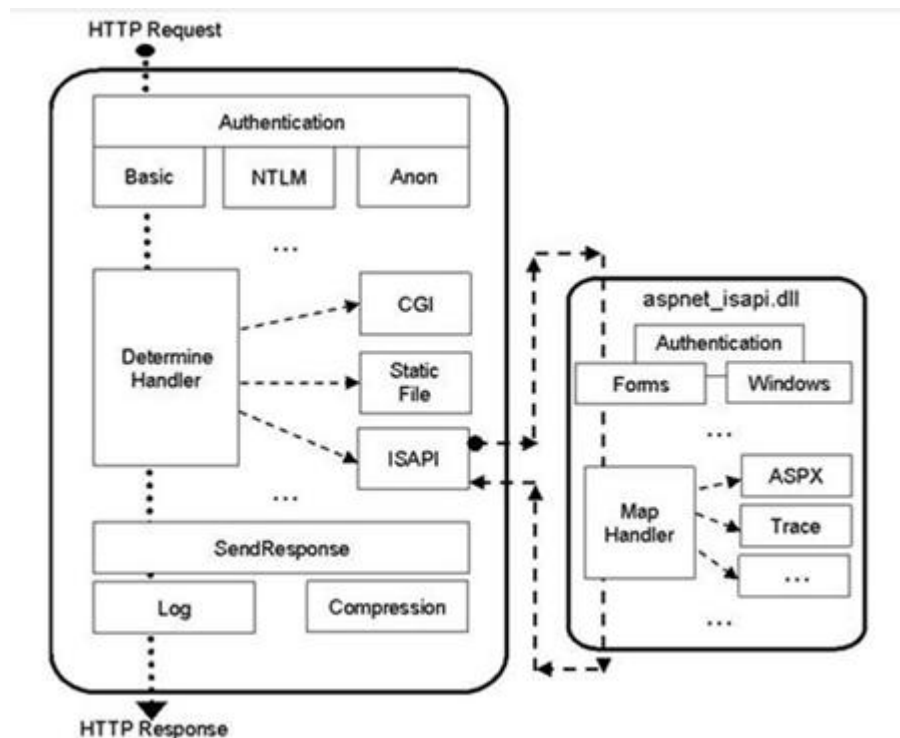
Az oldal közepén az eddig elért játék eredmények láthatóak a játék sorszámával, a pontváltozással, a tipp, feleletválasztó kérdések helyességének arányával, a szerzett területtel és az elért helyezéssel.

Jobb szélén lehet belépni a játékba. Itt ki lehet választani, hogy milyen tantárgy kérdéseire szeretne válaszolni, ha az „Általános” opciót hagyja kiválasztani, akkor bármely témakörből kaphat kérdéseket. Láthatja, hogy éppen hányan várákoznak az adott típusú játéokra. A „Belépek” gombra megjelenik az eltelt idő, és ha elegendő számú játékos, vagyis minimum három várákozik, akkor véletlenszerű kiválasztással bekerülnek egy játékba. Amennyiben túl kevesen várákoznak, akkor sem marad ki a játék élményéből, mivel a rendszer robot játékosokat generál ellenfélnek. Amennyiben már egy megkezdett játékból kilép a felhasználó, akkor lehetősége van a „Visszalép” gombbal újra bekapcsolódni. A beléptetés és sorsolás részletes működését, valamint a játék megjelenítése mögötti logikát a Játékmenet megjelenítése című alfejezetben fejtem ki részletesen.

6.2.6. IIS

Az ASP.NET alapú weboldalak futtatásához IIS (Internet Information Services) webkiszolgálóra van szükség. Az IIS-nek egy belső folyamatkezelője van, amely fogadja a beérkező kéréseket, feldolgozza és lefuttatja az üzleti folyamatokat, majd visszaküldi a kliensnek a választ. Az IIS támogatja az ASP.NET által nyújtott funkcionalitásokat, mint például a form vagy windows alapú azonosítást, a szerepkörök

kezelését a felhasználók számára, a kimenő adatok gyorsítótárazását, az URL-ek átirányítását, valamint az oldalak nyomon követését, debuggolását és hibakezelését. A 6.3. ábra [6] részletesen bemutatja ezt a folyamatot.



6.3. ábra Az IIS funkciói

A Web.config fájlban lehet paraméterezni az IIS beállításait, az adatbázis kapcsolatokat, az autentikáció módját, a hibakezelést, valamint hogy melyik oldal legyen a kezdőlap és melyik jelenjen meg hibák esetén hibakódoktól függően.

6.3. Játék logika

A BusinessLogic rétegben található osztályok tartalmazzák a rendszer működéshez szükséges adatokat és funkciókat. Ezek kommunikálnak a DataAccessLayer-rel, valamint fogadják a weboldalak által küldött kéréseket és számíttják ki a választ.

6.3.1. Player osztály

A Player osztály tartalmazza a játékosokkal kapcsolatban nyilvántartandó adatokat, úgymint a nevét, összpontszámát, játékban elért pontszámát, a kisorsolt színét, a kérdésekre adott helyes és összes válaszainak számát, a nyert területeinek számát, a végeredményét és az állapotát, hogy még a játékban tartózkodik-e vagy már kilépett.

A példány létrejöttkor megkapja a játékos nevét, majd beállítja a szükséges mezők alapértékét, valamint lekérdezi az összpontszámot a DataAccessLayerHelper

osztály segítségével. Minden kérdés után meghívódnak azon metódusai, amelyek alapértékre állítják az adott típushoz tartozó értékeket. Tipp kérdéseknél kiszámítja a helyes válaszhoz képest számított eltérést.

A játék végeztével meghatározza a tipp és feleletválasztós kérdések helyes válaszainak arányát, a megszerzett területek százalékát és ezek átlagtól számított eltérése alapján pozitív vagy negatív pontszámot ad hozzá az eddigi összpontszámhoz.

Ez az érték súlyozva van az eltérés arányában. Tehát ha sokkal jobb eredményt ér el, mint az átlag, akkor a szerezhető 50 pontból ennek arányában részesül, és fordított esetben ennyit veszít. Miután ez a pontszám kiszámításra került elmentjük az eredményeit.

6.3.2. *RobotPlayer osztály*

A Player osztály leszármazottja, felülírja a virtuális metódusait, így futásidőben dől el, az objektum típusa alapján, hogy melyik metódust hívjuk meg. Ezáltal nem kell a rendszerben külön kezelni az objektumokat a létrehozásukat követően. Az alapértékek beállításakor egy véletlenszerű idő generálódik a robot számára. A játék izgalmasabbá tételéhez kidolgoztam egy egyszerű algoritmust, amellyel a tipp válaszknál a helyes választ egy véletlenszerűen generált százalékos értéknek megfelelően módosítom. Ezáltal a robot játékosok a jó választól kis mértékben fognak eltérni, ennek köszönhetően a játékosok is motiválva lesznek, hogy minél pontosabb válaszokat adjanak.

A várakozási időkkel kapcsolatban azt a döntést hoztam, hogyha két robot játékos harcol egy területért, akkor ne várjuk meg, amíg letelik a teljes idő, hanem úgy állítottam be, hogy a rendszer mindig úgy kezelje, mintha már válaszoltak volna, ezáltal nem kell mindig végig várni az időt, hanem a valódi játékosok folytathatják a játékot. A robot játékosok eredményeit értelemszerűen nem mentem az adatbázisba.

6.3.3. *GameQuestion osztály*

Az őse a tipp és feleletválasztó kérdéseknek, definiálja a virtuális metódusokat, melyeket leszármazottai meg fognak valósítani. Ezek felelősek a véletlenszerű kérdések kiválasztásáért tantárgynak megfelelően, valamint a válaszok számának beállításáért a játékosoknál. A konstruktorban jön létre a már feltett kérdések sorszámát tartalmazó HashSet. Ez a generikus gyűjtemény gyors keresést biztosít.

A GameTipQuestion és a GameChooseQuestion osztályok feladata nem csupán az egyes kérdések kezelése, hanem a területfelosztás, illetve területfoglalás játékrészek managelése. Ezáltal a játékosok adatait és a rangsorokat is módosítják.

6.3.4. GameTipQuestion osztály

A GameQuestion leszármazottja, felülírja az őse virtuális metódusait, mivel a különböző kérdés típusoknál különböző megvalósításra van szükség. Betölti egy véletlenszerű tipp kérdést az adatbázisból, és tárolja azt, amíg sor nem kerül a következő kérdésre, valamint elmenti a sorszámát a már használt kérdések közé. Miután letelt az idő nullázza azon játékosok válaszait, akik nem válaszoltak időben a kérdésre. A játékosokat válaszaik helyessége és a válaszigó alapján egy egyszerű algoritmussal rangsorolja. Ezt követően növeli a helyes és összes válaszaik számát. A területfelosztáskor az első helyezett dupla pontot kap, a második egyet. Területfoglaláskor a győztes egy pontot kap.

