Pannon Egyetem
Műszaki Informatikai Kar
Villamosmérnöki és Információs Rendszerek Tanszék
Mérnökinformatikus BSc

SZAKDOLGOZAT

Életmód-támogató szakértői rendszer automatizált adatbázis-frissítése

Tobik János

Témavezető: Dr. Vassányi István
Konzulens: Szálka Brigitta
2017

Szakdolgozat témakiírás

Nyilatkozat

Alulírott Tobik János hallgató, kijelentem, hogy a dolgozatot a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszékén készítettem a mérnökinformatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy a dolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Veszprém, 2017. december 4.

Tobik János

Alulírott Dr. Vassányi István témavezető kijelentem, hogy a dolgozatot Tobik János a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszékén készítette a mérnökinformatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védésre bocsátását engedélyezem.

Veszprém, 2017. december 4.

Dr. Vassányi István

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Vassányi Istvánnak, hogy tapasztalatával, tanácsaival és tudásával segítette szakdolgozatom elkészítését.

Hálával tartozom továbbá családomnak és barátaimnak, hogy támogattak és véleményükkel, ötleteikkel, bíztatásukkal hozzájárultak a szakdolgozat megvalósításához.

TARTALMI ÖSSZEFOGLALÓ

A Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén működő Egészségügyi Informatikai Kutató-Fejlesztő Központ már több éve fejleszt egy életmód-támogató szakértői rendszert, a Laviniát. Segítségével könnyedén nyilvántartható mindennapi táplálékbevitelünk és személyes visszajelzések révén elsajátítható a helyes táplálkozás. Ennek következtében a szoftver felhasználói hatékonyabban tarthatják magukat az egészséges életmódhoz.

A Lavinia számos élelmiszerre és tápanyagra vonatkozó adatot biztosít a felhasználója számára. Dolgozatom témája egy olyan ETL alkalmazás kidolgozása, amellyel automatizált módon oldható meg a Lavinia által használt relációs adatbázisban szereplő élelmiszerek, tápanyagértékek és mértékegységek frissítése. Ehhez egy amerikai adatbázis rendszeresen megjelenő új verzióiban megtalálható adatokat kell felhasználni a migrációs folyamat megvalósítására. A rendszer implementálása a Java programozási nyelv és keretrendszerei segítségével történt. Az elkészített szoftver több tízezres nagyságrendben képes az adatok kezelésére.

Az elkészült migrációs alkalmazás jelentős mértékben segítheti elő a Lavinia mögötti adatbázis fejlődését és tartósságát. Ezáltal a felhasználók számára a lehető legnagyobb és legpontosabb adathalmaz áll rendelkezésre.

Kulcsszavak: életmód, ETL, Java, relációs adatbázis, migráció

ABSTRACT

The Medical Informatics Research and Development Center at the Department of Electrical Engineering and Information Systems, University of Pannonia has been developing a lifestyle-support expeert system, called Lavinia. Using the application it is easy to manage the daily nutrition intake and by giving a personal feedback it makes the user able to acquire healthy nurture. Consequently the users are able to sustain a healthy lifestyle more efficiently.

Lavinia provides data for its users referring to several food and nutrients. The topic of my thesis is the elaboration of an ETL application which makes it able to update automatically the relational database of Lavinia, including food, nutritional values and measurement units. For this, it is a must to use the new versions of a regularly updated US database to accomplish the migration process. The implementation of the system has been performed using the Java programming language and its interfaces. The final software is able to handle thousands of data.

The complete migration application significantly helps the improvement and the durability of database behind Lavinia. Thus, the largest and the most accurate set of data is available for the users.

Keywords: lifestyle, ETL, Java, relational database, migration

Tartalomjegyzék

1. Bevezetés 1
2. Felhasznált technológiák 3
2.1. PostgresSQL 3
2.2. Java 4
2.2.1. JDBC 4
2.2.2. JavaFX 5
2.2.3. Apache Commons IO 6
2.2.4. JUnit 6
2.3. Git 7
3. Rendszerkövetelmény 8
3.1. Specifikáció 8
3.2. Adatbázisok ismertetése 11
3.2.1. Relációs adatbázis 11
3.2.2. USDA adatbázisa 12
3.2.3. Lavinia adatbázisa 15
4. Tervezés 18
4.1. Rendszer terve 18
4.2. Modulok 19
4.2.1. Adatbázis interfész 20
4.2.2. Grafikus felhasználói interfész 21
4.2.3. Modell osztályok 21
4.2.4. Vezérlő osztályok 21
4.2.5. Naplózás 22
4.3. Frissítési folyamat terve 22
4.4. Lokális környezet 25

5. Megvalósítás 28
5.1. Felhasználói felület 28
5.2. Adatbázis beállítása 28
5.3. Fájlok feldolgozása 31
5.4. Adatbázis frissítése 33
5.5. Valós idejű megfigyelés 41
5.6. Naplózás 42
5.7. További lehetőségek 44
6. Tesztelés 45
6.1. Unit teszt 45
6.2. Adatbázis oldali tesztek 46
6.3. Rendszerteszt 47
7. Összefoglalás 49

1. Bevezetés

Magyarországon nagyon sokan szenvednek olyan betegségektől, amelyek okozója az egészségtelen életmód. Ezen betegségek sokszor halálhoz is vezethetnek. Ilyen életmód-társult betegségek az elhízás, a metabolikus szindróma, a 2. típusú cukorbetegség, a stroke, a magas vérnyomás, a szívinfarktus, rosszindulatú daganatok, depresszió. A betegségek kialakulásának valószínűségét jelentősen lecsökkenthetjük, ha egészséges életvitelt folytatunk. Az egészséges életmód egyik alapja a tudatos táplálkozás.

A technológia és az internet fejlődésével egyre több és több forrás áll rendelkezésre azok számára, akik szeretnének tudatos életmódot folytatni. Ezek között rengeteg olyat találni, amelyek orvosoktól, szakértőktől vagy épp betegektől származik. Emellett számtalan applikáció létezik okostelefonokra, amelyekkel nyilvántarthatjuk táplálékbevitelünket, vagy napi étrend javaslatokkal és életmód tanácsokkal gazdagodhatunk.

Dolgozatom témája egy ilyen rendszerhez kapcsolódik. Ez az alkalmazás a Lavinia életmód-tükör. Több vizsgálatot is folytattak cukorbetegek segítségével az életmód-támogatás hatékonyságának ellenőrzésére. Egyszerű és gyors kezelőfelületének köszönhetően megközelítőleg napi öt percre csökkenthető a táplálkozási naplózás. Ezen felül a veszélyesen magas és alacsony vércukorszint értékek előfordulása jelentősen csökkent.

A szoftvert a Pannon Egyetem Műszaki Informatikai Karán működő Egészségügyi Informatikai Kutató-Fejlesztő Központ készítette. Fő feladatom új adatok importálása, vagy meglévők módosítása a Lavinia mögötti adatbázisba. Ehhez egy amerikai adatbázis új verzióit kell felhasználnom. Így az adatbázis frissítése révén a Lavinia alkalmazást használók számára a lehető legtöbb és legpontosabb adatok állnak rendelkezésre.

Az US Department of Agriculture által szolgáltatott adatbázis adatait az Amerikai Egyesült Államokban található Agricultural Research Service, röviden ARS, az USDA egyik legfőbb kutatócsoportja biztosítja. A minél pontosabb eredmények érdekében több ezer kutató végzi munkáját, hogy megoldást találjanak

a mezőgazdasági problémákra, amelyek hatással vannak az amerikai emberek mindennapjaira. Az ARS magas prioritással vezeti a kutatást, hogy kifejlesszék a megoldásokat, amelyek az egész nemzetet érintik. A kutatók egyre több és több élelmiszert vizsgálnak meg és akár ugyanazt a vizsgálatot többször is elvégzik egy adott termék esetében, hogy minél pontosabb adatokkal tudjanak szolgálni. Hozzáférést biztosítanak az ételek és más mezőgazdasági termékek vizsgálatainak eredményeiről. Az információkból felállított adatbázis bárki számára elérhető és felhasználható, természetesen jogtiszta módon. Rendszeres jelleggel frissítik az adatbázist. A legelső verzió 1993-ban készült el SR10-es verzió néven. Azóta számos revízió került ki az USDA kutatóinak köszönhetően, a jelenleg utolsó 2015-ös SR28-as frissítéssel bezárólag. **(forrás USDA)**

A Lavinia életmód-tükör adatbázisa szintén a Pannon Egyetem Műszaki Informatikai Karán működő Egészségügyi Informatikai Kutató-Fejlesztő Központ által fejlesztett MenuGene táplálkozás-tudományi szakértői rendszer szolgáltatásait és adatbázisát használja fel. Az eddigiek során dietetikusok tartották karban az adatbázist. Az USDA adatai szintén forrásul szolgáltak ez idáig is, azonban csak manuális módon volt lehetőség a rendszeres frissítésre, amely jelentős kézi munkát vett igénybe. **(forrás Lavinia)**

2. Felhasznált technológiák

Az alábbi fejezetben ismertetem a dolgozatom kivitelezéséhez felhasznált technológiákat. Fontos szempont volt számomra, hogy széles körben használható legyen az alkalmazás, de legfőképpen Windows-os környezetben. Ebből az okból kifolyólag választottam a Java programozási nyelvet és a hozzákapcsolódó keretrendszereket, interfészeket. A felsorolt technológiák egy részét tanulmányaim során volt lehetőségem elsajátítani, de számos új tapasztalattal gazdagodtam az újonnan megismert technológiákkal kapcsolatban.

2.1. PostgresSQL

A PostgreSQL, más néven Postgres egy relációsadatbázis-kezelő rendszer, amelyet a Lavinia rendszere is használ. Szabad szoftver, melynek fejlesztését önkéntesek végzik közösségi alapon. A munka elsődleges koordináló oldala a postgresql.org. Kezdetben a Berkeley Egyetemen indult meg a fejlesztése a nyolcvanas években, majd a kilencvenes évek közepére elhagyta az egyetem falait és nyílt forráskódúvá vált.

A relációsadatbázis-kezelő rendszer (RDBMS) egy olyan adatbázis-kezelő rendszer, amelynek logikai adatbázisát szoftverkomponensei kizárólag a relációs adatmodellek elvén épülnek fel, illetve kérdezhetőek le. Kizárólag a relációs adatmodell alapú megközelítést támogatja. A relációsadatbázis-kezelő rendszerek szabványos adat hozzáférési nyelve az SQL (Structured Query Language). Az SQL segítségével könnyen és érthetően leírhatók akár az összetettebb CRUD (Create, Read, Update, Delete) funkciók is.

Az adatbázis adataihoz való hozzáférést, manipulációt, valamint az adatszerkezet tanulmányozásához a pgAdmin 4 programot használtam. Ez egy ingyenesen elérhető szoftver a PostgreSQL fejlesztőitől. E program melletti választásomat indokolta az, hogy biztosítja az egyszerű kezelőfelület és a szükséges funkciókat a feladatom során.

2.2. Java

A letölthető fájlok kezelésére, az adatbázis elérésére és manipulálására olyan szoftver kell, amely vezérli az adatátvitelt és közben erről tájékoztatást nyújt a felhasználó számára a folyamatról. Ezért döntöttem a Java nyelven történő implementálásról a feladat során.

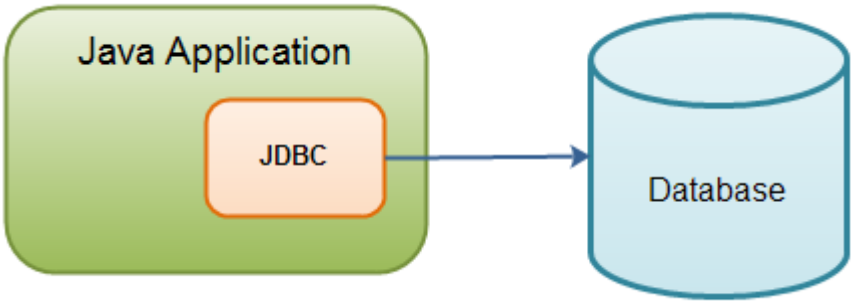
A Java egy általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a kilencvenes évek elejétől. Jelenleg az Oracle gondozásában áll. A Java alkalmazásokat jellemzően bájtkód formátumra alakítják. A bájtkód futtatását a Java virtuális gép (Java Virtual Machine) végzi, ami vagy interpretálja a bájtkódot, vagy natív gépi kódot készít belőle, és azt futtatja.

A program fejlesztéséhez a NetBeans IDE fejlesztőkörnyezetet választottam. A Java 8-as környezet szükséges a program futtatásához.

A feladat implementálásához a Java nyelvet választottam. A specifikáció során említett funkciókat könnyű vele megvalósítani. A fájlkezelést, az adatbázis kapcsolatot, a grafikus megjelenítést és az ezek közötti adatátvitelt egyaránt egyszerű felépíteni. Emellett fontos szempont volt, hogy széles körben használható legyen a megvalósított szoftver. A Java erőssége, hogy platformfüggetlen, csupán a Java Runtime Environment szükséges a programok futtatásához. Fejlesztési szempontból viszont elengedhetetlen a Java Development Kit, amely tartalmazza az előbb említett környezetet. Habár a Java Virtual Machine gyorsasága nem éri el a hardware közeli nyelvekét, ez nem jelent számottevő hátrányt a működésben.

2.2.1. JDBC

A Java Database Connectivity, röviden JDBC egy API a Java programozási nyelvhez, amely az adatbázishozzáférést támogatja. A JDBC definiálja az adatbázisok lekérdezéséhez és módosításához szükséges osztályokat és metódusokat, miközben igazodik a relációs adatmodellhez. Egyik fajtája a PostgreSQL JDBC interfész, amelyet a PostgreSQL fejlesztői adtak ki. A JDBC működési elvét az alábbi ábra szemlélteti.

