Debreceni Egyetem
Informatikai Kar

TANULÓI NYILVÁNTARTÓ SZOFTVER FIREBIRD ADATBÁZIS ALKALMAZÁSÁVAL

Témavezető:
Dr. Bajalinov Erik
Tudományos főmunkatárs

Készítette:
Juhász Gergely József
Informatikatanári

Debrecen
2009

Tartalomjegyzék

1. Bevezetés ... 3
2. A Firebird mint adatbázis-kezelő rendszer rövid bemutatása ... 4
3. Tanulo v2.0 követelményei, és adatbázis szerkezete ... 5
3.1. Firebird parancsértelmező ... 7
3.2. Néhány szó a tranzakciókról ... 7
3.3. Az adattárolás – táblák ... 8
3.4. Tanulo v2.0 táblaszerkezete ... 10
3.5. Generátorok ... 12
3.6. Triggerek ... 13
3.7. Tárolt eljárások ... 16
3.8. Kivételek ... 18
3.9. A Tanulo v2.0 programban lévő tárolt eljárások: ... 19
1.) Új osztály felvétele és módosítása ... 19
2.) Osztály törlését megvalósító tárolt eljárás ... 21
3.) Tanár törlése az adatbázisból ... 22
4.) Tanár felvétele és módosítása tárolt eljárás ... 22
5.) Tantárgy elvétele tanártól ... 23
6.) Tárgy felvétele és módosítása ... 23
7.) Tantárgy hozzárendelése osztályhoz metódus ... 24
8.) Tantárgy hozzárendelése tanárhoz ... 25
9.) Tantárgy elvétele osztálytól ... 25
10.) Tantárgy törlése tárgyak közül ... 26
3.10 Adatbázis szerkezet összegzése ... 26
4. Tanulo v2.0 vázlatos működése, főbb metódusai, felülete ... 27
4.1. Kapcsolódás az adatbázishoz, adatbázis komponensek ... 27
4.2. Használt adatbázis komponensek és néhány fontosabb metódus ... 30
4.3. A program felülete részlegesen ... 33
5. Összegzés ... 37
Ábrajegyzék ... 38
Irodalomjegyzék ... 39

1. Bevezetés

Dolgozatom célja, egy számítógépen futó szoftver és hozzá tartozó adatbázis megalkotása és részleges bemutatása, amely képes hatékonyan és egyszerűen a mindennapi adminisztrációs tanári munkát egyszerűbbé tenni. Mielőtt bármit is mondanék a dolgozat miértjéről, és keretében készített programról, szólnék egy pár szót előző mondatommal kapcsolatban. Egy olyan alkalmazás, ami képes hatékonyan és egyszerűen… Manapság a szoftverek piacán nagyon sok megvásárolható és szabad szoftvert találunk, ami az élet szinte minden területére kínálnak segítséget. Ilyen az oktatás adminisztrációja is. Ezen szoftverek közül, ami „jól" működik, nagy része az oktatási intézmények számára részlegesen vagy egyáltalán nem érhető el anyagi vonzatuk miatt, vagy ami elérhető, használati bonyolultsági szintje miatt nem alkalmazzák, mert sok esetben egy főállású adminisztrátort igényelne. Az általam készített szoftver jelenleg képes nyilvántartani osztályok, a benne lévő tanulók, az osztályokat tanító tanárok, és az egyes tanárok által tanított tantárgyakat. Az alkalmazás nem csak adminisztrátorok, hanem tanárok számára is jól használható.

Döntésem, hogy ilyen témát választottam, az, hogy magam is egy középiskolában tanítok négy éve, és folyamatosan problémáim voltak saját osztályom tanulóinak adatnyilvántartásával, az egyes osztályokban tanított tantárgyakkal stb. A probléma részlegesen megoldható táblázatkezelő szoftver segítségével.

Ugyan rendelkezésre áll egy Oktatási Minisztérium által elérhető szoftvert használni az iskola adminisztrátorának, de a magas bonyolultsági szintje miatt ezt nem alkalmazzák, és nem is alkalmazták. Továbbá egy olyan szoftver megalkotása a cél, amit nem csak adminisztrátor, hanem tanárok is alkalmaznak.

Említettem, hogy a program ***jól*** használható… A Tanulo v2.0 nevű szoftver előzménye egy szintén általam készített iskolai adminisztrációs program, amelyet kettő évvel ezelőtt szeptemberben adtam a kollegák részére használatra. Még ma is használják, diákok nyilvántartására, különböző dokumentumok nyomtatására, stb.. Amiért kell a fejlesztés, annak fő oka, hogy a program előző verziója lokális adatbázist használ, tehát ha módosítunk bármilyen adatot egy gépen futó Tanulo programban, akkor azt nem érzékeli a többi gépen lévő Tanulo adatbázis. Jelenleg fejlesztés alatt álló szoftverem elődje Paradox adattáblákat alkalmazott az adatok nyilvántartására, továbbá az egyes lekérdezéseket nem SQL lekérdezésekkel, hanem rekord léptető műveletek segítségével oldotta meg.

Mondanom sem kell, hogy már 400 diák adatainak tárolása esetén a lekérdezések jelentősen lelassultak. Ennek fő oka az SQL mellőzése. Továbbá paradox adattáblák alkalmazása esetén nem ajánlják több ezer rekord nyilvántartását stabilitási problémák miatt.
A Tanulo v2.0 kliensprogramja Delphi v7.0 környezetben készült, Firebird hálózati adatbázist használ, minden lekérdezés és módosítás SQL parancsok alkalmazásával történik.
A következőkben bemutatom a Firebird lényegét, „szépségit", Tanulo v2.0 programmal szemben támasztott követelményeket, az általam kreált adatbázis szerkezetét, a Firebird által használható elemeket – mint tárolt eljárások, triggerek, stb. – az én általam készített példákra támaszkodva felületesen kifejtem, továbbá a Tanulo v2.0 vázlatos működését és főbb metódusait, programfelület néhány ablakát. Nem kívánok élni a teljesség igényével, amikor az RDBMS és adatbázis tervezésével kapcsolatos objektumok elméleti hátterét, a program felületét és egyéb részleteket fejtem ki. A dolgozat szöveges részének fő célja nem a grafikus felület bemutatása, hanem egy stabil Firebird adatbázis létrehozása, a szükséges elméleti háttér bemutatásával, és ennek elérése, használata Delphi programozási környezet segítségével.

2. A Firebird mint adatbázis-kezelő rendszer rövid bemutatása

A Firebird adatbázis-kezelő egy nyílt forráskódú relációs (RDBMS) adatbázis-kezelő rendszer. Alapját a Borland Interbase 6.0 adja, ami még ingyenes RDBMS. Akár üzleti, akár nyílt forráskódú felhasználásra teljesen ingyenes. A Firebird egy 20 éve fejlesztés alatt lévő technológiát használ, így eléggé kifejlett és megbízható. Annak ellenére, hogy a 2008 közepén kiadott Firebird 2.1.2 szerver telepítőjének mérete mindössze 6,7 MB, mindent nyújt, amit elvárhatunk egy nagykaliberű adatbázis kezelőtől. Néhány KB-tól egészen számtalan GB-ig bármekkora adatbázist kezel kiváló teljesítménnyel, és elhanyagolható karbantartási igénnyel.
Főbb jellemzői:

- Tárolt eljárásokat és Triggereket teljes mértékben támogatja
- Tranzakció-kezelés
- Külső kulcsok (Foreing key) támogatása
- Kidolgozott nyelv a tárolt eljárások és triggerek kezelésére (PSQL - PostgreSQL)
- Támogatott operációs rendszerek: Linux, Windows, Solaris, MacOS, stb..

Főbb jellemzők után, nagyon fontos, hogy a Firebird egy mindenre kiterjedő parancssoros eszközzel van csomagolva. Ezekkel létrehozhatunk új adatbázist, megtekinthetünk adatbázis

statisztikákat, futtathatunk SQL utasításokat, illetve szkript-eket, csinálhatunk az adatbázisáról biztonsági mentést, melyet vissza is állíthatunk, stb..
A kliens programok több módon is kapcsolódhatnak az adatbázishoz: native/API, dbExpress driver-ek, ODBC, OLEDB, .Net provider, JDBC native type 4 driver, Python modul, PHP, Pearl, stb.
A Firebird egy szintés fontos legfontosabb tulajdonsága, hogy kliens – szerver modell alapján készült. Ez különbözik az olyan programtól, mint a Paradox, Visual dBASE és az Access, melyek a fájl-szerver modellt használják. A fájl-szerver modell esetén minden adatbázis intelligencia a kliens szoftverben van. Az adatbázis a fájlszerveren található. Ha az adatbázison valamilyen műveletet akarunk végrehajtani, a feldolgozáshoz mindent a kliens gép memóriájába kell másolni. Ha például egy táblán olyan lekérdezést akarunk végrehajtani, amely a tábla sorainak két százalékát adja vissza, a hálózaton keresztül minden egyes sort át kell másolni a kliens gépre, annak érdekében, hogy a soroknak azt a bizonyos két százalékát megkaphassuk. Kliens-szerver modellben az adatbázis intelligenciájának jelentős része az adatbázis szerveren van. Ha az egyik kliensnek valamilyen műveletet kell elvégezni az adatbázison, a kérést megfogalmazzuk a kliens gépen, majd a hálózaton keresztül elküldjük a szerverhez. Ennek feldolgozása a szerveren történik. Az előző példánál maradva a sorok kiválogatása (SELECT) a szerveren történik, csupán a soroknak azon két százalékát kapjuk meg a hálózaton keresztül, amelyek a lekérdezésnek megfelelnek. A kliens-szerver modell nagymértékben csökkenti a hálózati adatforgalmat. Jobb adatbázis integritást és nagyobb teljesítményt biztosít, mert az adatbázis kezelő szoftver egy helyre koncentrálódik. A fájl-szerver modell esetében minden egyes gépnek nagyteljesítményűnek kell lennie, mert csak így tudja kezelni az adatbázis szoftvert. A kliens-szerver modell esetében elegendő, ha a szerver rendelkezik nagy teljesítménnyel, a kliens gépek kisebb teljesítményűek is lehetnek.
A Firebird szerver telepítője szabadon letölthető a http://www.firebirdsql.org/ címről.