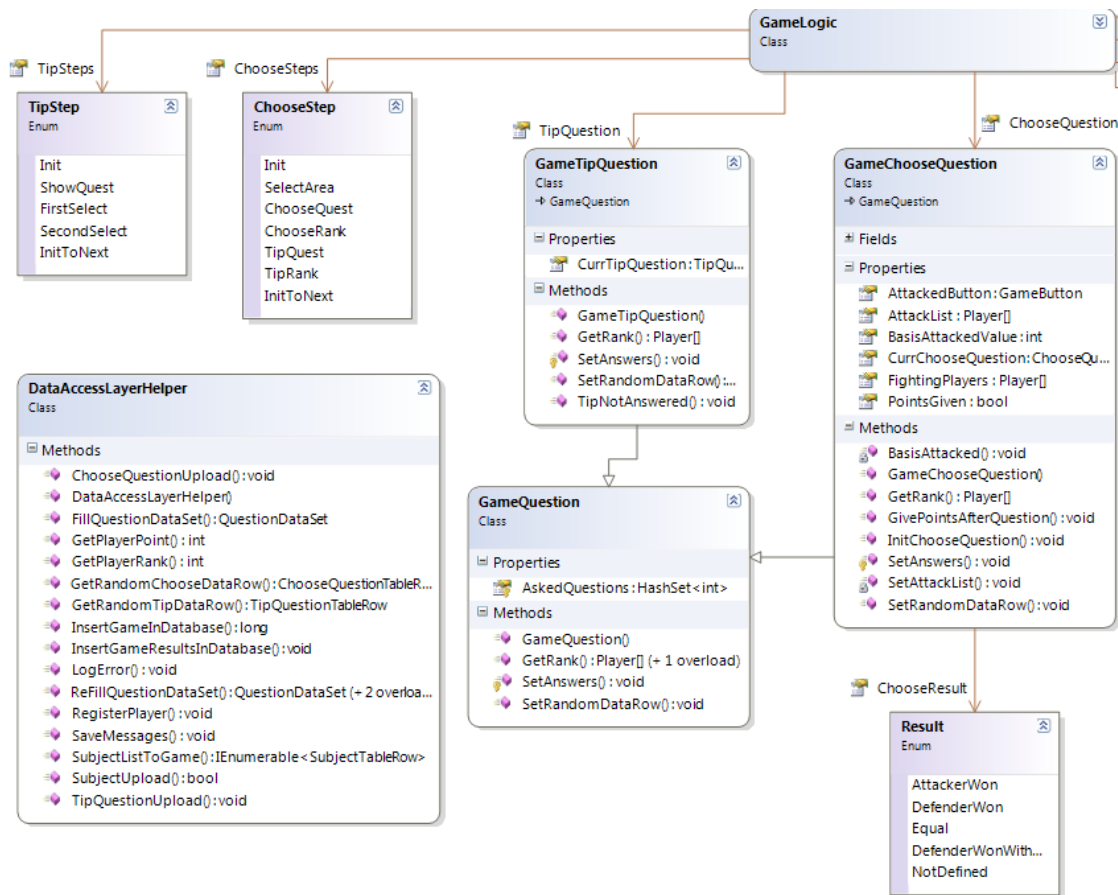
6.3.5. GameChooseQuestion osztály

Szintén a GameQuestion osztály leszármazottja. Az objektum létrejöttkor létrehozza a véletlenszerű támadási sorrendet, mellyel négy szakaszra osztja a területfoglalási fázist, majd beállítja a harcban álló játékosok, az aktuális kérdések és a támadott terület értékét alapállapotra. Minden kérdés után meghívja a játékosok inicializáló metódusait. Ez az osztály is megvalósítja a kérdések betöltését az adatelérési rétegből és tárolja a már feltett kérdéseket.

A feleletválasztós játékrészben lehetőség van az ellenfél bázisának megtámadásra is, ezt az eshetőséget is le kell kezelni. Amennyiben a támadó jól válaszol, a bázis értékének egy részét megkapja pontként. Három nyert támadás után a védekező összes területe a győztes játékoshoz kerül, és több kérdésre már nem válaszolhat, kiesett a játékból.

A feleletválasztós kérdések során a harcnak többféle végkimenetele lehet. Ha a támadó nyert, akkor 400 pontot kap és a terület az ő színét viseli, a védő pontszáma pedig a terület értékével csökken. Amennyiben a védő nyer, 100 pontot kap jutalmul. Ha mindketten jól válaszolnak, akkor kapnak egy tippelős kérdést, amelyben már mindenképp eldől a végeredmény, azonos válasz esetén a válaszigó dönt. Ha a támadó és a védő is rossz választ ad, akkor nem kapnak pontot. Ezen állapotok megkülönböztetésére egy enumot használok.

A DataAccessLayerHelper és ezen osztályok osztálydiagramja és kapcsolatai a GameLogic osztállyal a 6.4 ábrán látható.



6.4. ábra A kérdések osztálydiagramja

6.3.6. GameButton osztály

Ennek az osztálynak csupán annyi a feladata, hogy a vizuálisan megjelenített gombok állapotát tárolja a lekérések között. Megadható a gomb azonosítója, színe amely meghatározza, hogy melyik játékoshoz tartozik, valamint az értéke, amely mutatja, hogy elfoglalásával hány pontot lehet szerezni.

6.3.7. GameLogic osztály

Az egész játékmenet ütemezésért és vezérléséért felelős osztály. Itt tárolódnak a konstans változók, melyekkel paraméterezhető a játék, mint például a tipp és feleletválasztós kérdések maximális száma, a harctér mérete, a várakozási idő az egyes kérdések között, a szabad, a már elfoglalt területek és a bázis értéke. Ez az osztály a játék ideje alatt végig az Application memóriában marad, ezért itt tárolódnak a játékosok és a kérdések adatai, valamint a gombok színe és értéke. Elmentem az objektum létrejöttének idejét is, hogy a jövőben az esetleg bent ragadt játék objektumokat is fel lehessen szabadítani.

Az aktuális tantárgyat is beállíthatjuk, így a kérdések eszerint lesznek szűrve. Egy timerre is szükség van, amely az egész játékmenetet ütemezni fogja, és mellyel lehet majd figyelni az eltelt időt. Két enummal lehet majd figyelni a tipp és feleletválasztós kérdések állapotát, hogy már létrejöttek, megjelentek, válaszoltak rá a játékosok, megtörtént az inicializálás a következő kérdéshez.

Miután összegyűlt elegendő játékos egy játékhoz, illetve robotokkal helyettesítve lettek, létrejön egy GameLogic objektum. Ekkor beállítja az alapértékeket, elindul a timer és elkészül a harctér, melyen minden játékos bázisának helye a játékos színével és a bázis alapértékével kisorsolásra kerül. Egy algoritmus biztosítja, hogy nehogy két játékos bázisa azonos helyre kerüljön.

A belépést követően a rendszer vár egy kis időt, amíg az összes játékos számára betöltődik a kezdőképernyő, ha ez nem történik meg valamely játékos számára, a játék akkor is megy tovább, és később csatlakozhat.

A játék folyamán végig le kell kezelni azt az esetet, ha a játékos nem figyel oda és nem válaszol a kérdésre, vagy nem választ területet. Ezekben az esetekben is a játéknak mennie kell tovább.

A játékmenet két fő részre osztható, a területfelosztásra és a területfoglalásra. A területfelosztás folyamán tipp kérdésekre válaszolhatnak a játékosok és ezek eredménye alapján választhatnak területet maguknak, amíg a szabad területek el nem fogynak. Minden kérdésnél ugyanaz a folyamat fut le, melynek aktuális állapotát a tipp kérdésekre vonatkozó enum mutatja.

Először a kérdést betöltjük egy GameTipQuestion példányba, majd megjelenik a játékosok számára. Ha a definiált időn belül nem érkezik válasz valamely játékostól, akkor a rendszer nullázza a választ. Miközben megtekinthetik a helyes választ, valamint a válaszaik és a gyorsaságuk alapján felállított rangsort, a legjobb választ adó két területet választhat magának, a második egyet. Ebben az esetben is előfordulhat, hogy nem jelöli meg egyik területet sem a határidőn belül, ilyenkor a rendszer véletlenszerűen választ számára a szabad területek közül. A folyamat végén beállításra kerülnek az alapértékek a következő kérdéshez. Ez a ciklus mindannyiszor lefut, amíg az összes terület el nem fogy.

Ha már nem maradt szabad terület elkezdődik a területfoglalás. Ekkor létrejön egy objektum a GameChooseQuestion osztályból. Ez létrehozza a támadási sorrendet tartalmazó tömböt, mely alapján lesz majd kiválasztva a támadó játékos az adott kérdéshez. Ha a pontszáma nulla, az azt jelenti, hogy már elfoglalták a bázisát és kiesett a játékból, ezért átugorjuk ezt a lépést, és a sorrendben a következő játékos támadhat. Ebből következik, hogyha már csak egy játékos van játékban, akkor nincs több kérdés és vége.

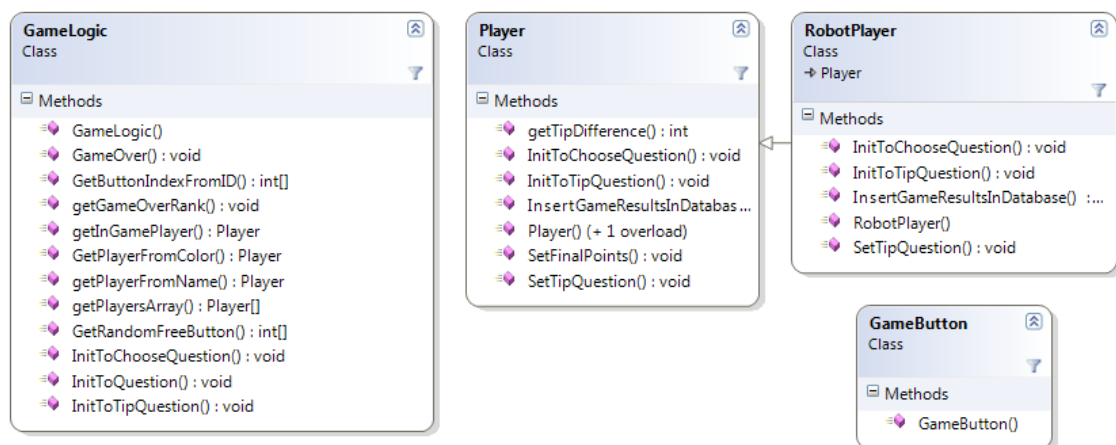
Ha a támadó még játékban van, akkor először területet kell választania, amelynek megszerzéséért harcolni fog. Itt is fenn áll a lehetőség, hogy nem választ időben, ilyenkor a rendszer a többi játékos területei közül véletlenszerűen választ számára. Ekkor a GameChooseQuestion objektum tárolja a támadott területet és a harcban álló játékosokat, és ezen túl a rangsorszámításnál is csak őket veszi figyelembe. Ezt követően megkapják a kérdést a négy választási lehetőséggel. Ha nem választott, a rendszer a választát érvénytelennek veszi, így nem nyerhet az adott fordulóban. Ezt követően megtekinthetik a rangsort, és ha a támadó nyert, akkor megkapja a területet, ha a védő, akkor megtarthatja.