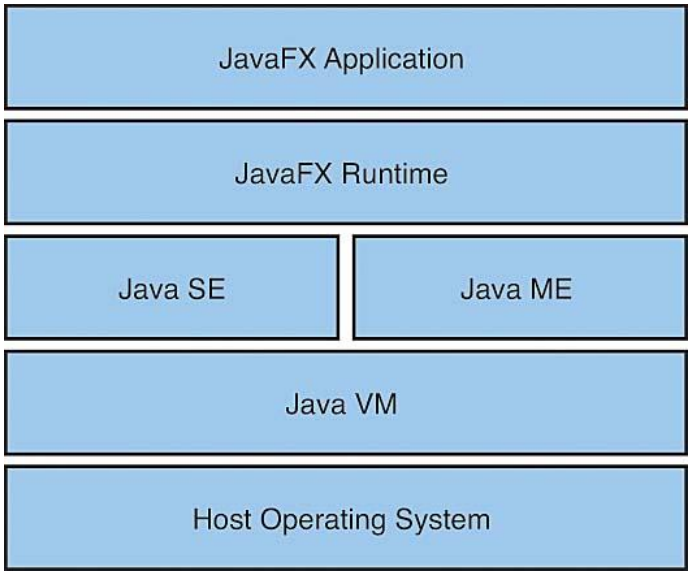


ábra JDBC architektúra

2.2.2. JavaFX

A JavaFX egy szoftver platform az asztali alkalmazások létrehozásához, amely a Swing mellett a Java Standard Edition alapértelmezett GUI könyvtára. A JavaFX applikációkat bármilyen asztali, mobil eszközön vagy böngészőben lehet futtatni. A grafikus felületet egy XML fájl definiálja, amelyet FXML fájlban tárolunk. Az XML fájl segítségével pontosan leírhatók az objektumok pozíciója és egyéb tulajdonságaik. Minden objektumhoz megadható egy egyedi azonosító, amellyel hivatkozhatunk rá a forráskódban. A keretrendszer támogatja az audió, a videó és az animáció implementálását is. A grafikus felület a Scene Builder program segítségével egyszerűen összeállítható és nem kell aggódnunk az XML fájl összeállításában, mert automatikusan legenerálja azt.

A JavaFX platform magában foglal egy fordítóprogramot, számos könyvtárat és fejlesztői eszközöket, beleértve a plug-in modulokat a különböző fejlesztő környezetekhez. Előnye abban nyilvánul meg, hogy Java platformon fut, emiatt a JavaFX alkalmazások ki tudják használni a Java adta lehetőségeket és együtt tud működni a többi interfésszel. A platform architektúráját az 5.1. ábra reprezentálja.



5.1. ábra JavaFX architektúra

2.2.3. Apache Commons IO

Az Apache Commons IO egy Java könyvtár fájlműveletekhez, melyet az Apache Foundation felügyelete alatt fejlesztettek. Osztályok sokaságát biztosítja a fejlesztők számára, hogy egyszerűbb, rövidebb és érthetőbb kód íródhasson a fájlok kezelésére. A feladatom során a fájlműveletek, különösképpen az olvasás rendkívüli szerepet játszanak a rendszer részeként, hiszen fájlok által beolvasott tartalommal kell frissíteni az adott adatbázist. A megvalósított szoftverben használhattam volna az alapértelmezett Java osztályokat a fájlműveletekhez, de ez a könyvtár jelentős mértékben megkönnyíti a munkát implementálás közben.

2.2.4. JUnit

A JUnit egy alapvető nyílt forráskódú keretrendszer a Java programozási nyelvhez, unit tesztelés céljából. Számos osztályt, annotációt és funkciót biztosít, hogy megfelelő teszteket írjon fejlesztő. A unit tesztelés támogatja a Test Driven Development (Teszt Vezérelt Fejlesztés) módszertan metodikáját, miszerint egy új funkció implementálása előtt megírjuk az ahhoz tartozó unit tesztet. Ezáltal a produkciós kód megírása nélkül is ismerhető a funkció elvárt működése. Ilyen formában a tesztek egyfajta specifikációként, dokumentációként is szolgálhatnak.

2.3. Git

A Git egy nyílt forráskódú, elosztott verziókezelő szoftver. Feladata, hogy a projekt fájljainak különböző verzióit tárolja és megossza a projekt felhasználói között. Azért döntöttem a Git használata mellett, mert implementálás közben nagyon hasznos, ha egy fájl korábbi verziójához szeretnék hozzáférni és használni. Véleményem szerint kevés olyan rendszer van, amely felveheti a versenyt a Gittel mind a hatékonyságban és mind az egyszerű kezelésben.

3. Rendszerkövetelmény

Ebben a fejezetben mutatom be a megvalósítandó rendszerrel szemben támasztott követelményeket. Röviden ismertetem a részfeladatok sajátosságait és funkcionalitásait, amelyekre szükség van a specifikált működés érdekében.

3.1. Specifikáció

Dolgozatom célja egy olyan ETL alkalmazás létrehozása, amely kapcsolatot biztosít két meglévő, strukturálisan jelentős módon eltérő adatbázis között. Az ETL mozaikszó az Extract, Transform, Load kifejezésekből származik, ami annyit tesz, hogy Kinyerés, Átalakítás, Betöltés. Az említett adatbázisok a US Department of Agriculture röviden USDA, Nutrient Database for Standard Reference adatbázisa és a Lavinia életmód-tükör mögötti MenuGene adatbázis.

Az eddigiek során csak manuális módon volt lehetőség a MenuGene adatbázisát frissíteni az USDA folyamatosan frissülő adataival. Ez meglehetősen sok munkát és időt igényelt. Annak érdekében, hogy MenuGene lépést tudjon tartani az USDA-val, egy olyan rendszert kell megvalósítani, amely automatizálva végzi el a frissítést. Az alkalmazás nagy segítséget nyújthat az adatbázis karbantartóinak és az életmód-támogató szakértői rendszer felhasználóinak egyaránt.

A megvalósítandó rendszer fő feladata, hogy egy migrációs folyamat képpen az USDA adatbázisán végrehajtott frissítéseket a Lavinia mögötti adatbázison is elvégezze. A kiépített kapcsolat segíti a cél-adatbázist, hogy naprakész legyen, és az aktuális adatokkal szolgáljon a különféle élelmiszerek, tápanyagok, tápanyagtartalmak és mértékegységek tekintetében. Ez azt jelenti, hogy az élelmiszerekhez minél pontosabb értékek társuljanak, amely során új adatok jelenhetnek meg, vagy az eddigi értékek módosulhatnak, vagy épp törlésre kerülhetnek.

A Lavinia jelenleg relációs adatbázist használ a receptek, az ételek, a tápanyagok és a további adatok tárolására, melyet a PostgreSQL relációsadatbázis-kezelő rendszerrel valósít meg. Az USDA adatbázisa viszont szöveges fájlok

formájában érhetőek el és tölthetők le. A rendszeres jelleggel publikált frissítések két külön részre bonthatók. Egy új revízió egyrészt tartalmazza az egész adatbázist, az újonnan frissített értékekkel, másrészt pedig tartalmazza az előző verzióval szembeni változtatásokat. Előbbi eset nyilvánvalóan nagyobb adathalmazt takar, hiszen a teljes adatbázist tartalmazza, szemben a verziók közötti változásokat összefoglaló fájlokkal, melyek nagysága töredéke a komplett adatbázisnak.

Az adatbázis-frissítéseként megvalósuló tranzakciót, tárolt eljárások formájában kell végrehajtani. Manapság szinte az összes adatbázis-kezelő rendszer támogatja a tranzakció kezelést. A tárolt eljárást tekinthetjük egy függvénynek, amelyet az adatbázis szerver fordít le, tárol és hajt végre. Használatának előnye, hogy jelentős mértékben lecsökkenti az adatforgalmat a kliens és az adatbázis szerver között. Emiatt alacsonyabb futási idővel fog végrehajtani a kívánt művelet. Továbbá teljes mértékben független a kliens programnyelvétől. Egységes felületet biztosít, így ha bármilyen módon meghívjuk a tárolt eljárást, ugyanazt a funkcionalitást érjük el.

Nem szabad figyelmen kívül hagyni, hogy a cél-adatbázison való frissítés végrehajtásának függetlennek kell lennie a folyamat időpontjától és számától. Elengedhetetlen, hogy bármikor és bármennyiszer futtatható állapotban legyen, továbbá támogassa az egymás utáni adatbázis-frissítéseket.

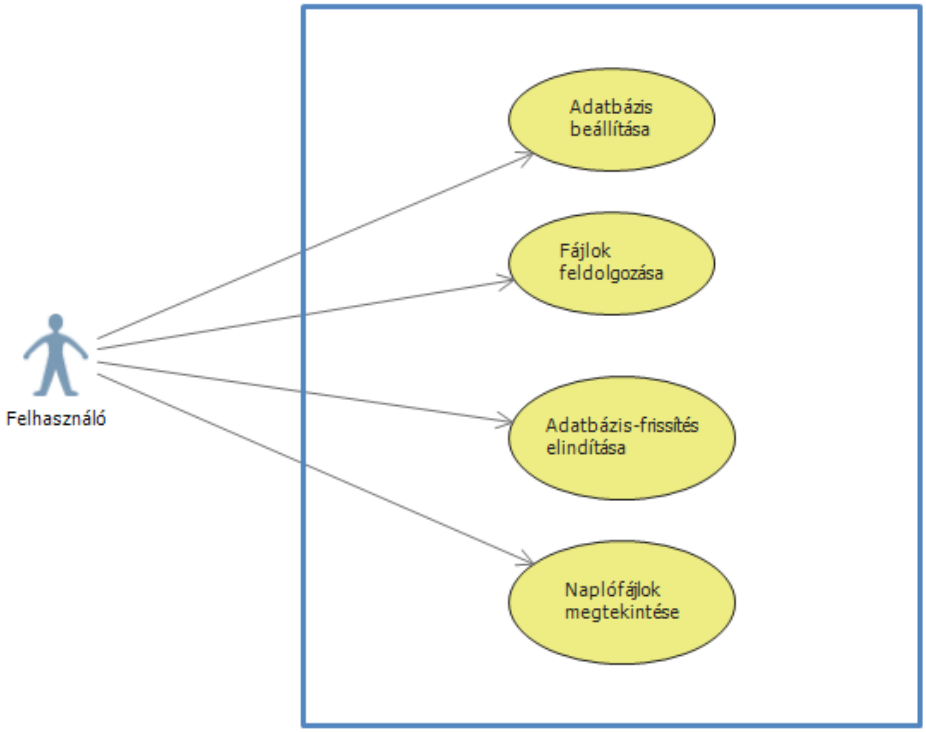
Elengedhetetlen a megfelelő hibatűrés a szoftver üzemeltetése során. Az adatbázis visszagörgetése magas prioritást kap bármilyen fellépő hiba esetén a frissítés során. A rendellenesség előtti műveleteket vissza kell állítani a korábbi állapotra. Ez azt jelenti, hogy az egész frissítési folyamatnak problémamentesen kell lefutnia, ellenkező esetben az adatbázis visszaáll az előző állapotára. A visszagörgetést mind a szoftver oldalon és mind az adatbázis oldalon meg lehet valósítani.

A szoftver nélkülözhetetlen részét képezi az adatbázis-frissítés folyamatának naplózása valamilyen formában. A jelentésnek szöveges formátumban kell készülnie úgy, hogy könnyen értelmezhető legyen a felhasználó számára. Tartalmaznia kell egyaránt a sikeres és sikertelen műveletek kimenetelét, miszerint az esetleges későbbi hibakeresés alkalmával könnyű dolga legyen a

felhasználónak a fennálló problémák kijavítására. Legegyszerűbb megoldás, ha egy közismert karakterkódolással ellátott szöveges fájlokba mentjük a naplófolyamatokat. A naplófájl felépítésének a lehető legegyszerűbbnek kell lennie a könnyű olvashatóság és értelmezhetőség kedvéért.

A felhasználóbarát működés érdekében szükség van egy könnyen kezelhető és átlátható grafikus felhasználói felületre, amely arra szolgál, hogy tájékoztassa a felhasználót a műveletek kimeneteléről. A grafikus interfész fő feladata, hogy a felhasználóval valós időben közvetítse a szöveges fájlok feldolgozásának folyamatát, illetve az adatbázis-frissítés eredményét, mindezt jól látható és értelmezhető megjelenési formában.

A specifikációból pontosan kiderül, hogy az alkalmazás egy adminisztrátori szerepkört betöltő felhasználónak készül, akinek hozzáférési joga van a Lavinia adatbázisához. Egyéb jogosultságú aktorok nem kapnak szerepet. A felhasználóhoz kapcsolódó főbb használati eseteket az alábbi use case diagramm mutatja be.



ábra Használati eset diagram

3.2. Adatbázisok ismertetése

A specifikációban említett adatbázisok meghatározottak voltak számomra, így új adatbázis létrehozására nem volt szükség. Ebből az okból kifolyólag csak a meglévő adatbázisok felépítését és struktúráját kellett megismernem, valamint az USDA adatbázis frissítésének koncepcióját. Mindkét adatbázis a relációs adatmodell elve alapján épül fel.

3.2.1. Relációs adatbázis

A relációt egy táblázattal lehet szemléltetni, amely sorról sorra tárolja a kívánt adatokat. A relációs adatbázis pedig ilyen relációk halmaza. A relációk egyedi nevet kell, hogy viseljenek ugyanúgy, mint az oszlopok egy reláción belül. A reláció architektúrája azonos struktúrájú rekordokat fejez ki, ezen sorok sorrendje nem fontos. Ezekben a sorokban tároljuk a logikailag összefüggő adatokat, viszont kettő vagy több megegyező sor nem fordulhat elő, ezzel elkerülve a redundanciát. Oszloponként csak meghatározott típusú mennyiségek tárolhatóak, például numerikus, szöveges vagy dátum formátumban. Egy relációnak megadhatunk kényszereket is, amelyek korlátozzák az adott adathalmazban előforduló értékeket.

A relációs adatbázis-kezelő rendszerek (angolul Relational Database Management System) kizárólag a relációs adatmodell alapján vannak megvalósítva. Ezen rendszerek segítségével építhetjük fel, kezelhetjük adatbázisainkat. Egy adatbázis-kezelő rendszer három rétegből épül fel. Legalsó réteg a fizikai réteg. A fizikai réteg definiálja az adat fizikai tárolását az adott hardver-egységen. Felette helyezkedik el a logikai réteg, amely a logikai kapcsolatokat határozza meg a tárolt információk között. Végül a legfelső szint a fogalmi vagy alkalmazási réteg. Ez biztosítja az adatokhoz való hozzáférési felületet a felhasználók részére. A relációs adatbázis-kezelő rendszerek nem valósulhatnak meg az ACID tulajdonságok nélkül. Ezen tulajdonságok az atomicitás (atomicity), a konzisztencia (consistency), az izoláció (isolation) és a tartósság (durability). Ezáltal rendszer garantálja, hogy több művelet egy műveletként hajtódjon végre, nem valósulhatnak meg részlegesen és ezek után is konzisztens állapotban marad. Tranzakciók egymástól függetlenül, elszeparáltan is

futtathatóak és sikeres mentés után, hardver vagy egyéb hiba esetén is tartós marad a változás.

