Debreceni Egyetem

Informatika Kar

Java Enterprise Computing

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Szunai János
programtervező informatikus BSc.

Debrecen

2009

Tartalom

1 Bevezetés **3**

2 Felhasznált technológiák ismertetése **5**

2.1 A Java programozási környezet 5

2.1.1 Az n rétegű architektúra 6

2.2 Hibernate 7

2.3 EJB 10

2.4 AJAX 11

2.4.1 GWT 12

2.5 Jasper Reports 15

3 Készletnyilvántartó alkalmazás követelményeinek feltárása és tervezése **17**

3.1 Adatbázis modell 17

3.1.1 Termék 17

3.1.2 Leltár 18

3.1.3 Partner 18

3.1.4 Számla 18

3.1.5 Számlatétel 19

3.1.6 Szállítólevél 20

3.1.7 Szállítólevél tétel 20

3.2 Rendszer modell 20

3.2.1 Üzleti folyamatok 22

3.2.2 Összefoglalva az alkalmazás feladatai 23

4 Készletnyilvántartó alkalmazás megvalósítása **24**

5 Készletnyilvántartó alkalmazás felhasználói kézikönyve **28**

5.1 Felületi elemek 28

5.2 Árucikkek megtekintése 30

5.2.1 Új árucikk felvétele 30

5.2.2 Árucikk módosítása 30

5.3 Partnerek megtekintése 30

5.3.1 Új partner felvétele 30

5.3.2 Partner módosítása 31

5.4 Eladás 31

5.5 Vétel.....31
5.6 Tétellista kitöltése.....31
5.7 Szállítólevelek.....32
5.8 Számlák.....33
5.9 Riportok.....34
5.10 Leltár.....35
5.10.1 Korábbi leltárak megtekintése.....35
6 Összefoglalás.....37
7 Ábrajegyzék.....38
8 Irodalomjegyzék.....39
9 Függelék.....40

1 Bevezetés

Diplomamunkámban szeretnék az egyetemi évem alatt megismert Java technológiák közül néhányat bemutatni és példát szolgáltatni gyakorlati alkalmazásukra. Mindezt egy alkalmazás fejlesztésén keresztül szeretném bemutatni, így látható, hogy a különböző technológiák hogyan integrálhatók egy stabil és sokrétű alkalmazás létrehozása céljából.

Célom egy készletnyilvántartó alkalmazás elkészítése volt, amely egy kereskedő segítségére szolgál a készlet kezelésében és a hozzá kapcsolódó járulékos feladatok elvégzésében.

Egy kereskedő számára mindig fontos, hogy tisztában legyen a lehetőségeivel. Sok féle termék nagy számban történő forgalmazása mellett az adminisztráció már igen nehéz, nem beszélve a korábbi forgalom ellenőrzéséről. Ezért gondoltam, hogy kifejlesztek egy olyan programot, amellyel kezelhető az árukészlet változása a beszerzések és eladások folyamán, valamint ezen üzletek később egy helyről visszakereshetők lesznek.

Ezzel a problémával apukám szembesült. A korábban használt nyilvántartó program már nem elégítette ki az igényeit. A piacon fellelhető hasonló nyilvántartó rendszerekkel is az a gond, hogy ritkán illeszkednek pontosan a felhasználó követelményeihez és nagyon ritka közöttük a szabadon használható szoftver. Ezért döntöttem úgy, hogy kifejlesztek egy olyan alkalmazást, amely megkönnyíti a mindennapi adminisztrációt.

Manapság az üzleti alkalmazások körében igen elterjedtek a Java Enterprise Edition platformra épülő webes alkalmazások. Felhasználók széles bázisa használhatja az alkalmazást konkurens módon. Ezt megfelelő alkalmazásszerverek biztosítják. De az is előnyei közé tartozik, hogy megfelelő architektúra mellett a hibatűrése is magas lehet – bár a jelenlegi probléma megoldásánál nem tartottam fontosnak az alkalmazás több szerveres környezetre történő optimalizálását.

Laikus felhasználónak nincs szüksége különösebb előkészületekre, a program szerverre történő telepítése után egy böngésző program segítségével bárhonnan használható a rendszer.

Szem előtt kellett tartanom, hogy olyan technológiákat használjak a részproblémák megoldására, amelyek szintén ingyenesen felhasználhatóak. A Java kézenfekvő választás volt ingyenessége és széles támogatottsága miatt. Alkalmazásszervernek Glassfish-t használok, amely a Sun cég terméke és szintén ingyenesen használható. Az ilyen rendszerek kritikus pontja az adatbázis, amelynek megválasztása kezdetben némi fejtörést okozott. Mivel a program egy kisvállalkozás számára készül, figyelembe kellett vennem, hogy az erőforrások

korlátozottak. Ezért használom a Derby adatbázist, amely része a Glassfish alkalmazásszervernek. Ezzel elértem, hogy a program olyan számítógépre telepítve is üzemképes, amely nem elsősorban szerverként funkcionál.

2 Felhasznált technológiák ismertetése

A felhasznált technológiák kiválasztásakor törekedtem olyanokkal dolgozni, amelyekben némiképp jártas vagyok korábbi tanulmányaim után, továbbá néhány új dolog felhasználására, amelyekkel az adott probléma megoldása során fogok részletesebben megismerkedni.

2.1 A Java programozási környezet

A Java nyelv megjelenése óta dinamikusan fejlődik. Egyre több feladat megoldására alkalmassá vált, ezért széles körben elterjedt. Mára a Java platformnak 3 kiadása van jelen a piacon. A Java Standard Edition (SE) hagyományos asztali alkalmazások és appletek készítését támogatja. Erre a Java már a kezdetektől alkalmas volt. A Java Micro Edition (ME) a mobil eszközökre történő fejlesztést támogatja. Jellemzője, hogy nem rendelkezik minden, az SE-ben megtalálható implementációval a céleszközök erőforrásainak korlátozottsága miatt, viszont vannak eszköz specifikus implementációk benne, amelyek a Micro Edition környezetben érhetők el. A legnagyobb méretű kiadás a Java Enterprise Edition (EE), amely elosztott, sok felhasználós vállalati méretű rendszerek fejlesztését támogatja. A Java EE-nek része a Java SE, tehát az Enterprise technológiák mellett a Java SE API is használható.

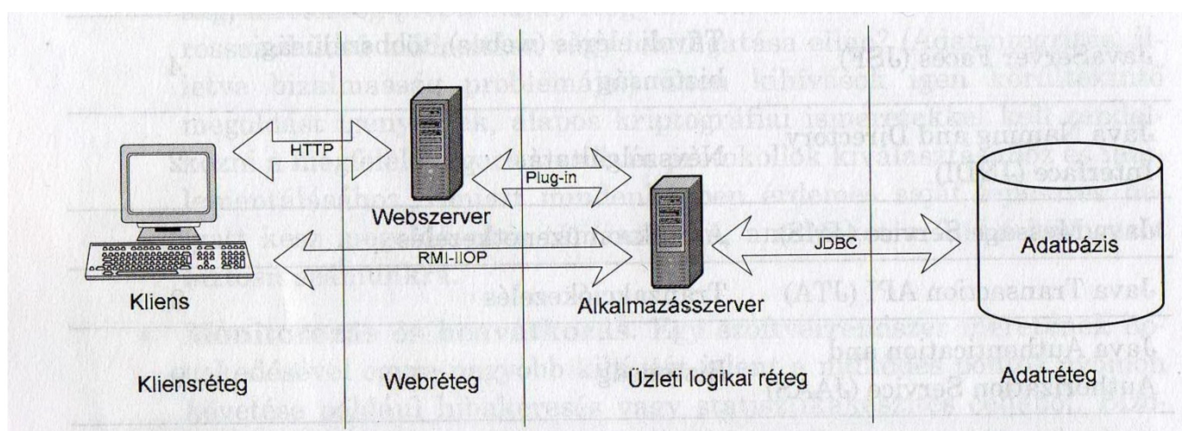
A Java általános jellemzője, hogy platform független, amelyet úgy képes megvalósítani, hogy a programjainkat egy köztes nyelvre (IL) fordítja, majd ez a kód egy futtató környezetben fut. A futtató környezet biztosítja, hogy a kód minden környezetben ugyanúgy viselkedjen.

Miben nyújt többet az Enterprise Edition, amiért nem elég a Standard Edition? Nagyméretű információs rendszerek fejlesztése során sok követelmény merül fel, amelyek lényegükben hasonlóak. Vállalati rendszerek fejlesztését a Java EE úgy támogatja, hogy az ilyen problémákra globális megoldást nyújt. A futtató környezet hivatott különféle szolgáltatásokat biztosítani, amelyek megoldással szolgálnak az egyes problémákra. Mondhatjuk, hogy a futtató környezet egyfajta keretrendszert biztosít az alkalmazások számára.

A Java EE futtató környezete az alkalmazásszerver. A teljesség igénye nélkül néhány probléma, amelyekre megoldással szolgál: perzisztencia, tranzakciókezelés, skálázhatóság. Ilyen komplex problémakörök megoldására nem érdemes saját implementációt elkészíteni, sokkal egyszerűbb a programozó dolga, ha mindezen rendszereket készen kapja. Ezeket

hívjuk *middleware-szolgáltatás*oknak. Az alkalmazásszerver több technológiát definiál, amelyek közösen lefedik ezeket a követelményeket. Ezen technológiák mindegyike saját API-n keresztül vehető igénybe.

2.1.1 Az n rétegű architektúra



1. ábra Az n rétegű architektúra

A rétegelt architektúrák jellemzője, hogy minden réteg egy jól definiált feladatot lát el, csakis az alatta lévő rétegtől vesz igénybe szolgáltatásokat és csak a fölötte lévő rétegnek nyújt szolgáltatásokat.

Az adatréteg feladata az adatok tárolása perzisztens formában és a rajtuk végezhető elemi műveletek biztosítása. Leggyakrabban ezt a réteget relációs adatbázis valósítja meg. Mára már valós alternatíva az objektumorientált vagy az XML adatbázis, de ezek még nem annyira elterjedtek, bár a támogatásukat a legtöbb rendszer biztosítja.

Az üzleti logikai réteg biztosítja az alkalmazási terület igényeinek megfelelő működést. Az üzleti szabályoknak megfelelően hívja meg az adatréteg szolgáltatásait. Ez az a réteg, amelyet az alkalmazásszerverre telepítünk Java EE esetében, így igénybe veheti a szabványos *middleware-szolgáltatás*okat.

A kliens réteg biztosítja a kapcsolatot a felhasználóval. A felhasználó felületen végbement interakciók hatására meghívja a megfelelő üzleti logikai funkciókat, majd esetlegesen frissíti a felhasználói felületet. A klienseknek két típusa van, vékony és vastag kliens. A köztük lévő határ ma már eltolódni látszik a gazdag webes kliensek jóvoltából, melyeknek köszönhetően vastag klienshez hasonló felülettel találkozhatunk a böngészőn belül.

Az eddig ismertetett architektúrát szokták 3 rétegű architektúrának nevezni, viszont ha vékony klienseket is ki akarunk szolgálni, szükségünk lesz egy webes rétegre, amely a böngészőktől érkező hívásokat értelmezi, és ezek alapján hívja az üzleti funkciókat. Java EE esetén ez is telepíthető az alkalmazásszerverre, viszont általános megoldás hogy külön webszervert alkalmaznak a feladatok ellátására.

2.2 Hibernate

Vállalati alkalmazásoknál elengedhetetlen az adatréteg kezelésének megfelelő megválasztása. Mára több olyan keretrendszer is létezik, amelyek megvalósítják a perzisztencia kezelést. A Hibernate ezek közül is talán az egyik legszélesebb körben használható. Flexibilis keretrendszer, sok környezethez hozzá lehet igazítani a megfelelő konfigurációs paraméterek beállításával. .Net rendszerhez is létezik adaptációja, a Java vonalon pedig több kiegészítése is létezik, amelyekkel támogatja az annotációkkal való konfigurálás vagy a standard JPA megvalósítást. Kezdettől fogva nagy konkurenciájú rendszerek kezelésére tervezték, klaszteres környezetben is jól működik. Jelen alkalmazásban csak a Core rendszert használom.