Amennyiben döntetlen az állás a feleletválasztós kérdés alapján, akkor egy tippelős kérdés következik, amely során végigfut a tippelős kérdéseknél megismert folyamat, megjelenik a kérdés, időt kapnak a válaszra, utána megjelenik a rangsor mely alapján már egyértelmű a harc kimenetele. Az eredmény alapján a játékosok megkapják a pontokat és a támadott terület tulajdonosa és értéke is aktualizálódik.

Jelen esetben a területfoglalás négyfordulós, ezért összesen tizenkétszer zajlik le a feleletválasztós kérdéseknél ismertetett folyamat. A fordulók száma szabadon paraméterezhető.

A játék végeztével meghívásra kerülnek a játékosok pontszámító metódusai és lementődnek az eredmények az adatbázisba, kivéve, ha robot játékosról van szó. A játékot is lementjük a győztes nevével. Az első és a második helyezett a megszerzett pontokon felül még jutalmat kap. Ha már az összes játékos kilépett a vége előtt, akkor is befejeződik a játék és mentődnek az addigi eredmények.

A GameLogic és a Player osztály osztálydiagramja a 6.5. ábrán látható.



6.5. ábra A GameLogic osztály osztálydiagramja

6.4. A játékmenet megjelenítése

6.4.1. Az aktuális adatok tárolása

Egy windows form-os alkalmazásnál a változók értékei akár az alkalmazás bezárásáig is megmaradnak, viszont az ASP.NET weboldalak az oldal újratöltésekor elvesztik a változóik értékét. Erre a problémára megoldást kellett találnom, mivel a játékmenet folyamán 15-20 percig is szükség van az állapotok és az aktuális eredmények megőrzésére.

Az ASP.NET az értékek tárolására többféle lehetőséget is kínál. Az egyik az Application memória, amelyben az egész alkalmazás számára elérhető adatok tárolhatók.

A Session memória az adott felhasználó adatait tárolja, de mások ezt nem láthatják. Alkalmas az egyedi információk megőrzésére, viszont az egyes játékosok nem tudnak adatokat cserélni.

Nekem olyan technológiára van szükségem, amely csak a játékosok egy csoportjának teszi elérhetővé a konkrét játékhoz tartozó állapotokat, a szerzett területeket, aktuális eredményeket.

Végül arra a megoldásra jutottam, hogy a GameLogic objektumot az Application memóriában tárolom egy egyedi azonosítóval. Ez a példány tartalmazza a játékhoz szükséges összes aktuális információt, vagyis a területek felosztását, a játékosok adatait, az adott kérdést és a játék állapotát, hogy éppen melyik fázis, milyen kérdés következik. Ennek az objektumnak az azonosítóját a játékosok összerendelésekor elmentem a felhasználók Session memóriájába, ezáltal az egy játékhoz tartozók ugyanazt a területet, ugyanazt a kérdést és egymás pontjait láthatják. Ezzel a megoldással kisebb csoportok számára elérhetővé válik egy konkrét játék.

6.4.2. Belépés a játékba

A főoldalon a különálló területeket egy-egy AJAX-os UpdatePanel-be helyezem, mivel így nem kell az egész oldalt újra tölteni, hanem csak az éppen frissülő részét az oldalnak. Így amikor a Chat ablakban új üzenet érkezik, akkor csak a beszélgetések területe töltődik be újra, a felhasználót nem zavarja meg az oldal folyamatos villogása a teljes betöltődés miatt, valamint a kiszolgálót is kevésbé terheli meg egy kisebb rész megjelenítése.

A főoldal jobb szélén található a Játék elnevezésű terület, ahol a felhasználó kiválaszthatja a kívánt tantárgyat. Ennek a listája a DataAccessLayerHelper osztályban levő metódustól megkapott adatokkal töltődik fel, amely azon tantárgyakat tartalmazza, amelyekhez elegendő tippelős és feleletválasztós kérdés áll rendelkezésre egy teljes játékmenethez.

Az egyes tantárgyakra várakozókat egy Dictionary objektumban tárolom, amely a választott tantárgy nevéhez egy listát tartalmaz, amelyben a várakozó felhasználók

találhatók. Így biztosítva van, hogy az adott érdeklődési körű játékosok kerüljenek be egy játékba. Akkor kerülnek be a listába, amikor megnyomták a Belépés gombot, de a Mégsem gombbal még visszavonhatják tagságukat, amíg be nem kerülnek a játékba. Amíg várakozik, láthatja az eltelt időt és a többi, az adott kategóriában jelentkező játékos számát. A várakozási idő a felhasználó Session memóriájában tárolódik.

A rendszer folyamatosan figyeli a várakozó játékosok számát, és amennyiben elegendően várakoznak, akkor kis idő elteltével véletlenszerűen kiválaszt három játékost. A legutóbbi játék száma mindig tárolódik az Application memóriában, így a rendszer létre tudja hozni az újabb bejegyzést a memóriában, amelyben elhelyez egy új GameLogic objektumot, melynek konstruktorában átadja a kiválasztott játékosokat és a tantárgyat. Ezt követően eltávolítja a kiválasztott játékosokat a tárgyhoz tartozó várakozási listából.

Amennyiben nem áll rendelkezésre elegendő számú játékos, és a játékos várakozási ideje meghaladja a 20 másodpercet, akkor hogy ne várassuk tovább, a hiányzó játékosok helyett robotokat generálunk. A RobotPlayer objektumokat átadhatjuk a GameLogic osztály konstruktorának, mivel a Player osztály leszármazottjai, és ezen túl normál játékosokként lesznek kezelve, csak futási időben fogja a futtató környezet a típusnak megfelelő virtuális metódusokat meghívni, így egyszerűen megvalósítható az eltérő viselkedés. Ezután a várakozó egy vagy két játékos törlődik a listából.

Nem elég, hogy a felhasználók logikailag bekerültek egy játék objektumba, de azt is el kell érni, hogy a böngészőjük frissüljön, és átirányításra kerüljenek a Game oldalra. Mindezek mellett ugyanazokat az információkat kell látniuk azt is figyelembe véve, hogy különböző böngészőjük és eltérő hálózati hozzáférésük lehet.

A Játék UpdatePanel frissítésekor a rendszer ellenőrzi, hogy a még nem elindult GameLogic objektumok közül valamelyik tartalmazza-e a Membership-től lekérdezett felhasználói nevet, tehát hogy az adott játékos, akinél meg van nyitva az oldal tagja-e éppen induló játéknak. Amennyiben igen, a Session memóriájába lementjük a játék sorszámát, így ezen keresztül tudja majd lekérdezni a rá vonatkozó játék információkat. A várakozási idő törlődik, hogy ne maradjon feleslegesen a Session memóriában, majd pedig a böngésző át lesz irányítva a Game oldalra.

Ezzel a módszerrel megoldható, hogy a rendszer azonosítsa az egy játékhoz tartozó felhasználókat, és csak az ő böngészőjüket irányítsa át a számukra közös játéktérre.

Ha a játékos kilépett a játékból, akkor a játék végéig még van lehetősége visszalépni a Vissza gombbal. A Session memóriában tárolom, hogy kilépett-e. A gomb megnyomása után az eddigi játék állapotát törlöm, hogy újra be tudjon kapcsolódni a játékba, majd pedig átirányítom a Game oldalra. Ha van a Session memóriában játék eltárolva, és ez a játék még létezik az Application memóriában, akkor visszaléphet, de ha már a játék befejeződött, akkor töröljük a felhasználó Session memóriájából is, és ezután már csatlakozhat új játékhoz.

A várakozási időt megjelenítő számlálókhoz és az állapotok figyeléséhez egy timerre van szükség, amely frissíti a belépéshez szükséges kontrolokat tartalmazó UpdatePanelt. Az UpdatePanel betöltődésekor ellenőrződik, hogy a játékos milyen állapotban van, tehát éppen várólistán van, vagy visszaléphet még egy zajló játékba.

6.4.3. A területfelosztás játékmenete

Miután sikeresen létrejött a GameLogic objektum, és a játékosok böngészői átirányítottak a Game oldalra, akkor megtörténik a játék oldal betöltése. Ha az oldalt megnyitni kívánó felhasználó nem rendelkezik a Session-jében az adott játék azonosítójával, akkor visszairányítjuk a főoldalra.

Az oldal öt UpdatePanel-ből áll, hogy mindig csak a megfelelő terület frissüljön, ne kelljen minden lekérésnél az egész oldalt újratölteni. Az UpdateArea a gombokat tartalmazza, amelyek a területeket jelképezik. A gombok elnevezése megfelel a GameLogic objektumban definiált kétdimenziós tömbnek, hogy könnyen lehessen indexelni a megfelelő területet. A gombok színe jelzi a terület tulajdonságát, és a szöveg pedig hogy hány pontot ér.

Az UpdatePanelAttackLabel tartalmazza a játékosok támadási sorrendjét a Területfoglalási szakasz során. Az aktuális támadó a játékos eredeti színével jelenik meg, a többi szürkével. Egyszerre csak egy forduló sorrendje, vagyis három játékos jelenik meg. Az a terület minden feleletválasztós kérdés után frissül.