Néhány relációs adatbázis-kezelő rendszer, amelyek közt van kereskedelmi, illetve nyílt forráskódú is:

- Oracle
- MS SQL Server
- DB2
- PostgreSQL
- MySQL
- SQLite

3.2.2. USDA adatbázisa

Az USDA National Nutrient Database for Standard Reference adatbázisa a legfőbb adatforrás az élelmiszer összetételek tekintetében az egész Egyesült Államokban. A 90-es évek elejétől elektronikus formában is elérhető adatbázisuk. Ez az adatbázis minden egyes revíziónál letölthető fájlok alakjában találhatóak meg az USDA honlapján. Egy revízió publikálásánál egyrészt elérhető az újonnan frissült teljes adatbázis, amely tartalmazza a régi és az új adatokat egyaránt. Másrészt letölthetőek csak az előző verzióhoz képesti változások, amely magába foglalja az újonnan hozzáadott, megváltoztatott vagy törölt adatokat. Ezek mellett letölthető egy rövidített adatbázis szerkezet is szöveges illetve Microsoft Excel fájl formátumban. Viszont ez nem tartalmazza az összes tápanyagot és ily módon nem felhasználható a rendszer számára. A számomra felhasználható fájlok egyaránt rendelkezésre állnak ASCII (ISO/IEC 8859-1 szabvány) formátumban kódolva és Microsoft Access adatbázis formájában is.

Az USDA által használt relációs adatbázis főbb tábláit a következőkben mutatom be.

Mezőnév	Típus	Leírás
NDB_No	alfanumerikus	élelmiszer egyedi azonosítója
FdGrp_Cd	alfanumerikus	élelmiszer-csoport kódja
Long_Desc	alfanumerikus	leírás
Shrt_desc	alfanumerikus	rövidített leírás
ComName	alfanumerikus	egyéb megnevezés
ManufacName	alfanumerikus	gyártó megnevezése
Survey	alfanumerikus	tanulmányokban felhasznált-e
Ref_desc	alfanumerikus	fogyasztásra alkalmatlan összetevő leírása
Refuse	numerikus	fogyasztásra alkalmatlan összetevő százalékban kifejezve
SciName	alfanumerikus	élelmiszer tudományos neve
N_Factor	numerikus	nitrogénből proteinbe való átalakítás tényezője
Pro_Factor	numerikus	proteinből kalória számítás tényezője
Fat_Factor	numerikus	zsírból kalória számítás tényezője
CHO_Factor	numerikus	szénhidrátból kalória számítás tényezője

táblázat Food Description tábla

A Food Description tábla magába foglalja az élelmiszerek különböző megnevezéseit, jellemzéseit vagy éppen a fogyasztásra alkalmatlan részeik mértéket az egyéb tudományos adatok mellett.

Mezőnév	Típus	Leírás
NDB_No	alfanumerikus	élelmiszer egyedi azonosítója
Nutr_No	alfanumerikus	tápanyag egyedi azonosítója
Nutr_Val	numerikus	mennyiség 100 grammban
Num_Data_Pts	numerikus	analízisek száma
Std_Error	numerikus	átlagtól való eltérés
Src_Cd	alfanumerikus	adattípust jelző kód
Deriv_Cd	alfanumerikus	meghatározás módja
Ref_NDB_No	alfanumerikus	referencia azonosító
Add_Nutr_Mark	alfanumerikus	vitamin vagy ásvány hozzáadásának megerősítése
Num_Studies	numerikus	tanulmányok száma
Min	numerikus	minimum érték
Max	numerikus	maximum érték
DF	numerikus	szabadságfokok
Low_EB	numerikus	alsó hibahatár
Up_EB	numerikus	felső hibahatár
Stat_cmt	alfanumerikus	statisztikai megjegyzés
AddMod_Date	alfanumerikus	hozzáadás vagy módosítás dátuma
CC	alfanumerikus	adat minőségi tényezője

táblázat Nutrient Data tábla

Az élelmiszerekre vonatkozó tápanyag-értékeket és információkat a Nutrient Data tábla tárolja.

Mezőnév	Típus	Leírás
Nutr_No	alfanumerikus	tápanyag egyedi azonosítója
Units	alfanumerikus	mértékegység
Tagname	alfanumerikus	rövidített név
NutrDesc	alfanumerikus	tápanyag neve
Num_Dec	alfanumerikus	helyiérték szám kerekítéshez
SR_Order	numerikus	sorrend

táblázat Nutrient Definition tábla

A különféle tápanyagokat leíró adatok a Nutrient Definition táblában találhatóak meg.

Mezőnév	Típus	Leírás
NDB_No	alfanumerikus	élelmiszer egyedi azonosítója
Seq	alfanumerikus	sorszám
Amount	numerikus	mennyiség
Msre_Desc	alfanumerikus	mértékegység leírása
Gm_Wgt	numerikus	súly grammban kifejezve
Num_Data_Pts	numerikus	adatpontok száma
Std_Dev	numerikus	eltérés

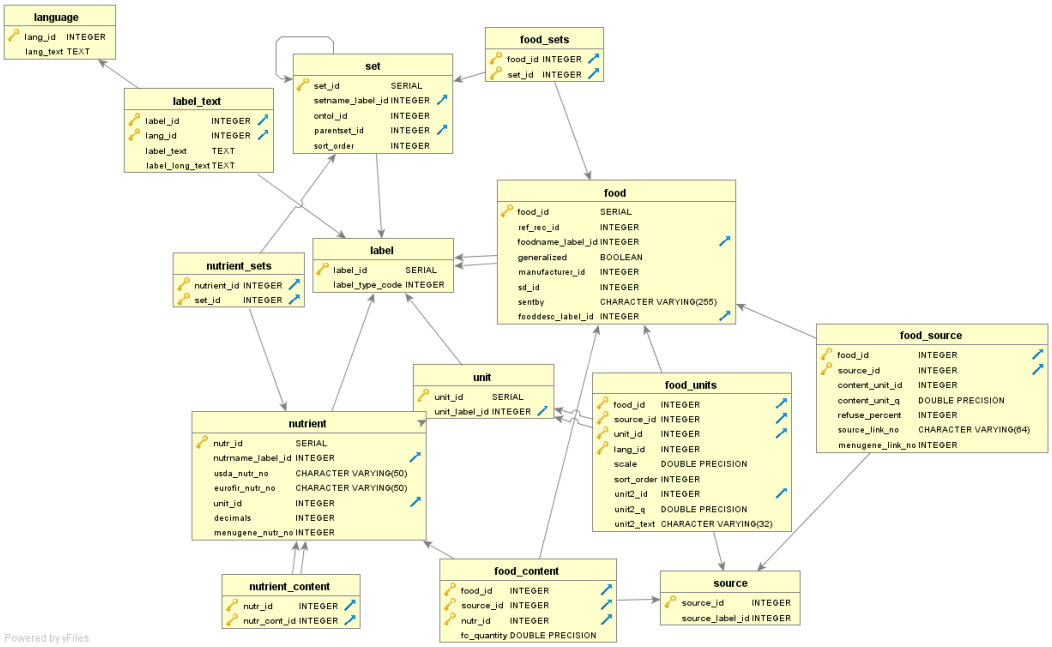
táblázat Weight tábla

A Weight tábla tartalmazza minden egyes élelmiszerhez a különféle közönséges mértékegységek tömegét grammnyi pontossággal kifejezve.

3.2.3. Lavinia adatbázisa

A Lavinia az említett MenuGene szakértői rendszer adatbázisát használja fel. Ez szintén a relációs adatmodell alapján készült a PostgreSQL relációs adatbázis-kezelő rendszer segítéségével. Sémája előre meghatározott volt számomra, így a létrehozni kívánt szoftvernek ehhez igazodnia kell. Az adatmodell felépítése és szerkezete, a kapcsolatok módosítása nem megengedett, kizárólag adatmanipulációt hajthat végre az alkalmazás. A rendszer megtervezése előtt

elsőként ennek tanulmányozásával foglalkoztam. A rendelkezésemre álló adatbázis struktúráját a következő ábra mutatja be.



ábra ER diagram

Ezek közül ki is emelném a legfontosabbakat. Az egyik olyan táblája, amely összefügg az USDA adatbázisával, a food_source tábla. Többek között ez tárolja az USDA-ból átvett élelmiszerek egyedi azonosítóit. Illetve ezen termékek összetevői hány százalékban alkalmatlanok emberi fogyasztásra. Továbbá meghatározza, hogy dekagrammban értendők a tápanyagokhoz rendelt tömeg értékek. A food tábla írja le az egyes élelmiszereket, amelyekre akár receptek is hivatkozhatnak a későbbiek során. A másik reláció, mely kapcsolatba állítható az USDA-val a nutrient tábla, mely a tápanyagokat reprezentálja. Szintén megtalálható bennük az USDA-ból átvett tápanyagok egyedi azonosítója. További fontos relációk között van a food_content és a food_units. Ezek kapcsoló tábla szerepét töltik be a struktúrában. Előbbiben kerül tárolásra, hogy melyik élelmiszer melyik tápanyagokat milyen mennyiségben tartalmazza. Ez a mennyiség 100g termékben lévő tápanyag mennyiségét mutatja meg. Utóbbi pedig egy egységnyi étel tömegét definiálja dekagrammban.

A két adatbázis felépítésének tanulmányozása után kiderül, hogy jelentősen eltér a szerkezetük. Az Egyesült Államokban megszokott angolszász

mértékegységrendszer helyett, az USDA adatbázisa is metrikus, más néven SI mértékegységrendszert használ. Így az efféle átváltások könnyen kivitelezhetőek. Az USDA által tárolt mennyiségek grammban értendők, míg a Lavinia adatbázisa dekagrammot használ.

4. Tervezés

Az ETL mozaikszó nem egy ismeretlen fogalom az informatika világában. Olyan cégek és vállalatok számára létfontosságú lehet ezen alkalmazások használata, amelyek többek közt az üzleti intelligenciával (angolul Business Intelligence, röviden BI) is foglalkoznak, mint például bankok vagy nagyvállalatok. Ezek az alkalmazások lehetővé teszik, hogy információt nyerjünk ki és felhasználásukkal, olyan üzleti döntések születhetnek, amelyek növelik az üzleti teljesítményt. Belső üzleti folyamatok elemzése és optimalizálása után jelentős profit érhető el és előnyhöz juttathat a piaci versengésben. Az üzleti intelligencia jövője nem kétséges, hiszen múltbeli és jelenlegi elemzések is végezhetőek, valamint a jövőre tekintve is készülhetnek előrejelzések. Számos alkalmazási területe van. Ezek közé tartozik a riport és dashboard készítés, statisztikai elemzések létrehozása, valamint az adatbányászat.

4.1. Rendszer terve

A tervezés folyamata rendkívüli szerepet játszik a rendszer megvalósítása előtt. A rendszer tervezési fázisához hozzátartozik a kliens oldal és a szerver oldal megtervezése mellett a két oldal viszonyának koncepciója is. Továbbá nem elhanyagolható a grafikus felhasználói felület terve, hiszen a felhasználó ezen keresztül kommunikál az alkalmazással, különféle interakciókkal.

A szoftver több modulból fog felépülni. Ezek a modulok a későbbi implementációban reprezentálják a specifikációban említett szegmenseket és funkcionalitást. A rendszer főbb funkcióit megvalósító alkotórészei a következők:

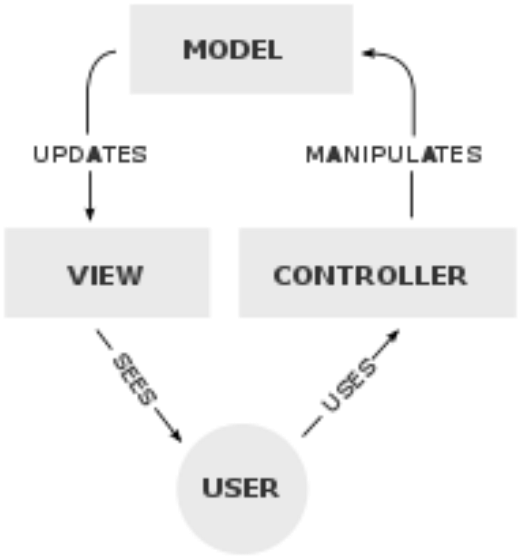
- adatbázis interfész
- grafikus felhasználói interfész
- modell osztályok
- vezérlő osztályok
- naplózás

Java nyelven nem okoz gondot elkülöníteni a különböző feladatokat ellátó modulokat. Az azonos feladatokért felelős osztályokat egy könyvtárba, úgy

nevezett Java package-be lehet szervezni. Ily módon könnyebben átláthatók az implementált részegységek.

4.2. Modulok

A modulok tervezésénél törekedtem az MVC, vagyis a Model-View-Controller (magyarul modell-nézet-vezérlő) architektúra elvének betartására. Ez azt jelenti, hogy az adatot függetlenné tesszük az azt megjeleníteni képes felületektől. Ez a vezérlő bevezetésével lehetséges. A modell réteg reprezentálja az információt, amelyet az alkalmazás kezel vagy tárol. A nézet a megjelenítő szerepet tölti be, amely a kívánt alakban ábrázolja az adatot. Ez rendszerint egy grafikus felület a felhasználó számára. A vezérlő réteg pedig a felhasználótól érkező interakcióra reagálva módosíthatja az adatot. Ennek egyik előnye, hogy a rendszer több rétegből épül fel, így könnyen átlátható és értelmezhető a szoftver szerkezete. A logikailag nem összetartozó szegmensek el vannak különítve egymástól. Ezáltal a jövőbeni módosítások csak az adott összetevőt érintik. Továbbá nagyobb rendszerek esetén, egyszerre több fejlesztő is dolgozhat az egyes rétegeket megvalósítva, egymástól független módon. Az én programom esetében az interfészek és a vezérlő osztályok felelnek meg a controllernek. A grafikus felhasználói felület tesz eleget a view rétegnek. Valamint a leíró osztályok egyeznek meg a model réteggel. A Model-View-Controller tervezési minta elvét a következő ábra mutatja be.