3. Tanulo v2.0 követelményei, és adatbázis szerkezete

Az adatbázissal és kliens-szoftverrel szemben támasztott követelmények:

- Legyen képes tanulók iskolai adatnyilvántartásához szükséges adatokat tárolni
- Tudjon tárolni tantárgyakat
- Tanárokat

- Osztályokat
- Az osztályokban diákokat
- Az osztályhoz osztályfőnököt
- Tanár – Tantárgy összerendelés megvalósítása
- Osztály – Tantárgy összerendelés megvalósítása
- Az osztályok, mint évfolyam és mint osztály vannak nyilvántartva, pl.: 9.A
- Egy osztálynak egy osztályfőnöke lehet
- Nem lehet két azonos osztály
- Egy osztály akkor törölhető, ha nincs hozzárendelve tanuló
- Egy tanár taníthat több tárgyat
- Tárgy felvitelekor nem vehetünk fel két azonos nevű tárgyat
- Egy tantárgy tartozhat több tanárhoz, mivel egy tárgyat több tanár is tanít egy iskolában
- Egy osztályban nem szerepelhet kétszer ugyanaz a tárgy ugyan annál a tanárnál
- Egy osztályban egy tárgy többször is szerepelhet különböző tanárok esetén
- Tárgy törlésekor annak törlődnie kell az osztályoktól és a tanároktól is.
- Tanár nem törölhető, ha osztályfőnök, vagy valamelyik osztályban tanít
- Ha egy tárgyat törlünk egy tanártól, akkor annak törlődnie kell az osztályok tárgyai közül is, de csak azoknak, amelyek az éppen tanártól elvett tárgyak.

Ha megvalósul a megfelelő adattárolás a fent leírtak alapján, akkor több lekérdezési szempont is megfogalmazható.

Egy iskolai adatnyilvántartó szoftvert említünk, akkor gondolunk pl. az osztályzatok nyilvántartására is. A Tanulo v2.0 jelenlegi állapotában erre nem képes, ez egy fejlesztési lehetőség.

A fent felsorolt követelményeknek való megfelelés ugyan megoldható lenne, ha minden vizsgálatot és szabálynak való megfelelést a kliens program végezne. Azonban ezek jóval egyszerűbben és gyorsabban megoldhatóak és megfogalmazhatóak az adatbázis oldalán a megfelelő eszközökkel. Ha ezt választjuk, akkor a kliens gépeknek jóval kevesebb memóriát kell fordítaniuk a program futtatására, mert minden adatbázis műveletet a szerver gép végez.

Tehát a Firebird adatbázison belül a probléma megoldására szükségünk lesz: adattáblákra, triggerek és tárolt eljárásokra, generátorokra, kivételek létrehozására és kezelésére.

Mielőtt megnéznénk, a felsorolt objektumokat hogyan tudjuk létrehozni, és milyen lehetőségeink vannak, tisztázzuk, hogy mi segítségével tudjuk ezt megtenni.

3.1. Firebird parancsértelmező

Lehetőségünk van karakteres és grafikus felületen is létrehozni és karbantartani adatbázisainkat. A Firebird szerver telepítője tartalmaz egy parancssori programot, ami parancsértelmezője a Firebird által kínált utasításoknak. Enek neve: Firebird ISQL Tools.
Ez elérhető a Firebird szerver telepítési mappájának „bin" könyvtárában, isql.exe. Megjegyezném, hogy ezen mappa tartalmaz több kis programot is, amivel karbantarthatjuk adatbázisunkat, pl. biztonsági mentés készítése és visszaállítása.
Számos grafikus felületű program is rendelkezésünkre áll Windows és Linux környezetben is, ezek egy rész szabadon elérhető és használható, még mások szabadon csak Triál verzióként használhatóak. Ilyen programok:

- FlameRobbin – szabadon elérhető és használható
- IBExpert – Triál verziója szabadon letölthető
- JDBStudio – Java nyelven íródott szabadon letölthető és használható több platformon.

Ne elejtsük el, hogy ezen grafikus programok átláthatóbbá és egyszerűbbé teszik adatbázisunk létrehozását és felügyeletét, de a triggerek, tárolt eljárások, és sok más objektumot nekünk kell létrehozni a begépelt kód alapján.

3.2. Néhány szó a tranzakciókról

Az adatok integritásának megtartásához a relációs adatbázisok egyik legfontosabb eszköze a tranzakció. A tranzakció egyetlen - adott esetben összetett - feladatot jelent, amely kívülről egyetlen egészként tekintendő, és amit több tulajdonság jellemez:

- A tranzakción belül az alkalmazás egy vagy több műveletet hajthat végre, amelyeket adott sorrendben kell végrehajtani. Egy művelet rendszerint egyetlen SQL utasításból áll, ez lehet: SELECT, INSERT, UPDATE, vagy DELETE, vagy pedig ezek kombinációja.
- Amennyiben a tranzakcióban szereplő valamennyi műveletet sikeresen befejeztünk, a tranzakció kapcsán elvégzett változtatásokat véglegesíteni kell (COMMIT). Amíg a változásokat nem véglegesítjük, azok a többi felhasználó számára még láthatatlanok.

- A tranzakciót azonban törölni is lehet, vagyis az elvégzett változtatásokat visszavonhatjuk (ROLLBACK). Az alkalmazásnak ezt kell tennie, ha például a tranzakció egyes műveletei sikertelenek voltak, vagy ha a felhasználó úgy dönt, hogy mégsem szeretné véglegesíteni az általa végrehajtott módosításokat.

Tehát ha egy végrehajtott adatbázis műveletet azok egy csoportját (egy tranzakciót) akarunk véglegesíteni, akkor azt a COMMIT utasítással tehetjük meg, ha visszavonni szeretnénk, akkor a ROLLBACK műveletet használjuk.

3.3. Az adattárolás – táblák

Adatokat akarunk tárolni, ezt megtehetjük adatbázis használatával. Ma korszerűnek mondhatók az RDBMS – relációs adatbázis-kezelő rendszerek. Relációs adatmodell alapeszköze a tábla. A tábla a relációnak nevezett matematikai szerkezet speciális példánya, és a nevét is innen kapta. A reláció kétdimenziós, sorokból és oszlopokból álló mátrix, amely homogén és dinamikus adatszerkezet. A táblák nem táblázatok, hanem halmazok, amelyeknek nincs első vagy utolsó soruk, ennek következtében a szekvenciális adatfájlokból ismert BOF (fájl eleje) és az EOF (fájl vége) fogalomnak nincs értelme. Ha a halmazon végrehajtunk egy műveletet, akkor annak minden elemén egyszerre hajtjuk végre. A táblákhoz és oszlopaihoz megszorításokat köthetünk, amelyeknek mindig érvényben kell maradniuk.

Egy tábla tehát mátrix minden oszlopát a relációelméletben tulajdonságnak tekintjük (rekord alapú rendszerben mezőnek nevezik). A relációs adatbázisban minden relációnak (táblának) egyedi névvel kell rendelkeznie, továbbá a reláción belüli oszlopnak is hasonlóan egyedi névvel kell rendelkeznie.

A mátrix – tábla – sorait az Firebird-ben ***sor***nak hívjuk. (Rekord alapú rendszerekben rekordnak hívjuk). Minden sor összetartozó adatcsoportot jelent, és ennek a sornak az egyes oszlopbeli értékei az adott tulajdonság értékeit jelentik.

Egy oszlopban a felvehető értékek tartományát domain-nek nevezzük. Ha egy oszlop értékeit adott tartományon belül tartjuk, ezt a domain integritásának nevezzük. Az InterBase a PRIMARY KEY, a FOREIGN KEY, a UNIQUE és a CHECK megszorítással biztosítja a domainek integritását.

Mivel minden sornak egyedinek kell lennie, a relációs modell megkívánja, hogy legyen egy olyan különleges oszlop (csoport), amely egyedileg azonosítja a sort. Egy táblában több ilyen csoport is lehet, ezeket hívjuk kulcsoknak.

A lehetséges kulcsok halmazából választunk egyet, ez lesz majd az elsődleges kulcs. Ha egyetlen oszlopból áll, akkor egyszerű kulcsnak nevezzük, ha pedig többből, akkor összetett kulcsról beszélünk. Ha van egyszerű kulcs, azt érdemes választani. A reláció elméletben minden táblához meg kell határozni egy elsődleges kulcsot. Ha ilyen nincs, akkor a teljes sor azonosítja a rekordot. Az elsődleges kulcs értéke nem lehet NULL. Összetett kulcs esetén egyetlen oszlop értéke sem lehet NULL. Az elsődleges kulcstulajdonságban lévő minden értéknek egyedinek kell lennie. Az elsődleges kulcs automatikus védelmet nyújt a többszörös értéktárolás ellen. Adatművelet esetén - ha az adatművelet érinti az elsődleges kulcsot - az adatbázis kezelő ellenőrzi, hogy ne következzen be kulcsütközés.

Az idegen kulcs egy másik tábla elsődleges kulcsára történő hivatkozás.
Döntő fontosságú az adatbázis szempontjából az elsődleges és az idegen kulcs elv. Az idegen kulcs és a hivatkozott elsődleges kulcs közötti kapcsolat adja azt a mechanizmust, amely az adatok egységességét és integritását biztosítja.
Az idegen kulcs egy vagy több oszlop, amely egy reláció elsődleges kulcsára hivatkozik. A hivatkozás azt jelenti, hogy ha beírok egy értéket idegen kulcsnak, a Firebird ellenőrzi, hogy ilyen elsődleges-kulcs érték valóban létezik-e a hivatkozott táblában.

A relációs adatbázis-kezelő rendszer minden oszlopa egyedi információt tárol. Az oszlopok meghatározott típusúak. Az oszlop tartalmazhat szöveg, szám, dátum vagy más típusú adatot.

Adattípusok

Típus neve	Tárolható adatok	méret
CHAR(n)	Szöveg	Fix tárolási oszlopméret
VARCHAR(n)	Szöveg	Változó tárolási oszlopméret
SMALLINT	Egész szám	2 byte, -32,768-tól +32,767
INTEGER	Egész szám	4 byte, -2,147,483,648-től +2,147,483,647
FLOAT	Valós szám	4 byte
DOUBLE	Valós szám	8 byte
NUMERIC(n,m)	Egész vagy valós	Rögzített a tizedespont helye, az n és m értéke 1 és 15 között lehet
DECIMAL(n,m)		
DATE	Dátum	32 bites szó
TIME	idő	
TIMESTAMP	Dátum és idő	Két 32 bites szó
BLOB	bitfolyam	dinamikus

A Firebird-ben lehetséges az oszlop tömbként történő definiálása. Többdimenziós tömb definiálható egydimenziós tömbök tömbjeként.

Miután megnéztük, hogy a táblákat milyen szabályrendszer mellet hozhatjuk létre, nézzük meg a Tanulo v2.0 táblaszerkezetét.