Az UpdatePlayersScore terület a játékosok nevét tartalmazza a színükkel, valamint a játékok során elért összes, és az aktuális pontszámmal. Az aktuális pontszám minden kérdést követően frissül.

Az UpdateQuestionPanel tartalmazza a tippelős vagy feleletválasztós kérdéseket tartalmazó div-eket, valamint a kérdéseket követően megjelenő div-et, amely a helyes választ, a játékosok helyezését, válaszát, és válaszidejét jeleníti meg. A tippelős kérdéseknél megjelenik a kérdés szövege, egy textbox, amelybe a választ lehet írni. A beírt érték validálva van, hogy csak pozitív egész számokat lehessen bevinni, eltérő adat esetén az oldal nem küldi el a szervernek. Miután a játékos bevitte az adatot és enter nyomot, vagy a küldés gombra kattintott a mező értéke már nem módosítható. A feleletválasztós kérdéseknél a kérdéssel négy gomb jelenik meg, melyek a válaszokat tartalmazzák.

Az UpdateElapsedTime a kérdés megjelenésétől számított eltelt időt vagy a területválasztásra kapott időt jeleníti meg, melyből a játékos tájékozódhat, hogy mennyi ideje van még. Ez a számláló mindig nullázódik, amikor az új kérdés megjelenik.

Az ASP.NET weboldalak esetében felmerül a http kommunikáció okozta azon probléma, hogy a szerver csak a kliens által küldött kérésekre tud válaszolni, nem tudja szándékosan frissíteni. Erre a problémára azt a megoldást találtam, hogy egy timer helyezek el a weboldalon, amely aszinkron kéréseket küld a kiszolgáló felé. Az oldalon a tick eseményre vizsgálom a játék aktuális állapotát a szerveren, és ennek megfelelően frissítem az UpdatePaneleket. Az egyes szakaszokhoz tartozó állapot vizsgálatok a lépéseknek megfelelő metódusokban vannak elhelyezve.

A belépéskor a GameLogic objektumban is elindul a timer, amely vár egy kicsit, amíg a felhasználóknak betöltődik az oldal, ezt egy változó beállításával vizsgálja a rendszer, mikor az oldal megjelenik. A területfelosztás szakasz során a tipp kérdéseknél mindig ugyanaz a folyamat zajlik le. A játékos Session memóriájában tároljuk, hogy a nála megjelenített oldalon a játék milyen státuszban van. Ez az állapot összhangban van a GameLogic objektumban beállított állapottal.

Az oldal betöltődésekor az UpdateArea-ban beállítódnak a gombok a GameLogic-ban tárolt objektumok alapján. A SetButtons metódus mindig betölti a gombok színét és szövegét az aktuális állapot szerint a panel frissítésekor.

Első lépésben betöltődik az Application memóriából egy véletlenszerű tipp kérdés. Ezt követően a szerver tovább lépteti az állapotot a kérdés megjelenítésére. Ekkor megjelenik az UpdateQuestionPanel a betöltött tipp kérdéssel, és az eltelt időt megjelenítő, folyamatosan frissülő UpdateElapsedTime panellel. Amikor a játékos elküldi a választ, akkor megáll a stopper, elmentődik a beírt érték, és a Pushed állapota igazra állítódik. A beviteli kontrolokat letiltom, hogy ne vihessen be más értékeket.

Ha az összes játékos beküldte a választát, akkor nem kell végig várni, amíg letelik az idő, mert a timer frissülésekor a rendszer megvizsgálja a Pushed értéket, és ha mind igaz, akkor egyből továbblép a rangsor kiszámítására és megjelenítésére. Ekkor a kérdés eltűnik és megjelenik a rangsor, miközben az első két helyezett területet választhat. Egy felirat jelzi, hogy éppen ki van soron.

A terület gombjai csak az éppen választó játékos számára engedélyezettek. A játékosok a helyezés alapján kapnak választási lehetőséget. A győztes két, míg a második helyezett egy lehetőséget kap. Addig választhatnak, amíg ezek el nem fogynak. A megnyomott gomb ID-ja alapján megkapjuk a GameLogic-ban tárolt gomb tömbben a kiválasztott indexét, majd ennek beállítjuk a játékos színének megfelelő háttérszínt tartalmazó CSS class-t, a játékos megkapja a gomb értékét, és csökkentjük a választási lehetőségeit.

Amennyiben a játékos nem választ időben, vagy nem használja fel a területeit, akkor a GameLogic osztályban lefut a véletlenszerű választási rutin, amely annyi szabad területet sorsol ki, és jelöli ki a játékosnak, amennyi megmaradt. Ugyanez az eljárás a második helyezett esetében is. Mindeközben a GameLogic-ban meghatározott játék állapot lépteti a Session memóriában tárolt állapotot, amely az egyes részek megjelenítését aktiválja.

Miután a nyert területek felosztásra kerültek, a TipStep folyamat a végéhez ér, ezzel egy tippelős kérdés szakasz lezárul. A kontrolok visszaállítódnak alaphelyzetbe az új kérdéshez. Ez a ciklus zajlik le addig, amíg a meghatározott számú kérdést el nem érjük. Az utolsó tipp kérdésnél már csak egy terület marad, amelyet a nyertes kap meg, ezt a rendszer könnyen kezeli, mivel miután a választási lehetőségek elfogytak a rendszer átlépi a második játékos választási szakaszát. Ezen szakasz végén átlépünk a Területfoglalási játékrészbe.

6.4.4. A területfoglalás játékmenete

A GameLogic felveszi a ChooseStep enum SelectArea kezdőértékét, majd amíg a választás zajlik, betöltődik a kérdés. A támadási listában következő játékos számára engedélyezettek csak a terület gombok, és csak olyat választhat, amely nem az övé, a sajátjai letiltódnak. Miután megjelölte a megszerezni kívánt gombot megjelenik a kérdés a négy választási lehetőséggel. A gombok ID-ja magában hordozza a válasz sorszámát, így bármely gomb lenyomásakor az eseménykezelő metódus ki tudja olvasni a küldő értékét, majd ezt továbbküldi a szervernek, és letárolódik a játékos válaszaként. Ha nem választ, akkor elvesztette a fordulót.

Megjelenik a rangsor, de csak a szemben álló játékosok szerepelnek benne. Ha döntetlen az állás, akkor létrejön egy TipQuestion objektum, ami kiválaszt egy véletlenszerű tipp kérdést. Ennek eredménye eldönti a küzdelem kimenetelét, és eldől a terület sorsa. A tárolt GameButton objektum értékeit beállítjuk a nyertes színe és az elfoglalt terület értéke alapján. Majd a felületen lévő gombot ezek szerint változtatjuk.

A kontrolok ezután beállítódnak a következő kérdéshez, és a támadás jogát a sorrendben következő játékos kapja meg. A bázis támadására három lehetőség van, amennyiben sikeres, minden fordulónál veszít a terület az értékéből mindaddig, amíg el nem éri a nullát. Ha ez megtörténik, akkor a támadott játékos összes területe a támadóhoz kerül és a pontszámát is a támadó kapja meg. Ezzel a játékos kiesett, a kontrolok tiltottak lesznek számára, és a kérdésekből is kimarad, amikor sorra kerülne, akkor a lista automatikusan a következő játékosra ugrik. de a játék végezetéig még megtekintőként bent maradhat.

6.4.5. A játék vége

Amikor az utolsó támadás is lezajlott a játék véget ér. A Player metódusai kiszámítják a játékos végeredményét, lementik az adatbázisba, majd ez alapján a GameLogic rangsort állít fel, és ezt az eredményt megjeleníti a játékosok számára. Ekkor a timer leáll a GameLogic objektumban és az oldalon is. A kilépés gombra kattintva a játékos Session memóriájából törlődik a játék azonosítója, így már nem tud többször visszalépni, és az InGame státusza is false-ra állítódik.

Amikor már az összes játékos kilépett, az Application memóriából törlődik a játék objektum. A kilépés gombot a játék vége előtt is használhatják a játékosok, ilyenkor megőrződik a Session-ben a játék azonosítója, és kap még egy változót, amely azt jelzi, hogy kilépett, de még nem fejezte be a játékot. A játék elhagyása 100 pont levonással jár, valamint a későbbi megválaszolatlan kérdések miatt veszíthet az összpontszámából, amelyet a játék végeztével lementünk az adatbázisba. A főoldalról a visszalépés gombbal, amíg van aktív játékos a játékban visszaléphet. Amennyiben nem maradt játékos, az utolsó felhasználó kilépésekor a játék adatai mentődnek és a játék objektum törlődik az Application memóriából, és a kilépett játékosok Session memóriájából is törlődik a játék azonosítója, így már beléphetnek új játékba.

7. IMPLEMENTÁCIÓ

7.1. Programnyelv kiválasztása

7.1.1. Visual Basic .NET és C#

Az ASP.NET keretrendszerben számos .NET-et támogató programnyelvvvel lehet fejleszteni, de a legismertebb a Visual Basic és a C#. Mindkét nyelv a Microsoft fejlesztése, fordításkor ugyanarra a köztes nyelvre fordulnak, és .NET Framework futtatja őket. Legfőképp szintaktikai eltérések vannak, de a főbb funkcionalitásoknak mindkét nyelvben megtalálhatók a megfelelői. Az összes .NET nyelvnek azonos a futtató környezete, a köztes nyelve, az operációs rendszerhez való hozzáférhetősége, a szemétyűjtése, kivételkezelése és hibakövetése. A vizsgálathoz szükséges információkat a [7] és [8] forrásokból merítettem.