ábra MVC architektúra

4.2.1. Adatbázis interfész

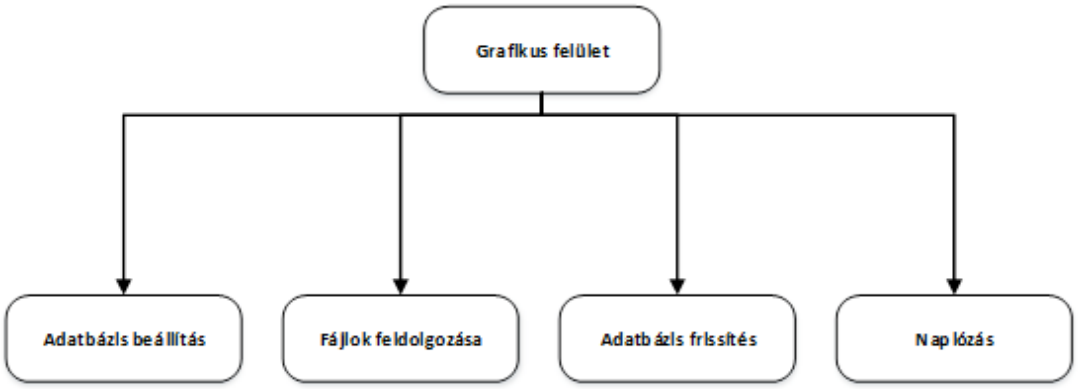
Az adatbázis interfész egy gyakran használt komponens a rendszerben. Az adatbázissal való kommunikáció érdekében megfelelő kapcsolatot kell kiépíteni azzal, akár több esetben is. Ez a modul szolgál az adatbázis-frissítés alapjául. Kapcsolatkiépítés után biztosítania kell az adatmanipulációt. Ez azt jelenti, hogy a kommunikáció során az adatbázis-szerveren lévő tárolt eljárásokat hívja meg a szükséges paraméterekkel. Ezeket a tárolt eljárásokat előbb létre is kell hozni, amely szintén az interfész feladata. Hiszen kezdetben a szerver nem rendelkezik ezen funkciókkal, viszont későbbi futtatásoknál ez nem okoz gondot.

A rendszerkövetelmények alapján hamar kiderült számomra, hogy a céladatbázis frissítését azokkal a fájlokkal kell végrehajtanom, amelyek az USDA adatbázis verziói közötti változásokat foglalják össze. A frissítési folyamat során a felhasználni kívánt fájlokat három kategóriába sorolhatjuk. Név szerint ezek a FOOD, a NUTR és a WGT kategóriák. Ezek foglalják össze az élelmiszerekre, a tápanyagokra és az egységnyi súlyokra vonatkozó információkat. A fájlnevek meghatározzák, hogy milyen változást tárolnak. Vannak olyanok, amelyek új adatokat tartalmaznak és hozzá kell adni az adatbázishoz, mint az ADD_FOOD.txt, az ADD_NUTR.txt és az ADD_WGT.txt fájlok. Emellett vannak olyan fájlok, amelyek már létező adat változását írják le. Ezek a CHG_FOOD.txt, a CHG_NUTR.txt és a CHG_WGT.txt fájlok. Végül a törölni kívánt adatokat a DEL_FOOD.txt, a DEL_NUTR.txt és a DEL_WGT.txt fájlok fedik le. A DEL_FOOD.txt és DEL_NUTR.txt fájlok kivételével, szerkezeti szempontból a FOOD kategóriájú fájlok megegyeznek az USDA adatbázisában említett Food Description tábla felépítésével, mint ahogy a NUTR azonos a Nutrient Data táblával és a WGT a Weight tábla struktúrájával. Továbbá nem szabad figyelmen kívül hagyni, hogy egy-egy USDA frissítés során előfordulhatnak az ADD_NDEF.txt és a CHG_NDEF.txt fájlok is. Ezek új tápanyagok felvitelét vagy a meglévők változtatását írják le és struktúrájuk megegyezik a Nutrient Definition táblával.. Ezen fájlokra alapozva kell a tárolt eljárásokat a megfelelő paraméterlistával megtervezni.

4.2.2. Grafikus felhasználói interfész

A grafikus felhasználói felület teremti meg a kapcsolatot a szoftver és a felhasználó között. A felhasználó interakciói befolyásolják a rendszer működését. Ezt a komponenst az egyik legmodernebb technológiával, a JavaFX-el valósítom meg. Könnyen kezelhető, ami jelentősen elősegíti az implementálás menetét.

Felhasználói szempontból rendkívül fontos a grafikus felület létrehozása, mert ez alapozza meg az első benyomást a szoftverrel kapcsolatban. Emiatt próbáltam a lehető legegyszerűbb és legkönnyebben értelmezhető felületet felépíteni. A felület négy fülből épül fel, amelyek a különböző funkcionalitásokat hivatottak megteremteni. Ennek elrendezését az xx.yy. ábra szemlélteti.**(forrás)**



ábra Grafikus felületen elhelyezett funkcionalitások

4.2.3. Modell osztályok

Modell osztályok segítségével a feldolgozott szöveges fájlok adatainak megfelelő leíró osztályokat valósítok meg az implementálás alatt. Ezen osztályok a fájlok szerkezetét testesítik meg. Ezekkel az osztályokkal könnyű reprezentálni és átadni az adatokat a különböző vezérlő osztályoknak.

4.2.4. Vezérlő osztályok

A vezérlő osztályok fogják össze az egész rendszert. Összeköttetést létesítenek a modell osztályok és az interfészek között. Segítségükkel valósulhat meg a megfelelő adatáramlás az egyes komponensek között. Ezen felül vezérlő osztályok

valósítják meg és irányítják a folyamatokat. Ebbe bele tartozik a szöveges fájlok feldolgozása és az adatbázis-frissítés tranzakciójának vezérlése.

4.2.5. Naplózás

Az adatbázison végrehajtott műveletek mindegyikéről készül jelentés. Ennek során a program időbélyeggel ellátott fájlt hoz létre, amely sorai az elvégzett műveletek sikerességét vagy sikertelenségét szemléltetik. A naplófájlokból így kiderül, hogy mi és miért okozta a rendellenességet.

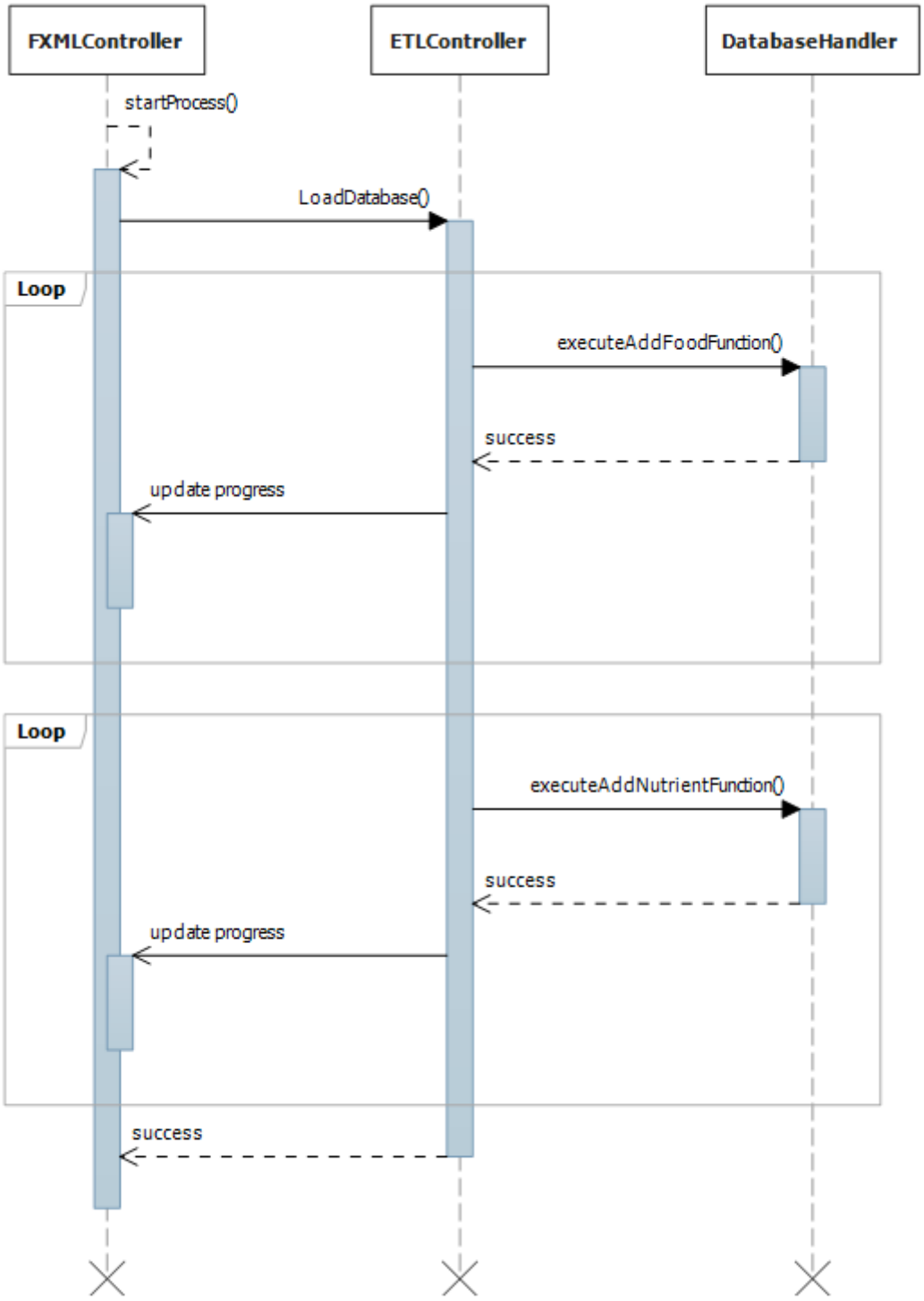
4.3. Frissítési folyamat terve

Az adatbázisok megismerése után az adatok frissítésének sorrendjének megtervezése volt a feladatom. Ehhez a céladatbázisban fellelhető táblák közötti kapcsolatokat kellett elemeznem. Ugyanis a külső kulcs kényszerek miatt egy adott sorrendben kell végrehajtani az INSERT és DELETE SQL utasításokat.

Ideális esetben, ha minden fájltípus megtalálható az USDA frissítésében, az adatbázis-frissítési folyamata az új tápanyagok importálásával kezdődik. Ez azért fontos, mert a későbbi adatok hivatkozhatnak az újonnan importált tápanyag egyedi azonosítójára. A teljes folyamat szisztémáját a következő pseudo kód reprezentálja.

tranzakció létrehozása
ciklus az ADD_NDEF listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az CHG_NDEF listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az ADD_FOOD listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az ADD_NUTR listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az ADD_WGT listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az CHG_FOOD listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az CHG_NUTR listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az CHG_WGT listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az DEL_WGT listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az DEL_NUTR listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
ciklus az DEL_FOOD listáján
    lista elemmel tárolt eljárás hívása
ciklus vége
tranzakció véglegesítése

Az xx.yy. ábrán látható szekvencia diagram ugyanezt a folyamatot mutatja be, amely az adatbázis-frissítés csak egy részletét képviseli.



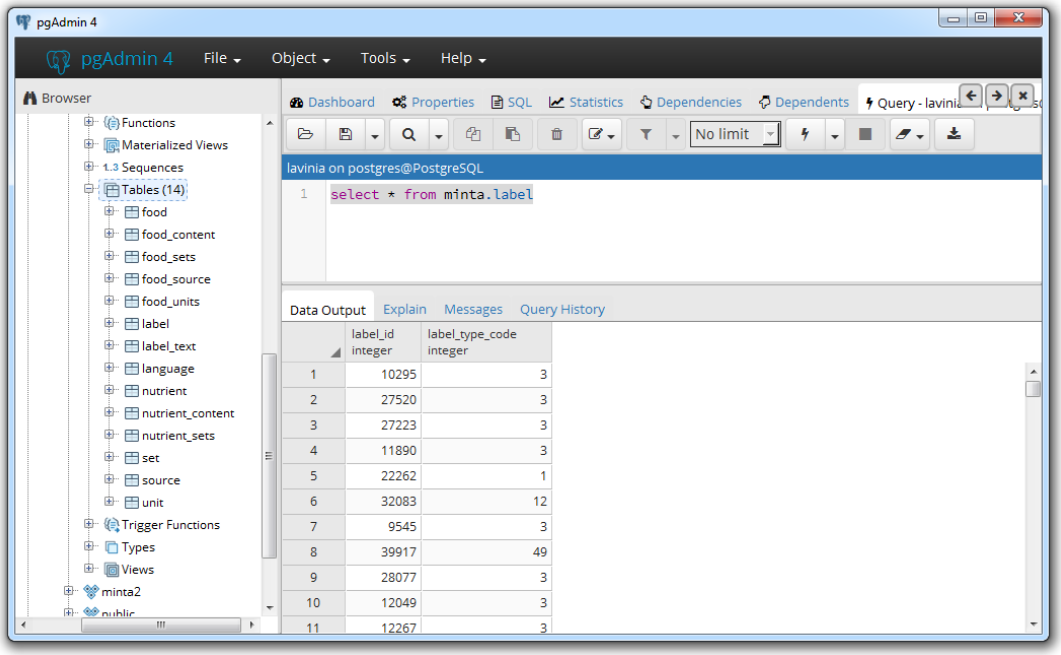
ábra Szekvencia diagram

A diagrammon jól látható, hogy ciklikus hívás történik. A rendszer a fájlok feldolgozásából származó adatok mindegyikére meghívja a megfelelő tárolt eljárást.

4.4. Lokális környezet

Tervezés során rendelkezésemre állt a Lavinia adatbázis adatainak egy kis halmaza, ami pár száz rekordot jelent. A kapott ábrák és séma leírások mellett ezek az adatok is segítettek az adatbázis tanulmányozásában és szerkezeti felépítésének megértésében.