3.4. Tanulo v2.0 táblaszerkezete

Táblát létrehozni a CREATE TABLE utasítással lehetséges

Tábla feladata:	Egyes diákokat nyilvántartó tábla	
tábla neve:	**table_diak**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
nev	varchar(30)	
diak_cim	varchar(50)	
diak_cim_varos	varchar(20)	
szul_hely	varchar(20)	
szul_ido	date	
sz_szam	varchar(15)	
diak_tel	varchar(20)	
anyja_nev	varchar(30)	
gondviselo_nev	varchar(30)	
gondviselo_cim	varchar(50)	
gondviselo_tel	varchar(20)	
diak_taj	varchar(15)	
diak_OM	varchar(20)	
diak_diak	varchar(20)	
osztaly_ID	integer	FK
mj	varchar(1000)	
tanult_nyelv_ID	integer	FK
akt_tanev	smallint	

A tanulók táblát a következő Firebird – SQL kóddal hoztuk létre:

```
CREATE TABLE TABLE_DIAK (
  ID                INTEGER NOT NULL,
  NEV               VARCHAR(40),
  DIAK_CIM          VARCHAR(60),
  DIAK_CIM_VAROS    VARCHAR(30),
  SZUL_HELY         VARCHAR(30),
  SZUL_IDO          VARCHAR(15) CHARACTER SET NONE,
  SZ_SZAM           VARCHAR(20),
  DIAK_TEL          VARCHAR(30),
  ANYJA_NEV         VARCHAR(40),
  GONDVISELO_NEV    VARCHAR(40),
  GONDVISELO_CIM    VARCHAR(60),
  GONDVISELO_TEL    VARCHAR(30),
```

```
  DIAK_TAJ          VARCHAR(20),
  DIAK_OM           VARCHAR(20),
  DIAK_DIAK         VARCHAR(20),
  OSZTALY_ID        INTEGER,
  MJ                VARCHAR(1000),
  TANULT_NYELV_ID   INTEGER,
  EVFOLYAM          INTEGER
);
```

Tábla feladata:	osztályokat nyilvántartó tábla	
tábla neve:	**table_osztaly**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
osztaly_szam	varchar(3)	
osztaly_betu	varchar(3)	
osztaly_szak	varchar(50)	
osztalyfonok_ID	integer	FK

Tábla feladata:	Lehetséges tanulni kívánt nyelvek	
tábla neve:	**table_nyelv**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
nyelv	varchar(30)	

Lehetséges tanulni kívánt nyelvet külön nyilvántartjuk egy táblában, mert ez nem osztályonként, hanem diákonként eltér.

Tábla feladata:	Lehetséges Tanulandó tantárgyak	
tábla neve:	**table_targy**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
targy_nev	varchar(20)	

Tábla feladata:	Osztály és Tantárgyakat összekötő tábla, segítségével megmondjuk,hogy melyik osztály milyen tárgyakat melyik tanárnál tanulja	
tábla neve:	**table_oszt_targy**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
tagy_ID	integer	FK
osztaly_ID	integer	FK
tanar_ID	integer	FK

Tábla feladata:	Tanárokat nyilvántartó tábla	
tábla neve:	**table_tanar**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
név	varchar(30)	

Tábla feladata:	A tábla segítségével megmondjuk, hogy melyik tantárgyat milyen tanár tanítja	
tábla neve:	**table_tanar_targy**	
tulajdonság azonosító	**tulajdonság típus**	**egyéb paraméter(ek)**
ID	integer	not null; PK
targy_ID	integer	FK
tanar_ID	integer	FK

3.5. Generátorok

A generátor adatbázis objektum, amely segít megadni azokat az értékeket, amelyeket egy adott oszlopba kell beírni. A generátorok feladata az, hogy olyan egyedi azonosító sorszámot hozzanak létre, amelyeket rendszerint a tábla elsődleges kulcsaként adunk meg, tehát egyedi értékeket tudunk kreálni a segítségével.

Generátor létrehozásához a CREATE GENERATOR utasítást használjuk. A generátornak olyan egyedi nevet kell adnunk, amelyre akkor hivatkozunk, amikor értékét szeretnénk növelni.

A generátor növeléséhez és az új érték eléréséhez használjuk a GEN_ID( ) függvényt.

A GEN_ID( ) függvénynek két független változója van, és alakja a következő:

GEN_ID(GenerátorNeve, LéptetésiÉrték)

A GenerátorNeve paraméter a meghívandó generátor neve. A LéptetésiÉrték paraméter az az érték, amellyel a generátor pillanatnyi értékét növelni vagy csökkenteni kell. Ha ez az érték pozitív, a generátor értéke nő, ha negatív, csökken. A GEN_ID( ) függvény meghívható triggerből, tárolt eljárásból, vagy valamilyen alkalmazásból, ha INSERT, UPDATE, vagy DELETE utasítást használunk.

Fontos, hogy a generátor értékének növelése nem olyan művelet, amelyet visszacsinálhatunk azzal a tranzakcióval, amely meghívta a generátort. Ez azért van így, mert a generátor működése független a Firebird tranzakció modelljétől.

A generátorok képesek kezelni egyidejűleg több felhasználót, akiknek többsége tranzakciókat indít és hagy jóvá (COMMIT). Csupán kevésszámú tranzakciót kell visszaállítani (ROLLBACK).

Ha az alkalmazásunk olyan rendszert igényel, amelyben nem lehetnek üres helyek az elsődleges kulcsban, saját értékadó rendszert kell készíteni, amely megvizsgálja az egyes számok foglalt voltát.

A szakdolgozati programban 7 generátor található, feladatuk a 7 tábla egyszerű elsődleges kulcsainak értékeit megfelelően létrehozni.

GEN_diak_ID
GEN_osztaly_ID
GEN_nyelv_ID
GEN_targy_ID
GEN_oszt_targy_ID
GEN_tanar_ID
GEN_tanar_targy_ID

A GEN_diak_ID generátort a következő kóddal hozzuk létre:

```
CREATE SEQUENCE "GEN_diak_ID";
ALTER SEQUENCE "GEN_diak_ID" RESTART WITH 0;
```

3.6. Triggerek

A trigger olyan, az adatbázisunkban tárolt program, amely automatikusan végrehajtásra kerül, ha bizonyos fajta események fordulnak elő. Ezek az események táblához és sorhoz kötődnek.

Például, megadhatunk olyan triggert, amely meghívásra kerül, ha egy sort hozzáadunk, módosítunk vagy törlünk. Ha a tábla több során hajtunk végre műveletet, a táblához rendelt trigger minden érintett soron végzett művelet alkalmával végrehajtásra kerül.

A tárolt eljárás végrehajtása a szerveren és nem a kliens gépen történik.

A triggerek az adatbázis integritás megtartásának alapvető eszközei. Például megadható, hogy a trigger elinduljon, mielőtt új sort adunk hozzá a táblához. Ekkor a trigger megállapíthatja, hogy a sor teljes-e, valamint hogy belsőleg konzisztens-e, és ha nem, akkor megtagadja a művelet végrehajtását. Megadhatunk olyan triggert is, amely akkor lép működésbe, ha a felhasználó olyan sort akar törölni az adatbázisból, ahol a törlendő sort tartalmazó tábla egy a többhöz kapcsolat elsődleges kulcsa. A trigger meghatározza, hogy kapcsolódik-e ehhez a sorhoz az idegen kulcsú táblában sor vagy sorok, és ettől függően végrehajtja a sortörlést, vagy megtagadja.

Adatbázisunkban a CREATE TRIGGER utasítással hozhatunk létre triggert, melynek alakja a következő:

```
CREATE TRIGGER TriggerNeve
    FOR TáblaNeve
    TriggerTipusátJelzőSzó
AS
    LokálisVáltozókDeklarálása
BEGIN
    TriggerTest
END
```

A trigger készítésekor megadhatjuk, hogy azonnal aktív legyen-e vagy inaktív. Alapértelmezésben a trigger aktív, ha nem adunk meg erre vonatkozóan semmit. Az inaktív trigger nem tudja végrehajtani a hozzá rendelt utasítást.

Meg kell adnunk, hogy a trigger a soralapú művelet végrehajtása előtt vagy után lépjen működésbe. A BEFORE INSERT trigger a sornak a táblához történő hozzáadása előtt, míg az AFTER INSERT annak hozzáadása után indul.

A triggert úgy kell megadni, hogy a következő utasítások egyikére működjön:

- Az INSERT trigger sorbeszúrás előtt vagy után lép működésbe
- Az UPDATE trigger sor módosításakor lép működésbe
- A DELETE trigger sor törlésekor lép működésbe

Az eljárástesten belül helyi érvényességű változókat is megadhatunk. Mint minden strukturált programozási környezetben, itt is ezek a változók addig élnek, amíg az eljárás fut. Az eljáráson kívülről nem láthatók, és az eljárás befejeződésével megsemmisülnek. A helyi változókat közvetlenül az AS záradék után adjuk meg, a DECLARE VARIABLE utasítás segítségével.

Az triggertest BEGIN utasítással indul, melyet a helyi változók megadása követ, és END utasítással fejeződik be. A BEGIN és END utasítást ugyancsak alkalmazni kell minden olyan utasításblokkban, amelyek logikailag összetartoznak, ilyenek például a ciklusokban szereplő utasítások.

A Firebird triggerekhez a következő SQL bővítményeket biztosítja:

```
DECLARE VARIABLE
BEGIN … END
SELECT … INTO : VáltozóLista
Változó = Kifejezés
/* megjegyzés */
EXECUTE PROCEDURE
FOR select DO …
IF Feltétel1 THEN … ELSE …
WHILE Feltétel2 DO …
```

Ha az adatbázis parancsértelmezője szkriptet dolgoz fel, normál esetben egymás után beolvas és feldolgoz minden egyes utasítást a lezáró karakterig, amely alapesetben a pontosvessző (;). Minden utasítást elemez, lefordít, belső ábrázolássá átalakít, majd végrehajt.

Trigger és tárolt eljárások esetén is, azonban nem támaszkodhatunk az utasítás végét jelző pontosvesszőre. Az eljárás egy SQL utasításokat tartalmaz, melyek a CREATE TRIGGER eljárás belsejébe vannak beágyazva. Minden egyes beágyazott utasítás pontosvesszővel végződik, ezek azonban nem a CREATE TRIGGER eljárás végét jelzik.

Annak érdekében, hogy az Firebird rendesen értelmezhesse és fordíthassa a tárolt eljárásokat, az adatbázis szoftvernek rendelkeznie kell lehetőséggel a CREATE TRIGGER eljárás befejezésének jelzésére, amely viszont eltér a CREATE TRIGGER eljárás belsejében található utasítások befejezésének jelzésétől.

Erre szolgál a SET TERM utasítás, melynek alakja a következő:

```
SET TERM ÚjLezáróKarakterek JelenlegiLezáróKarakterek
```

Nézzünk meg egy triggert a Tanulo v2.0 adatbázisából:

```
SET TERM ^ ;
CREATE TRIGGER TRG_UJ_DIAK FOR TABLE_DIAK
ACTIVE BEFORE INSERT POSITION 0
AS
begin
   new.ID = gen_ID(gen_diak_ID,1);
   post_event 'uj_diak';
end
^
SET TERM ; ^
```

Láthatjuk, hogy a trigger a table_diak táblához van rendelve, a trigger neve TRG_UJ_DIAK, a tábla INSERT műveletére fog aktiválódni, méghozzá beszúrás előtt (BEFORE), és ha több olyan trigger van ami ezen tábla insert művelete előtt hajtódna végre, akkor azok közül ez fog először, mivel POSITION 0 paramétert kapott.