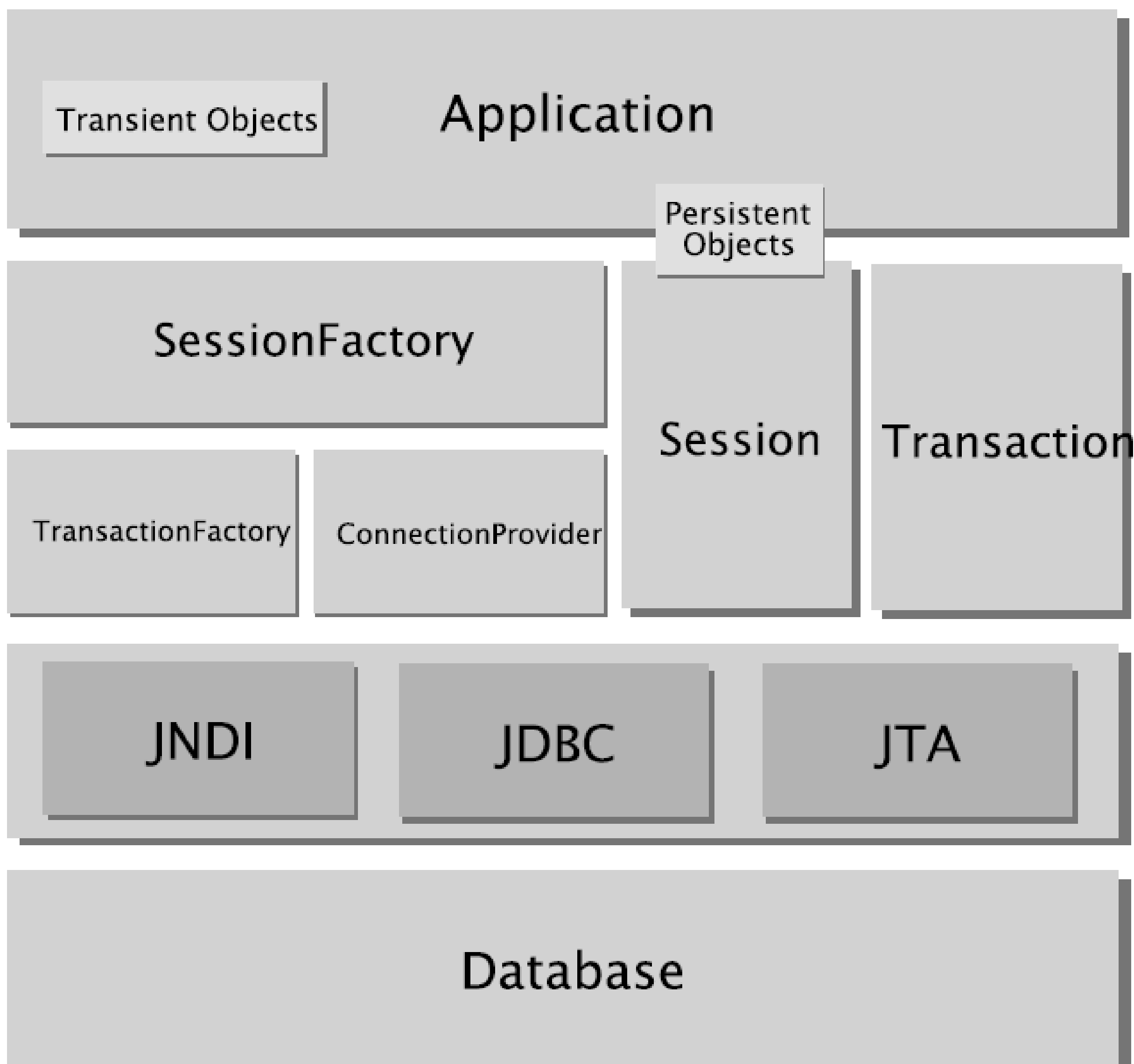
Hibernate megvalósítja az objektum-relációs leképezést, objektumok létrehozását és tárolását objektum orientál paradigma mentén. Az adatbázisban tárolt értékekkel a programban úgy dolgozhatunk, mintha egyszerű osztályok lennének. Támogatja az öröklődést, a kollekciókat, asszociációt stb. Mindemellett saját lekérdező nyelvvel is rendelkezik (*HQL*), amely lényegesen megkönnyíti az objektumokkal való munkát.

A tárolás és visszaolvasás menete igen leegyszerűsödött, egyetlen metódushívással megoldható. Mindezek mögött viszont a háttérben ott van a hagyományos SQL vonal, az utasításokat a Hibernate maga generálja. Ez azért hasznos, mert a konfigurációs paraméterek alapján generálódnak az SQL utasítások, ezzel elérhető, hogy a használt adatbázistól függetlenül írhatunk programot Hibernate-ben.

A lekérdező nyelv is hasonlóan a célnak megfelelően lett kifejlesztve. Ebben objektumokkal dolgozunk és szintaktikája nagyon hasonló a hagyományos SQL lekérdezésekhez. Segítségünk lehetnek még a kritérium lekérdezések, amelyekkel objektumorientált módon fogalmazhatunk meg lekérdezéseket. Ha ez mégsem lenne számunkra megfelelő, lehetőségünk van natív SQL utasításokat is végrehajtani, bár ezek a legtöbb esetben sértik a kód hordozhatóságát.

Megfelelő paraméterezés mellett SE és EE környezetben is könnyen használható, az adatbázis kapcsolatok kezelését elvégzi a rendszer.

Az alábbi ábra egy enterprise környezetben futó alkalmazás architektúráját mutatja be:



2. ábra Hibernate alkalmazás architektúrája

Látható, hogy a JDBC/JTA API-k rétege az alkalmazás perzisztenciáért felelős rétege alatt helyezkedik el, a részletekről a Hibernate fog gondoskodni. A fent jelölt modulok feladatai:

- **SessionFactory**: Egy állandó *cache* a már kialakított adatbázis- leképezés tárolására, kliens a *ConnectionProvider* felé valamint *Session*-öket hoz létre. Továbbá opcionálisan fenntarthat egy *cache*-t a tranzakciók között újrafelhasználható adatok számára.

Létrehozása költséges, egyszer történik általában az alkalmazás indulásakor, aztán minden modul ezt a példányt használja.

- **Session**: Egy JDBC csatlakozást lefedő objektum, mely egy párbeszédet reprezentál az alkalmazás és a perzisztens tároló között, továbbá *Transaction* objektumokat kezel és cache-t biztosít a perzisztens objektumok számára. Létrehozása kevésbé költséges. Javasolt a *session-per-request* minta használata, miszerint a felhasználó kérése eljut a szerverhez, ott létrejön egy *Session*, végbemennek az adatbázis műveletek, aztán a *Session* bezárul, és lényeges még hogy egy ilyen kérés minden művelete egy *Session*-ön belül végbemenjen. A *Session* létrejöttekor nem nyílik feltétlen adatbázis kapcsolat, hanem csak ha majd szükség lesz rá.
- **Persistent Objects**: Perzisztens állapotot és üzleti funkciókat megvalósító objektumok. Lehetnek JavaBean vagy POJO (egyszerű Java objektum) objektumok, viszont speciálisak abból a szempontból, hogy pontosan egy *Session*-höz lehetnek hozzárendelve. Amint a Session lezárásra kerül, az alkalmazás szemszögéből újra hagyományos objektumokká válnak.
- **Transient objects**: Olyan objektumok, melyek adott pillanatban nincsenek *Session*-höz rendelve. Lehetnek az alkalmazás által példányosított nem perzisztens, vagy leendő perzisztens objektumok, esetleg egy már lezárt Session által létrehozott objektumok.
- **Transaction**: Szokásos tranzakció fogalom, atomosan végrehajtandó feladatsorozat. Lefedi a JDBC, JTA vagy CORBA tranzakciót. Egy Session több tranzakciót is magába foglalhat
- **ConnectionProvider**: JDBC kapcsolatokat épít ki és kezel. Olyan általános interfészt biztosít az alkalmazás számára, mely elfedi *Datasource*-ot vagy *DriverManagert*; ezek az adatbázissal való, már implementációfüggő kommunikációt hivatottak biztosítani. Kiterjeszthető vagy újraimplementálható a fejlesztő által.
- **TransactionFactory**: *Transaction* példányokat állít elő. Kiterjeszthető vagy újraimplementálható a fejlesztő által.

A Hibernate több kiegészítő interfészt biztosít a programozó számára, melyek implementálásával egyedi, specializált perzisztenciakezelést valósíthat meg.

2.3 EJB

Az Enterprise JavaBean-ek (EJB) a Java EE alkalmazásban központi szerepet betöltő, szabványos felülettel rendelkező, szolgáltatásokat nyújtó szerveroldali komponensek. Egy alkalmazás újrafelhasználható építőelemei. Gyors és egyszerű fejlesztést tesznek lehetővé, transzparensen biztosítják az alkalmazásszerver által nyújtott middleware-szolgáltatásokat.

Jelenleg három típusú EJB létezik.

- **Session bean**: Üzleti folyamatokat reprezentálnak. Minden egyes biztosítani kívánt szolgáltatásnak megfeleltetünk egy metódust, majd az összetartozó metódusok összegyűjtjük egy session bean-be.
- **Entity bean**: Üzleti modell egy entitását reprezentálja. Az tárolásért felelős adatbázissal tartja a kapcsolatot, megfeleltethető az adatbázis egy sorával. Mára már nem sűrűn használják, átvették a helyét különböző egyéb gyártótól származó perzisztencia kezelő rendszerek, egyszerűbb felépítésük és könnyebb tanulhatóságuk miatt.
- **Message-driven bean *(MDB)***: Az aszinkron üzenetkezelésre nyújt megoldást. Szorosan kapcsolódik a JMS-hez, mivel használta úgy történik, hogy üzenetsorba küldünk egy feldolgozásról szóló üzenetet, majd az arra regisztrált MDB elkezdi a feldolgozást, miközben a hívás helyén tovább folytatódik a feldolgozás.

Minden az alkalmazásszerver egy speciális részében, az EJB konténerben fut. Egy EJB komponens része az *bean osztály*, ez tartalmazza az üzleti logikát, itt kell implementálni a klienseknek nyújtandó szolgáltatások metódusait. Szükség van még *interfész*ekre – távoli vagy lokális –, amelyekben a kliensek felé publikálhatjuk a metódusokat, ugyanis azok sosem érhetik el közvetlenül az implementációs osztályt, csak a kliens oldali csonkot, amely egy generált fájl. Alkalmazásszerver által generált osztály még az *EJB objektum*, amely *middleware-szolgáltatások* hívását biztosítja. A *home objektum*, amely gyártó szerepet tölt be, helyátlátszó módon tudjuk példányosítani a kívánt EJB-t.

Az EJB használatát lényegesen leegyszerűsítette az EJB 3.0 verzió. Itt már csak a számunkra lényeges osztályokkal kell foglalkozni, csak az implementációs osztállyal és a használni kívánt interfésszel kell csak foglalkozni. Kötelezően implementálandó metódusok sincsenek, csak azokat kell implementálni, amik lényegesek az elképzelt működéshez. Ha mégis szeretnénk olyan metódust, amelyet korábban implementálni kellett, azt is megtehetjük és a megfelelő annotációval jelölve betöltheti a régi szerepét. Egyéb szükséges információkat, amiket eddig egy külön telepítés leíró hordozott, szintén annotációk segítségével beleírhatjuk

a kódba. Továbbá azok a komponensek, amelyek szintén konténerben futnak, egyszerűbben használatba vehetik az EJB-t a függőség injektálás segítségével.

2.4 AJAX

Az AJAX (Asynchronous JavaScript and XML) napjaink kedvelt technológiái közé tartozik. Ez nem egy új programozási nyelv, hanem inkább egyesíti korábbi technológiák előnyeit, ezért is könnyű megtanulni. A neve kicsit megtévesztő lehet, ugyanis nem teljes egészében aszinkron technológia és nem is minden esetben használ XML-t. Az alábbi technológiákat foglalja magába:

- **HTML**: Formok készítése és elemek azonosítása, elérése az alkalmazáson belül.
- **JavaScript**: Ez a kód az AJAX alkalmazások alapja.
- **DHTML**: Dinamikus HTML, az oldal futás közbeni frissítése <div>, <span> és egyéb dinamikus elemek segítségével.
- **DOM**: Document Object Model, célja a HTML és XML dokumentumok struktúráján történő feladatvégzés.

AJAX-szal a JavaScript közvetlen a szerverrel tud kommunikálni egy úgynevezett *XMLHttpRequest* objektum segítségével. Ezzel az objektummal a JavaScript az oldal újratöltése nélkül tud adatokat küldeni a szervernek. Aszinkron http adatátvitelt használ, így lehetőség van információ részletek cseréjére, amelyből később összeáll az új oldal.

Hagyományos webes alkalmazásnál egy felhasználó interakció egy http kérést generál, ez elmegy a szerverhez, majd a válasz megérkeztével egy új oldal kerül a felhasználó elé. Így a felhasználónak az egyes tevékenységek között várakoznia kell. Ezt küszöböli ki az AJAX alkalmazás úgy, hogy egy *AJAX motor*t iktat a felhasználó és a szerver közé. Ez lassíthatná a feldolgozást, mert egy újabb komponens kerül a feldolgozási sorba, viszont a felhasználó ennek ellenére azt fogja tapasztalni, hogy a felület nagyobb arányban elérhető a korábbiakhoz képest.