Annak érdekében, hogy mélyebben megismerjem az adatbázist, fel kellett építenem azt egy lokális környezetben. Tesztelhetőség szempontjából fontos, hogy hasonlítson az éles helyzetben használt adatbázisra. Ehhez a követelményekben meghatározott PostgreSQL relációs adatbázis-kezelő rendszer fejlesztői által kiadott, ingyenesen elérhető PgAdmin 4 nevű programot telepítettem. A szoftverrel való megbarátkozás könnyedén ment, hiszen tanulmányaim során megismerhettem más hasonló relációs adatbázis-kezelő rendszereket is. Ezek után a program segítségével létrehoztam az üres adatbázist a kapott séma alapján, majd a mentett adatokkal feltöltöttem. Létrehozáskor minimális hibát tapasztaltam. Ezek főleg hivatkozások voltak olyan táblákra, amelyek nem voltak a séma részei. Ennek az lehet az oka, hogy az adatbázis azon táblái álltak rendelkezésemre, amelyeket a dolgozatom során kell felhasználnom. A hibákat okozó külső kulcs kényszerek megszüntetése után nem lépett fel egyéb hibajelenség. A végrehajtott módosítások nincsenek hatással a megvalósított adatbázis-frissítés folyamatára. A PgAdmin 4 grafikus kezelőfelületét az alábbi ábra mutatja be.



ábra PgAdmin 4 kezelőfelület

Az adatbázis létrehozásához és adatokkal való feltöltését SQL parancsok segítségével hajtottam végre. Az SQL a Structured Query Language, magyarul strukturált lekérdezőnyelv elnevezésből származik. Használatát a relációs adatbázis-kezelő rendszerekhez köthetjük. A nyelv alkalmazásával képesek vagyunk az adatot definiálni, kezelni, lekérdezni és vezérelni a kívánt adatbázison. Szintaxisa logikusan épül fel, könnyen megérthető és átlátható. A PgAdmin felületén megadott különböző SQL parancsokkal vizsgáltam meg a kapott adatbázist.

Ezek után az adatbázis-frissítésre szánt fájlokat vettem szemügyre. Tervezés során el kellett döntenem, hogy a frissítés folyamatát hogyan végezze el a rendszer. Egyik megvalósítási ötlet az volt, hogy a fájlokban lévő élelmiszerekkel összetartozó tápanyag értékeket és tömeg értékeket összekapcsoljam egymással és az így kapott adatokat használjam fel a művelethez. Ennek előnye, hogy az egybevonás eredménye képpen kevesebb alkalommal kell meghívni a tárolt eljárásokat. Viszont nagy adatmennyiség esetén az összevonás folyamata lelassíthatja a szoftvert működését. Továbbá a szöveges fájlok nem feltétlenül tartalmazzák az élelmiszerekhez tartozó egyéb adatokat. Előfordulhat, hogy csak a tápanyag értékeket találjuk meg vagy csak a tömegre vonatkozó információkat.

Másik reális ötlet pedig az volt, hogy a fájlokat egymástól függetlenül, egyesével felhasználva valósuljon meg a frissítés. Ennek előnye abban rejlik, hogy rendkívül egyszerű. Ezáltal pontoson átlátható a rendszer működése és monitorozható, hogy éppen melyik adatot használja fel. Hátránya viszont az, hogy teljes mértékben a fájlokra hagyatkozik, és nem hajt végre ellenőrzést az összetartozó adatokra tekintve.

Az automatizált adatbázis-frissítés által megoldani kívánt probléma méretét jól reprezentálja az USDA által kiadott frissítésekben található fájlok sorainak száma. Példa képpen a SR28-as nevű revízió fájljaiban fellelhető adatok számát mutatom be.

Fájl	**Darabszám**
ADD_FOOD	370
ADD_NUTR	35 899
ADD_WGT	538
CHG_FOOD	753
CHG_NUTR	14 271
CHG_WGT	341
DEL_FOOD	199
DEL_NUTR	11 426
DEL_WGT	328

táblázat Fájlok adatainak darabszáma

Az ábra jól mutatja, hogy tízezres nagyságrendekben kell gondolkodni. Végül arra jutottam, hogy a második ötlet megvalósítása mellett döntök. Ennek oka, a koncepció egyszerűsége, illetve az adathalmaz meglehetősen nagy számossága, amelyet a rendszer felhasznál.

5. Megvalósítás

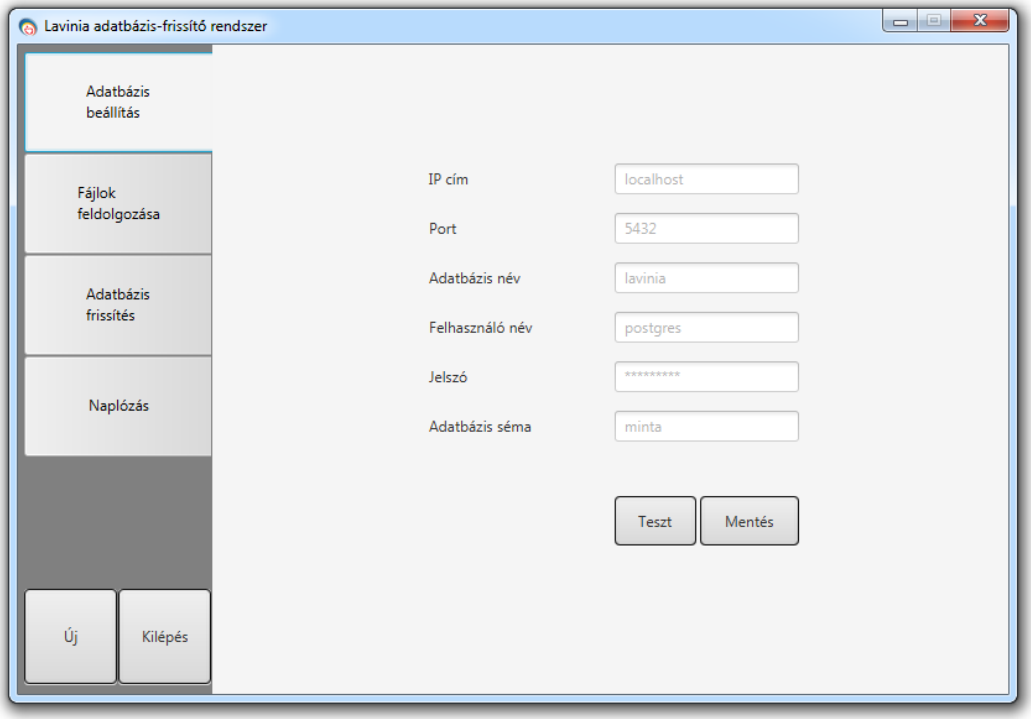
A tervezési fázis után maga a megvalósítás következik. Dolgozatom ezen fejezetében ismertetem az implementált rendszert, beleértve a grafikus felületet és a különböző funkcionalitásokat.

5.1. Felhasználói felület

A grafikus felhasználói felület az ismertetett JavaFX technológiával készült. Más grafikus felületet megvalósító keretrendszerekkel szemben, a kezelőfelületet egy XML kiterjesztésű fájl írja le. Ennek megírására különösebb programozási ismeret nem szükséges és egy egyszerű szövegszerkesztő alkalmazással kivitelezhető. Szerkezete hierarchikus, és könnyedén olvasható. A Scene Builder nevű programnak köszönhetően nem kell ennek létrehozásával foglalkoznunk. A szoftverben úgy nevezett Drag & Drop módszerrel tudjuk behúzni a kívánt JavaFX komponenseket a felületre. A felület mentésével a Scene Builder legenerálja számunkra az XML fájlt. Ezáltal könnyen, gyorsan és hatékonyan tudunk létrehozni akár bonyolult grafikus interfészeket is.

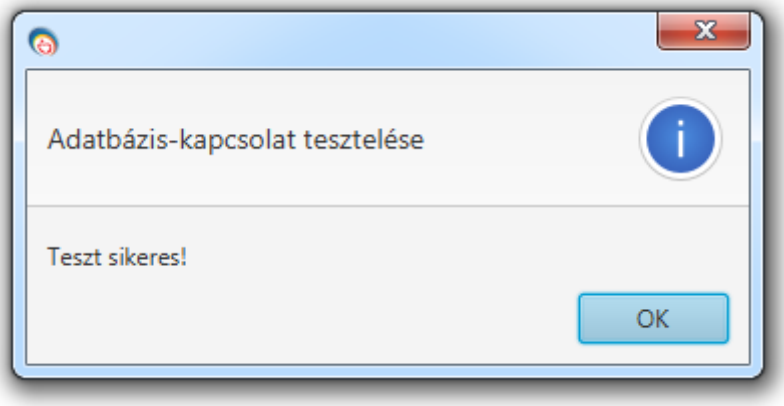
5.2. Adatbázis beállítása

A program indításával az adatbázis beállításával találjuk szembe magunkat. Ezt megelőző külön bejelentkező ablak létrehozását nem láttam szükségesnek. Hiszen csak abban az esetben férhetünk hozzá a Lavinia adatbázisához, ha ismerjük annak attribútumait. Így ez is értelmezhető egyfajta biztonsági pontnak. Ezek az attribútumok a hoszt ip címe, a port száma, az adatbázis neve, a séma neve, valamint az adatbázishoz társított felhasználó neve és a hozzátartozó jelszó. Ideális esetben az éles rendszerben, kizárólag az adatbázis adminisztrátorai számára állnak rendelkezésre ezen információk.

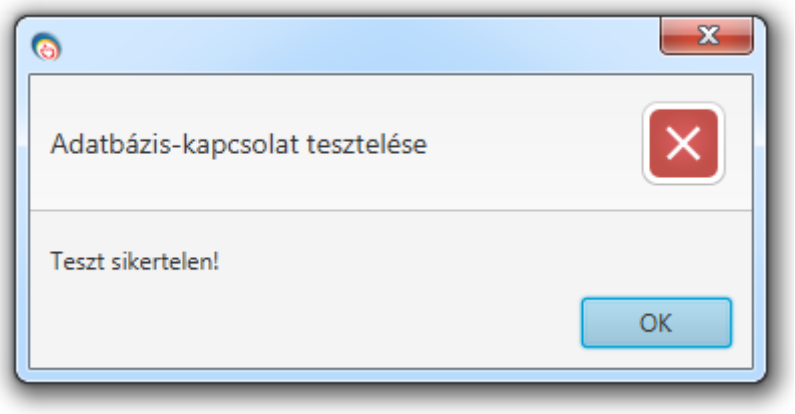


ábra Adatbázis beállításának felülete

A „Teszt” gomb megnyomásával lehetőségünk van lefuttatni egy kísérletet a megadott paraméterekkel, hogy azok helyesek-e és a rendszer képes-e csatlakozni az adatbázishoz. A teszt kimenetele helyes vagy helytelen lehet. Ezt a szoftver a megfelelő rendszerüzenetek egyikével jelzi is a felhasználó számára. A teszt csatlakozás végeredményét az xx.yy. és az xx.yy. ábra mutatja be.



ábra Rendszerüzenet: sikeres teszt



ábra Rendszerüzenet: sikertelen teszt

A „Mentés” gomb megnyomásával a rendszer szintén leteszteli, hogy képes-e csatlakozni az adatbázishoz a megadott adatok alapján. Ennek végeredményéről a felhasználó ugyanúgy rendszerüzenet formájában kap visszajelzést. Sikeres teszt esetében a rendszer elmenti az adatokat és a felhasználót átirányítja a következő fülre.

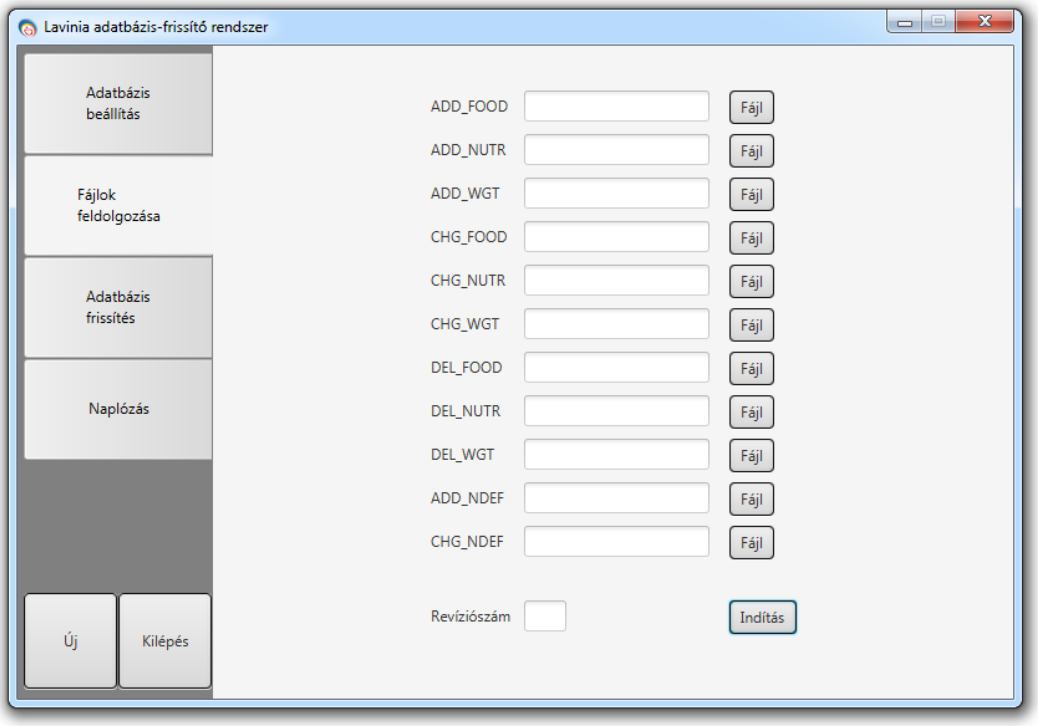
Ezen funkciók megvalósításához szükség van a kommunikációs kapcsolat létrehozására a szoftver, mint kliens és az adatbázis-szerver között. Az összeköttetés kiépítéséhez a Java Database Connectivity interfész feladata. Az interfész által definiált osztályok a `java.sql` csomagban találhatóak meg. Magát a kapcsolatot a `Connection` osztály reprezentálja. Az osztály segítségével többek között kinyerhetünk információkat az adatbázisunkról, SQL lekérdezéseket és tárolt eljárásokat hozhatunk létre, beállíthatjuk commitok automatizáltságát és a tranzakciós szinteket. Egy ilyen osztályba tartozó objektumot a `DriverManager` osztály `getConnection()` metódusával inicializálhatunk, amelyet felparaméterezhetünk adatbázisunk attribútumaival.