A trigger fő blokkjának első sora meghívja a GEN_diak_ID nevű generátort. Segítségével előállítja az újonnan beszúrandó sor ID tulajdonságának értékét. A NEW kulcsszó utal az újonnan kapott értékre, tehát a NEW.id, az új ID tulajdonság értéke lesz. A NEW-hoz hasonló környezeti változó az OLD, ami UPDATE és DELETE művelet esetén értelmezhető, a módosítandó vagy törlendő sor valamelyik tulajdonságának régi értékére hivatkozik.
Második sora, a POST_EVENT utasítás egy eseményt küld a kliens programnak, hogy lefutott ezen trigger, tehát minden éppen futó Tanulo szoftver értesül új diák felvételéről.
A szoftver adatbázisában lévő triggerek:

TRG_uj_diak
TRG_uj_osztaly
TRG_uj_nyelv
TRG_uj_targy
TRG_uj_oszt_targy
TRG_uj_tanar
TRG_uj_tanar_targy
TRG_diak_modositas
TRG_diak_torlese

3.7. Tárolt eljárások

A tárolt eljárás olyan program, amelyet az adatbázissal együtt tárolunk. Az adott adatbázison belül speciális adat, és metaadat feldolgozást hajt végre.
Ha végiggondoljuk egy adatbázis koncepcióját, rájövünk, hogy nem csak egyszerűen adatokat tárolunk a táblákban. Annak érdekében, hogy megtartsuk ezen adatok integritását, és ezeket hatékonyan használhassuk, képesnek kell lennünk arra, hogy műveleteket végezzünk ezekkel az adatokkal, módosítsuk és átalakítsuk ezeket.
Ezen feladatok végrehajtásához a Firebird rendelkezik egy tárolt eljárás programozási nyelvvel, amely arra készült, hogy optimálisan kezelhessük a táblázatokban tárolt adatokat. Rendelkezik SQL kiegészítéssel, amely támogatja a változókat, a megjegyzéseket, a deklaráló utasításokat, a feltétel ellenőrzést, az elágazást és a ciklusokat. Olyan nyelv, amely arra készült, hogy az adatbázison belül fusson. A tárolt eljárásban használható utasítások használhatóak triggerekben is.
A tárolt eljárás végrehajtása a szerveren és nem a kliens gépen történik.
A tárolt eljárások teljes mértékben használhatják a SQL adatmódosító nyelv (DML – INSERT, UPDATE, DELETE) utasításait.
Minden eljárás önálló programegység, melyet többféleképpen hajthatunk végre:

- Interaktív módon az isql parancssorból,

- SELECT utasítás részeként. A Firebird támogatja a tárolt eljárások alkalmazását, hogy lekérdezéseknek és allekérdezéseknek értéket adjanak vissza,
- Másik eljárásból, vagy adatbázis triggerből,
- C++, Delphi vagy más környezetben írt alkalmazásokból.

Tárolt eljárásokat a következő esetekben használjunk:

- Ha a műveletet teljes egészében elvégezhetjük a szerveren és a művelet végrehajtása alatt a felhasználó erről információt nem igényel. Jegyezzük meg, hogy a tárolt eljárás meghívásakor kérhet be adatokat, és adhat vissza adatokat a meghívó alkalmazásnak.
- Ha a művelet nagyszámú sor feldolgozását igényli.
- Ha a műveletet az alkalmazáson belül vagy a különböző alkalmazásokban több különböző modul vagy folyamat hívja meg.

Tárolt eljárást a CREATE PROCEDURE utasítással hozhatunk létre, amelynek alakja a következő:

```
CREATE PROCEDURE EljárásNeve BemenőParaméterLista
    RETURNS VisszaadottParaméterLista
    AS
        HelyiVáltozókDeklarálása
    BEGIN
        EljárásTest
    END
```

Eljárás Neve egyedi név az adatbázison belül.

A BemenőParaméterLista olyan változókat tartalmazó felsorolás, melyeket a hívó alkalmazás ad át az eljárásnak. Ezeket a változókat azért használjuk, hogy módosíthassuk viselkedését.

A VisszaadottParaméterLista olyan értékeket tartalmaz, amelyet a folyamat adhat vissza az őt meghívó alkalmazásnak, ilyen lehet például egy számítás eredménye.

Minden lista alakja a következő:

```
ParaméterNeve1   ParaméterTípusa,
ParaméterNeve2   ParaméterTípusa,
...
ParaméterNeveN   ParaméterTípusa,
```

Az eljárástesten belül helyi érvényességű változókat itt is megadhatunk a triggerekhez hasonlóan.

Az eljárástest BEGIN utasítással indul, melyet megelőz a helyi változók megadása, és END utasítással fejeződik be, a triggertesthez hasonlóan.

Tárolt eljárás végrehajtásának legegyszerűbb módja az EXECUTE PROCEDURE eljárás használata. A SUSPEND utasítással az eljárás végrehajtását felfüggesztjük, és a kimenő változó tartalmát átadjuk a hívó utasításnak, tehát minden tárolt eljárás testjének utolsó utasítása a SUSPEND.

Mielőtt megnézzük a nyilvántartó program által tárolt eljárásokat, szólnék néhány szót a kivételekről.

3.8. Kivételek

A kivételek olyan "nevet kapott hibaüzenetek", amelyeket az adatbázisban tárolunk tárolt eljárásokban és triggerekben történő alkalmazás céljából. A kivételt létrehozása után meghívhatjuk.

Ha az eljárást úgy írtuk meg, hogy csak egyetlen integer értéket kell visszaadnia, a Firebird nem engedi meg, hogy helyette szöveges hibaüzenet jelenjen meg. Amennyiben azonban kivételt hozunk létre, a normál visszatérési érték törlésre kerül, és helyette a Firebird kivételt adja vissza.

Ha egy kivétel kiváltódik, akkor az addig végzett műveletek az aktuális tranzakción belül, visszavonásra kerülnek (ROLLBACK), továbbá, a kivételhez rendelt szöveg automatikusan elküldésre kerül azon kliensprogram felé, akinek műveletvégzésének hatására kiváltódott a kivétel.

Kivételt a CREATE EXCEPTION utasítással hozhatjuk létre. Az utasítás alakja a következő:

```
CREATE EXCEPTION KivételNeve
'Kivételkezelő üzenet szövege';
```

Az EXCEPTION utasítással a meghívó eljárást értesítjük kivétel fennállásáról. A hívó alkalmazás lehet trigger, tárolt eljárás, ISQL kliens, vagy bármilyen más program. Az EXCEPTION utasítás alakja a következő:

```
EXCEPTION KivételNeve;
```

A Tanulo programban alkalmazott kivételek:

EXP_mar_felvett_tantargy	Kiváltódik, ha egy osztályhoz olyan tárgyat akarunk hozzáadni, amit már tanul.
EXP_osztalyfonok_nem_torolheto	Kiváltódik, ha tanárt törlünk és osztályfőnök! Ekkor nem lesz törlés.
EXP_foglalt_of	Kiváltódik, ha új osztály vagy osztály módosítás esetén olyan osztályfőnököt akarunk, aki már osztályfőnök
EXP_letezo_osztaly	Kiváltódik, ha osztály felvétel esetén már létező osztályt választunk.
EXP_letezo_targy	Kiváltódik, ha olyan nevű tárgyat akarunk felvenni, ami már létezik
EXP_kozbe_valaki_felvette	Kiváltódik, ha valaki azt a tárgyat már hozzárendelte a tanárhoz, amit én épp most akarunk felvenni.
EXP_mar_nem_letezo_targy	Kiváltódik, ha olyan tárgyat akarok rendelni egy osztályhoz, amit pár másodperce valaki megszüntetett.
EXP_nem_ures_osztaly	Kiváltódik, ha osztály törlésekor, még vannak tanulók az osztályban.
EXP_tanit_targyat	Kiváltódik tanár törlése közben, ha tanít tárgyat!
EXP_kozbe_toroltek	Tárgy-tanár hozzárendeléskor előfordulhat, hogy a kiválasztott tantárgyat ez elmúlt néhány másodpercben valaki törölte.
EXP_kozbe_tanart_toroltek	Kiváltódik, ha tanár – tantárgy hozzárendelésekor a tanárt akihez tárgyat akarunk adni közben valaki törölte
EXP_toroltek_az_osztalyt	Kiváltódik, ha osztályhoz tárgyat rendelünk, de közben valaki tölri azt az osztályt, amihez éppen adni akarunk tárgyat

3.9. A Tanulo v2.0 programban lévő tárolt eljárások:

1.) Új osztály felvétele és módosítása

A következő tárolt eljárás meghívásával van lehetőségünk új osztályt felvenni az adatbázisba, továbbá osztály adatait – csak osztályfőnök nevét és szak megnevezést – módosítani. A bemenő paraméterek tartalmazzák az új osztály adatait. Az osztály_id bemenő paraméter értéke ha –1 akkor új osztály felvételéről van szó, ha nem -1, akkor módosítás, mégpedig azon osztály módosul, amelyiknek az ID-je osztaly_id-vel egyenlő. Ha a kívánt osztályfőnök már más osztály osztályfőnöke, akkor nem végezhető el a művelet. Ekkor egy kivétel váltódik ki. Figyeli azt is, hogy azonos elnevezésű osztály nem vehető fel, ekkor szintén egy kivétel fog

kiváltódni. Ha kivétel váltódik ki, akkor az adatbázis művelet leáll, és esetleges eddigi módosítások visszavonásra (ROLLBACK) kerülnek.

```
SET TERM ^ ;
CREATE OR ALTER PROCEDURE PRC_OSZTALY_FELVETELE (
    osztaly_of integer,
    osztaly_szak varchar(50),
    osztaly_betu varchar(3),
    osztaly_id integer,
    osztaly_szam varchar(3))
as
declare variable lehet_modositani_of smallint;
declare variable megvaltozott_of_id integer;
declare variable uj_of_id integer;
declare variable foglalt_oszt integer;
declare variable of_id integer;
begin
  of_id=-1;
  osztaly_betu = upper(:osztaly_betu);
  select table_osztaly.osztalyfonok_id from table_osztaly where table_osztaly.osztalyfonok_id =
:osztaly_of INTO :of_id;
  if (:of_id > -1 and :osztaly_id = -1) then
  begin
     exception exp_foglalt_of;
  end
  if (:osztaly_id > -1) then
  begin
      uj_of_id = -1;
      select table_tanar.id from table_tanar INNER JOIN table_osztaly ON table_tanar.id =
table_osztaly.osztalyfonok_id where table_osztaly.osztalyfonok_id = :osztaly_of INTO :uj_of_id;
      megvaltozott_of_id = -1;
      select table_osztaly.osztalyfonok_id from table_osztaly where table_osztaly.osztaly_szam =
:osztaly_szam and table_osztaly.osztaly_betu = :osztaly_betu INTO :megvaltozott_of_id;
      lehet_modositani_of = 0;
      if (:megvaltozott_of_id = :osztaly_of or  uj_of_id = -1) then lehet_modositani_of = 1;
      if (lehet_modositani_of = 0) then exception exp_foglalt_of;
  end
  if (:osztaly_id = -1) then
  begin
    foglalt_oszt = -1;
    select table_osztaly.id from table_osztaly where (table_osztaly.osztaly_szam = :osztaly_szam)
AND (table_osztaly.osztaly_betu = :osztaly_betu) INTO :foglalt_oszt;
    if (:foglalt_oszt > -1) then
    begin
      exception exp_letezo_osztaly;
    end
  end
  if (:osztaly_id = -1) then
      INSERT into table_osztaly(osztaly_szam, osztaly_betu, osztaly_szak, osztalyfonok_id)
values(:osztaly_szam, UPPER(:osztaly_betu), :osztaly_szak, :osztaly_of);
  if (:osztaly_id > -1) then
      UPDATE table_osztaly SET osztaly_szam = :osztaly_szam, osztaly_betu = :osztaly_betu,
osztaly_szak = :osztaly_szak, osztalyfonok_id = :osztaly_of WHERE table_osztaly.id = :osztaly_id;

  suspend;
end^
SET TERM ; ^
```