Az alkalmazás indulásakor betöltődik a JavaScript-ben írt *AJAX motor* is. Ezután nem minden tevékenység hatására indul http kérés. Ehelyett egy JavaScript hívás történik, ami eljut az *AJAX motor*hoz. Ha a kérés megválaszolásához nem szükséges a szerver beavatkozása, akkor nem is jut el a szerverig, kliens oldalon a JavaScript végzi a feldolgozást. Ilyen lehet például az adatvalidáció. Ha pedig a szerver válasza szükséges, a motor a kérést

aszinkron kérésként továbbítja, mialatt a felhasználó tovább használhatja az alkalmazást. Minden híváshoz tartozik egy *callback* függvény, ami a beérkezett válasz feldolgozását végzi.

Hátrány adódhat viszont a böngészők különbözőségéből, ugyanis nem minden böngésző ugyanúgy értelmezi megjeleníteni kívánt oldalak kódját. Hagyományos AJAX alkalmazás készítésénél erre fokozottan oda kell figyelni.

Mivel feldolgozást JavaScript modulok végzik, probléma lehet, ha a böngészőben le van tiltva a JavaScriptek futtatása. Ez esetben nem lesznek elérhetőek az oldal JavaScriptes szolgáltatásai, esetlegesen az egész program elveszti a funkcionalitását.

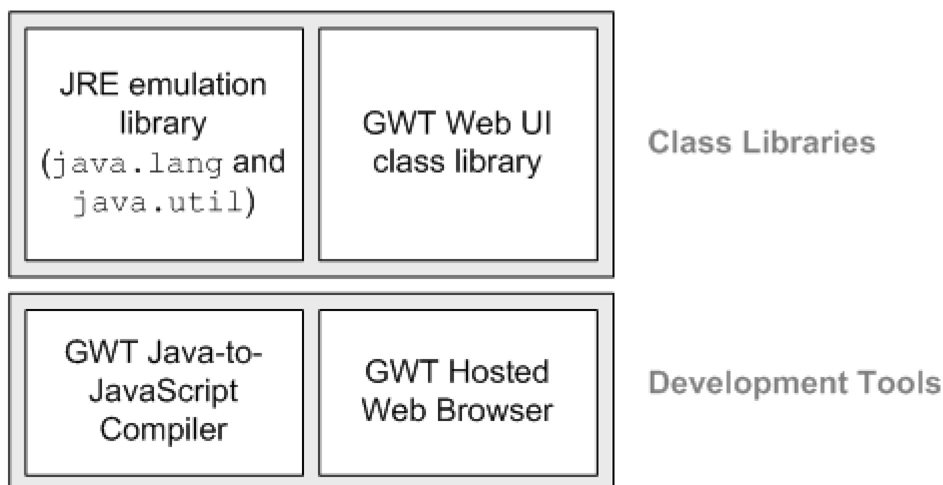
2.4.1 GWT

A *Google Web Toolkit* egyszerűsíti AJAX alkalmazások előállítását. Java kódot kell megírni, melyből a GWT fordító optimalizált JavaScript kódot állít elő. Így lehetőség van jóval komplexebb alkalmazások írására, amelyek később is átláthatóbbak, karbantarthatóbbak lesznek, mint egy robosztus AJAX-os JavaScript kód, nem beszélve arról, hogy egy Java nyelvhez szokott fejlesztőnek sokkal kevesebb időbe kerül így a felület elkészítése. Ha a GWT által kínált lehetőségek mégsem elégítik ki az igényeket, lehetőség van natív JavaScript kód beillesztésére is *JSNI* interfészen keresztül.

Alkalmazás felületének kialakításához úgynevezett *widget*-eket használhatunk. Egy *widget* a GWT alkalmazás olyan megjeleníthető része, amelyet a böngészőben fogunk látni. Java kódban állíthatjuk a példányaiból generált és megjelenített HTML elem tulajdonságait. Mint minden más UI keretrendszerben, itt is vannak panelek, amelyek a komponensek elrendezését határozzák meg. Rendkívül széles a felkínált *widget*ek köre, de ha nem vagyunk velük megelégedve, kiterjeszthetjük őket vagy újakat is implementálhatunk.

Architektúra:

4 fő komponense van a GWT-nek: Java-ból JavaScript-re fordító, hosted módú böngésző és két Java osztálykönyvtár



3. ábra GWT komponensek

- **A fordító:** Java kódból előállítja a JavaScript kódot. Mielőtt alkalmazásszerverre feltöltenénk az alkalmazást (*web mód*), el kell végezni a fordítást. *Hosted mód*hoz nem szükséges.
- **Hosted módú böngésző:** Lehetőségünk van a kód hatásának megtekintésére a JavaScript-re fordítás elvégzése nélkül, ekkor a kód Java-ként fut, melyet a JVM interpretál. *Hosted mód*ban mivel JVM interpretálja az alkalmazást, hagyományos módszerekkel végezhetjük a debugolást.
- **JRE emulációs könyvtárak:** A széles körben használt osztályok JavaScript implementációi (*java.lang.** és *java.util* bizonyos osztályai). Egyéb osztályok használata GWT-ből nem támogatott.
- **GWT Web UI osztály könyvtár:** Interfészek és osztályok gyűjteménye, melyek segítségével létrehozhatjuk a felhasználói interfész *widget*-jeit. Gombok, szövegek, beviteli mezők és hasonló eszközök vannak. Ezek a GWT legfőbb elemei.

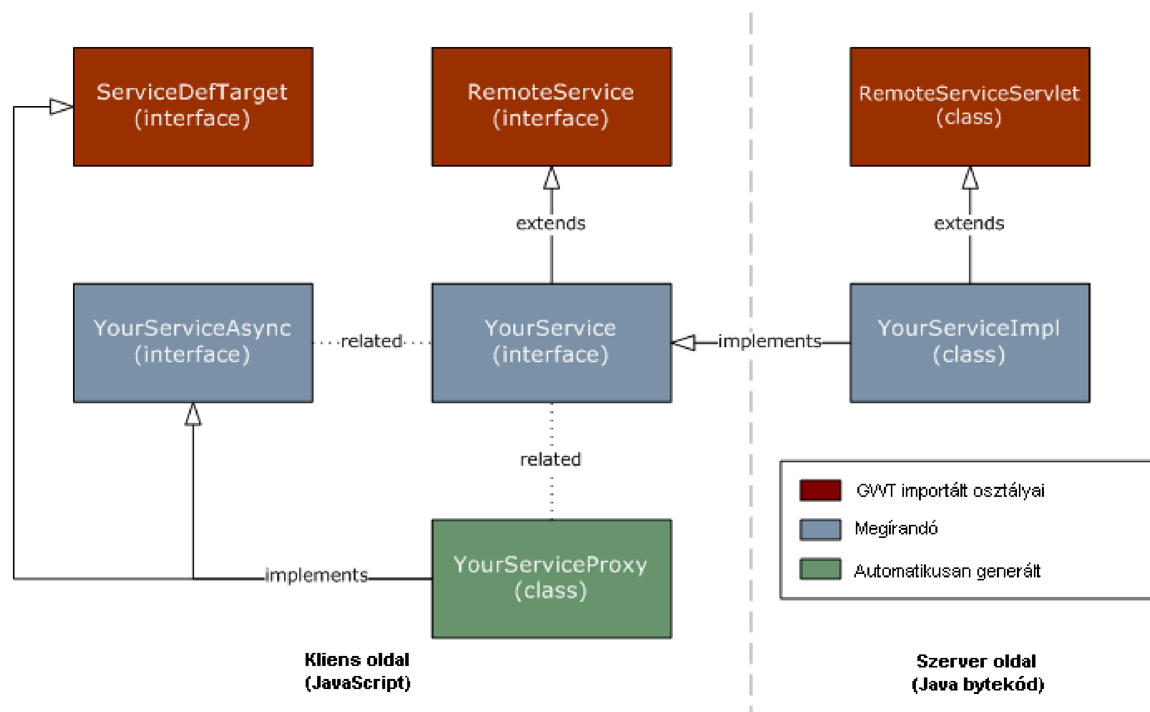
A fordítás során automatikusan létrejönnek az elterjedt böngészőket támogató implementációk, így a programozónak nem kell foglalkoznia minden támogatott böngészőhöz külön-külön implementáció megírásával. Az optimalizálás úgy történik, hogy a JavaScriptbe már nem kerülnek bele az esetleges használatlan importok és bizonyosan hívásra nem kerülő kódrészletek. Ha nemzetközi local-okat támogatóan írtuk meg az oldalakat, azokhoz is külön-külön generálódik a script mindegyikhez. Minden egyes kliensnek, akik megjelenítik az oldalt, csak az adott böngésző specifikus és adott local-hoz tartozó implementációt kell betöltenie.

Egy GWT alkalmazás önálló egységekből épül fel, amelyeket moduloknak hívunk. Minden modult egy-egy XML fájl definiál. Egy modul megfelel egy hagyományos webalkalmazás egy oldalának.

Az alkalmazás hálózaton keresztül jut el a felhasználóhoz, annak böngészőjébe, ott már kizárólag JavaScript-ként fut. Mindent, ami a böngészőben történik, kliens oldali folyamatnak nevezünk. Kliensoldalon korlátozottan használhatjuk pl. a reflection technikáit, JavaScript-ben nincs objektum finalizáció és másfajta lebegőpontos ábrázolást használ mint a Java.

RPC:

A GWT Java Servleteken keresztül biztosít hozzáférést a szerver erőforrásaihoz, ehhez szükség van a *gwt-servlet.jar* osztálykönyvtárra. A szerver oldali kódot, amelyet a kliensből meghívunk, *service*-nek nevezzük (szolgáltatás). Csak a logikát kell megírni meg néhány kötelezően implementálandó osztályt és interfészt, a többi osztályt pedig a keretrendszer generálja, valamint szerializálja a kommunikációban résztvevő objektumokat, ehhez az objektumoknak implementálniuk kell az *IsSerializable* interfészt.



4. ábra GWT RPC komponensei

Kliens oldalon definiáljuk a service szinkron interfészét, ebben van benne minden hívható metódus specifikációja. Szerver oldalon implementáljuk a szinkron interfészben definiált metódusokat. Hogy meg is lehessen ezeket hívni, a kliens oldalon szükség van még

egy aszinkron interfészre a szinkron interfész alapján, csak ezeknek a metódusoknak void a visszatérési értéke és a paraméter lista végén egy *callback* objektumot is kapniuk kell. Az elnevezési konvenciókat követni kell, a service neve mindhárom esetben megjelenik, a szerver oldali osztály *Impl* végződést kapott, a kliens oldali aszinkron interfész pedig *Async* végződést.

Az implementációs osztálynak kötelezően ki kell terjesztenie a *RemoteServiceServlet* osztályt. A *callback* objektumban előírhatunk teendőket sikeres és sikertelen végrehajtás esetére is. Sikertelen végrehajtásnak tekinthető, ha a hívás során bármilyen nem kezelt kivétel történik.

2.5 Jasper Reports

Igazi üzleti alkalmazás nincs riportok nélkül. Ezen funkciók biztosítására is kész keretrendszerek állnak rendelkezésünkre. A Jasper Reports üzleti felhasználásra is ingyenes, csak a dokumentációért kell fizetni, ugyanis a javadoc dokumentációból lényegi információkhoz nem jutunk hozzá.

A riport szerkezetét egy speciális felépítésű xml fájl tárolja (*JRXML*). Ezt le kell fordítani, így jön létre az a sablon objektum (*JASPER*), amit a Java használ a riport létrehozására. A fordítás történhet futási vagy fordítási időben, a lefordított objektum újrafelhasználható. Az xml-ben tároljuk a riport adatait biztosító lekérdezést, a lefordított riport objektumnak később csak az adatforrást és esetleges egyéb paramétereket kell átadni, hogy a sablonunkat feltöltsük.

A riportsablon elkészítéséhez elég egy egyszerű szövegszerkesztő, amelyben összeállítjuk az xml-t. Ez viszont igen aprólékos munka, egyszerűbb mód ha a rendszerhez fejlesztett grafikus designer eszközt, az *iReport*-ot használjuk.

A riportot többféle kimeneti formátumban megkaphatjuk – HTML, PDF, CSV, stb. –, vagy akár ki is nyomtathatjuk. Arra azonban ügyelni kell, hogy nyomtatás esetén csak azokat a nyomtatókat éri el a program, amelyek a szerver géphez vannak csatlakoztatva.