Ezután az objektum felhasználható a kliens oldalon, hogy SQL parancsokat hajtson végre az adatbázison. Az elvégzett műveletek után fontos a kapcsolat bontása a hibák elkerülése végett. Emellett az operációs rendszer feleslegesen ne tartsa fent a kapcsolatot és ne foglalja a memóriát, ha nem szükséges. Ezt a `close()` metódussal tehetjük meg, amit a `Connection` objektumon hívhatunk meg. Ha az adatbázis-szerveren a műveletek végrehajtása során bármilyen hiba

adódik, akkor azt *SQLException* osztály formájában kaphatjuk el és kezelhetjük le.

5.3. Fájlok feldolgozása

A Lavinia adatbázis-hozzáférés után az USDA adatbázis frissítését tartalmazó fájlokat kell feldolgozni. A funkcióhoz a grafikus felhasználói felületet úgy terveztem meg, hogy minden fájlt hozzá lehessen adni a rendszerhez. A címkék jelzik melyik helyre melyik fájlnak kell kerülnie. Fontos megjegyezni, hogy pontosan azt a fájlt adjuk meg, amelyet a címke mutat. Hiszen a fájlok szerkezete eltérő és mindegyiket más módon dolgozza fel a szoftver. Az állományokat az elérési úttal adhatjuk meg, amit a rendszer több féle képpen is támogat. Felhasználhatóság szempontjából a legegyszerűbb a Drag & Drop módszer. Ezt azt jelenti, hogy egy fájl menedzselő program segítségével a kívánt fájlt behúzzuk a megfelelő címkével ellátott szövegdobozba. A szoftver érzékeli a Drag & Drop eseményt és lekéri a fájl teljes elérési útját. Ennek megvalósításához a célterületnek előbb egy Drag Over esemény által érzékelnie kell, hogy a felhasználó egy állományt akar behúzni. Ezután a Drag Dropped esemény bekövetkezésekor kinyerhető a fájlhoz tartozó információ, mint például az elérési út. Majd a rendszer megjeleníti a kapott elérési utat a szövegdobozban. Másik módszer a címkékhez társított „Fájl” gomb megnyomásával végezhető el. A gomb megnyomása után egy fájl választó ablak jelenik meg a képernyőn, amelyen a fájlt tartalmazó könyvtárba navigálhatunk és választhatjuk ki a feldolgozni kívánt fájlt. Végezetül a felhasználó saját maga is begépelheti az elérési utat, de nyilvánvalóan ez nem egy felhasználóbarát megoldás. Ebből az okból kifolyólag implementáltam az előző eljárásokat. A felületen helyet kapott még a revízió szám megadása is. A felhasználó itt határozhatja meg, hogy éppen melyik USDA verzió általi frissítést szeretné végrehajtani. Ennek a mezőnek a kitöltése kötelező. Hiányában a fájlok feldolgozását nem végzi el a program és üzenet formájában közli a felhasználóval a mező kitöltését.



ábra Fájlok feldolgozására szánt felület

Mivel szöveges állományokról van szó txt kiterjesztésben, ezért Java nyelven nem okoz gondot a fájlok beolvasása. Ehhez az Apache Commons IO könyvtárat használtam. A könyvtár osztályai nagyon hasznosnak bizonyulnak a fájlkezelés elvégzéséhez. A fájl sorain egy iterátor segítségével lehet végigmenni. Egy sorban lévő különböző mezőket egy „^” jel választja el egymástól. Továbbá az alfanumerikus kódolású mezők „~” jellel vannak körülvéve. Ennek megfelelően kellett implementálnom a sorok mezőkre bontását és a szükséges adatok kinyerését. Az egyszerűség kedvéért a beolvasás során mindegyik mezőt szövegként kezeltem és később, ha kellett típuskonverziót hajtottam végre.

A könnyebb kezelhetőség miatt, a beolvasott szöveges fájlok szerkezetének megfelelő leíró osztályokat hoztam létre. Ezek az osztályok reprezentálják az egymáshoz tartozó adatokat. A leíró osztályok segítségével könnyű eltárolni és tovább adni az adatokat a különböző vezérlő osztályoknak. Továbbá minden fájl számára implementáltam egy listát, amelyekbe a fájlfeldolgozása alatt mentem a fájlok soraiból kinyert adatokat. Ez azt jelenti, hogy egy lista elem egy sort valósít meg a hozzáillő fájlból. Ezeket a listákat az `ArrayList` osztállyal implementáltam,

ami egy átméretezhető tömböt jelent számos tagfüggvényt tartalmazva a különféle listaműveletekhez.

Az „Indítás” gombra kattintva a rendszer felismeri, ha valamelyik mezőt üresen hagyta a felhasználó. Ezt egy rendszerüzenet formájában meg is jeleníti számára a képernyőn. Ebben az esetben folytatni is és megszakítani is lehet a feldolgozás folyamatát. A művelet során a megfelelő fájl soraiból a szükséges adatok az említett listákba mentésre kerülnek. Ezen listák elemeit később a vezérlő osztályok kapják meg feldolgozás céljából.

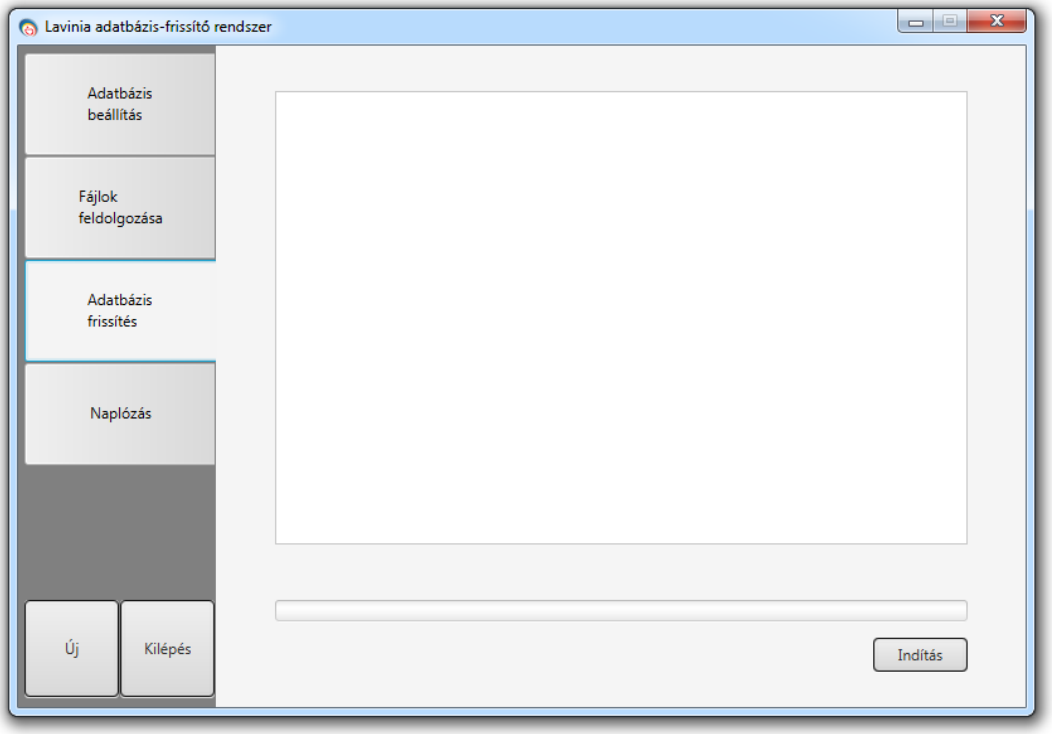
5.4. Adatbázis frissítése

Már a tervezési fázis során kiderült számomra, hogy az adatbázis-frissítés folyamata meghatározó eleme lesz a rendszernek. Így a rendszer tervezése alatt erre fordítottam a legtöbb időt. A rendszer magjaként elengedhetetlen, hogy maga a folyamat jól meg legyen tervezve, többek között mind a kliens oldalon, mind az adatbázis-szerveren lévő tárolt eljárások tekintetében. Ehhez figyelembe kell venni a feldolgozni kívánt adat nagyságát is, amely jelentősen befolyásolhatja a szoftvert működés közben.

Modularitás szempontjából a komponenst úgy terveztem és implementáltam, hogy elkülönüljön a többi modultól. Ez azt jelenti, hogy egy osztály tartalmazza az adatbázis kezeléséhez vonatkozó funkciókat. Beleértve az adatbázis kapcsolat létrehozását, a tárolt eljárások megalkotását és használatát a szerveren. Egy másik osztály pedig magát a tranzakciót valósítja meg ezen függvények hívásával. Ennek az az előnye, hogy más rendszerekbe is beilleszthető és kisebb módosítások után használható is akár.

A rendszert a specifikációtól eltérve úgy terveztem meg, hogy az első fellépő hibánál ne álljon le a folyamat és ne görgesse vissza az adatbázist a korábbi állapotra. Ehelyett úgy implementáltam, hogy észlelje a hibát, de menjen végig a folyamat, kijelezve az összes hibát. Majd csakis a folyamat végén hajtódjon végre rollback az adatbázison. Ez abból a szempontból előnyös, hogy elég egyszer lefuttatni a frissítést, és a logfájlok alapján ki lehet javítani az összes felmerülő hibát. Erről az ötletről azonban le kellett mondanom. Egyrészt a fájlok adatai között

lévő összefüggések számos fals pozitív hibát jeleznének. Másrészt a hibajavítások és tesztek során szembesültem azzal, hogy a Postgres adatbázis-kezelő rendszer megszakítja a folyamatban lévő tranzakciót egy megjelenő hiba esetén és ignorálja az utána bekövetkező parancsokat. Ez kiküszöbölhető lehetne az úgynevezett savepoint-ok bevezetésével. Ezen savepoint-ok segítségével visszaállítható a tranzakció, annak egy adott állapotára. Viszont ebben az esetben is számottevően kialakulhatnak fals hibák. Ebből az okból kifolyólag visszatértem az alap ötletre, miszerint az első hiba előfordulásánál a frissítési folyamat megáll, és a program végrehajtja az adatbázis visszagörgetését.



ábra Adatbázis-frissítési felület

Az adatbázis-frissítés komponens szálkezelésen alapul. Az „Indítás” gomb megnyomásával a szoftver elindít egy új szálat, amin a frissítés folyamata fut. A többszálasítás előnye, hogy egyes alkalmazásoknál növeli a hatékonyságot. Több szálat egymás mellett futtathatunk szinkron vagy éppen aszinkron módon, különböző feladatokat végrehajtva. Ez gyorsabb futási időt is eredményezhet. Az egy időben futtatható szálak száma a processzor teljesítményétől függ. Hátránya viszont, hogy nehezíti az implementálást, valamint a szálak együttes kezelése is nehéz feladatnak bizonyulhat. A szoftver esetében a grafikus felhasználói felület

futtató fő szál mellett az adatbázis-frissítés a háttérben futhat egy másik szálon. Ez azt váltja ki, hogy a frissítés mellett a grafikus felület is elérhető és képes kezelni a felhasználói interakciókat. Ellenkező esetben a frissítési folyamat egy szálon futna grafikus felülettel. Emiatt a felület nem képes reagálni a felhasználói interakciókra addig, amíg a művelet be nem fejeződik. A szálkezeléshez a *javafx.concurent* és a *javafx.application* könyvtárak komponenseit használtam fel. Segítségével össze tudtam hangolni a grafikus felhasználói felületet a háttérben futó folyamatokat megvalósító szálakkal.

Fontos megjegyezni, hogy a szálaknak kommunikálniuk is kell egymással. Hiszen a frissítést végző mellékszálnak folyamatosan adatot kell biztosítani a folyamat állapotáról a felhasználói felület részére. Ez abban nyilvánul meg, hogy a rendszer folyamatosan információt közöl a felhasználóval, többek között az elvégzett frissítések számáról, valamint az esetleges hibaüzenetről. Ezzel szemben visszafelé, a fő szálról nem kell adatot szolgáltatni a mellék szál számára. A grafikus felületet módosítani kizárólag azon a szálon lehet, amelyiken a felület fut. Ez a legtöbb grafikus keretrendszert megvalósító technikára igaz. A JavaFX keretrendszer lehetőséget ad, hogy a fő szálon hajtsuk végre a kívánt változtatásokat. Ehhez a *Platform* osztály *runLater()* statikus metódusát kell felhasználni, amelynek egy `Runnable` típusú paramétert kell megadni. Ez lehet egy *Runnable*–t implementált osztály vagy egy lambda kifejezés is. A kapott paramétert a fő szálon futtatja az alkalmazás, illetve másik szálról is meghívható a metódus. A paraméterben definiált szálnak kell megadni, hogy a grafikus felület mely objektumát módosítsa. Az eljárás a következő példa alapján adható meg.

```
Platform.runLater( new Runnable() {
   @Override
   public void run() {
      // futtatni kívánt kódrészlet
   }
});
```

A grafikus felületet megvalósító szál mellett újat is létrehozhatunk. Ehhez a *Task* osztályt használtam fel. Olyan feladatok elvégzésére van kitalálva, amelyeket

egy háttérben futó szálon kell végrehajtani. Teljes mértékben megfigyelhető módon lehet implementálni, a futó folyamat állapotáról végig képes információt biztosítani a grafikus fő szál számára. A rendszer esetében az adatbázis-frissítés folyamatát valósítottam ilyen módon. A háttérben futó szál folyamatosan jelzi a frissítés állapotát a felhasználói felületen. Ennek több formája is látható a felhasználó számára. Egyrészt az úgynevezett `ProgressBar` JavaFX objektum vizuálisan szemlélteti a művelet előrehaladását. Ezenfelül a frissítés elindításakor megjelenik a felületen egy státusz jelző, amely az elvégzett és az összes frissítés hányadosát mutatja. Továbba a rendszer valós időben, szöveges formában is jelzi a fontosabb eseményeket egy `ListView` objektum felhasználásával. Ezek megvalósításához meg kell hívni a fő szálon az említett `Platform.runLater()` metódust a háttérben futó szálon. Mivel a mellékszálak létrehozása fontos szerepet kap a szoftverben, ezért példával is bemutatnám, hogyan lehet implementálni a háttérben futó folyamatokat.**(forrás concurency)**

```
Task task = new Task<Void>() {
   @Override
   protected Void call() {
      // futtatni kívánt kódrészlet
      return null;
   }
};
new Thread(task).start();
```

A rendszer hatékonysága érdekében az adatbázis adatainak elmentése után azonnal bontja is a létrehozott kapcsolatot. Ez azért fontos, hogy feleslegesen ne foglalja az erőforrásokat más elvégzendő feladatok elől. A rendszert úgy terveztem meg, hogy csak akkor legyen kapcsolat az alkalmazás és az adatbázis szervere között, amikor arra szükség van. Ez az adatbázis beállítása mellett a frissítési folyamat esetében fordul elő. Az adatbázis-frissítés tranzakcióját elindítva újra kapcsolatot kell létrehozni. Ezt egy `Connection` objektummal adom meg, amelyet továbbadva paraméterként, felhasználható a tranzakció kezelésére. A tranzakció megkezdése előtt viszont, létre kell hozni a tárolt eljárásokat az adatbázis-

szerveren. Miután megteremtettük az adatbázis-kapcsolatot, SQL és PL/SQL parancsokat is hajthatunk végre a szerveren. Ezt a JDBC keretrendszer támogatásával tehetjük meg a `Statement`, `PreparedStatement` és a `CallableStatement` osztályok egyikével.**(forrás tutorialspoint)**

- Statement - Általános célú hozzáférésre használható. Statikus SQL lekérdezések igénybe vételénél bizonyul hasznosnak. Nem paraméterezhető.
- PreparedStatement - Az SQL utasítás többszöri felhasználására alkalmazható. Akár futási időben is paraméterezhető.
- CallableStatement - Az adatbázis-szerveren lévő tárolt eljárások hívását segíti. Szintén paraméterezhető futási időben is.