2.) Osztály törlését megvalósító tárolt eljárás

Osztály törlését vihetjük végbe ezen eljárással, először megvizsgáljuk, hogy a törölni kívánt osztályba jelenleg van e tanuló. Ezt CURSOR segítségével tudjuk megtenni. A CURSOR arra szolgál, hogy egy SELECT utasítás eredményét sorról sorra feldolgozhassuk. A példában az eredmény első sorát dolgozzuk fel, ha van. Azért szükséges CURSOR-t használnunk, mert a SELECT utasítás több sort is visszaadhat az esetben, ha már többen járnak az osztályba. Ekkor hiba történne, ha nem CURSOR-t használnánk, mert a lekérdezés eredményét egy INTEGER típusú változóba tesszük, és több egész nem fér el egy egész helyén.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_OSZTALY_TORLESE (
    osztaly_id integer)
as
declare variable id integer;
declare c cursor for (select table_diak.id from table_diak where table_diak.osztaly_id = :osztaly_id);
begin
    id = -1;
    open c;
    fetch c INTO :id;
    close c;
    if (:id > -1) then
        exception exp_nem_ures_osztaly;

    DELETE from table_oszt_targy where table_oszt_targy.osztaly_id = :osztaly_id;
    DELETE from table_osztaly where table_osztaly.id = :osztaly_id;

  suspend;
end^

SET TERM ; ^
```

3.) Tanár törlése az adatbázisból

A paraméterül kapott tanár_id-vel megegyező tanárt törli. Nem törölhető, ha a tanár osztályfőnök, vagy valamilyen tantárgyat tanít. Ha valamelyik teljesül, akkor kivétel váltódik ki.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TANAR_ELTAVOLITAS (
    tanar_id integer)
as
declare variable tan_id integer;
declare variable osztf_id integer;
declare c cursor for (select table_oszt_targy.id from table_oszt_targy where table_oszt_targy.tanar_id
= :tanar_id);
begin
        osztf_id = -1;
        select table_osztaly.osztalyfonok_id from table_osztaly where
             table_osztaly.osztalyfonok_id = :tanar_id INTO :osztf_id;
        if (:osztf_id>-1) then
            exception exp_osztalyfonok_nem_torolheto;
        tan_id = -1;
        open c;
            fetch c INTO :tan_id;
        close c;
        if (:tan_id > .1) then
            exception exp_tanit_targyat;
        delete from table_tanar_targy where table_tanar_targy.tanar_id = :tanar_id;
        delete from table_tanar where table_tanar.id = :tanar_id;
  suspend;
end^

SET TERM ; ^
```

4.) Tanár felvétele és módosítása tárolt eljárás

Paraméterül kapott tanar_nev lesz az új tanár neve, ha a tanar_id = -1, különben a tanar_id-jű tanár neve módosul a tanar_nev-re.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TANAR_FELVETEL (
    tanar_nev varchar(30),
    tanar_id integer)
as
declare variable nev varchar(30);
begin
  if (:tanar_id = -1) then
        insert into table_tanar(nev) values(:tanar_nev);
  if (:tanar_id > -1) then
        update table_tanar SET table_tanar.nev = :tanar_nev where table_tanar.id = :tanar_id;
  suspend;
end^

SET TERM ; ^
```

5.) Tantárgy elvétele tanártól

Ezen egyszerű műveletet is tárolt eljárás meghívásával végezzük, mert törléskor törölnünk kell a table_tanar és table_tanar_targy táblából is.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TARGY_ELVETELE_TANARTOL (
    tanar_id integer,
    targy_id integer)
as
begin
    DELETE from table_tanar_targy where table_tanar_targy.tanar_id = :tanar_id AND
table_tanar_targy.targy_id = :targy_id;
    DELETE from table_oszt_targy where table_oszt_targy.tanar_id = :tanar_id AND
table_oszt_targy.targy_id = :targy_id;
  suspend;
end^

SET TERM ; ^
```

6.) Tárgy felvétele és módosítása

Tantárgyat vehetünk fel, vagy módosíthatunk az eljárás segítségével.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TARGY_FELVETELE (
    targy_id integer,
    targy_nev varchar(20))
as
declare variable id integer;
begin
  id = -1;
  targy_nev = upper(:targy_nev);
  select table_targy.id from table_targy where table_targy.targy_nev = :targy_nev INTO :id;

  if (:id > -1) then exception exp_letezo_targy;

  if (:targy_id = -1) then INSERT INTO table_targy(targy_nev) values(:targy_nev);
  if (:targy_id > -1) then UPDATE table_targy SET table_targy.targy_nev = :targy_nev WHERE
table_targy.id = :targy_id;

  suspend;
end^

SET TERM ; ^
```

7.) Tantárgy hozzárendelése osztályhoz metódus

Az alprogram meghívásával tantárgyat rendelhetünk osztályhoz. Fontos, hogy tárgy és osztály párt kell rendelnünk egy osztályhoz, mert pl. matematikát egy iskolában több tanár is tanít. Továbbá, ha van az adott osztálynál azonos tárgy-tanár páros, akkor nincs felvétel, kivétel váltódik ki. Fontos, hogy csak létező osztályhoz adhatunk tárgyat, ugyanis előfordulhat, hogy valaki törölte azt az osztályt, amihez éppen tárgyat akarunk rendelni.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TARGY_HOZZAADASA_OSZTALYHOZ (
    targy_id integer,
    osztaly_id integer,
    tanar_id integer)
as
declare variable toroltek_az_osztalyt integer;
declare variable letezik_e integer;
declare variable mar_van_ilyen_targy integer;
begin
  mar_van_ilyen_targy=-1;
  select table_oszt_targy.targy_id from table_oszt_targy where
     table_oszt_targy.osztaly_id = :osztaly_id AND
     table_oszt_targy.targy_id = :targy_id AND
     table_oszt_targy.tanar_id = :tanar_id INTO :mar_van_ilyen_targy;

  if (:mar_van_ilyen_targy > -1) then
      exception exp_mar_felvett_targy;

  letezik_e = -1;
  select table_tanar_targy.id from table_tanar_targy where table_tanar_targy.tanar_id = :tanar_id
AND table_tanar_targy.targy_id = :targy_id INTO :letezik_e;
  if (:letezik_e = -1) then
      exception exp_mar_nem_letezo_targy;

  toroltek_az_osztalyt = -1;
  select table_osztaly.id from table_osztaly where table_osztaly.id = :osztaly_id INTO
:toroltek_az_osztalyt;
  if (toroltek_az_osztalyt = -1) then
      exception exp_toroltek_az_osztalyt;

  insert into table_oszt_targy(targy_id, osztaly_id, tanar_id) values(:targy_id, :osztaly_id, :tanar_id);

  suspend;
end^
SET TERM ; ^
```

8.) Tantárgy hozzárendelése tanárhoz

Egy tanárhoz rendelhetünk tantárgyat akkor, ha az adott tanárhoz még nincs azonos nevű tárgy rendelve, továbbá a kívánt tantárgy „még” létezik, mert előfordulhat, hogy az elmúlt másodpercekben valaki törölte, esetleg azt a tanárt törölte valaki, akihez tárgyat akarunk adni.

```
SET TERM ^ ;
CREATE OR ALTER PROCEDURE PRC_TARGY_HOZZAADASA_TANARHOZ (
    targy_id integer,
    tanar_id integer)
as
declare variable letezo_tanar integer;
declare variable kozbe_toroltek integer;
declare variable nem_adtak_meg_hozza integer;
begin
    nem_adtak_meg_hozza = -1;
    select table_tanar_targy.id from table_tanar_targy where table_tanar_targy.targy_id = :targy_id
AND table_tanar_targy.tanar_id = :tanar_id INTO :nem_adtak_meg_hozza;
    if (:nem_adtak_meg_hozza > -1) then
        exception exp_kozbe_valaki_felvette;
    kozbe_toroltek = -1;
    select table_targy.id from table_targy where table_targy.id = :targy_id INTO :kozbe_toroltek;
    if (:kozbe_toroltek = -1) then
        exception exp_kozbe_toroltek;
    letezo_tanar = -1;
    select table_tanar.id from table_tanar where table_tanar.id = :tanar_id INTO :letezo_tanar;
    if (:letezo_tanar = -1) then
        exception exp_kozbe_tanart_toroltek;

    INSERT INTO table_tanar_targy(targy_id, tanar_id) values(:targy_id, :tanar_id);
  suspend;
end^
SET TERM ; ^
```

9.) Tantárgy elvétele osztálytól

A probléma megoldása egy egyszerű DELETE SQL utasítás, igazából felesleges tárolt eljárásban megvalósítani.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TARGY_TORLESE_OSZTALYTOL (
    targy_id integer,
    osztaly_id integer,
    tanar_id integer)
as
begin
  delete from table_oszt_targy where
     table_oszt_targy.targy_id = :targy_id AND
     table_oszt_targy.osztaly_id = :osztaly_id AND
     table_oszt_targy.tanar_id = :tanar_id;
  suspend;
end^
SET TERM ; ^
```

10.) Tantárgy törlése tárgyak közül

Segítségével tantárgyat törölhetünk az adatbázisból, de ha ezt tesszük, akkor a törölt tárgyat el kell távolítani azon osztálytól, aki ezt tanulja, azon tanároktól, akik ezt tanítják, és a table_targy táblából.

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE PRC_TARGY_TORLESE_TARGYAK_KOZUL (
    targy_id integer)
as
begin

  delete from table_oszt_targy where table_oszt_targy.targy_id = :targy_id;
  delete from table_tanar_targy where table_tanar_targy.targy_id = :targy_id;
  delete from table_targy where table_targy.id = :targy_id;
  suspend;
end^

SET TERM ; ^
```

3.10 Adatbázis szerkezet összegzése

Amint láthatjuk, a Tanulo v2.0 adatbázisában sok olyan probléma, amit elsőre talán triggerel oldanánk meg, tárolt eljárással van megvalósítva. Amiért én így döntöttem: A módosító vagy beszúró SQL utasítás így sok esetben, az adatbázisban van eltárolva, és meghívva. Ha triggerekkel végeztettük volna a fenti munkák nagy részét, akkor az SQL utasítások a kliensprogramban vagy szöveges file-kban a kliensprogram mellet lennének tárolva. Ezen megoldás esetén, a kliensprogramban, „csak” eljárás hívások szerepelnek bizonyos bemenő paraméterekkel ellátva. Ha bármi módosítás kell tenni a későbbiekben SQL utasításokat érintve, akkor azt nem kell megtenni minden egyes kliensprogramokban, vagy frissíteni minden kliensprogramot (ha tegyük fel, már több gépen használják), hanem egyetlen egy helyen, az adatbázisban, amely az egyetlen Firebird szervert futtató gépen található.