Az előzőekben tárgyalt technológiák mellett az alábbi elemekből épül fel a webalkalmazásom:

- **GlassFish v2.1**
- **Apache Derby 10.2.2.1 (a GlassFish része)**
- **Hibernate 3.3.1**
- **EJB 3.0**
- **Java Servlet**
- **GWT 1.5.3**
- **GWT-Ext 2.0.6**
- **JasperReports 3.1.2**

A fejlesztés során NetBeans 6.5 fejlesztőkörnyezetet használtam az éppen elérhető iReport és GWT kiegészítésekkel a munkám megkönnyítése érdekében.

3 Készletnyilvántartó alkalmazás követelményeinek feltárása és tervezése

Az alkalmazás követelményeinek feltárásában segítségemre volt egy korábbi, Clipper nyelven írt program, amelynek használata azért vált nehézkessé, mert DOS alá íródott. A készlet egyszerűen a forgalmon keresztül volt kezelve és a riportok helyes kezelése érdekében minden évben új készletkezelést kellett indítani.

A készletkezelést eladásokon és vételeken keresztül oldom majd meg, amelyekben a visszárus forgalmakat is kezelni szándékozom. Eladásoknál is lehetséges szállítólevélen keresztüli, vagy közvetlen számlás értékesítés. A riportokhoz a készletmozgáson, áfa és készletértéken alapuló kimutatásokon kívül új szolgáltatásként megvalósításra kerül a vevők tartozásainak összesítése. Készletmozgás és fizetendő áfa lekérdezhető valamilyen intervallumra, a készletérték és a vevői tartozások mindig az aktuális állapotot tükrözik. Alapvető követelmény még a készlet elemeit alkotó árucikkek adminisztrációja, partnerek karbantartása, valamint a leltározás.

3.1 Adatbázis modell

Az alkalmazás biztos működéséhez rendkívül fontos az alapul szolgáló adatbázis pontos megtervezése. A kiinduló program relációs adatbázis modellje igen egyszerű volt, három darab táblán alapult a program működése. A tárolást igyekeztem objektumorientált alapokra helyezni, ezért kisebb egységekkel modelleztem a probléma résztvevőit. A tárolandó osztályok meghatározása után meg kellett határozni az egyedekre jellemző tulajdonságokat. Ezek tudatában már megtörténhetett az objektum-relációs leképezés a Hibernate segítségével.

3.1.1 Termék

Ez reprezentálja a készlet elmeit. Az egész rendszer működése valamilyen kapcsolatban van ezen adatokkal.

- **cikkszám**: elsődleges kulcs
- **név**
- **nettó vételár**
- **nettó eladási ár**
- **áfa**: 0 és 1 közé eső valós érték

- **készlet**: termékből készleten lévő mennyiség
- **vámtarifaszám**

Bruttó árak tárolásának nem láttam értelmét, ugyanis az meghatározható a nettó ár és az áfa szorzataként.

3.1.2 Leltár

Készleten lévő mennyiségtől való eltérést tárol. Egyes leltárak alkalmával a termék készleten lévő mennyisége mennyivel lett módosítva.

- **leltárazonosító**: elsődleges kulcs
- **dátum**: bejegyzés dátuma
- **termék**: külső kulcs
- **eltérés**: egész érték

3.1.3 Partner

Ez az alkalmazásban egy absztrakt osztály, két leszármazottja van a **vevő** és az **eladó**. Az adatbázisban az utóbbi két entitás egy táblában van tárolva. Adatbázis szinten a lentebb felsorolt jellemzők mellett megjelenik még egy diszkriminátor is, melyet a Hibernate nem hoz fel az alkalmazás szintjére. Számlákkal illetve szállítólevelekkel állhatnak kapcsolatban.

- **partnerazonosító**: elsődleges kulcs
- **név**
- **város**
- **irányítószám**
- **helyiség**
- **házszám**
- **emelet**
- **ajtó**
- **telefonszám**
- **számlaszám**
- **adószám**

3.1.4 Számla

Valóságban megjelenő számlát tükröz. Viszont absztrakt osztály, konkrét leszármazottai az **eladási számla** és **vételi számla**. Ezek eladásokhoz kötődnek, felvételükkor a készlet

automatikusan változik. Ezek készpénzes számlák rögzítésére alkalmasak. Előbbiek leszármazottja az **átutalásos számla** – mindkét típusú számlából létezik átutalásos is, amelynél megadható egy határidő és a kiegyenlítés állapota. Szintén egy táblában tárolódik az összes számla.

- **számlaszám**: elsődleges kulcs
- **partner**: eladási számlánál vevő, vételi számlánál beszállító
- **tételek**: adatbázisban külső kulccsal vannak a számlához kapcsolva a tételek oldaláról
- **kiállítás dátuma**
- **kedvezmény**: jelenleg nem használt

Átutalásos számlák esetén további jellemzők:

- **fizetési határidő**
- **fizetve**: logikai érték

Készpénzes számlákat mindig fizetett státuszúnak tekinthetjük, átutalásosaknál a kiegyenlítést külön jelezni kell.

3.1.5 Számlatétel

Számla tételei, amelyek mindig pontosan egy számlához kapcsolódnak. Mivel a termékek tulajdonságai az idő folyamán változhatnak, viszont egy kiállított számla tételei állandóak, ezért kiállítás alkalmával egy-egy számlatételben le kell másolni az eladott termék adatait. Ezzel biztosítható, hogy a visszanyitott számla minden esetben ugyanazt az eredményt fogja mutatni.

- **tételazonosító**: elsődleges kulcs
- **cikkszám**
- **név**
- **mennyiség**
- **nettó ár**
- **áfa**
- **vámtarifaszám**

3.1.6 Szállítólevél

Valóságban megjelenő szállítólevelet tükröz. Szállítólevél kiállításakor a készletet csökkenteni kell a szállított tételekkel. Fizetés esetén hozzá lehet kapcsolni egy számlát, amelynek kiállításakor a készlet változatlan marad.

- **szállítólevélszám**: elsődleges kulcs
- **partner**: vevő
- **tételek**: adatbázisban külső kulccsal vannak a szállítólevélhez kapcsolva a tételek oldaláról
- **kiállítás dátuma**
- **kedvezmény**: jelenleg nem használt
- **kapcsolt számla**: ha nem állítottunk még ki számlát a szállítólevélhez, akkor null

3.1.7 Szállítólevél tétel

Szállítólevél tételei, amelyek mindig pontosan egy szállítólevélhez kapcsolódnak. Mivel a termékek tulajdonságai az idő folyamán változhatnak, viszont egy kiállított szállítólevél tételei állandóak, ezért kiállítás alkalmával egy-egy szállítólevél tételben le kell másolni az eladott termék adatait. Ezzel biztosítható, hogy a visszanyitott szállítólevél minden esetben ugyanazt az eredményt fogja mutatni.

- **tételazonosító**: elsődleges kulcs
- **cikkszám**:
- **név**
- **mennyiség**
- **nettó eladási ár**
- **áfa**
- **vámtarifaszám**

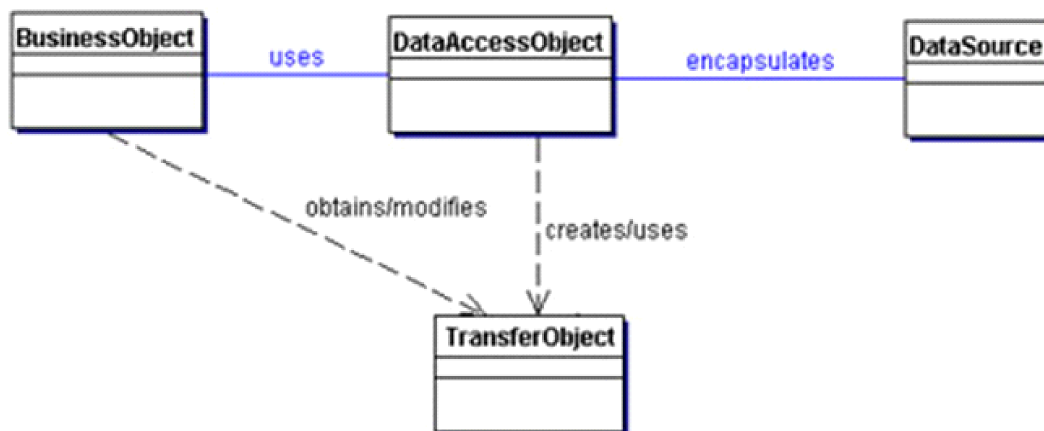
3.2 Rendszer modell

A mai alkalmazásoknál általános elvárás, hogy *MVC* (*Model-View-Controller*) minta szerint épüljenek fel, amely már a Smalltalk óta elfogadott tervezési minta. Nagy adatmennyiséggel dolgozó alkalmazásoknál gyakori igény a felhasználói felülethez (*nézet*) és az adatábrázoláshoz (*modell*) tartozó követelmények szétválasztása. Így az adatok szervezése függetlenné válik a felhasználói felülettől és a felület sem függ az adat struktúrájától. Ezt az

MVC úgy éri el, hogy az adatelérést és az üzleti logikát külön választja a felhasználó felülettől. Ezek közé még beiktat egy *vezérlő*t, ami a felhasználói interakciókra reagál.

Nagy mennyiségű adattal dolgozó alkalmazásoknál nem mindegy, hogyan is oldjuk meg az adatkezelést. Ennek egyik megoldása lehet a *Data Access Object* (*DAO*) tervezési minta, amely egy interfész beiktatásával biztosít kapcsolatot az üzleti komponens és az adatbázis között. Moduláris fejlesztésnél igen hatékony, ugyanis világosan elkülöníti az üzleti logikát és az adatkezelést. Meghatározott interfészen keresztül van kapcsolatunk az adatbázissal, nem kell tudni hogyan is történik a perzisztencia. Így a logika érintetlenül hagyásával változtathatunk a perzisztencia módszerén is, ha az interfész változatlan marad.

A DAO interfészben meghatározunk a hozzáférés módjait, ehhez tartozik egy implementációs osztály. Az üzleti logikát megvalósító osztálynak a *DAO* csak *szállító objektum*okat – más néven *value object* (*VO*) – ad át, ezzel biztosítva azt, hogy entitás ne kerüljön tranzakciós környezeten kívülre, így az üzleti logika leprogramozása kisebb körültekintés igényel, a tranzakciókezelés és a teljes adathozzáférés be van zárva a *DAO*-ba. Többször használnak *gyártó osztály*okat a *DAO*-k létrehozására, amely mindig az éppen kívánt objektumot szolgáltatja a felhasználónak.



5. ábra Data Access Object

DAO tervezésének több módja van. Ha több adatbázissal kommunikál egy alkalmazás, egy *DAO*-n keresztül elérhetjük egy adatbázis összes szolgáltatását. Másik lehetőség – amit én is használni fogok, hogy egy entitáshoz kapcsolódó műveleteket gyűjtünk össze egy *DAO*-ba.

3.2.1 Üzleti folyamatok

Az alkalmazás alapja a készletkezelés, ezért ennek megtervezésére komoly gondot fordítottam. Módosítható entitások a termékek és a partnerek. Új terméket felvenni mindig csak 0 kezdő készlettel lehet és a készleten történt minden egyes változtatásnak nyoma kell hogy legyen a programban. Termék és partner törlése nem értelmezett, mert hivatkozások lehetnek rá.

A készlet változtatása nyomot hagyó eseményeken keresztül történik, ami annyit jelent, hogy egy tranzakción belül felvesszük a változás módját rögzítő eseményeket az adatbázisba, majd ennek sikeressége után egy adatbázis trigger megfelelően módosítja az adott termék készleten lévő darabszámát. Ilyen esemény lehet vétel vagy eladás, amelynek nyoma egy számla vagy egy szállítólevél. Különleges eset még, ha egy szállítólevélhez szeretnénk számlát kiállítani. Jelenleg ezt úgy támogatja a rendszer, hogy a számla adatait megadhatjuk, de a tételek ugyanazok lesznek, mint a szállítólevélen, résszámla kiállítása nem támogatott. Ilyen kapcsolt számla kiállításakor természetesen nem fut készletmódosító trigger. Szállítólevél és számla tételeinek tárolásakor az adott termék aktuális tulajdonságai lemásolódnak, ugyanis egy termék neve és ára az idő folyamán változhat, viszont egy kiállított számla tételei minden megtekintés alkalmával ugyanazok kell, hogy legyenek.

További funkció, amely hatással van a készletre, a leltár. Ezzel lehet korrigálni a készleten lévő darabszámot az éppen aktuális helyzethez képest. Itt is előbb minden termékre letároljuk az eltérés mértékét, majd ez alapján a trigger aktualizálja a darabszámot.