Mivel a tárolt eljárásokat csak egyszer kell létrehozni az adatbázis szerverén, ezért statikus jellegű SQL lekérdezéssel megoldható a létrehozásuk. Ebből az okból kifolyólag a `Statement` és a `PreparedStatement` osztály is egyaránt alkalmazható. Egy SQL lekérdezés futtatását az adatbázis-szerveren az alábbi példakód mutatja be.

```
try {
   Connection conn = Connect();
   String query = "UPDATE MYTABLE SET NAME=’name’ WHERE id = 
1"
   PreparedStatement ps = conn.prepareStatement(query);

   ps.executeUpdate();

   conn.commit();
   ps.close();
   conn.close();
} catch(SQLException ex) {
   ex.printStackTrace();
}
```

Tárolt eljárásokat a PL/SQL (Procedural Language/Structured Query Language) procedurális programozási nyelv alapján hozhatunk létre. Lényege, hogy több, logikailag összefüggő eljárást egy funkcióban valósítsunk meg. Így a komplexebb feladatokat az adatbázis-kezelő rendszer végzi el, amely gyorsabb és hatékonyabb megoldást jelenthet a sorozatos SQL hívásokkal szemben. A nyelv alapja az Ada programozási nyelv. Emellett értelemszerűen tartalmazza az SQL nyelv elemeit is, többek között a SELECT, INSERT, DELETE, UPDATE utasításokat.**(forrás wiki plsql)**

A tárolt eljárások létrehozása után elindulhat az adatbázis-frissítés folyamata a háttérben futó szálon. Ez jelenti a tranzakció kezdetét. A rendszer mielőtt bármilyen műveletet végrehajtana a Lavinia adatbázisán, elvégzi a tranzakció beállításait. Ezt meg lehet tenni mind az adatbázis-szerveren, mind a kliens programkódjában. Én az utóbbit választottam az egyszerűség végett, hiszen a JDBC keretrendszere ebben is segítséget tud nyújtani. Az adatbázis-kezelő rendszer a módosításokat először egy cache-ben tárolja addig, amíg mentésre vagy törlésre nem kerül a változtatás. Emiatt egy tranzakció kétféleképpen érhet véget. Egyrészt a `COMMIT` SQL utasítással, ami megerősíti a korábbi SQL parancsok általi változásokat az adatbázisban. Így az adatbázis új állapota mentésre kerül. Másrészt a `ROLLBACK` SQL utasítás vethet véget a tranzakciónak. Ezzel a paranccsal vissza lehet vonni a tranzakció összes módosítását. Ezáltal az adatbázis visszaáll a korábbi állapotára. Ennek következtében elsősorban az automatikus mentés beállítást állítottam manuálisra. Ez azt jelenti, hogy az alkalmazás letiltja az alapértelmezett automatikus commit opciót az adatbázis szerverén és a forráskódban kell megadni, hogy a tranzakció folyamán mikor mentse a változtatásokat. Ez az adatbázis-kapcsolatot reprezentáló `Connection` objektum `setAutoCommit()` metódusával valósítható meg, amely egy `boolean` típusú értéket vár paraméterként. Ezután a tranzakció izolációs szintjét állítom be. Az izolációs szint határozza meg, hogy a végrehajtás alatt lévő tranzakciók, valamint tárolt eljárások milyen hatással vannak egymásra és milyen állapotban látják az adatbázist. Az SQL szabvány négy izolációs szintet definiál. Ezek a read uncommited, a read commited, a repeatable read és a serializable. Az izolációs szintek közötti különféle viselkedésekre vonatkozó megszorításokat a következő táblázat mutatja be.

Izolációs szint	Piszkos olvasás	Nem megismételhető olvasás	Fantom olvasás
Read uncommited	X	X	X
Read commited		X	X
Repeatable read			X
Serializable			

táblázat Izolációs szintek

A szoftver esetében a read uncommited izolációs szintet választottam. Annak ellenére, hogy ez a legmegengedőbb szint, a rendszer megfelelő futásához elengedhetetlen. Ennek oka a gyors és hatékony működés. Emellett a feldolgozott fájlokat egymástól függetlenül használja fel az adatbázis frissítésére, de nyilvánvalóan vannak összefüggések két fájl tartalma között. Új élelmiszer esetén importálni kell a hozzátartozó tápanyagtartalmat és az egységnyi tömeg értékeket. Így előfordulhat az az eset, hogy a tranzakció elején módosított adatokat később felhasználja további adatok változtatására. Mivel az egész frissítési folyamat egy tranzakciót jelent, ezért csak a végén van módosítás mentés a `COMMIT` paranccsal. Emiatt a tranzakció elején módosított adatok nem kerülnek elmentésre, és a továbbiakban nem lehetne rájuk hivatkozni. A read uncommited izolációs szint megengedi viszont, hogy a nem mentett adatokat is lehet olvasni a tranzakció során. Erre jó példa egy új élelmiszer adatainak felvitele az adatbázisba. Először az ADD_FOOD.txt fájl adatait importáljuk, majd ezt követően az ADD_NUTR.txt és az ADD_WGT.txt fájlokban lévő információt. Utóbbiakban található adatok hivatkoznak az élelmiszerek egyedi azonosítójára, viszont az újonnan importáltak nincsenek commitolva a tranzakció végéig. Így az említett tranzakciós szint használata nélkül nem lehetne ezen azonosítókra hivatkozni. A tranzakció izolációs szintjét a *`Connection`* osztály *`setTransactionIsolation()`* metódusával határozom meg. Paraméterként az osztályban definiált *`TRANSACTION_READ_UNCOMMITED`* konstanst adom meg.

A tranzakció megkezdése előtt egy biztonsági pontot is meghatározok. Ehhez a JDBC interfészben definiált *`SavePoint`* osztályt használtam. A Postgres adatbázis-kezelő rendszer támogatja a savepoint-ok használatát, melyekkel olyan

pontokat tudunk kijelölni a tranzakción belül, amelyekhez hiba esetén vissza lehet görgetni a végrehajtott módosításokat. Én a tranzakció legelejére helyeztem egy ilyen savepoint-ot, így ha bármi hiba felmerülne a frissítés folyamata közben, az összes módosítást visszagörgeti az adatbázis-kezelő rendszer.

A tranzakció beállításai után a frissítési folyamat következik. Ezt a tervezési fázisban bemutatott koncepció alapján implementáltam. Vagyis ciklikus hívás történik minden egyes tárolt eljárásra. Az adatbázis-szerveren lévő tárolt eljárás hívását a korábban említett `CallableStatement` osztály segítségével valósítható meg. Ezt példakód alapján be is mutatnám.

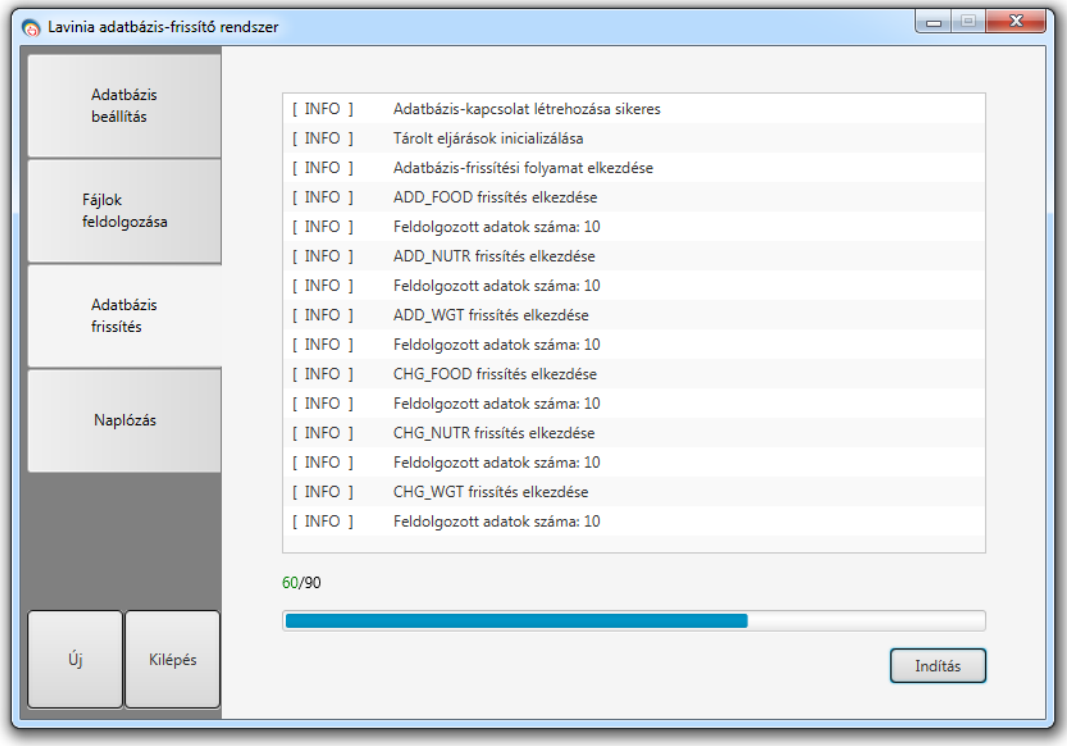
```
try {
   Connection conn = Connect();
   CallableStatement cs =
                  conn.prepareCall( "{? = call add( ? )}" );
   cs.registerOutParameter(1, Types.INTEGER);
   cs.setInt(2, 10);

   cs.execute();

   conn.commit();
   cs.close();
   conn.close();
} catch(SQLException ex) {
   ex.printStackTrace();
}
```

Az alkalmazás minden sikeres tárolt eljárás hívását tudatja a fő szállal. A felhasználó a grafikus felületen, ezt a folyamat előrehaladását mutató objektumokon látja. Viszont rendellenesség esetén a megfelelő hibatűréssel kell rendelkeznie a szoftvernek. A frissítési művelet kizárólag a JDBC keretrendszer segítségével valósul meg. Emiatt teljes mértékben kontrollálható a hibák kezelése. A hibák az adatbázis szerverén keletkeznek, amiket a Java oldalon van lehetőség felülvizsgálni. Probléma esetén a meghívott metódusok kivételt dobnak

SQLException formájában. A kivétel elkapásával megfelelően kezelhető a megjelenő hiba, majd a kivétel tovább dobásával a vezérlőosztályok befolyásolhatják a folyamat menetét. A rendszer esetében a legfontosabb a tárolt eljárások hibakezelése. Ebből az okból kifolyólag számos utasítás rendellenes működése kivételt dob a Postgres szerveren. Ez a *RAISE EXCEPTION* utasítással tehető meg a tárol eljárás kódjában. A kivételt pedig kliens oldalon a try-catch blokk implementálásával lehet elkapni. Tervezés során fontos volt felismerni milyen hibák léphetnek fel a rendszer működése közben. A kapott adatbázis tanulmányozása közben arra jutottam, hogy elsődlegesen hivatkozási hibák fordulhatnak elő. Ez jelentheti egy bizonyos adat hiányát, vagy épp többszörös meglétét. Adathiány esetén a szoftver nem képes azt pótolni, emiatt alkalmazni kell a követelményekben ismertetett hibakezelést. Amennyiben bármilyen hiba lépne fel, a rendszer visszagörgeti az adatbázist a frissítés előtti állapotra. Ez a *rollback()* metódussal implementálható, amely paraméterként a korábban létrehozott *SavePoint* objektumot kapja meg. Ennek következtében a tranzakció véget ért. Így a szoftver bontja is a kapcsolatot az adatbázissal.

5.5. Valós idejű megfigyelés

Az alkalmazás lehetőséget biztosít a felhasználónak, hogy valós időben kövesse a folyamat főbb elemeit. A frissítés közben informatív üzenetek jelennek meg a felületen, mutatva az aktuális állapotot. Az üzenetek alatt szintén megfigyelhető egy állapotjelző számpáros. Az első szám zöld színnel kiemelve mutatja meg az elvégzett, sikeresen frissített elemek darabszámát. A második szám pedig az összes adat számosságát jelöli, amelyek részt vesznek a frissítés folyamata közben. Ezt az állapotot vizuálisan is szemlélteti a számok alatt található folyamatjelző csík. Ennek segítségével könnyebben érzékelhető az adatbázis-frissítés pillanatnyi státusza. Az alkalmazást a frissítési folyamat közben az 5.9. ábra mutatja be.



ábra Adatbázis-frissítés folyamat közben

5.6. Naplózás

Az alkalmazás minden adatbázis-frissítés jelentősebb történéseit naplófájlokba menti. Ennek előnye, hogy a felhasználó vissza tudja követni az egész folyamatot, beleértve a sikeres és sikertelen műveleteket. Sikertelen futás esetén visszakereshető mi a hiba és mi okozta azt.

A naplózáshoz nem használtam előre megírt könyvtárat vagy külső eszközt. Emiatt saját osztályt hoztam létre a feladathoz. Minden egyes fájl neve tartalmazza a tranzakció elindításának pontos időpontját. Például a *logfile_2017_11_17T18_52_13.txt* nevű szöveges állomány, amely a 2017. november 17-én 18 óra 52 perc 13 másodperckor elindított adatbázis-frissítés eseményeit tartalmazza.