A fent leírt kódok mennyisége talán soknak tűnhet egy ilyen egyszerű probléma megoldása esetén, mint diákok, tantárgyak és tanárok tárolása, de ha ezt jól megtervezzük, akkor a kliensprogram kódjából jócskán spórolunk, nem is beszélve a kliensprogram erőforrás szükségleteiről. Továbbá adatbázisunk hatékonyan, gyorsan és hibamentesen fog működni az adatok tárolását tekintve.

A fejezet címének a Tanulo v2.0 követelményei és adatbázis szerkezete címet adtam. Korszerű, így Firebird adatbázis használata esetén, a szerkezet magába foglalja táblákon túl a

generátotorok, triggerek, kivételek, tárolt eljárások stb. halmazát is, mivel ezek segítségével valósítjuk meg a konzisztens, anomália-mentes, integritási hivatkozásokban megfelelő adatbázist.

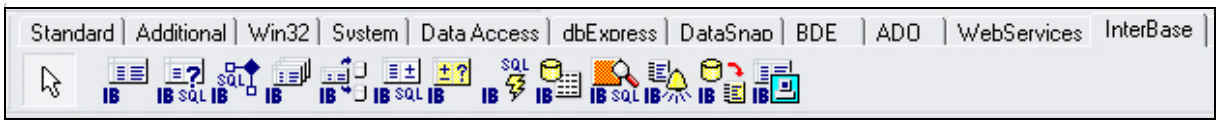
4. Tanulo v2.0 vázlatos működése, főbb metódusai, felülete

4.1. Kapcsolódás az adatbázishoz, adatbázis komponensek

Amint már kiderült, a Tanulo v2.0 két részből áll, az adatbázisból és szoftverből, ami a kliensprogram. A szoftver Borland Delphi 7.0 környezetben készült, és folyamatosan készül. Az alkalmazás Firebird adatbázishoz csatlakozik, tehát olyan programozási eszközökre van szükség, amelyek segítségével az adatbázis elérhető és programozható. A Firebird RDBMS a Borland InterBase 6.0 rendszerből fejlődött ki. A Borland cég a Delphi-hez olyan komponenseket, osztályokat készített és szolgáltat, amelyekkel elérhető és programozható az Interbase adatbázis. Ezek a Delphi Interbase palettáján találhatóak. Az Interbase 6.0 és 6.5 –s verziója az adatbázis file-nak GDB kiterjesztést adott, a 7.0-tól IB-t. A Firebird adatbázis-file kiterjesztése FDB. A Delphi Interbase komponensei azonban csak GDB vagy IB kiterjesztésű file-t hajlandó kezelni abban az esetben, ha a file elérési útját adjuk meg közvetlenül. Másik lehetőség, megadjuk TCP eléréssel a szerver címét, ami LOCALHOST is lehet, és az adatbázisunk ALAIS nevét. És ezen utóbbi megoldás kínál lehetőséget a Firebird programozóknak, hogy el tudják érni a Firebird adatbázis file-kat az IB komponensek használatával. Az IB komponensekkel teljesen kompatibilis a Firebird szerver.

Akkor hogyan érhetjük el Delphi környezetből az FDB file-kat?

Ha a szükséges objektumokat tervezési időben példányosítjuk, akkor azok elérhetőek az Interbase palettán.



Delphi 7, InterBase paletta

Adatbázisunkhoz való kapcsolódására kell két IB komponens a Delphi Interbase palettájáról. Egy TIBDatabase osztálypéldány és TIBTransaction példány. A TIBDatabase objektum Database adattag értékének kell adnunk az adatbázisfile eléréséhez szükséges paramétereket. Ez pl.:

localhost:tanulo_v20 vagy 192.168.1.2:tanulo_v20.

A kettőspont előtt azon gép címe szerepel, ahol fut a szerver, utána lévő rész az ALIAS neve, ami mutat a szerveren lévő adatbázis file-ra. A szervernek kijelölt gépre feltelepítjük a Firebird szerverprogramot. Ez lehet akár az a gép, amin dolgozunk (ekkor a szerver címe: LOCALHOST), majd a Firebird telepítési mappa gyökerében találunk egy aliases.conf file-t. Ebbe kell beírnunk a következőt:

Alias_neve = FDB_file_elérési_útja

Pl.:

tanulo_v20 = d:\tanulo_v20\db\tanulo_v20.fdb

Ha ez megtörtént, akkor állítsuk még be az IBDatabase objektum DefaultTransaction adattagjának értékét, ez legyen az IBTransaction objektum name tulajdonságában szereplő karakterlánc. Továbbá állítsuk be a Params tulajdonság értékének a következőt:

user_name=sysdba
password=masterkey
lc_ctype=WIN1250

Ezzel megadtuk, hogy mi az adatbázis-kapcsolódáshoz szükséges felhasználónév, jelszó, és adatbázisban használt karakterkészlet. Ha az objektum LoginPrompt adattagjának értékét false–ra állítjuk, akkor automatikusan csatlakozik az adatbázishoz, amikor a Connented tulajdonság értékét igazra állítjuk, vagy kiadjuk az IBDatabase1.Open utasítást. A kapcsolódás előtt állítsuk be az IBTransaction objektum DefaultDatabase értékét az IBDatabase objektum name karakterlánc érékére, továbbá a Params tulajdonságához adjuk hozzá a következő sorokat:

read_committed
rec_version
nowait

Ez azért fontos, mert ha végrehajtunk egy COMMIT utasítást (erről szó lesz a következőkben), akkor annak eredményét a többi, éppen adatbázishoz kapcsolódott kliensprogram is lássa.

A Tanulo v2.0 kliensprogramja induláskor az IBDatabase Database tulajdonságának értékének egy file tartalmát olvassa be, ami a szoftver gyökerében lévő database.src file első sora. Ezen első sor:

Szerver_címe:alias_neve

Ennek segítségével a szöveg-file első sorának módosításával megadhatjuk, hogy hol van az adatbázis, és milyen ALIAS név mutat rá.
Sikeres kapcsolódás után indíthatunk SQL utasítások segítségével lekérdezéseket, vagy adatbázis módosító SQL műveleteket a szerver felé. Ezekre használatosak a TIBDataSet és TIBQuery osztályok példányai. Tárolt eljárást hívhatunk meg a TIBStordProc objektum-példány használatával a következő módon:

IBStoredProc1.ExecProc

Az utasítás kiadása előtt állítsuk be az objektum StoredProcName tulajdonságát, ami az adatbázisbeli meghívandó tárolt eljárása neve, továbbá a Parms tulajdonságot, ami egy dinamikus tömb, ennek elmei legyenek sorrendben az eljárás bemenő paramétereinek értékei.
Meghívás előtt ajánlott kiadni az

IBStordProc1.Prepare

utasítást, amellyel előkészítjük a bemenő paramétereket a tárolt eljárás számára.
Ne felejtsük el, hogy ha elvégeztünk egy módosító adatbázis műveletet, akkor ki kell adnunk a használt objektum által küldött COMMIT vagy ROLLBACK utasításokat. Pl.:

IBDataSet1.Transaction.Commit

Vagy

IBStoredProc.Transaction.Commit

Lekérdező SELECT utasítást az aktuális objektum Open metódus meghívásával tudunk érvényesíteni, pl.:

IBQuery1.Open

Természetesen az Open metódus meghívása előtt az SQL.Text adattagnak adjuk meg mint karakterláncot az SQL utasítást.
Ha IBQuery vagy IBDataSet objektum segítségével alkalmazunk INSERT, DELETE vagy UPDATE SQL utasítást, akkor a következő módon futtassuk:

IBQuery1.ExecSQL

Program bezárásakor fontos, hogy hajtódjon végre az IBDatabase példány Close metódusa.
Továbbá fontos, hogy ha végrehajtunk egy COMIIT vagy ROLLBACK utasítást, akkor zárolódik a kapcsolat az adatbázis és program között, mert véglegesítünk egy tranzakciót. Tehát minden adatmegjelenítő, ami esetlegesen egy SELECT utasítás eredményét tárolja, mert a hozzá kapcsolódó IBQuery vagy IBDataSet objektum nyitva van, akkor azok tartalma

nullázódik, tehát nem mutatják a számunkra értékes információt. Ekkor futtassuk le az adott IBQuery vagy IBDataSet objektum Open metódusát.
Tárolt eljárás hívása után is futtassuk le az IBStoredProc objektum COMMIT vagy ROLLBACK metódusát, pl.:

IBStoredProc.Transaction.Commit

Van lehetőségünk arra, hogy ha a szerveren valamilyen esemény, pl. adatmódosítás, törlés, új elem felvétele történik, akkor az adatbázis-szerver egy üzenetet küldjön ki a kliensprogramoknak. Értelemszerűen az nem kap üzenetet, aki a tranzakciót indította, és véglegesítette. Ezen üzenet feldolgozható a TIBEvents osztály egy példányával. A példány database tulajdonság értékének adjuk meg az IBDatabase name adattagjának értéként, ami segítségével kapcsolódunk az adatbázishoz. Az IBEvents objektum Events adattagjának több string értéket tudunk adni az IBEvents.Enevents.Add(s:string) metódus segítségével. Ezek legyenek az adatbázisban definiált események nevei. Ha ez megvan, akkor az osztálypéldány OnEventAlert metódusa alá programozhatunk, ami lefut, ha egy esemény érkezik az adatbázisból. Ezen esemény neve elérhető az EventName bejövő paraméter segítségével.

4.2. Használt adatbázis komponensek és néhány fontosabb metódus

A fejlesztés során az adatbázishoz, mint LOCALHOST szerverhez kapcsolódom, amit nagyon egyszerűen a database.src file-ban bármikor módosíthatunk.
A Tanulo v2.0 programban SELECT utasítások megjelenítésére TDBGrid objektum-példányait használom. Ezek TDataSource osztály példányaihoz kacsolódnak, amelyek TIBDataSet objektumokhoz, ezek pedig a dbTanulo nevű TIBDatabase osztálypéldányhoz.
Tárolt eljárások meghívására TIBStoredProc osztály példányát használom, ami a dbTanulo nevű objektumhoz kapcsolódik.
SELECT lekérdezéshez, INSERT, DELETE és UPDATE utasítások kiadásához használok lokális IBQuery objektumot.
Események fogadásához használok TIBEvents osztálynak egy példányát. Fontos, hogy ha valaki felvesz egy diákot, akkor azt azonnal érzékelje a többi párhuzamosan futó Tanuo program, ezért, ha végrehajtódik egy INSERT vagy UPDATE műveletet, akkor egy „uj_diak” nevű eseményt küld az adatbázis az éppen futó kliensprogramoknak.