7.1.2. Visual Basic .NET

Előnye, hogy a számos nyelvi elem megkönnyíti a fejlesztés bizonyos részeit. Például a függvények törzsét automatikusan beilleszti, egyszerűen eldönthető egy stringről, hogy számmá konvertálható-e, egyszerű dátum deklaráció. A programozó által létrehozott osztályokat a My névtérben helyezi el, így nem lehet őket összekeverni a rendszer osztályaival. A COM komponensek és a más nyelvek osztályainak felhasználása jobban kezelhető a futásidejű kötés miatt, mint például a C#-ban. A névterek projekt szinten deklaráálódnak, így nem kell minden fájlban megadni, mint a C#-ban. A feltételes kivételkezelés is könnyen megvalósítható a „When expression” blokkokkal.

A változók könnyen konvertálhatók, ha egy stringet értékül adunk egy integer változónak, ha konvertálható, akkor integerré alakítja anélkül, hogy függvényeket kellene meghívunk a konverzióhoz. Logikai értéket is megadhatunk integer változónak, amely 1-et vagy -1-et ad vissza. A számokat szöveggé alakítja. Két változó összehasonlításakor a fordító konvertálja, majd összehasonlítja az adott értékeket egy egyszerű „=” jel használatával anélkül, hogy függvényekkel kellene konvertálnunk és összehasonlítani az értékeket. A tulajdonságok metódusai kaphatnak paramétereket is. A Visual Basic .NET nem case sensitive, tehát nem kell ügyelni a kis és nagy betűk eltérésére.

7.1.3. C#

Lehetőség van nem biztonságos kód blokkok elhelyezésére, mint például a C++, amit nem tud ellenőrizni a fordító, de szükség van rájuk. Létre lehet hozni osztályokat és interfészeket logikailag több részben, eltérő helyeken, melyeket majd a fordító egy fájlként állít össze. Statikus osztályok készíthetők, amelyek csak statikus tagokat tartalmazhatnak. A számokkal történő műveletek nem kerülnek automatikus ellenőrzés

alá, így gyorsabb lesz a kód, de a túlcsordulások megelőzésére a programozó felügyelet alá helyezheti az adott kódrészt, így elkaphatók és helyben kezelhetők a számítási problémák.

A C# case sensitive nyelv, ez az elnevezések értelmezését megkönnyíti, mivel egy osztály neve kezdődhet nagy, a példánya és a változók kis betűvel, így első ránézésre tudhatjuk a változó típusát is. A C#-ban a sor végét pontosvessző jelöli, így könnyen formázható, mindig tudni lehet, hogy melyik sor hova tartozik és hol ér véget. A VB.NET-ben bizonyos esetekben nehézkes formázni a sorokat úgy, hogy logikailag átláthatóak legyenek. A kapcsos zárójelek is a könnyebb áttekinthetőséget segítik.

A C#-ban könnyebb a hibákat felfedezni, például ha egy változó nem kapott kezdőértéket és hivatkozunk rá, vagy ha egy kódrészlet sosem fog lefutni a feltételek miatt. A kommentezés is fejlettebb, több soros, XML kommenteket is lehet készíteni, a summary-k segítségével a használt osztályhoz, vagy metódushoz írt megjegyzéseket láthatjuk a tooltip ablakban, így könnyen megtudhatjuk, hogy mire is való pontosan. A régiók kezelése is rugalmasabb, akár egy metódus is felosztható régiókra, ezáltal logikailag jobban elkülönül.

7.1.4. Összehasonlítás

A C# és a VB.NET főként szintaktikailag térnek el egymástól, mindkét programnyelvben megoldhatóak ugyanazok a feladatok, ha ismerjük a keretrendszert. Valójában megszokás kérdése, hogy ki melyik programozási nyelvet szereti, melyik áttekinthetőbb számára. Az előző vizsgálatok alapján arra az elhatározásra jutottam, hogy a C# nyelv lesz megfelelő a fejlesztéshez, mivel számomra áttekinthetőbb a kód felépítése a zárójeleknek és a pontosvesszőknek köszönhetően, valamint fontosnak tartom a kommentezés támogatását. A kis és nagybetűk megkülönböztetése számomra megkönnyíti a változók felismerését és szerepének kikövetkeztetését.

7.2. Konvenciók

A .NET keretrendszer egyaránt támogatja a case-sensitive és a nem case-sensitive nyelveket, a köztes nyelvre fordítást követően azonos módon kezeli őket, viszont a kis és nagy betűk megkülönböztetése a programozók számára megkönnyítheti a kód megértését. Az egységesítés miatt és a könnyebb átláthatóság érdekében különböző konvenciók jöttek létre a változók, metódusok, osztályok elnevezésére. A rendszer fejlesztése során ezeket a szabályokat alkalmaztam, hogy a programkódom mások számára is könnyen érthető legyen. A rendszerben egységesen mindenhol angolul neveztem el a változókat. A továbbiakban bemutatom a kis és nagy betűk használatának helyeit és módszereit.

7.2.1. *Pascal casing*

Az azonosítók megkülönböztetésére használt egyik módszer a Pascal casing. Azt a formát nevezzük így, ahol az első betű és a további szavak első betűje a kifejezésben nagy betűs. Ezt a módszert használtam a publikus tagok, névterek, típusok, enumok, továbbá a metódusok, események és interfészek elnevezésére.

7.2.2. *Camel casing*

A másik módszer a Camel casing, melyben a szavak első betűjét kis betűvel írjuk, a többi tag kezdőbetűjét pedig naggyal. Ez jó módszer az osztályok és a példány szintű tagok megkülönböztetésére, mivel az osztályt írhatjuk Pascal case-szel, a belőle létrehozott példányt pedig Camel case-szel, így első ránézésre tudhatjuk, hogy a változó mely osztályból lett példányosítva. A metódusok paramétereit is ezzel a módszerrel neveztem el.

7.2.3 *Kommentek*

A kommenteket, mivel a programot áttekintők, illetve továbbfejlesztők magyar anyanyelvűek lesznek, ezért a könnyebb érthetőség kedvéért magyarul írtam. A publikus láthatósági szintű osztályokat és metódusokat summary-kkal láttam el, így az osztálykönyvtárakat, osztályokat használók a fejlesztőkörnyezet által megjelenített XML kommentekből tájékozódhatnak az adott elem pontos működéséről. Mindezek mellett a programkód nagy részét kommentekkel láttam el, hogy az esetleges jövőbeni fejlesztők is könnyedén megértsék a működést és továbbfejleszthessék.

7.3. Fejlesztési döntések

7.3.1. *Felhasználók kezelése*

Amint az adatbázis megtervezésénél írtam, a felhasználók kezelésénél meg kellett hoznom azt a döntést, hogy magam készítem el a bejelentkezéshez szükséges felületet, a mögöttes ellenőrzési és tárolási logikákat, vagy már meglévő komponenseket használok. Ebben a kérdésben arra a döntésre jutottam, hogy biztonságosabb, és kézenfekvőbb az ASP.NET által nyújtott Membership és Role providereket használni, melyek már stabil és alaposan tesztelt háttérrel nyújtanak a felhasználók kezelésére.

Minden igényemet viszont nem tudták kielégíteni, mert a játékkal kapcsolatban több plusz információra is szükségem volt, ezért alaposabban tanulmányoztam ezeket a komponenseket, és kiegészítettem az adatbázist is a számomra szükséges mezőkkel. A regisztrációs felületen és a mentési logikában bevezettem a beiratkozási év tárolását, hogy ez alapján következtetni tudjunk arra, hogy a játékos mennyi ideje az egyetemünk tagja, ezáltal mennyire adhat könnyen választ a kérdésekre.

7.3.2. Megjelenítés

A Visual Studio 2010 által nyújtott gyári beállítású megjelenítés a játék számára igen szegényesnek bizonyult, ezért jelentősen átalakítottam. A bejelentkezéshez kapcsolódó oldalakon az adatbevitelt jobban áttekinthetővé tettem, a fejléct is átalakítottam, és az oldalakon történő navigáláshoz menüt alakítottam ki. Az adminisztrátori oldalak és az oda navigáló linkek csak az adminisztrátori szerepkörrel rendelkező felhasználók számára elérhetőek.

Az új kérdések létrehozására szolgáló oldalon egyszerű formákat hoztam létre a szükséges validálásokkal és hibaüzenetekkel, hogy a kérdések a megfelelő formátumban kerüljenek a rendszerbe.

Az elért eredményeket és az adminisztrátor számára megjelenített kérdés, játékos, üzenet és hiba adatokat lapozható GridView kontrollokban jelenítem meg, melyek a Linq To Sql adatkapcsolathoz vannak kötve, ezáltal az adatbázisból mindig csak annyi adatot hozunk fel, amennyit egy oldalon megjelenítünk, így nem terhelődik feleslegesen a rendszer és nem akadozik a megjelenítés.

Az oldal többi részétől elkülönülten frissülő tartalmakat, mint például a chat, a játéktér, az eredményeket tartalmazó gridek UpdatePanelek-be helyeztem, ezáltal az oldal több részétől elkülönülten, kisebb adatforgalommal frissülnek. Az UpdatePanel a ScriptManager segítségével az oldalnak csak azt a részét rendereli, generálja újra, amely valóban változott. Az UpdatePanel frissülését különböző eseményekhez kötöttem a használatától függően. Az eredmények kiválasztásánál a Combobox-ból történő választás váltotta ki a frissítést, a chat esetében az üzenet elküldése, a játékosok belépésekor a timer, a játékmenet közben pedig a játék szakaszainak állapotától függően az oldal hívta meg az egyes területek frissítését.