Tisztázni kellett még mi az, amit a vevő részéről tartozásnak tekintünk. Készpénzes számla a kiállítás pillanatától mindig kiegyenlített. Szállítólevél kiállításával nem történik fizetés, tehát ha nincs hozzá kapcsolt számla, akkor az összege tartozás, egyébként figyelmen kívül hagyható. Átutalásos számlák mindaddig tartozást tükröznek, amíg kiegyenlítésüket nem jelöltük.

3.2.2 Összefoglalva az alkalmazás feladatai

- ✓ Termék és partner felvétele, megjelenítése, módosítása
- ✓ Leltározás, korábbi leltárak megjelenítése
- ✓ Szállítólevél felvétele, megjelenítése, szállítólevélhez számla kiállítása
- ✓ Számla felvétele, megjelenítése, átutalásos számla kiegyenlítése
- ✓ Áruforgalom és fizetett áfa lekérdezése egy adott intervallumra
- ✓ Jelenleg fennálló vevői tartozások és készletérték megjelenítése

4 Készletnyilvántartó alkalmazás megvalósítása

A kezdeti specifikáció alapján hozzákezdtem az alkalmazás elkészítéséhez. A fejlesztés folyamata leginkább az inkrementális modellhez áll közel. Az architektúra létrehozása után minden egyes új funkciót külön kezdtem implementálni és egymás után kerültek bele a rendszerbe. Ha egy funkció már a várakozásoknak megfelelően működött, akkor következhetett a következő megvalósítása.

A fejlesztés a Hibernate konfigurálásával és az osztályok objektum-relációs leképezésével kezdődött. Adatbázis kapcsolatnak egy JNDI névvel regisztrált adatforrás (*DataSource*) van megadva, így egyszerűbb az alkalmazás üzembe helyezése. A kívánt adatbázis elérési paramétereit az alkalmazásszerveren beállítjuk egy adatforrás objektumban, majd azt regisztráljuk. Esetleges adatbázis változások nem érintik majd a kódot, csak a szerver konfigurációjában kell ezeket megváltoztatni.

A session-öket JTA segítségével kezelem és konténer menedzselt tranzakciókat használok. Ez a logika leprogramozásánál megkönnyíti a dolgot, ugyanis az alkalmazásszerverbe való integrálhatóság lehetővé teszi, hogy egy Hibernate sessiont egy JTA tranzakcióhoz kössünk. Konténer menedzselt tranzakciók esetén az EJB deklaratívan végzi a tranzakciókezelést – indítás, véglegesítés, visszagörgetés. Így aktuális session kérésekor a Hibernate mindig a megfelelő tranzakcióhoz kapcsolt session-t adja vissza. Az első *getCurrentSession()* hívás után a session hozzákapcsolódik az aktuális tranzakcióhoz. Ezen a session-ön nem lehet végrehajtani explicit véglegesítést vagy visszagörgetést. Ha a session-ön végzett műveletek kezeletlen kivétel kiváltása nélkül végbementek, a tranzakció véglegesítődni fog. Ha kezeletlen kivétel váltódik ki, az egész tranzakció visszagörgetődik. A tranzakció vége előtt a session tartalma az adatbázisba íródik, aztán ha vége a tranzakciónak a session bezárul. Mindezen feladatok elvégzéséhez az alkalmazott beállítások mellett semmi féle kódot nem kell írni.

A Hibernate egyik képességét kihasználva automatikusan hozom létre a tároláshoz szükséges táblákat. Az osztály leképezéseket tartalmazó XML-ek alapján a *SessionFactory* minden egyes létrehozásakor ellenőrzi a szükséges táblák meglétét, ha nem talál ilyeneket, az XML alapján létrehozza.

Ezután következett az alkalmazás felületének kialakítása. Egységes felületet terveztem, az egyes funkcióknak meg vannak az elérései, viszont ezek akkor váltak használhatóvá, ha a hozzá tartozó inkremenst integráltam az alkalmazásba.

A felület összeállításához GWT mellett GWT-Ext-et is használtam. Ez egy widget-könyvtár, amely a GWT-nél szélesebb választékban tartalmaz elemeket, mindezeket kész stílussal, így azonnal használatba vehetjük. A GWT widgetek között sok olyan van, amelyekhez még utólagosan létre kell hozni a kívánt stílust. Bár így nagyobb szabadságot kap a fejlesztő a felület összeállítsa során, én most nem erre helyeztem a hangsúlyt, így elégnek találtam az előre elkészített GWT-Ext-es widgeteket. (*24. ábra*) Ez a felület inkább hasonlít egy hagyományos asztali alkalmazás felületére, de ezekkel az eszközökkel így volt egyszerűbb megoldni, mindamellett megmaradtak a vékony kliens által nyújtott előnyök.

Az adatok továbbítása során a *DAO* minta alkalmazásával végre kell hajtani egy átforgatást, mikor az objektum átkerül az üzleti logika metódusaihoz vagy az üzleti logikától érkezik olyan objektum, amit tárolni kell. Mikor átforgatjuk az entitást *VO*-ba, nem kell minden adattagot átmásolni, csak azokat, amikre szükség van a feldolgozás szempontjából, így teljesítménynövelés érhető el azzal, hogy kisebb objektumokat kell a hálózaton küldözgetni, bár ez a jelenleginél jóval nagyobb alkalmazásokban jelent javulást. Egy entitáshoz több *VO* is tartozhat, függően az elvégezni kívánt művelettől.

Ezekhez a *VO*-khoz szükséges még létrehozni másik, hasonló szerkezetű tároló objektumokat (*WebVO*), amelyek a kliens oldalon fogják tárolni az objektum adatait. A *WebVO*-ból fordítás során JavaScript objektum lesz, így oldható meg a kliens oldali reprezentáció. Minden *GWT service* csak olyan objektumot adhat vissza vagy fogadhat el paraméterként, amely kliens oldalon kezelhető. Így a service-ben is szükséges egy átforgatás, amely elvégzi a konverziót *VO* és *WebVO* között.

A felület kialakításánál át kellett gondolnom, hogyan is kezeljem az aszinkron hívásokat olyan esetekben, amelyek tulajdonképpen nem aszinkron feldolgozást feltételeznek. GWT-vel az üzleti logika szolgáltatásait csak aszinkron hívásokon keresztül lehet igénybe venni. Ezt hasznos is lehet például leltározásnál. Elindítjuk a leltározási folyamatot, eközben pedig foglalkozhatunk tovább egyéb dolgokkal. Ha viszont tárolt értékeket kérdezünk le, amelyeket meg akarunk tekinteni és később dolgozni velük, a lekérdezés folyamata alatt nem szerencsés, ha egyéb funkciókat kezd használni a felhasználó és valahogy tudatni kell vele, hogy a kérése feldolgozás alatt van.

Erre a problémára olyan megoldást találtam, hogy ha egy felhasználói interakció olyan szolgáltatást használ, amelynek eredményét utána szeretnénk megtekinteni vagy felhasználni, és mindezek előtt valami egyéb tevékenység veszélyeztetné a felület vagy az adatbázis

konzisztenciáját, az aszinkron hívás elküldésekor egy értesítés kerül megjelenítésre, hogy a feldolgozás folyamatban van. Ha megkaptuk a szervertől az eredményt, az értesítés eltűnik és folytatható a munka a megfelelő adatokon. Egyéb esetekben – mint a leltáros példánál – az aszinkron hívás elküldését követően tovább folytatható a munka, a feldolgozás befejeztével ugyan kapunk visszajelzést a szervertől, de sikeres végrehajtás esetében nem hajtunk végre egyéb tevékenységeket. A hibákat természetesen mindkét esetben jelezni kell a felhasználónak.

Az első problémába akkor ütköztem, mikor számla és szállítólevél kiállításánál egy listában szerettem volna megjeleníteni a választható termékeket. Viszont az erre szánt eszköz, a GWT-Ext-es ComboBox vagy előre megadott adatokkal tölti fel a listát, vagy valamilyen http címről kapott adatokkal tud dolgozni. Ekkor döntöttem úgy, hogy Java servleteket is használok az alkalmazásban. Különösebben új problémát ez nem jelentett, ugyanúgy lehet EJB-ket használni így is, csak nem visszatérési értékben adom vissza az eredményt, hanem a response kimenetére írom.

Ennek megoldására a *JSON* formátumot használom. Ez egy ember számára könnyen olvasható és gépek számára könnyen létrehozható és értelmezhető adatátviteli formátum. Ismeri az objektumokat, tömböket és egyszerű értékeket és mindezt szövegként reprezentálja.

Ilyen módon lett megoldva a vevői tartozások megjelenítése is.

Mivel így szövegeket kell továbbítania a servletnek, figyelembe kell venni azt a tényt, hogy a Java servletei szöveg továbbításánál latin 1-es kódolást alkalmaznak, így az ékezetes betűk helytelenül fognak megjelenni a hívás helyén. Ennek megoldására egy servlet *Filter*-t írtam, amely minden hívás esetén beállítja a request karakterkódolását UTF-8-ra vagy az éppen megadott kódolásra.

Gondolkodnom kellett még azon, hogyan jelenítsem meg a számlákat és a szállítóleveleket. HTML oldalon való megjelenítésük nem tűnt praktikusnak, mert megjelenítés estén feltételezhető, hogy ki is nyomtatják őket. A nyomtatás eredménye viszont böngészőnként más és más lehet. Ezért döntöttem a PDF kimenetei formátum mellett. A servletnek átadom a számla vagy szállítólevél lekérdezéséhez szükséges információkat, az összeállítja a PDF dokumentumot majd a response-ban visszaadja a kliensnek. Az így megkapott dokumentumra minden esetben rákerül egy vízjel, hogy jelezze az nem eredeti példány.

Fel kell tüntetni továbbá a dokumentum kiállítóját is a partner mellett. Ez megtehető úgy, hogy a sablonba hard-kódoljuk a kívánt adatokat, viszont én azt a megoldást választottam, hogy a partnerek táblába felveszem a vállalkozás tulajdonosának adatait is és innen lekérdezhetővé válik. Ezt az alkalmazás üzembe helyezésekor kell megtenni. Ennek a felületen nem kell megjelennie a partnerek között, nem hoztam létre hozzá új osztályt és az új diszkriminátor érték miatt a Hibernate a partnerek kezelése során nem vesz tudomást erről a sorról.

A böngészők különbözőségéből már sok probléma adódott és most én is belefutottam egy ilyenbe. A vevők tartozásait megjelenítő GWT-Ext-es *GridTree* ugyan helyesen működik FireFox alatt, de az Internet Explorer hibát jelez a komponens megjelenítése közben. A vevők összesített tartozását így is meg lehet tekinteni, de a tételes megjelenítés csak FireFox alatt működik. Sokáig kerestem a hiba okát, de JavaScriptet debugolni igen nehézkes.

A GWT-Ext legutóbbi kiadása ezen év elején jelent meg, ebben újdonság a *GridTree*. Így megírtam a tapasztalt hibát a fejlesztőknek és remélem a következő verzióban már javítva lesz.

Az alkalmazás jelenleg nem kezel felhasználói csoportokat, nincs beléptetés. Mindenki, aki hozzáfér az alkalmazáshoz, ugyanolyan jogokkal rendelkezik.

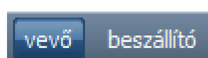
5 Készletnyilvántartó alkalmazás felhasználói kézikönyve

Bár az alkalmazást úgy készítettem, hogy használata egy hasonló alkalmazást még nem látott felhasználónak is egyértelmű és elhibázhatatlan legyen, ebben a részben a használattal kapcsolatos tudnivalókat gyűjtöm össze.

5.1 Felületi elemek

Az alkalmazás használata közben a következő felületi elemekkel találkozik a felhasználó:

- **Gomb**

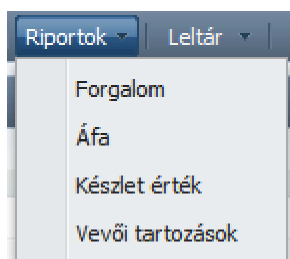


6. ábra Gomb

Megnyomására végbemegy az általa jelölt akció.

Van ilyen gomb a menüsorban és a felület egyéb helyein.

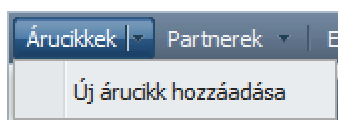
- **Menügomb**



7. ábra Menügomb

A gomb, amelynek felirata mellett egy lefelé mutató nyíl van, kattintásra lenyit egy menüt, amelyből választhatóak az akciók.

- **Osztott menügomb**



8. ábra Osztott menügomb

A gomb, amelynek felirata és a nyíl között egy függőleges elválasztó van.