A következő függvény segítségével bármilyen SELECT utasítást hajthatunk végre, amelynek eredményét egy Tmemo osztály példányában adja vissza.

Az alprogram négy paramétert kap, az első az sqlFile nevű logikai változó, aminek értéke, ha igaz, akkor a második paraméterben kapott s string tartalma egy SQL lekérdezést tartalmazó file. Ha hamis, akkor az s stringben egy SELECT utasítás található. Ennek eredményét beleteszi egy, a harmadik paraméterként kapott, már létező Memo objektum Text tulajdonságába. A negyedik paraméter egy ibq_parameterek típusú változó, ami egy variant típusú elemeket tartalmazó dinamikus tömb. Ennek amennyi eleme van, azokat paraméterül megkapja a SELECT SQL lekérdezés (paraméterezett SQL utasítások). Nyílván, ha nulla az elemszáma, akkor nem kap paramétereket a lekérdezés.

```
function Tmain_form.query(sqlFile:boolean; s:string;
memo:Tmemo;params:ibq_parameterek):Tmemo;
var ibq:TIBQuery;
   ds:TDataSource;
   dbG:TDBGrid;
   i,j:integer; s1:string;
begin
   memo.Parent:=main_form;
   memo.Visible:=false;
   dbg:=TDBGrid.Create(nil);
   ibq:=TIBQuery.Create(nil);
   ds:=TDataSource.Create(nil);
   ds.Name:='dbData_source';
   ibq.Database:=DM.dbTanulo;
   ibq.Name:='ib_Query';
   ds.DataSet:=ibq;
   dbG.DataSource:=ds;
   ibq.SQL.Clear;
   if not sqlFile then ibq.SQL.Add(s);
   if sqlFile then ibq.SQL.LoadFromFile(DM.prog_DIR+'sqls\'+s);
   if high(params)>-1 then
     for i:=low(params) to high(params) do ibq.Params[i].Value:= params[i]  ;
   ibq.Open;
   memo.Lines.Clear;
   ibq.First;
   while not ibq.Eof do
   begin
      s1:='';
      for j:=0 to dbG.Columns.Count-1 do
      begin
          s1:=s1+dbG.Columns.Grid.Fields[j].AsString+';';
      end;
      memo.Lines.Add(s1);
      ibq.next;
   end;
   result:=memo;
   ibq.Close;
   ibq.Free;
end;
```

A következő eljárás hasonlóan működik, mint az előző, csak ez INSERT és UPDATE utasításokat küld az adatbázisnak.

```
procedure Tmain_form.query_mod(sqlFile:boolean; s:string; params:ibq_parameterek);
var ibq:TIBQuery; i:integer;
begin
    ibq:=TIBQuery.Create(nil);
    ibq.Database:=DM.dbTanulo;
    ibq.Transaction:=DM.trTanulo;
    ibq.SQL.Clear;
    if sqlFile then
        ibq.SQL.LoadFromFile(DM.prog_DIR+'sqls\'+s);
    if not sqlFile then
        ibq.SQL.Add(s);
    for i:=low(params) to high(params) do
        ibq.Params[i].Value := params[i];
    ibq.prepare;
    ibq.ExecSQL;
    ibq.Transaction.Commit;
    ibq.Free;
    DM.dataset_Tanulok.Open;
end;
```

Alábbi függvény bármilyen tárolt eljárás meghívására képes:

```
function Tmain_form.StoredProc(sp_nev:string; params:ibq_parameterek):boolean;
var E:Exception;  ibsp:TIBStoredProc;
   i:integer; ok_felvetel:boolean;
begin
   ibsp:=TIBStoredProc.Create(nil);
   ibsp.Database:=DM.dbTanulo;
   ibsp.Transaction:=DM.trTanulo;
   ibsp.StoredProcName:=sp_nev;
   ibsp.Prepare;
   ok_felvetel:=true;
   for i:=low(params) to high(params) do
   begin
       ibsp.Params[i].Value:=params[i];
   end;
   ibsp.Prepare;
   try
      ibsp.ExecProc;
    except on E:Exception do
    begin
       MessageDlg('H I B A    T Ö R T É N T ! '+#13+'
'+#13+main_form.hiba_kiir(E),mtError,[mbOk],0);
       ibsp.Close;
       ok_felvetel:=false;
       ibsp.Transaction.Rollback;
    end;
   end;
   if ok_felvetel then ibsp.Transaction.Commit;
   ibsp.Free;
   StoredProc := ok_felvetel;
end;
```

A függvény paraméterül kapja a meghívandó adatbázisbeli eljárás nevét, és egy dinamikus tömböt, amiben az eljárás bemenő paramétereinek kell lennie. Az alprogram visszatér egy logikai értékkel, amely hamis, ha az eljárás futása során hiba történt, igaz, ha nem. Ha hiba történik az eljárás futása során, akkor nagy valószínűséggel egy általunk készített kivétel váltódott ki, amelynek hibaüzenetét megjeleníti a felhasználó felé. Ennek „szép” formára hozását egy külön metódus végzi, mivel az adatbázis által küldött üzenet nem csak az, amit mi adtunk meg a kivétel létrehozásakor.

A program további metódusai és teljes forráskód elérhető a mellkelt CD-n.

4.3. A program felülete részlegesen

A Tanulo v.2.0 indításakor sikeres kiszolgálóhoz való csatlakozás után a következő ablakot kapjuk:

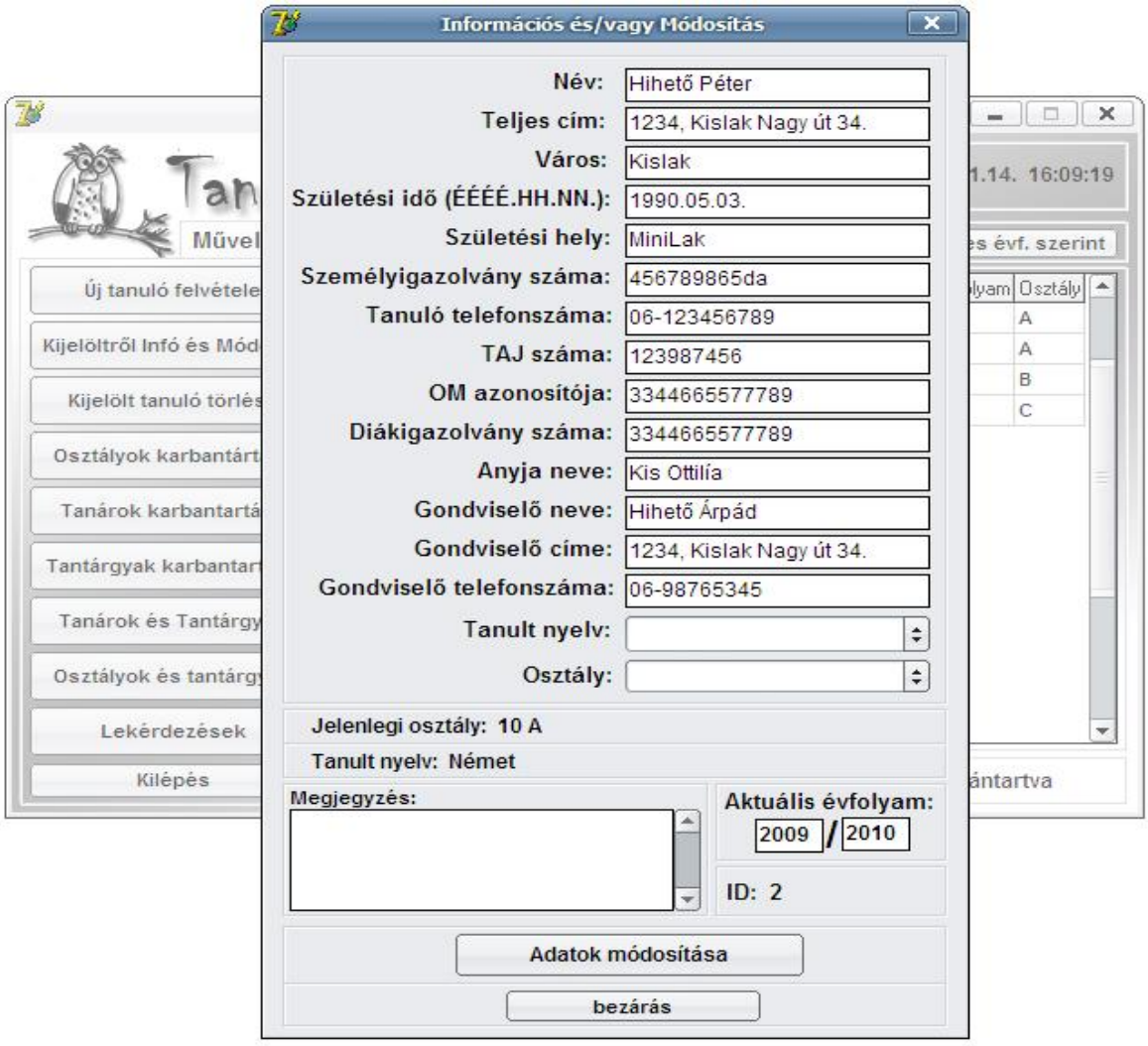
1. ábra: Tanulo v2.0 kezdő és fő -képernyője

Sikertelen kapcsolódás esetén a következőt látjuk:

2. ábra: Sikertelen kapcsolódás

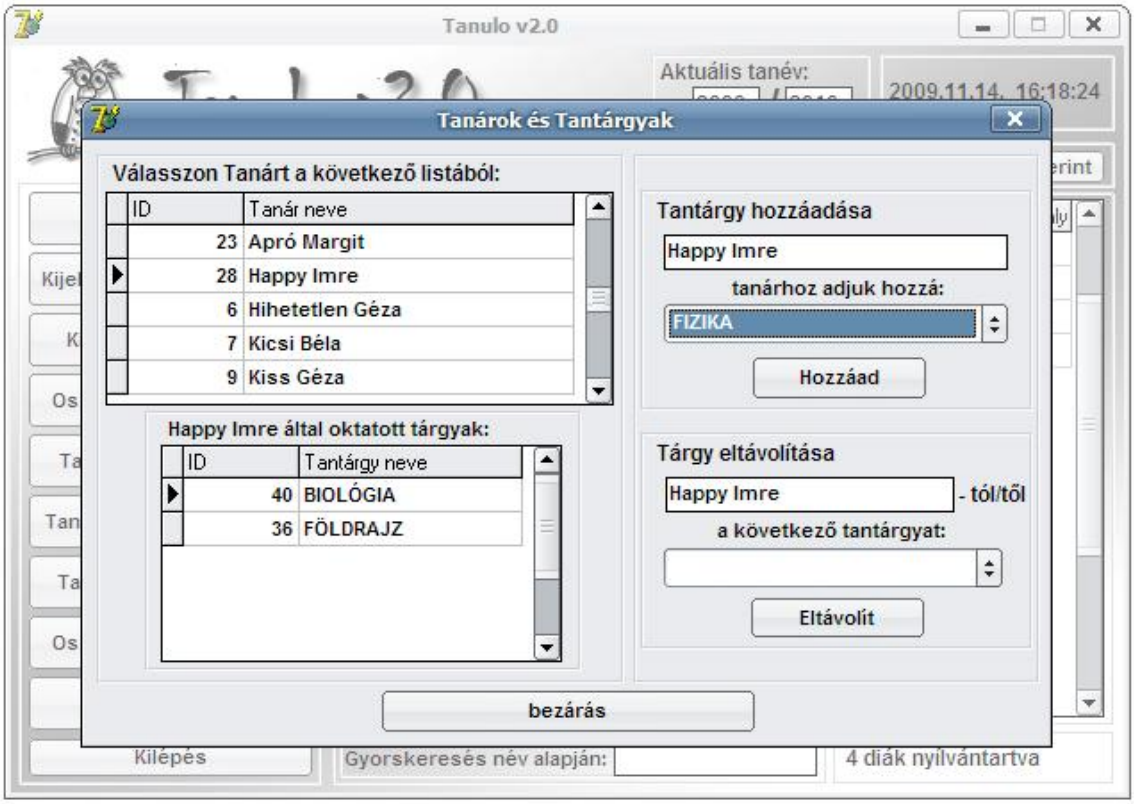
Láthatjuk az 1. ábrán, hogy a program által kínált funkciókat a fő ablak bal oldalán található gombok segítségével érhetjük el. Továbbá az induló ablakban láthatjuk az aktuális dátumot, időt, tanévet és jelenleg adatbázisban nyilvántartott diákok listáját. Van lehetőségünk név szerint gyorskeresésre.

A következő ablak új tanuló felvételekor és létező tanuló módosításakor jelenik meg:



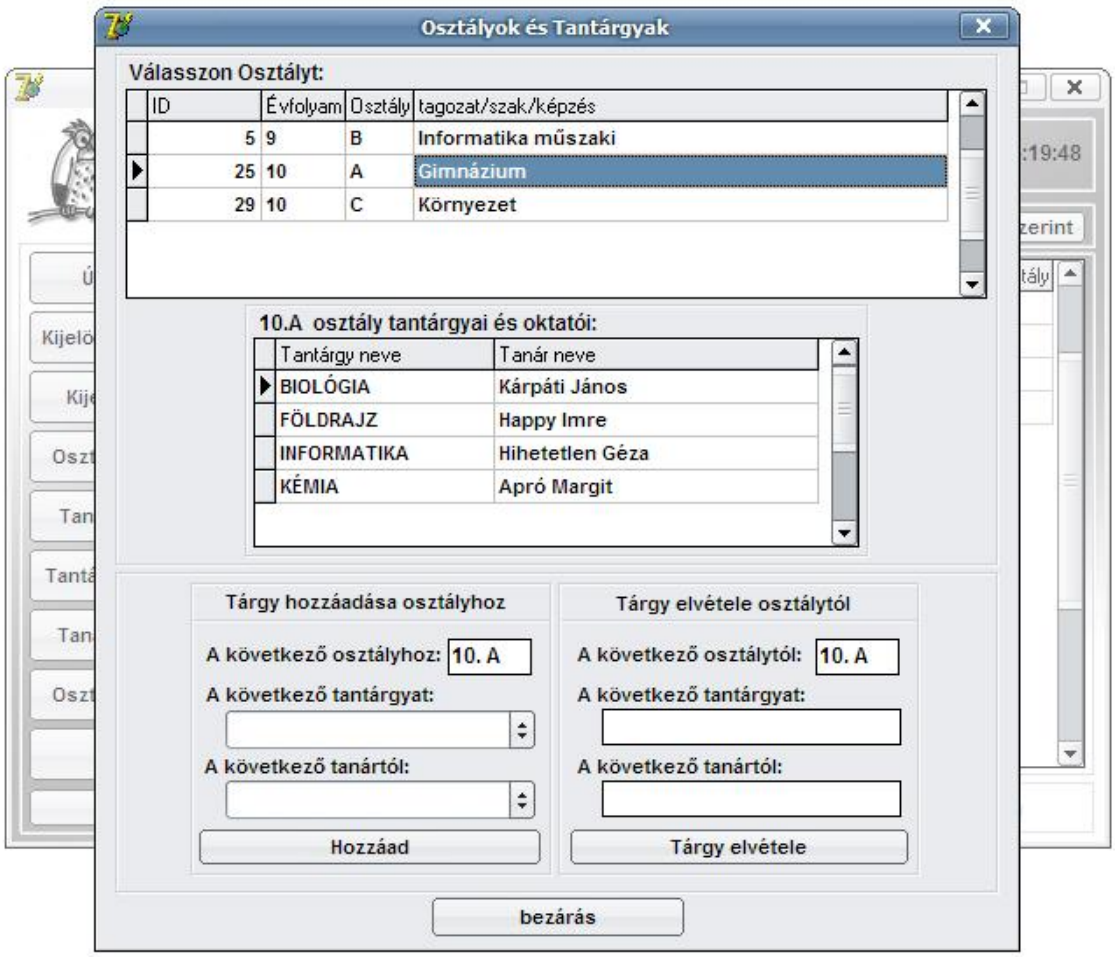
3. ábra: tanuló felvétele és módosítása

Tanárok és tantárgyak gombra kattintva az egyes tanárokhoz rendelhetünk hozzá tantárgyakat. Fontos, hogy ha egy tanárnál ne jelentkezzen többször ugyan az a tanegység. Mint látható, a program a hozzárendelhető és eltávolítható tárgyak listáját combobox alkalmazásával oldja meg, de előfordulhat, hogy egy párhuzamosan futó Tanulo program éppen előttünk pár másodperccel (a combobox feltöltése után) hozzáadja ugyanazon tanárhoz azt a tanegységet, amit mi akarunk. Ezért ezt figyelni kell az INSERT művelet végrehajtáskor. Szintén figyelnünk kell, hogy a hozzáadandó tárgyat valaki már törölte – e. Mint megismerhettük, ezen problémákat egy tárolt eljárás segítségével lekezeljük, és az INSERT műveletet elvégzi helyettünk.



4. ábra: Tanárok és tantárgyak

Szükséges az egyes osztályokhoz is hozzárendelnünk a tantárgyakat. Egy osztálynál szerepelhet több azonos nevű tanegység, de ezeknek különböző tanárnál kell, hogy legyenek nyilvántartva. Hasonlóan a fent említett tanár – tantárgy problémához, figyelnünk kell, hogy létezik –e a tárgy, amit már kiválasztottunk a combobox segítségével, továbbá létezik –e az osztály, akihez tárgyat kívánunk adni.



5. ábra: osztályhoz tantárgy rendelése

A programban található további ablakok, és hibaüzenetek megnézhetőek, ha futtatjuk a mellékelt CD lemezen lévő Tanulo v2.0 programot.

5. Összegzés

A Tanulo v2.0 szoftver fejlesztése során célom a minél egyszerűbb felhasználói felület, és minél hatékonyabb adatbázis készítése és kezelése. A program fejlesztése folyamatos, több helyen is említettem, hogy „jelenlegi állapotában”. A dolgozatban kifejtett Tanulo program verziójában nem szerepelnek olyan konkrét lekérdezések, amelyek igazán hatékonyak lennének az osztályfőnökök számára, gondolok itt pl. lakóhely, születési idő stb. szerinti lekérdezésre, vagy adott diáknak iskolalátogatási igazolás, diákigazolvány igénylő dokumentum közvetlen nyomtatása stb. Ezeket már megvalósítottam a Tanulo v1.x verziókban. Mint említettem, előző verziók adatbázis szerkezete és használatának hatékonysága meg sem közelítette a 2.0-s verziót. Ezen új kiadás is ki lesz bővítve a felsoroltakkal, és egyéb olyan lekérdezésekkel, műveletekkel, amelyeket a tanári kar és vezetőség jónak és hasznosnak tart.

A szoftver jelenlegi példányának tesztelése még nem következett be. A program által nyújtott funkciók hasznossága és kezelhetősége az már tesztelődött az előző verziók alapján, és pozitívan vizsgáztak. Azt, hogy hálózatban, párhuzamos adatbázis-műveletek mellett hogyan működik, azt otthoni környezetben három számítógépen próbáltam tesztelni. Próbáltam a lehető legtöbb szélsőséges esetet szimulálni, ezek kezelését megfelelően kezelik tárolt eljárások és kivételek. Az „éles” működtetés alatt valószínűleg majd merülnek fel hibák, amelyek javítva lesznek. Ismételten hangsúlyozom, ami nagyon „szép” az, hogy a problémák nagy része esetén nem kell a kliensprogram kódjához nyúlni, hanem elég az adatbázisban a megfelelő eljárások javítása, módosítása.

Összegezve, Tanulo v2.0 program jelenlegi állapotában működőképes, a megfelelő osztályfőnöki munka kiegészítésére még bővíteni kell, esetleges minimális módosításokat kell végrehajtani.

Szakdolgozatom konkrét célja a Tanulo v2.0 program elkészítésével és bevonásával egy komolyabb Firebird adatbázis létrehozása lefektetett követelmények mellett, továbbá felületesen Delphi programozási környezetből az adatbázis elérése és használatának bemutatása. Nem szántam nagy terjedelmet a program felületének bemutatására, mivel ez nem is volt célom. Firebird adatbázis létrehozásához szükséges objektumokkal részletesebben foglalkoztam, és néhány, kimondottan az adatbázissal való kommunikációt megvalósító függvényt és eljárást bemutattam. Bővebben kifejtettem a Delphi és Firebird adatbázis kapcsolódásának egyfajta lehetőségét, az InterBase komponensek segítségével.

Végül ismételten hangsúlyoznám, hogy a Firebird adatbázis tervezése, használata, a szerver futtatása otthoni és üzleti felhasználásra is teljesen ingyenes, annak a mai kapitalista szemlélet ellenére, hogy egy hatékony, hálózati kliens – szerver modellt megvalósító adatbázisrendszerről beszélünk.

Ábrajegyzék

Delphi 7, InterBase paletta 27
1. ábra: Tanulo v2.0 kezdő és fő -képernyője 33
2. ábra: Sikertelen kapcsolódás 34
3. ábra: tanuló felvétele és módosítása 34
4. ábra: Tanárok és tantárgyak 35
5. ábra: osztályhoz tantárgy rendelése 36

Irodalomjegyzék

- Szabó János: Az InterBase * FIREBIRD UNIX / LINUX / WINDOWS ComputerBooks kiadó, 2005
- www.prog.hu programozási portál
- www.firebird.org
- www.firebirdnews.org
- http://www.softwareonline.hu
- http://www.w3schools.com/SQl/

Köszönetnyilvánítás

Szeretném kifejezni köszönetemet Dr. Bajalinov Erik Konzulens tanáromnak, aki segítségével hozzájárult szakdolgozatom szöveges, és szoftver részének megalkotásához.