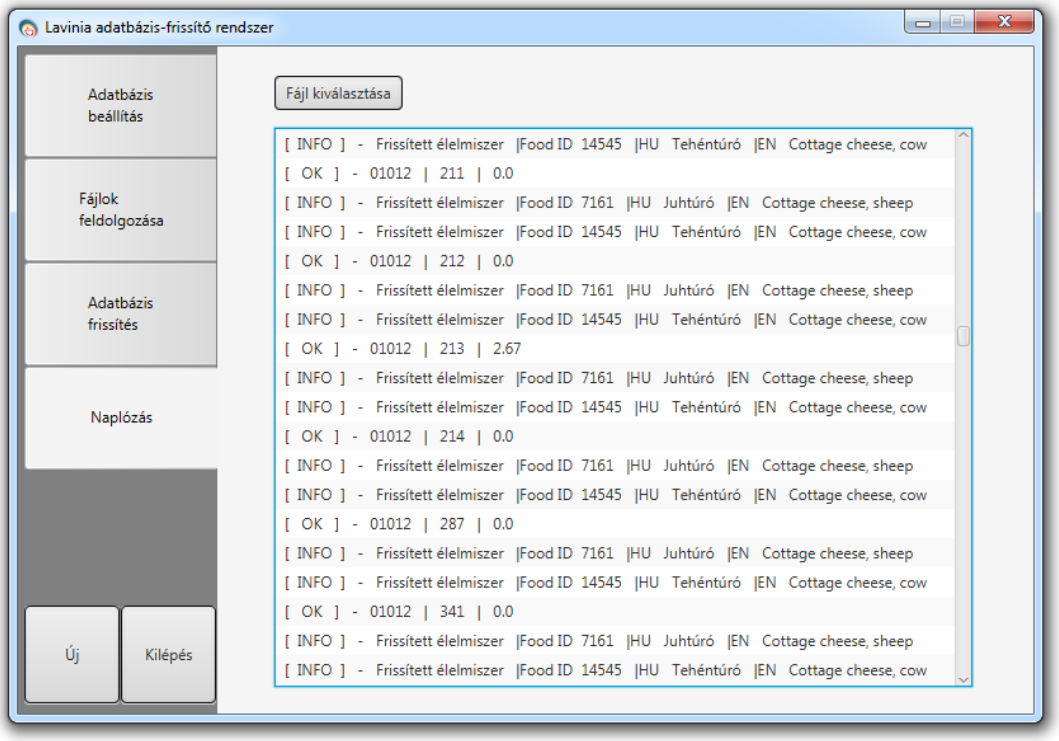
A naplófájlokat a program könyvtárában található LOG mappában találhatjuk meg. Ha ez a mappa nem létezik, az azt jelenti, hogy a szoftverrel még nem történt adatbázis-frissítés. Ilyenkor a rendszer automatikusan létrehozza a mappát. Egy naplófájl magában foglalja egyrészt a felhasznált USDA adatokat, és

hogy ezek melyik élelmiszert befolyásolták az adatbázisban. Ezt az alábbi példa mutatja.

```
[  OK  ]  -  01012  |  210  |  0.0
[ INFO ]  -  Frissített élelmiszer  |Food ID  7161  |HU Juhtúró  |EN  Cottage cheese, sheep
[ INFO ]  -  Frissített élelmiszer  |Food ID  14545  |HU Tehéntúró  |EN  Cottage cheese, cow
```

Az adatok frissítésre vonatkozó állapotát a szögletes zárójelben lévő szó mutatja. A sikeres `[ OK ]` feliraton kívül, bármilyen rendellenesség esetén megtalálható a `[ FAILED ]` jelölés is. Ezt minden esetben egy `[ERRORMSG]` jelöléssel kezdődő sor követi, amely az adatbázis-szerveren bekövetkező hiba üzenetét ismerteti. Előfordulhat az az eset, amikor egy USDA azonosító több élelmiszerhez is tartozik a Lavinia adatbázisában. Ilyen alkalommal a frissítés természetesen mindegyik élelmiszerre hatással van és a naplófájlokban is megjelenik, mely élelmiszereket érintette a művelet. Erre jó példa a fent említett bejegyzés. Ez az `[ INFO ]` kezdetű sorokban figyelhető meg, amely az adatbázisban található egyedi azonosító mellett a magyar és az angol megnevezést is tartalmazza.

A felhasználó képes a program segítségével megnyitni az egyes fájlokat és megvizsgálni azok tartalmát. Ezt az 5.10. ábra mutatja be.



ábra Naplózási felület

A „Fájl kiválasztása” gombra kattintva megjelenik egy fájl választó ablak, amelynek az alapértelmezett kiindulási pontja a LOG mappa. A kiválasztott fájl megnyitásával a grafikus felület listaszerűen megjeleníti annak tartalmát.

5.7. További lehetőségek

Úgy gondolom az elkészült rendszert számos funkcióval lehetne még bővíteni, amelyek növelhetik a szoftver felhasználóbarátságát.

Már a tervezés fázisában gondolkodtam egy külön grafikus felületen, amelyen manuális módon lehet élelmiszert felvinni a Lavinia adatbázisába. Habár szembe megy a rendszer fő elvével, az automatizáltsággal, véleményem szerint hasznos lehet, ha csekély hiányosságok fordulnak elő az adatbázisban. Egy ilyen funkció segítségével könnyedén lehetne pótolni az elmaradt élelmiszereket.

További fejlesztésnek elképzelhető egy felület, amely a frissítésekből statisztikát vezet. Így a felhasználó egy visszajelzést kap a frissítések eredményeiről különböző diagrammok segítségével. Ezáltal megfigyelhető, hogy a frissítések mekkora hatással voltak az adatbázisra.

6. Tesztelés

Ebben a fejezetben a program tesztelését mutatom be. Tesztelés során elsődleges szempont volt, hogy a szoftver megfelel-e a specifikációnak és helyesen csinálja-e az abban megfogalmazott követelményeket.

6.1. Unit teszt

Először a rendszeren végzett unit tesztelési metodikát ismertetem, ugyanis már az implementálás fázisa közben is nagy szerepet játszott. A unit tesztelés a legalacsonyabb szintű tesztelés. Azokat az egységeket teszteli, amelyekből felépül a komplett rendszer. Pontos meghatározása nincs annak, hogy mit képvisel egy unit egy adott szoftverben. Viszont az megállapítható, hogy a legkisebb önálló egységként tesztelhető részét képezi a rendszernek. A teszt használatának célja, hogy alapot nyújtson a biztonságos, helyesen működő szoftverkomponensek implementálásához. Unit tesztelés során általában egy funkciót vagy metódust kell tesztelni. Lényege, hogy különböző bemenetek esetén, miként reagál a metódus és milyen viselkedést produkál az elvárthoz képest. Egy metódusra akár több teszt esetet is lehet írni, ha több féle képpen reagálna különböző bemenetek alkalmával. Fontos megjegyezni, hogy egymástól elszeparáltan futtassuk a teszteket, hogy ne alakuljon ki semmilyen függőség közöttük, ami nem a valós elvárt viselkedést mutatná. Valamint így könnyebb megállapítani a hiba helyét. A metódusok későbbi változtatása esetén szintén futtatható a unit teszt, ami biztosítja, hogy a változtatás során nem következett be hiba.

Unit teszteléshez a JUnit keretrendszert használtam, amelyet a NetBeans integráltan támogat. Ez egy nyílt forráskódú keretrendszer, amely különféle annotációkkal és metódusokkal teszi egyszerűbbé a unit tesztelés menetét. Ennek köszönhetően a szoftver fejlesztése is gyorsabb és növeli minőségét. A tesztek automatikus módon futtathatóak és azonnali visszajelzés érkezik a tesztek sikerességéről vagy épp sikertelenségéről a fejlesztő számára. Külön tesztosztályokat is létrehozhatunk az átláthatóság kedvéért, amelyek a teszt eseteket tartalmazzák. Egy teszt eset a tesztelt metódus egy bizonyos részét vagy

funkcionalitását foglalja magában és vizsgálja annak viselkedését. Egy teszt eset a következő példa alapján valósítható meg a JUnit keretrendszerrel.

```
public class UnitTestClass extends TestCase {

   @Before
   @Override
   public void setUp() {
   }

   @Test
   public void firstTest() {
      asserEquals( 0, 0 * 10 );
   }
}
```

Előfordulhat, hogy több teszt eset futtatásához szükség van ugyanarra az objektumra. A `setUp()` metódus felülírva és `Before` annotációval ellátva minden egyes teszt eset előtt lefut. Így nincs szükség minden teszt eset elején inicializálni az objektumok értékét, ehelyett egy függvény gondoskodik róla. Az `assertEquals()` metódus paramétereként egy elvárt, valamint egy aktuális értéket kell megadni, amelyeket megegyezés céljából megvizsgál.

A rendszer implementálása közben a grafikus felhasználói felületet nem tudtam unit tesztelni, hiszen az nem valósít meg semmilyen üzleti logikát. Emellett az adatbázison végrehajtott műveleteket sem voltam képes letesztelni a keretrendszerrel, ugyanis ebben az esetben az adatbázis szerverén történik a folyamat. Ennek következtében főként a fájlok feldolgozását és egyéb műveletek tudtam unit tesztelni.

6.2. Adatbázis oldali tesztek

Legfontosabb tesztelés az adatbázis-szerver oldali folyamatok tesztjei voltak, hiszen ez a rendszer központi eleme. Az adatbázis-frissítésre használt tárolt eljárásokat manuálisan teszteltem. Ehhez segítségemre volt a tervezési fázis alatt

kapott példa adatbázis, amely mindössze pár száz rekordot tartalmazott. Erre támaszkodva jelentősen könnyebb volt megvalósítani és tesztelni a tárolt eljárásokat.

Az implementálás fázisa alatt biztosították számomra a Lavinia teljes adatbázisának másolatát. Persze csak azon részét, amelyet a megvalósított szoftver befolyásol. Így már több mint hatszázezer rekordot tudtam importálni a lokálisan felépített adatbázisomba. Rendkívül nagy segítség volt ez a tesztelés szempontjából. Ezután a tárolt eljárások első futtatása már rendellenes működést mutatott. A kisebb adathalmaz tanulmányozása közben, azt a következtetést vontam le, hogy egy USDA élelmiszer egy rekordhoz tartozik a Lavinia adatbázisában. A tárolt eljárásokat is ennek megfelelően terveztem és írtam meg. A teljes adatbázison végrehajtott tesztelések viszont cáfolták ezt. Egy USDA azonosító több élelmiszerhez is tartozhat. Erre példa az amerikai adatbázisból vett „TURKEY BREAST MEAT” termék, mely „Pulykahús” és „Natúr pulykamell szelet” elnevezéssel is szerepel az adatbázisban, két külön rekordot alkotva. Ebből az okból kifolyólag újra kellett terveznem a tárolt eljárásokat.

6.3. Rendszerteszt

Utolsó tesztelési szint a rendszerteszt volt. Ezeket a grafikus felhasználói felülettel egyidejűleg teszteltem. Elsősorban az volt a cél, hogy megvizsgáljam a szoftver működőképességét egyben és komponensenként is. Ezt manuálisan végeztem el, ellenőrizve a grafikus felület helyes működését a különböző funkciókkal. A rendszertesztek alatt figyelemmel kísértem az alkalmazás működését a felhasználói interakciók során is, hogy milyen rendellenességet okozhatnak.

Továbbá megfigyeltem az egyes funkciót futási idejét is. A fájlok feldolgozása több tízezres nagyságrendnél is egy pillanat alatt bekövetkezik. Így az adatbázis-frissítés futási idejét emelném ki. A kísérlet alatt teszt adatokkal dolgoztam. Az elvégzett tesztek során elsődlegesen száz, majd ezer élelmiszert tartalmazó mintát használtam fel. A kísérleteket három-három alkalommal hajtottam végre. Ezen mérések átlagát a következő táblázatok mutatják be ezredmásodpercnyi pontossággal.

Modul	Futási idő (ezredmásodperc)
ADD_FOOD	22
ADD_NUTR	78
ADD_WGT	108
CHG_FOOD	97
CHG_NUTR	405
CHG_WGT	112
DEL_FOOD	62
DEL_NUTR	395
DEL_WGT	125

táblázat Futási idő száz tesztadatra

Modul	Futási idő (ezredmásodperc)
ADD_FOOD	134
ADD_NUTR	711
ADD_WGT	998
CHG_FOOD	871
CHG_NUTR	3932
CHG_WGT	995
DEL_FOOD	593
DEL_NUTR	4086
DEL_WGT	1111

táblázat Futási idő ezer tesztadatra

7. Összefoglalás

Dolgozatom célja egy olyan migrációs szoftver megalkotása volt, amely a rendszeresen publikált USDA adatbázis-frissítései alapján, automatizált módon hajtja végre a Lavinia által használt adatbázis frissítését. Ez rendkívül nagy adatfeldolgozással jár, hiszen a folyamat több tízezer rekordot érinthet a céladatbázisban.

Az alkalmazás grafikus felülettel biztosítja a felhasználóbarát működést, amely könnyen kezelhető és átlátható. Tartalmazza mindazon funkciókat, amelyekkel megvalósulhat a Lavinia adatbázisának frissítése. Az USDA által biztosított szöveges állományok gyorsan és könnyedén feldolgozhatóak. A frissítési folyamat az adatbázis-szerven megy végbe, egy nagy tranzakció formájában. A felhasználónak lehetősége van valós időben megfigyelni a művelet főbb eseményeit. A folyamat aktuális állapotát pedig több objektum is jelzi a grafikus felületen. A rendszer működése közben minden egyes elindított tranzakció naplózásra kerül szöveges fájlokként. Ezáltal bármilyen rendellenesség megjelenése esetén, egyértelműen meghatározható a hiba oka.

A szoftver sikeressége a tesztek során mutatkozott meg igazán. A több tízezres nagyságrendű adattal rendelkező szöveges fájlokat egy pillanat alatt képes feldolgozni. Emellett ezer tesztadaton végzett adatbázis-frissítési kísérlet megközelítőleg tizenhárom másodperc alatt következett be.

Úgy vélem az elkészült rendszer nagyban segítheti a Lavinia adatbázisának növekedését és tartósságát a jövőben. Ezáltal az élelmiszerekhez a lehető legpontosabb adatok fognak tartozni.