A szövegre kattintva végbemegy egy akció, a nyílra kattintva lenyílik egy menü, amelyből további akciókat lehet választani.

- **Táblázat**

cikkszám	megnevezés	készlet	jelenlegi készlet
102101	vegyesvágott kg	0 db	0 db
102512	csemege uborka sós kg	0 db	0 db

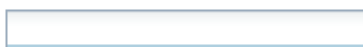
9. ábra Táblázat

A táblázatnak mindig van fejléce. Bizonyos rekordok szerint lehet rendezni a táblázat tartalmát. Ez a rekord fejlécére való kattintással elvégezhető.

A táblázat tartalmának megjelenítése esetenként eltérő lehet.

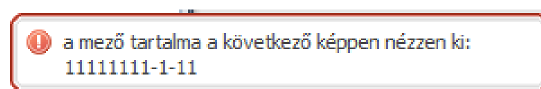
- **Beviteli mezők**

A beviteli mezők minden bevitel előtt validálva vannak, a hibás bevitel esélyét a lehet legkisebbre csökkentettem. A számot váró beviteli mezőkbe csak számot lehet bevinni. Tizedes tört bevitele esetén a tizedesvessző helyett tizedespontot kell használni.



10. ábra Kötelező mező

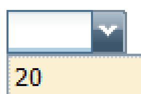
11. ábra Hibás mező



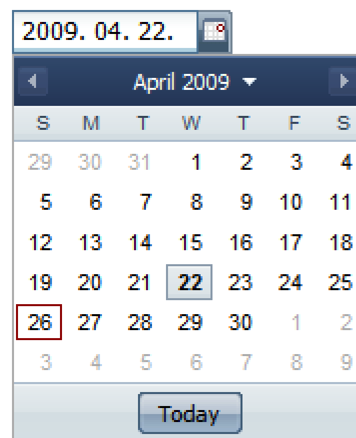
12. ábra Beviteli hiba

A kötelezően kitöltendő mezők kék háttérrel vannak jelölve. Ha egy mező validálása sikertelen, ezt a mező mellett megjelenő piros felkiáltójel jelzi. Az egeret erre a jelzésre mozgatva gyorstipp szerűen megjelenik a hiba oka.

13. ábra ComboBox



14. ábra Lenyíló lista



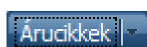
15. ábra Dátum beviteli mező

A ComboBox arra szolgál, hogy olyan értéket adhassunk meg, amelyet a ComboBox listája tartalmaz. Történhet listából való választással vagy begépeléssel. Gépelésnél két karakter bevitele után megjelenik a lista a még lehetséges értékekkel. Olyan értéket nem lehet gépeléssel bevinni, ami nincs a listában.

Lenyíló lista estén a lista elemeiből kell egyet kiválasztani.

Dátum beviteli mező esetén a kívánt dátumot megadhatjuk gépeléssel, amikor a következő formátumot kell használni: ***<év 4db szám>. <hónap 2db szám>. <nap 2db szám>.*** A mező melletti naptár gombra kattintva megjelenik egy grafikus naptár, amelyből egérrel lehet kiválasztani a kívánt dátumot. Ennek a nyelve még angol.

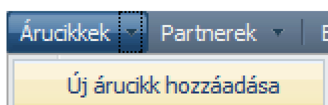
5.2 Árucikkek megtekintése



Az osztott gomb megnyomása után végbemegy az árucikkek lekérdezése, amely ezután megjelenik a böngészőben.

Bármelyik rekord szerint lehet rendezni a táblázatot.

5.2.1 Új árucikk felvétele

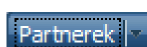


Új árucikket felvenni az árucikkek menüjéből lehet. Az ár megadása nem kötelező, ez a későbbiekben is megtörténhet.

5.2.2 Árucikk módosítása

Egy árucikk adatait módosítani lehet, amennyiben az árucikkek listájában a termék nevére vagy cikkszámára kattintunk. Ekkor megjelennek az árucikk adatai a felvételhez hasonló ablakban. A cikkszám és készlet módosítása nem megengedett.

5.3 Partnerek megtekintése

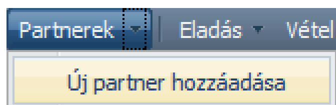


Az osztott gomb megnyomása után végbemegy a partnerek lekérdezése, amyel ezután megjelenik a böngészőben.

Alapértelmezetten minden típusú partner megjelenik (vevő, beszállító). A menü alatt lévő gombokkal lehet szabályozni milyen partnereket, szeretnénk látni.

A táblázat rendezhető név, cím és típus szerint.

5.3.1 Új partner felvétele

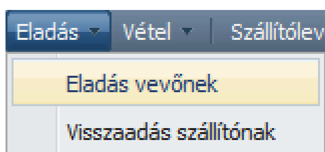


Új partnert felvenni az partnerek menüjéből lehet. Választható milyen típusú partnert szeretnénk felvenni. Vevőnél csak a név megadása kötelező, beszállítónál a számlaszám és az adószám is. Az összes többi tulajdonság opcionális, viszont a számlákon ezekkel az adatokkal fog szerepelni.

5.3.2 Partner módosítása

Egy partner adatait módosítani lehet, amennyiben az partner listájában a partner nevére kattintunk. Ekkor megjelennek a partner adatai a felvételhez hasonló ablakban. A partner típusának megváltoztatása nem megengedett.

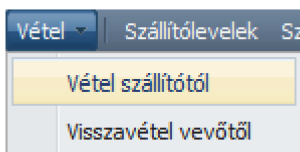
5.4 Eladás



Eladás minden olyan készletmozgás, ami számlával vagy szállítólevéllel igazolt és a készlet csökkenését eredményezi. Az ilyen tranzakciókról szóló számlák az eladási számlák között lesznek megtalálhatóak. Vevőnek történő eladáskor lehetőség van választani szállítólevél vagy készpénzes számla kiállítsa közül.

Eladni minden egyes terméket az aktuálisan az adatbázisban szereplő áron lehet. Ha másként szeretnénk az eladási dokumentumon szerepeltetni, eladás előtt meg kell változtatni a termék árát.

5.5 Vétel



Vétel minden olyan készletmozgás, ami számlával igazolt és a készlet növekedését eredményezi. Az ilyen tranzakciókról szóló számlák a vételi számlák között lesznek megtalálhatóak.

Vevőtől való visszavételnél ha az adatbázisban eltérő árat szeretnénk szerepeltetni a vételi számlán, előtte meg kell változtatni a termék árát.

Szállítótól történő vétel esetén szerepelhet a számlán az adatbázisban szereplő ár, vagy új ár is megadható, a termék ára pedig erre fog módosulni.

5.6 Tétellista kitöltése

Készletmozgási dokumentum felvételekor ezen lista elemei fognak megjelenni a dokumentumon. A listához új árucikket az ***Új tétel*** gombbal lehet hozzáadni. Ekkor egy üres sor kerül a táblázat elejére. A megnevezést ComboBox segítségével lehet megadni. Vételi számla esetén minden árucikk választható, egyéb esetben csak olyanok, amelyeknek meg van adva az ára. Árucikk kiválasztásakor a hozzá tartozó értékek betöltődnek a táblázatba, csak mennyiséget kell még megadni.

Egy tétel megnevezésének módosítása nem lehetséges. Ha hibás felvétel történt, úgy lehet javítani, ha az adott sor rendelt mennyiségét 0-ra állítjuk, és újra felvesszük a kívánt tételt.

Új tétel

cikkszám	megnevezés	mennyiség	nettó eladási ár
		0 db	
109021	baba törlő	0 db	283.33 Ft
109409	berbero maxi	40 db	575.00 Ft
109021	baba törlő	2 db	283.33 Ft

16. ábra Tétellista - általános

Vétel számla esetében a korábbi teendők annyival egészülnek ki, hogy a *megnevezés* és a *mennyiség* oszlopok mellett a *nettő vételár* és *nettó eladási* ár is kötelezően megadandó. Ha ez korábban meg volt adva a terméknél, megjelennek, lehet módosítani is. Ha nem volt korábban megadva ár a termékhez, mindkét ár megadása kötelező.

Új tétel

cikkszám	megnevezés	mennyiség	nettó vételár	nettó eladási ár
109409	berbero maxi	10 db		575.00 Ft

17. ábra Tétellista - vételi számlánál

5.7 Szállítólevelek

Szállítólevél részleteinek megtekintéséhez kattintsunk a szállítólevélszámra vagy a partner nevére.

Ha nincs a szállítólevélhez kapcsolt számla, egy felirat jelzi, hogy lehet hozzá kapcsolni. Erre kattintva egy számla felvételéhez hasonló panel jelenik meg, csak ebben az esetben a számlaszámot és dátumot lehet megadni.

Egyébként megjelenik a kapcsolt számla száma zöld színnel, ha az ki van egyenlítve, pirossal ha nincs.

A táblázat a kapcsolt számla számán kívül minden egyéb tulajdonág szerint rendezhető.

szállítólevélszám	név	kiállítás dátuma	végösszeg	kapcsolt számla száma
2009-04-BOD-0007	Rácz Miklósné	2009. 04. 24.	4,680.00 Ft	2009-04-E-0006
2009-04-BOD-0006	Ifj Orosz János	2009. 04. 23.	2,809.92 Ft	Számla kiállítása
2009-04-BOD-0005	Dorogi Magánsütöde Kft	2009. 04. 20.	9,799.99 Ft	2009-04-E-0005
2009-04-BOD-0004	Ifj Orosz János	2009. 04. 02.	4,580.16 Ft	Számla kiállítása
2009-03-BOD-0003	Hosszú és Társa Kft	2009. 03. 20.	5,345.10 Ft	2009-03-E-0004
2009-03-BOD-0002	Sebők Élelmiszer Kft	2009. 03. 10.	13,879.86 Ft	2009-03-E-0003
2009-02-BOD-0001	Dorogi Magánsütöde Kft	2009. 02. 12.	24,035.10 Ft	Számla kiállítása

18. ábra Szállítólevelek

5.8 Számlák

A számlákat két csoportban lehet megtekinteni, eladási és vételi számlák. A vevőktől történő visszavételről kiállított számlák a vételi számlák között, a szállítóknak történt visszaküldésről kiállított számlák az eladási számlák között jelennek meg. Ezeknek negatív a végösszege.

Ha fizetési határidő nincs feltüntetve, a számla készpénzes. Ha a fizetési határidő fel van tüntetve és zöld színű, akkor átutalásos a számla és már ki van egyenlítve. Ha a határidő piros színű, a kiegyenlítés még nem történt meg. Kiegyenlíthető egy ilyen számla, ha a fizetési határidőre kattintva megerősítjük a kiegyenlítést.

Számla részleteinek megtekintéséhez kattintsunk a számlaszámra vagy a partner nevére.

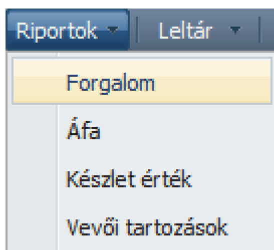
Bármely rekord szerint lehet rendezni a táblázatot.

vételi számlák | **eladási számlák**

számlaszám	név	kiállítás dátuma	végösszeg	fizetési határidő
2009-04-E-0006	Rácz Miklósné	2009. 04. 24.	4,680.00 Ft	2009. 04. 30.
2009-04-E-0005	Dorogi Magánsütöde Kft	2009. 04. 20.	9,799.99 Ft	2009. 05. 31.
2009-03-E-0004	Hosszú és Társa Kft	2009. 03. 31.	5,345.10 Ft	
2009-03-E-0003	Sebők Élelmiszer Kft	2009. 03. 31.	13,879.86 Ft	2009. 05. 01.
2009-02-E-0001	Fazekas Jánosné	2009. 02. 01.	1,847.95 Ft	

19. ábra Eladási számlák

5.9 Riportok



Az *Áfa* adott intervallumban eladott termékek után fizetendő áfa értékét adja meg. Összeadva az eladási számlák áfa értékét, pozitív értéket szolgáltat. Összeadva a visszavételi számlák áfa értékét, negatív értéket szolgáltat.

Készlet érték mindig az árucikkek aktuális ára alapján van számolva.