A játéktérületen a játéktér megjelenítését nem volt egyszerű megoldani, mivel a területeket szimbolizáló gombokat a NIK épülete felett szerettem volna elhelyezni. Asztali alkalmazás esetén ez egyszerű lett volna, mivel csak átlátszóvá kellett volna tennem a gombokat. Viszont az ASP-s kontrolloknak nincs ilyen tulajdonsága. Hosszú keresések után sikerült megtalálnom a CSS 3-ban bevezetett opacity tulajdonságot. Ezzel százalékosan lehet állítani az átlátszóság mértékét. A CSS 3 még nem hivatalos szabvány, a Visual Studio 2010 sem ismeri, viszont figyelmeztetéssel, de lefordul. A modern, népszerű böngészők mind ismerik ezt a tulajdonságot, ennek köszönhetően sikerült elérnem a tervezett megjelenést.

7.3.3. Újratervezés

Mivel a szakdolgozatom megírása mellett célul tűztam ki az ASP.NET keretrendszer használatának megtanulását, ezért beleütköztem számos, az új technológiák megismerésekor felmerülő problémába. Ezek miatt néhány dolgot másképp kellett megvalósítanom, mint ahogy terveztem. Nem használhattam azokat a módszereket, mint az asztali alkalmazások fejlesztésekor. Erre példa a fent említett

átlátszóság kezelése is. A html-re forduló kontrollok tulajdonságai és eseményei erősen korlátozottak, nehezebb elérni ugyanazt a működést és megjelenítést. A CSS, javascript és jquery technológiák nagy segítségemre voltak a bonyolultabb funkciók megvalósításában.

A változók használata is jelentősen eltér, mivel egy asztali alkalmazásnál a változók értéke korlátlan ideig rendelkezésre áll, viszont egy webes alkalmazásnál az oldalt a szerver minden lekérésnél újragenerálja, ezért ezen változókra nem lehet számítani huzamosabb információátvitel céljából. Az oldalak közti navigáció során is elvesznek a változók értékei. Egy több felhasználós asztali alkalmazásnál könnyedén megjeleníthetjük a felhasználók adatait mások számára, és eseményekkel frissíthetjük a számokra megjelenített információkat.

A rendszerben felhasználók kis csoportjának kell megjeleníteni ugyanazokat az adatokat, és külön kezelni az egyéni információkat. Erre a feladatra az ASP.NET Application és Session memóriájának kombinált használatát találtam alkalmasnak.

A közös GameLogic objektumot az Application memóriában tárolom így a játék folyamán végig elérhetőek a játék adatai, a játékosok pedig a Session memóriájukból kiolvashatják a GameLogic objektum Application memóriabeli címét, és az innen kapott adatok alapján építi fel a szerver a válaszként kapott egyedi Game oldalt. Ez az oldal a közös játéktérületet mutatja az adott játék résztvevőinek, valamint a rájuk vonatkozó egyéni pontszámokat.

7.4. Kompatibilitás

Az ASP oldalak formázásához a CSS szabványait használtam, mellyel a modern böngészőkön hasonló megjelenést értem el. A böngészők sokfélesége miatt a megjelenésben lehetnek apró eltérések, de a működés egyforma. A Google Chrome böngészőhöz optimalizáltam az alkalmazást, viszont a rendszer kompatibilis az Internet Explorer 8, 9-cel, a Mozilla Firefox-szal, az Operával és a Safarival. Mindezen böngészőkön az összes funkció elérhető és a játék végigjátszható, legfeljebb apró megjelenítési eltérések tapasztalhatóak.

8. A RENDSZER TESZTELÉSE

Mivel az ASP.NET fejlesztési környezet teljesen új volt számomra, ezért a fejlesztés folyamán folyamatosan teszteltem a megoldásaimat, és próbáltam a lehető leggyorsabb módszert megtalálni a rendszer elkészítéséhez. A tesztek folyamán többször rájöttem, hogy a választott technika lassú, ezért másik megoldást kerestem, és egyes részeket újra terveztem. Miután a fejlesztés elért egy közel végleges verziót, elkezdtem a rendszer átfogó tesztelését.

A rendszer tesztelésére kétféle módszert alkalmazok. Először a saját gépemen, a fejlesztőkörnyezetben végzek automatikus terheléses teszteket, majd pedig egy szerverre kitelepítve fogom egyszerre több felhasználó segítségével tesztelni. Az ezek folyamán szerzett eredmények és tapasztalatok alapján pedig kijavítom az esetleges hibákat.

8.1. Automatizált tesztek

8.1.1. Teszt környezet

Az automatizált tesztek elkészítéséhez a Visual Studio 2010 beépített tesztelési eszközeit használtam. Ezeket a teszteket a saját gépemen végeztem, mivel így még a fejlesztőkörnyezetben debuggolhatom az alkalmazást, könnyebb megtalálni a hibák pontos helyét a programkódban, mintha már az IIS-re kitelepített webes alkalmazástól kapott válaszok alapján kellene felderíteni. A teszt környezet az alábbi.

Hardver:

- Notebook: Acer Aspire 5920G
- Processzor: Intel Core 2 Duo T7300 (2.0GHz, 800MHz FSB(Intel GTL+, busz szélesség: 64bit, valódi órajel: 250MHz, sávszélesség: 2000MB/mp), 4MB L2 cache(On-Die, ATC, Full-Speed))
- Memória: 2 GB DDR2, 667MHz
- Virtuális memória: 3,4 GB
- Videokártya: NVIDIA GeForce 8600M GS (256MB beépített memória)
- Merevlemez: Western Digital 320GB, 5400rpm, SATA, 8Mb cache

Szoftver:

- Operációs rendszer: Microsoft Windows 7 Ultimate, 32 bit
- Keretrendszer: Microsoft .NET Framework 4.0
- Fejlesztőkörnyezet: Microsoft Visual Studio 2010
- Programnyelv: C#
- Alkalmazás típusa: Web Application

Adatbázis:

- Szerver: MS SQL 2008 R2 adatbázis szerver (Developer Edition)
- Környezet: Egy virtuális gépen fut, amelynek 1 processzora és 1 GB memóriája van és Windows Server 2008 R2 Enterprise Edition operációs rendszer működteti.

8.1.2. A tesztelés célja

Az alkalmazás működéséből eredően magát a játék folyamatot nem lehet automatizált tesztekkel ellenőrizni, mivel a válaszok kiválasztása és a játékmenet végigjátszása emberi gondolkodást igényel. Viszont a játékba lépési folyamatot már lehet ezzel a módszerrel ellenőrizni, és így kipróbálható, hogy nagyszámú játékos esetén a rendszer hogyan tűri a terhelést. Az oldalak válaszüdejéből megtudható, hogy elég hatékonyan működik-e a rendszer.

8.1.3. A tesztelés folyamata

Első lépésben megterveztem, hogy pontosan milyen folyamatot fogok a tesztben megvizsgálni. Egy felhasználót a rendszerbe történő belépést követően beléptetek egy játékba, majd amint belépett kiléptetem a játékból, majd a rendszerből is. Ezzel megtudhatom, hogy a belépés folyamán nagyszámú felhasználó esetén milyen problémák merülhetnek fel.

8.1.4. Webes alkalmazás teljesítménytesztelése

A folyamat rögzítéséhez a Visual Studio Web performance test eszközét használom, amellyel egy webes felületen a működő rendszeren rögzíteni lehet a tesztelés lépéseit. A rögzítés után kipróbáltam, hogy a teszt probléma nélkül lefut-e.

A teszt futtatásakor máris sikerült egy hibát felfedeznem. A viewstate adatok MAC titkosításának nem sikerült a validációja. Ez a hiba nem jött elő, amíg a fejlesztőkörnyezetből futtattam az alkalmazást, viszont a tesztesetek nem tudtak emiatt lefutni. Utánanéztem szakmai fórumokon [\[9\]](#), hogy mi is pontosan ez a probléma, és megtudtam, hogy a viewstate állapot titkosítási kulcsát a böngésző egy rejtett mezőben küldi vissza, és ha a postback hamar történne meg, minthogy ez a mező betöltődne, akkor a szerver nem kapja meg a titkosítás feloldásához a kulcsot, ezért hibát dob. Ezt a problémát a web.config fájlban az oldalak viewstate titkosításának kikapcsolásával megoldottam. A rendszer a játék folyamán nem küld felhasználói adatokat, ezért nincs szükség ezen információk biztonságos küldésére. Ezt a javítást követően már sikeresen lefutott a teszt.

8.1.5. Terheléses teszt

Miután elkészült a tesztelendő folyamat lépéseinek sorozata, a Visual Studio Load test eszközével terheléses tesztet készíték. Meg lehet adni, hogy hány virtuális felhasználó használja a rendszert, és hogy mennyi időn keresztül, milyen várakozási idővel. A belépés teszteléséhez 100 virtuális felhasználót hozok létre, amelyek 10 percen keresztül fogják az előzőleg létrehozott Web performance testet futtatni. Ezek a tesztek igen jól paraméterezhetők, meg lehet adni, hogy milyen hálózatot használjanak, és milyen böngészőn futtassák a teszteket, én az Internet Explorer 8-at választottam.

A teszt első futtatásakor több ezer hibaüzenetet kaptam. Ennek az volt az oka, hogy a Web performance tesztben az először beírt felhasználónév és jelszó került tárolásra, és most mind a 100 virtuális felhasználó ugyanezzel próbált belépni, amit a rendszer nem enged, ezért a teszt nem tudott tovább lépni a főoldalra. Szerencsére a Web performance tesztek lehetővé teszik, hogy az olyan helyeken, ahol a bevitt adatokban eltérések lehetnek, dinamikusan adjuk meg az adatokat. A belépéskor megadott adatokhoz létrehoztam egy adatbázist, amely a regisztrált felhasználói neveket és jelszavakat tartalmazza. Azért nem az eredeti adatbázist használtam, mert ott a jelszavak titkosítva vannak, ezeket a felhasználói információkat csak a teszt céljára hoztam létre.

A következő futtatáskor még mindig ezer feletti volt a hibák száma, mivel a teszt rögzítésekor a belépéskor történő várakozást nem vettem figyelembe, ezért nem találta a teszt a várt oldalt. Ezután a tesztlépéseket újragondoltam, és figyelembe vettem a várakozási időket is.

Ezt követően már hibák nélkül lefutott a teszt, melyet többször is megismételtem, majd a tesztek folyamán átlagosan a 8.1. táblázaton látható eredményeket kaptam.

Max User Load	100
Tests/Sec	8,71
Tests Failed	0
Avg. Test Time (sec)	11,4
Transactions/Sec	0
Avg. Transaction Time (sec)	0
Pages/Sec	61,3
Avg. Page Time (sec)	0,075
Requests/Sec	133
Requests Failed	0
Requests Cached Percentage	67,7
Avg. Response Time (sec)	0,043
Avg. Content Length (bytes)	5 085

Összesített eredmények

Oldalak eredményei

URL	Scenario	Test	Avg. Page Time (sec)	Count
http://localhost:3160/	Scenario1	WebTest1	0,13	5 325
http://localhost:3160/Game.aspx {GET}	Scenario1	WebTest1	0,098	5 237
http://localhost:3160/Main.aspx {GET}	Scenario1	WebTest1	0,078	5 250
http://localhost:3160/Account/Login.aspx {POST}	Scenario1	WebTest1	0,074	5 251
http://localhost:3160/Main.aspx {POST}	Scenario1	WebTest1	0,058	10 471
http://localhost:3160/Game.aspx {POST}	Scenario1	WebTest1	0,022	5 227

8.1. táblázat A terheléssel teszt eredménye

A táblázat alapján jól látható, hogy egyik oldalnál sincsen eltérően nagy válaszidő, és hogy a rendszer folyamatos terhelés mellett is helyt állt. A 8.2. ábrán a tesztelés folyamán készült grafikonok láthatóak, amelyek különböző szempontok szerint figyelik a rendszer állapotát.



8.2. ábra A terheléses teszt grafikonjai

Az első grafikonon egy összesítő ábra látható, a felső csík mutatja a felhasználók számát, amely a teszt folyamán folyamatosan 100. A középső vonal a másodpercenként megnyitott oldalak számát mutatja, amely 61 körüli, ez az érték a kezdetekben eléggé ingadozó, de később kissé stabilizálódik. Az oldalak válaszáideje az első betöltéskor még megugrott, majd 0,1 alatti értékre beállt. A tesztek válaszáideje 11 másodperc körüli volt, mivel 10 másodperces várakozási időket állítottam be a tesztek között. Az utolsó grafikonon a processzor és a memória terheltsége látható.

A memória kihasználtsága már a tesztek előtt is 65% körüli volt, a tesztek kezdetekor felment 75%-ra, majd lecsökkent 70%-ra. Ez az eredmény igazolja, hogy a DataSet használata a kérdések betöltéséhez nem növelte meg nagymértékben a rendszer memória igényét, figyelembe véve, hogy a fejlesztőkörnyezettel és a teszt környezettel együtt is csak maximum 10%-al nőtt meg a memóriahasználat.

A processzor használata az ASP.NET development szerver és a tesztek indításakor igencsak megugrott, háromszor is túllépte a 75%-os határt, majd 60%-on stabilizálódott. Ez köszönhető annak, hogy a fejlesztőkörnyezet, egyéb alkalmazások, például SQL szerver, és a terheléses tesztek egy gépen történő futtatása megterhelő a

hardver számára. Egy kétmagos processzortól 100 virtuális felhasználó menedzselése, plusz az ehhez szükséges egyéb rendszerek futtatása melletti átlagosan 60%-os kihasználtság elég szép eredmény. Egy weboldalak hosztolására kialakított szerver várhatóan könnyebben megbirkózik majd a feladattal.

8.1.6. Automatizált tesztek kiértékelése

A terheléses tesztek segítségével sikerült több hibára is fényt deríteni, majd ezek javítását követően a rendszer bizonyította, hogy szerény hardverkörülmények mellett, viszonylag nagyszámú felhasználó mellett is helyt áll. A tesztelt oldalaknak elég gyors a válaszüzeje, és hiba nélkül teljesítenek 100 virtuális felhasználó folyamatos terhelése mellett 10 percig. Ezek alapján a rendszer készen áll a szerverre telepítésre, és a valódi felhasználók által történő tesztelésre.

8.2. Felhasználói tesztek

8.2.1. Teszt környezet

A felhasználói tesztek elkészítéséhez a munkahelyem jóvoltából a céges szerveren tudtam hosztolni a webes alkalmazást. Ennek köszönhetően a tesztet végző felhasználók egy link segítségével el tudják érni a rendszert. Az adatbázist az egyetemi szerverről áttelepítettem a cégesre, hogy az IIS-szel egy szerveren legyen, ezáltal az adatforgalom gyorsabb lesz, kevesebb hálózati kockázat veszélyezteti a rendszer válaszüzeit. Az alkalmazást futtató operációs rendszer egy virtuális gépen fut. A teszt környezet az alábbi.

Hardver:

- Processzor: AMD Athlon II X3 440 Processor 3.01GHz
- Memória: 4 GB DDR3
- Virtuális memória: 4 GB

Szoftver:

- Operációs rendszer: Windows Server 2008 R2 Enterprise SP1, 64 bit
- Keretrendszer: Microsoft .NET Framework 4.0
- IIS 7.5

Adatbázis:

- Szerver: MS SQL 2008 R2 adatbázis szerver (Developer Edition)

8.2.2. A tesztelés célja

A rendszer terheléses tesztelésével sikerült fényt deríteni a belépéskor fellépő problémákra, valamint sikerült lemérni a rendszer terhelhetőségét. Ezek után az automatikusan nem tesztelhető részeket, a játéklógika és az egyéb funkciók működését fogom felhasználók bevonásával megvizsgálni. A felhasználói tapasztalatok segítségével remélhetőleg felderíthetők olyan funkcionális hibák, melyek a fejlesztés során nem jöttek elő.

8.2.3. A tesztelés folyamata

A funkcionális tesztek során 9 ismerősömet fogom megkérni, hogy lépjenek be a rendszerbe, majd egy időben indítsák el a játékot, majd játsszák végig. Ennek folyamán ki fog derülni, hogy milyen hatással van az alkalmazásra, ha egyszerre több felhasználó használja, különböző böngészőkkel, különböző platformokon. A játékmenet tesztelése mellett meg fogom kérni őket, hogy nézzék végig a ranglistákat, és töltsenek fel kérdéseket, hogy a rendszer ezen részei is tesztelve legyenek. A teszt alatt fel fogják jegyezni, ha hibákat találnak, vagy ha nem logikusan működnek az egyes funkciók.

8.2.4. Felhasználói tesztek

A felhasználói tesztek során kilenc ismerősöm használta egyidejűleg a játékot. Ennek folyamán fény derült néhány apróbb hibára. A feleletválasztós kérdéseknél a gombokon nem mindig látszottak a válaszlehetőségek, mert azok túl hosszúak voltak. Ennek korrigálására megnöveltem a gombok méretét, de mivel ez a lehetőség korlátos, ezért lecsökkentettem a feltölthető feleletválasztós válaszok hosszát.

Kiderült egy másik probléma is, amely komplexebb volt, és amikor egy felhasználóval és robot játékosokkal teszteltem, akkor nem jött elő. Amikor területfelosztáskor a győztes játékos gyorsan egymás után két területre kattint, akkor a rendszer néha csak egynek érzékeli, ezután egy harmadik területet is elfoglalhatott. Emiatt a végén hamarabb elfogytak a területek, így az utolsó kérdésnél már nem lehetett területet választani.

Ennek a problémának a megoldásához át kellett néznem a területfelosztás folyamatát, és rájöttem, hogy gyors dupla kattintáskor az egymás utáni hirtelen post backek miatt a rendszer csak egy elfoglalható területet von le a játékosról, de a terület színét átállítja. Ennek megoldására átalakítottam a területszám csökkentését a gombnyomáskor, így biztosan levonja a kijelölt területet mielőtt a játékoshoz rendelné.

A felhasználók jelezték, hogy kevés az idő a válaszadásra, hosszú kérdéseknél túl hamar letelik az idő, és már nincs idejük beírni a választ. Ennek kiküszöbölésére átállítottam a válaszadás időhatárát szabályozó konstans változót, így már elegendő idő áll rendelkezésre.

Amikor kevesen várakoztak egyszerre, és robot játékos ellen kellett játszaniuk, akkor a robot játékos a területfelosztáskor a rendelkezésre álló teljes időt felhasználta, hogy kiválassza a területet. Ez rontotta a játékélményt, mivel sokszor arra kellett várniuk, hogy a robot válasszon. Mivel ez a lépés automatikus, ezért megoldottam, hogy ne várja végig a teljes időt, hanem már három másodperc várakozás után kisorsolja a területeket számára a rendszer, így a játék rugalmasan mehet tovább.

8.2.5. Hibák felderítése

A felhasználói tesztek során megnéztem az adatbázisban összegyűjtött hibákat, melyeket a használat során naplózott a rendszer. Az adminisztrátorral a felületen keresztül is könnyen megtekinthető a hibák helye és részletes adatai. Ez a funkció igencsak hasznosnak bizonyult, mivel így ki tudtam nyomozni, hogy a felhasználók által jelentett hibáknak mi lehetett a pontos oka.

Észrevettem, hogy rengeteg bejegyzés van, ami a favicon.ico fájlt hiányolja. Minden oldal megnyitásakor hibát dobott az alkalmazás, mivel nem találta ezt a fájlt, amely a böngészők fejlécében megjelenítendő ikon a weboldalhoz. Ezért a NIK logóját elhelyeztem ezzel a névvel a megfelelő névvel, és a MasterPage-en beállítottam az ikon helyét megadó tag-et. Ennek köszönhetően az alkalmazás bármely oldalának megnyitásakor a böngésző lapjának fejlécében a NIK-es logó lesz látható.