Forgalom táblázatát bármelyik rekord alapján lehet rendezni.

cikkszám	megnevezés ▲	eladott mennyiség	vett mennyiség	visszaküldött mennyiség	visszavett mennyiség
202351	alufólia 10m	10 db	0 db	0 db	10 db
109021	baba törlő	0 db	30 db	0 db	0 db
109409	berbero maxi	0 db	100 db	0 db	0 db
200615	citromlé Zeus 1 l	20 db	0 db	0 db	0 db
200617	citromlé Zeusz 1.5 l	22 db	0 db	0 db	0 db
109899	comfex tampon 16 db	10 db	10 db	0 db	0 db
109894	comfex tampon 8 db	30 db	0 db	0 db	0 db
109883	comfex éjszakai betét	10 db	15 db	0 db	0 db

20. ábra Forgalom

A *Vevői tartozások* megjelenítésekor minden vevő esetén a neve mellett található tartozásának összértéke. A tételes lebontás – amely még Internet Explorer alatt nem tökéletes – a vevő neve melletti + szimbólumra kattintva elérhető. A *Szállítólevél* és *Számla* csoport mellett szintén ott van összegezve az adott kategóriában fennálló tartozás. Ezeket kinyitva pedig láthatóak a tartozást hordozó dokumentumok adatai.

vevő	kiállítás dátuma	végösszeg
⊟ Ifj Orosz János		7 390,08 Ft
⊟ Szállítólevél		7 390,08 Ft
2009-04-BOD-0004	2009. 04. 02.	4 580,16 Ft
2009-04-BOD-0006	2009. 04. 23.	2 809,92 Ft
⊟ Dorogi Magánsütöde Kft		33 835,09 Ft
⊟ Szállítólevél		24 035,10 Ft
2009-02-BOD-0001	2009. 02. 12.	24 035,10 Ft
⊟ Számla		9 799,99 Ft
2009-04-E-0005	2009. 04. 20.	9 799,99 Ft
⊞ Sebők Élelmiszer Kft		13 879,86 Ft

21. ábra Vevői tartozások

5.10 Leltár

Leltározás esetén megjelenik az árucikkek listája, mellette a készleten lévő darabszám és egy mező a jelenlegi készletnek. Kezdetben a *jelenlegi készlet* mező értéke egyenlő a készlettel. Ha eltérés van, a jelenlegi készlethez be kell írni az aktuális értéket.

A ***Leltároz*** gombbal elvégezhető a módosítás, az ***Alapállapot*** gomb visszaállítja a jelenlegi készlet mezőkön végzett módosítást.

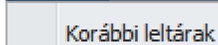
Ha nem történt változás és így történik a leltározás, egy megerősítés után bekerül a rendszerbe a *Nem volt eltérés* bejegyzés.

cikkszám	megnevezés ▲	készlet	jelenlegi készlet
131023	0,5 barack lekvár svéd tálban	0 db	0
131053	0,5 szilva lekvár svéd tálban	0 db	0 db
131013	0,5 vegyes lekvár svéd tálban	0 db	0 db
102601	alma paprika kg	0 db	0 db
202351	alufólia 10m	8 db	8 db
109021	baba törlő	24 db	24 db
109409	berbero maxi	6 db	6 db
200615	citromlé Zeus 1 l	86 db	86 db
200617	citromlé Zeusz 1.5 l	108 db	108 db
109885	comfex szárnyas betét	60 db	60 db
109899	comfex tampon 16 db	40 db	40 db
109894	comfex tampon 8 db	96 db	96 db
109804	comfex tisztasági betét 18 db	255 db	255 db
109884	comfex ultra szárnyas	270 db	270 db
109883	comfex éjszakai betét	76 db	76 db
103132	családi májas	6 db	6 db
103511	csemege kukorica	45 db	45 db
112301	csemege uborka 720 ml	34 db	34 db
192301	csemege uborka apró 4250ml	10 db	10 db
102522	csemege uborka kg	0 db	0 db

Alapállapot Leltároz

22. ábra Leltár

5.10.1 Korábbi leltárak megtekintése



Korábbi leltárak megtekintésekor időpontokba csoportosítva megjelennek az árucikkek készletértékén történt változások. Ha az érték pozitív, a korábbi érték növelése történ, negatív érték esetén csökkentés.

Alapesetben minden csoport kinyitva jelenik meg. Nagy adathalmaz esetén az áttekinthetőség érdekében ezeket be lehet zárni.

megnevezés	mennyiség	dátum
⊟ **dátum: 2009. 04. 26. 12:59 (1 bejegyzés)**		
Nem volt eltérés.		2009. 04. 26. 12:59
⊟ **dátum: 2009. 04. 13. 14:58 (74 bejegyzés)**		
zöldborsó 720ml	+16 db	2009. 04. 13. 14:58
zöldbab 720ml	+16 db	2009. 04. 13. 14:58
virtamp tampon	+85 db	2009. 04. 13. 14:58
viriana tampon 8db super	+87 db	2009. 04. 13. 14:58
viriana tampon 16 db super +	+32 db	2009. 04. 13. 14:58
viriana tampon 16 db super	+5 db	2009. 04. 13. 14:58
viriana tampon 16 db normál	+30 db	2009. 04. 13. 14:58
viriana tampon 16 db mini	+6 db	2009. 04. 13. 14:58
viriana eü.betét normál	+154 db	2009. 04. 13. 14:58
vegyesvágott 720ml	+20 db	2009. 04. 13. 14:58
vegyes íz Vitamor 10kg	+13 db	2009. 04. 13. 14:58
vegyes íz 570g	+16 db	2009. 04. 13. 14:58
tanga tisztasági betét	+62 db	2009. 04. 13. 14:58
tanga szárnyas tisztasági	+113 db	2009. 04. 13. 14:58
sűrített pradicsom 4550g 28%	+3 db	2009. 04. 13. 14:58
sűrített paradicsom 360g	+14 db	2009. 04. 13. 14:58
sűrített paradicsom 135g	+10 db	2009. 04. 13. 14:58
sűrített paradicsom 70g	+20 db	2009. 04. 13. 14:58

23. ábra Korábbi leltárak

6 Összefoglalás

A Készletnyilvántartó alkalmazás fejlesztése és az adódó problémák megoldása során sok hasznos tapasztalatra tettem szert.

Az alkalmazás még korán sem érte el végleges állapotát. Át van adva használatra és folyamatosan alakítani kell a felhasználói igényekhez. Ilyen például a kedvezményes eladások kezelése, amelynek lehetősége a mostani verzióba még nem került bele.

Üzleti folyamatok változása estén is módosítás szükséges. Ilyen például a nemsokára esedékes áfa-kulcs módosítás. A megvalósítás során ügyeltem rá, hogy a hasonló változások minimális módosítással elvégezhetőek legyenek.

Továbbfejlesztési lehetőség, hogy számlázó programmá alakítom. Ezen esetben az APEH által hitelesítve a programmal az egész cég nyilvántartása (elektronikus és papír alapú) vezethető.

Ehhez szükséges a számla minden egyéb papír alapú dokumentum kliens oldalon való nyomtatásának megoldása, amely applet alapú technológiákkal megoldhatónak látszik.

Itt ragadnám meg az alkalmat, hogy köszönetet mondjak Dr. Fazekas Gábornak az alkalmazás elkészítésében és jelen diplomamunka létrejöttében nyújtott segítségért és ötletekért.

7 Ábrajegyzék

1. ábra Az n rétegű architektúra ... *6*

2. ábra Hibernate alkalmazás architektúrája ... *8*

3. ábra GWT komponensek ... *13*

4. ábra GWT RPC komponensei ... *14*

5. ábra Data Access Object ... *21*

6. ábra Gomb ... *28*

7. ábra Menügomb ... *28*

8. ábra Osztott menügomb ... *28*

9. ábra Táblázat ... *28*

10. ábra Kötelező mező ... *29*

11. ábra Hibás mező ... *29*

12. ábra Beviteli hiba ... *29*

13. ábra ComboBox ... *29*

14. ábra Lenyíló lista ... *29*

15. ábra Dátum beviteli mező ... *29*

16. ábra Tétellista - általános ... *32*

17. ábra Tétellista - vételi számlánál ... *32*

18. ábra Szállítólevelek ... *33*

19. ábra Eladási számlák ... *33*

20. ábra Forgalom ... *34*

21. ábra Vevői tartozások ... *34*

22. ábra Leltár ... *35*

23. ábra Korábbi leltárak ... *36*

24. ábra Alkalmazás felülete ... *40*

25. ábra Intervallum megadása riportokhoz ... *40*

26. ábra Objektumorientált adatbázisséma ... *49*

8 Irodalomjegyzék

Google Web Toolkit [Online]. - http://code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=google-web-toolkit-doc-1-5.

GWT-Ext [Online]. - http://gwt-ext.com/.

Hibernate [Online]. - http://www.hibernate.org/.

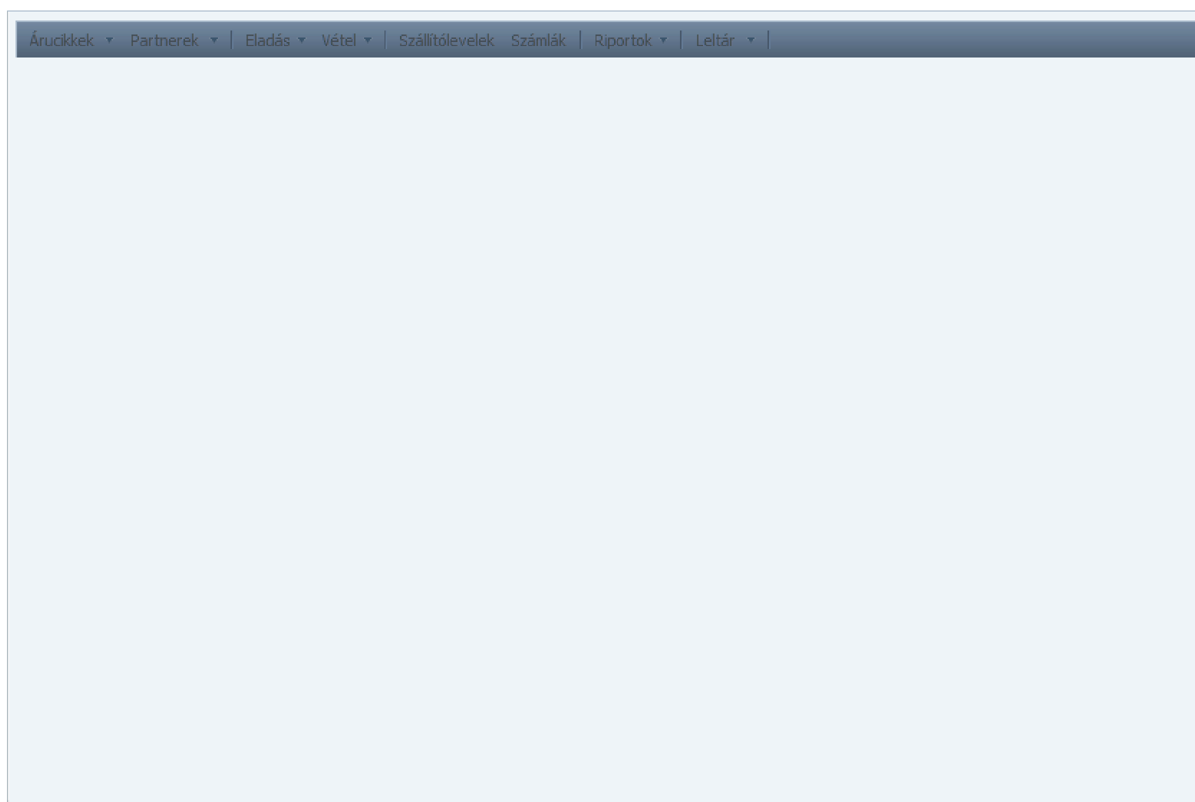
Jasper Reports [Online] / szerző Parragh Szabolcs. - http://parszab.hu/javaee/jasperreports/index.html.

Java EE [Online]. - http://java.sun.com/javaee/.

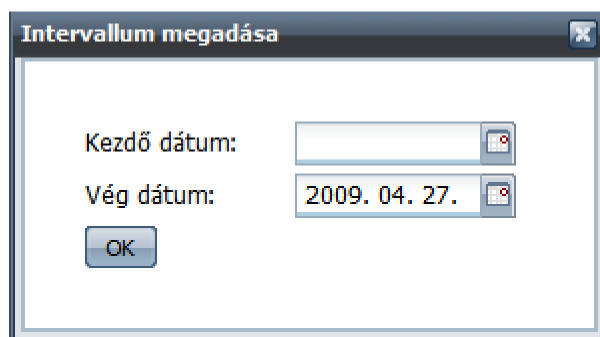
Szoftverfejlesztés Java EE Platformon [Könyv] / szerző Imre Gábor. - Bicske : Szak kiadó, 2007.

The definitive guide to JasperReports [Könyv] / szerző Teodor Danciu és Lucian Chirita. - New York : Apress, 2007.