8.2.6. Felhasználói tesztek értékelése

A felhasználókkal történő tesztelés során számos problémára fény derült, melyek a fejlesztés során nem tűntek fel számomra, és a robot játékosok elleni játékok során sem jöttek elő. A felmerült hibákat javítottam, és a felhasználók javaslatai alapján igyekeztem az alkalmazást ergonomikusabbá, a használók igényeinek megfelelővé tenni.

A módosítások után többször újra kipróbáltattam a rendszert az ismerőseimmel, így sikerült könnyebben használhatóvá tenni a rendszert, de a tesztelések és fejlesztések egy modern webes alkalmazás fejlesztésénél sosem érnek véget.

9. AZ ALKALMAZÁS ELEMZÉSE, TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

9.1. A rendszer elemzése, alkalmazása

A tesztek végeztével megállapítható, hogy a rendszer elég stabil, gyors és megbízható, valamint a megjelenítés meglehetősen egyszerű és hatékony, ezáltal alkalmas az informatikai ismeretek szórakoztató formában történő oktatására. A jövőben a mérnök informatikus hallgatók használhatják a zárthelyikre és vizsgákra történő felkészülés során, ezáltal könnyebben elmélyíthetik és bővíthetik meglévő tudásukat, felfedezhetik hiányosságaikat.

A rendszer felépítésének köszönhetően a felhasználók maguk is fejleszthetik az alkalmazást a kérdések és tantárgyak hozzáadási lehetőségének köszönhetően, ezáltal a tantárgyak köre, és a kérdések mennyisége dinamikusan növekedhet, valamint közelebb érezhetik magukhoz a játékot, ha a játszmák közben az általuk feltett kérdéseket viszont láthatják.

A kérdések bővítésének köszönhetően nem csupán az informatikai ismeretek oktatására alkalmas a szoftver, hanem a felhasználók igényei szerint bármilyen témakört be lehet vezetni, mellyel akár közgazdaságtani, természettudományi ismeretek is elsajátíthatók lehetnek, így az egyetem más karai számára is hasznos felkészülési és szórakozási lehetőséget jelenthet.

9.2. Továbbfejlesztési lehetőségek

A játék népszerűbbé válásával számos módon lehetne még felhasználó-barátiabbá, még élvezetesebbé tenni a web alkalmazást. Első sorban a felületen lehetne szépíteni, dinamikusabb, színeesebb designt készíteni. Be lehetne vezetni a chat lehetőségét az egyes játékokban is, nem csak a főoldalon, így a felhasználók a játék közben is tudnának beszélgetni egymással.

Az esetleges automatikus regisztrálások ellen lehetne védeni a rendszert szövegek képről történő felismertetésével, vagy e-mailes regisztráció megerősítéssel. A brute-force jelszófeltörések ellen korlátozni lehetne a belépési próbálkozások számát, és egy időkorlátot megadni az újra próbálkozáshoz. Az elfelejtett jelszavakat egy e-mailben küldött linkkel lehetne módosítani, amennyiben a titkos kérdésre tudja a felhasználó a választ.

Az okos telefonok elterjedése miatt idővel a telefonokon is elérhetővé lehet majd tenni az alkalmazást. Az Opera és Safari böngészőkkel már jelenleg is kompatibilis a rendszer, de az Android operációs rendszerrel rendelkező és az iPhone telefonok kijelzőjének felbontásához hozzá kell majd igazítani, hogy a játéktérület és a kérdések is jól láthatóak legyenek.

10. ÖSSZEFOGLALÁS

A szakdolgozatom célja az volt, hogy olyan szoftverrendszert készítsek, amely gyorsan és hatékonyan segíti a mérnök informatikus hallgatók felkészülését a zárhelyikre és a vizsgákra. Mindennek eléréséhez először megvizsgáltam a .NET keretrendszer által nyújtott adatbázis elérési lehetőségeket.

A szakirodalom feltérképezésével és összetett terheléses tesztekkel sikerült kiválasztanom egy kombinált megoldást, mely a játékhoz szükséges kérdéseket kellőképpen gyorsan rendelkezésre bocsátja, valamint az egyéb adatkezelési feladatokat is egyszerűen és könnyen továbbfejleszthető módon ellátja.

Megfogalmaztam a rendszer használati eseteit, szerepköröket definiáltam, majd specifikáltam az általuk elérhető funkciókat, kapcsolatukat a rendszerrel. Felépítettem a webes alkalmazás által nyújtott funkciókat, valamint a játékmenet folyamatát, a kérdés típusokat, a szabályokat és a pontozási rendszert.

Miután a működésbeli követelményeket megfogalmaztam, megvizsgáltam, hogy milyen technológiával lenne a legcélszerűbb megvalósítani a rendszert. Több módszer vizsgálatát követően az ASP.NET keretrendszert választottam, majd ennek megfelelően UML diagramok segítségével megterveztem a rendszer felépítését, részletesen átgondoltam az osztályok működését, az objektumok közti kommunikációt. A játékmenethez tartozó megjelenítési lehetőségeket is megvizsgáltam és a felépített logikához kapcsolva kiválasztottam a megfelelő módszereket.

Megvizsgáltam az ASP.NET-tel használt programnyelveket, majd kiválasztottam a célnak legmegfelelőbbet. A rendszer implementációja folyamán figyelembe vettem a C#-os fejlesztéshez tartozó elnevezési és kommentezési konvenciókat, ügyeltem arra, hogy a webes alkalmazás minél több böngészővel kompatibilis legyen. A fejlesztés során mivel az ASP.NET keretrendszer még új volt számomra, ezért egyes részek megvalósítását újraterveztem.

A fejlesztés alatt folyamatosan teszteltem a már elkészült részeket, majd amikor a rendszer már nagyrészt kész volt, rögzítettem a játékba történő belépés folyamatát, majd nagyszámú virtuális felhasználóval terhelési tesztet végeztem ezen oldalakon, hogy lemérjem, hol vannak teljesítménybeli problémák a rendszerben. Ezt követően a játékmenet funkcionális teszteléséhez megkértem néhány ismerőst, hogy egy időben teszteljék a játékot, és jegyezzék fel észrevételeiket. A tesztek során felmerülő hibákat kijavítottam, így a rendszer készen áll a használatra.

A szakdolgozatban kitűzött célt, miszerint egy hatékony, az informatikai ismereteket játékos formában oktató rendszert kell készítenem a tesztekben elért eredmények által bizonyítottan teljesítettem. Személyes célom az volt, hogy a szakdolgozat fejlesztése során elsajátítsam az ASP.NET keretrendszer használatát. Ezt is sikerült elérnem, mivel az ezen idő alatt megszerzett tudással el tudtam helyezkedni egy webes alkalmazások fejlesztésével foglalkozó vállalatnál.

11. SUMMARY

The purpose of my thesis was to prepare a software system which quickly and effectively helps to prepare of computer engineer students for tests and exams. To reach all of these first I have examined the database access possibilities supplied by the .NET framework.

I have succeed to choose a combined solution by the research of professional literature and complicated load tests, which solution provides the questions for the game in a sufficiently fast and effective way, furthermore does the other data handling tasks in a simply and easily continue to develop way.

I have drawn up the use cases of the system, defined the roles, then I have specified the reachable functions and the connections to the system. I built the supplied functions by the web application, the process of the game, the question types, the rules and the scoring system.

After I formulated the functional requirements I examined which technology would be the most practical for realize the system. After look into more methodologies I have chosen the ASP.NET framework, then accordingly to this I planned the architecture of the system by the help of UML diagrams, I thought over the details of the working of the classes and the communication between the objects. I analyzed the visualization possibilities which belonged to the game and I have chosen the suitable technologies connected to the built logic.

I examined the programming languages used with ASP.NET, so I have chosen the most suitable for my purpose. I took into consideration the naming and commenting conventions of C# during the implementation of the software, and I took care about the compatibility of the web application to more browsers. Forasmuch the ASP.NET framework was new for me, therefore I have replanned certain parts during the development.

In the course of programming I have permanently tested the completed parts, when the most of the system was prepared I have recorded the process of the accessing into the game and I made load tests with a large amount of virtual users on the pages to measure where are performance problems in the system. After this I have asked some of my friends to test the functional working of the system and play the game at the same time, and note their remarks. I fixed the problems which came up during the tests, thereby the system is ready for use.

The object I purpose to my thesis, that I have to prepare an effective system, which teaching informatics knowledge in a playful way, I have succeeded proved by the test results. My personal purpose was to learn the usage of the ASP.NET framework during I develop my thesis. I could succeed this also, forasmuch I was able to get a job at a web developing company with the knowledge I got along this time.

IRODALOMJEGYZÉK

- [1] Troelsen, A.: Pro C# 2008 and the .NET 3.5 Platform. *Apress*, 2007
- [2] http://en.wikipedia.org/wiki/.NET_Framework, 2011-03-18
- [3] Johnson, G.: Microsoft .NET Framework 3.5 – ASP.NET Application Development, *Microsoft Press*, 2009
- [4] Troelsen, A.: Pro C# 2010 and the .NET 4 Platform, *Apress*, 2010
- [5] <http://www.amazon.com/Programming-Entity-Framework-Building-Centric/dp/0596807260>, 2011-03-22
- [6] <http://learn.iis.net/page.aspx/243/aspnet-integration-with-iis-7/>, 2011-08-31
- [7] http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Visual_Basic_.NET, 2011-09-28
- [8] <http://www.vbrad.com/article.aspx?id=63>, 2011-09-28
- [9] <http://blogs.msdn.com/b/tom/archive/2008/03/14/validation-of-viewstate-mac-failed-error.aspx>, 2011-10-15