9 Függelék



24. ábra Alkalmazás felülete



25. ábra Intervallum megadása riportokhoz

```
<hibernate-configuration>
    <session-factory>
        <!-- Database datasource -->
        <property name="hibernate.connection.datasource">
            StockRegistryDataSource
        </property>
        <!-- Turns off autocommit mode -->
        <property name="hibernate.connection.autocommit">
            false
        </property>
        <!-- SQL dialect -->
        <property name="hibernate.dialect">
            org.hibernate.dialect.DerbyDialect
        </property>
        <!-- Set Hibernate's session context management to JTA -->
        <property name="hibernate.current_session_context_class">
            jta
        </property>
        <!-- Set Hibernates's transaction factory to container -->
        <property name="hibernate.transaction.factory_class">
            org.hibernate.transaction.CMTTransactionFactory
        </property>
        <!--Set transaction manager lookup class for glassfish -->
        <property name="hibernate.transaction.manager_lookup_class">
            org.hibernate.transaction.SunONETransactionManagerLookup
        </property>
        <!-- Disable the second-level cache -->
        <property name="hibernate.cache.provider_class">
            org.hibernate.cache.NoCacheProvider</property>
        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">
            true
        </property>
        <!-- Create the database schema on startup if necessary-->
        <property name="hibernate.hbm2ddl.auto">
            update
```

```
        </property>
        <!-- This is required for weblogic! -->
        <!--
            <property name="hibernate.query.factory_class">
                org.hibernate.hql.classic.ClassicQueryTranslatorFactory
            </property>
        -->
        <!-- Class mappings -->
        <mapping resource="edu/degreeWork/szuni/stockRegistry/product/Product.hbm.xml"/>
        <mapping resource="edu/degreeWork/szuni/stockRegistry/partner/Partner.hbm.xml"/>
        <mapping resource="edu/degreeWork/szuni/stockRegistry/accountItem/AccountItem.hbm.xml"/>
        <mapping resource="edu/degreeWork/szuni/stockRegistry/account/Account.hbm.xml"/>
        <mapping resource="edu/degreeWork/szuni/stockRegistry/inventory/Inventory.hbm.xml"/>
    <mapping
resource="edu/degreeWork/szuni/stockRegistry/billofdeliveryItem/BillOfDeliveryItem.hbm.xml"/>
    <mapping resource="edu/degreeWork/szuni/stockRegistry/billofdelivery/BillOfDelivery.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```


1. kódrészlet Hibernate config XML

```
<hibernate-mapping>
    <class name="edu.degreeWork.szuni.stockRegistry.account.Account" table="account" abstract="true"
discriminator-value="A">
        <id name="accountId" column="account_id"/>
        <discriminator column="account_type" type="character" not-null="true"/>
        <set name="items" cascade="all">
            <key column="accountitem_accountid"/>
            <one-to-many class="edu.degreeWork.szuni.stockRegistry.accountItem.AccountItem"/>
        </set>
        <property name="date" column="account_date" not-null="true"/>
        <property name="preference" column="account_preference" precision="2" scale="2" not-
null="true"/>
        <many-to-one class="edu.degreeWork.szuni.stockRegistry.partner.Partner" name="partner"
column="account_partnerid" unique="true"/>
```

```
        <subclass name="edu.degreeWork.szuni.stockRegistry.account.buy.BuyAccount" discriminator-value='B'>
            <subclass name="edu.degreeWork.szuni.stockRegistry.account.buy.RemittanceBuyAccount" discriminator-value='C'>
                <property name="deadline" column="account_deadline"/>
                <property name="payed" column="account_payed"/>
            </subclass>
        </subclass>
        <subclass name="edu.degreeWork.szuni.stockRegistry.account.sell.SellAccount" discriminator-value='S'>
            <subclass name="edu.degreeWork.szuni.stockRegistry.account.sell.RemittanceSellAccount" discriminator-value='T'>
                <property name="deadline" column="account_deadline"/>
                <property name="payed" column="account_payed"/>
            </subclass>
        </subclass>
    </class>
</hibernate-mapping>
```


2. kódrészlet Hibernate leképezés XML - számlák

```
/**
 * Szállítólevél DAO objektuma
 * @author Szunai János
 */
public interface BillOfDeliveryDao {

    /**
     * Új szállítólevél letárolása
     * @param billOfDeliveryVo szállítólevél
     */
    public void insert(BillOfDeliverySaveVo billOfDeliveryVo);

    /**
     * Összes szállítólevél lekérdezése
     * @return szállítólevelek listája
```

```
     */
    public List<BillOfDeliveryListVo> findAll();

    /**
     * Szállítólevél keresése szállítólevélszám alapján
     * @param billOfDeliveryId szállítólevélszám
     * @return szállítólevél
     */
    public BillOfDeliveryViewVo findByBillOfDeliveryId(String billOfDeliveryId);

    /**
     * Számla kapcsolása szállítólevélhez
     * @param billOfDeliveryId szállítólevélszám
     * @param account számla
     */
    public void insertAccountForBill(String billOfDeliveryId, Account account);

    /**
     * Kifizetetlen szállítólevelek lekérdezése
     * @return szállítólevelek listája
     */
    public List<BillOfDeliveryListVo> listUnpayedBills();
}
```

3. kódrészlet DAO interfész

```
/**
 * Szállítólevél DAO implmentációja
 * @author Szuni
 */
public class BillOfDeliveryDaoImpl implements BillOfDeliveryDao{

    /**
     * @inheritDoc
     * @return @inheritDoc
     */
```

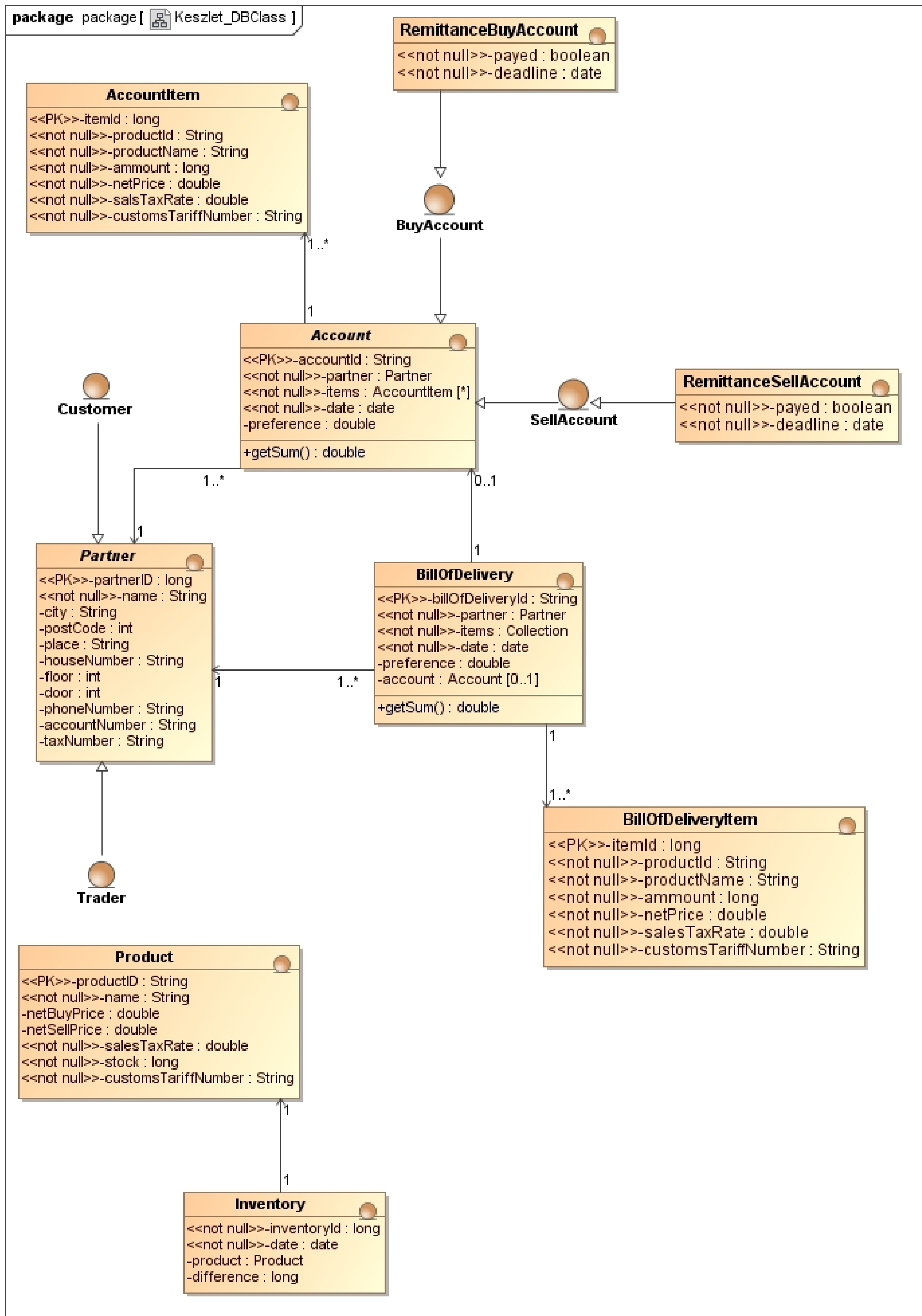
```
    public List<BillOfDeliveryListVo> findAll() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        List<BillOfDelivery> list = new ArrayList<BillOfDelivery>();
        list = session.createQuery("from BillOfDelivery order by billOfDelivery_date desc").list();
        return entityToListVo(list);
    }

    /**
     * @inheritDoc
     * @param billOfDeliveryId @inheritDoc
     * @return @inheritDoc
     */
    public BillOfDeliveryViewVo findByBillOfDeliveryId(String billOfDeliveryId) {
        …
    }

    /**
     * @inheritDoc
     * @param billOfDeliveryVo @inheritDoc
     */
    public void insert(BillOfDeliverySaveVo billOfDeliveryVo) {
        BillOfDelivery billOfDelivery = (BillOfDelivery) voToEntity(billOfDeliveryVo);
        Set<BillOfDeliveryItem> items = new HashSet<BillOfDeliveryItem>();
        for(ProductSellVo p : billOfDeliveryVo.getItems()){
            items.add(itemVoToEntity(p));
        }
        billOfDelivery.setItems(items);

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.save(billOfDelivery);
    }

    /**
     * @inheritDoc
     * @param billOfDeliveryId @inheritDoc
     * @param account @inheritDoc
```

```
     */
    public void insertAccountForBill(String billOfDeliveryId, Account account){
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        BillOfDelivery billOfDelivery = (BillOfDelivery) session.load(BillOfDelivery.class,
billOfDeliveryId);
        billOfDelivery.setAccount(account);
        session.update(billOfDelivery);
    }

    /**
     * @inheritDoc
     * @return @inheritDoc
     */
    public List<BillOfDeliveryListVo> listUnpayedBills() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        List<BillOfDelivery> list = session.createQuery("from BillOfDelivery as bod where bod.account
is null").list();
        return entityToListVo(list);
    }

    /**
     * Entity listából VO lista
     * @param list entity lista
     * @return vo lista
     */
    private List<BillOfDeliveryListVo> entityToListVo(List<BillOfDelivery> list) {
        …
    }

    /**
     * Entityből VO
     * @param billOfDelivery entity
     * @return VO
     */
    private BillOfDeliveryListVo entityToListVo(BillOfDelivery billOfDelivery) {
        …
```

```
    }

    /**
     * Entitből VO
     * @param billOfDelivery entity
     * @return VO
     */
    private BillOfDeliveryViewVo entityToViewVo(BillOfDelivery billOfDelivery) {
        return new BillOfDeliveryViewVo(
             billOfDelivery.getBillOfDeliveryId(),
             billOfDelivery.getPartner().getName(),
             billOfDelivery.getPartner().getPartnerID(),
             entityToItemVo(billOfDelivery.getItems()),
             billOfDelivery.getDate(), billOfDelivery.getPreference());
    }

    /**
     * VO-ból entitás
     * @param billOfDelivery VO
     * @return Entitás
     */
    private BillOfDelivery voToEntity(BillOfDeliverySaveVo billOfDelivery){
        …
    }

    /**
     * Szállíólevél elem VOból entity
     * @param vo VO
     * @return entity
     */
    private BillOfDeliveryItem itemVoToEntity(ProductSellVo vo){
        …
    }

    /**
     * Szállítólevél elem halmazból VO halmaz
```

```
     * @param items szállítólevél elem halmaz
     * @return VO halmaz
     */
    private Set<ItemVo> entityToItemVo(Set<BillOfDeliveryItem> items) {
        …
    }

    /**
     * Szállítólevél elemből VO
     * @param item szállítólevél elem
     * @return VO
     */
    private ItemVo entityToItemVo(BillOfDeliveryItem item){
        …
    }
}
```

4. kódrészlet DAO implementáció



26. ábra Objektumorientált adatbázisséma