Diplomamunka

Fekete Gyula

Debrecen
2011

Debreceni Egyetem

Informatikai Kar

Telefonközponti tarifázó rendszer

Témavezető:
Dr. Juhász István
egyetemi adjunktus

Konzulens:
Gazdag Edit
IT vezető

Készítette:
Fekete Gyula
programtervező matematikus

Debrecen
2011

Tartalomjegyzék

1. Bevezetés

Diplomamunkám témája egy nagyvállalati környezetben meglévő probléma megoldása az egyetemi tanulmányaim alatt megszerzett tudás alapján. A feladatom végrehajtásához szükséges volt a megfelelő hardverismeret, szoftverfejlesztési és adatbázis tervezési tapasztalat. Egy személyben próbáltam megvalósítani az agilis szoftverfejlesztési technikát, több –akár ismeretlen– szakterületet átfogni és megérteni. A feladat megoldása során sikerült az elméleti tudásom mellé olyan gyakorlati tapasztalatot is gyűjteni, melyek a későbbi munkáimban segítségemre lesznek.

A Tiszavasvári Alkaloida Vegyészeti Gyár Zrt. 1000 dolgozója és közel 500 telefonvégpontja komoly adminisztrációs munkát jelent az ezért felelős személyeknek. Úgy gondoltam rendelkezem azon eszközökkel, amelyekkel könnyíteni tudnék e probléma megoldásában, illetve olyan plusz szolgáltatásokat nyújtani, amelyekre eddig nem volt lehetőség.

Az elképzelt rendszer segít átlátni az üzemek kimenő hívásait, illetve adózási szempontból szétválasztani a magán és hivatalos hívásokat. A gyár területén belül, a mellékek közötti kapcsolással nincs különösebb teendőnk. Az üzem dolgozói saját pin kóddal rendelkeznek, amely a külső irányú telefonáláshoz szükséges. Ez, és/vagy a törzsszám alapján meghatározható a hívó személye. Lehetőség van magán, illetve hivatalos irányú hívás indítására, melyet egy előhívó segítségével adhatnak meg. A magán célú beszélgetéseket minden esetben tároljuk, majd hónap végén számlázásra bocsájtjuk (feltéve, hogy elért egy minimális összeget). A hivatalos irányú beszélgetéseget szintén eltároljuk. Felmerült az igény, hogy a dolgozóknak lehetőségük legyen az eddigi hívások online megtekintésére, illetve a vezetők megtekinthessék a beosztottaik telefonhívásait. A gyár területén több külsős cég is üzemel, feléjük a vállalat telefonszolgáltatást nyújt. A külsős cégeket csak mellék alapján tudjuk behatárolni, ezeket külön táblában tároljuk.

A feladat rengeteg érdekességet tartogat számomra, mint például azt, hogyan oldjam meg az éles adatokkal való tesztelést, hogyan írjak olyan biztonságos kódot, amely rezidensként futtatható, hogyan reprezentáljam a beosztási viszonyokat, vagy hogyan oldjam meg az adatok törvényben meghatározott tárolását. Kihívás volt az is,

hogy minél optimálisabb kódot írjak, és lehetőség szerint igazodjak a környezethez. (pl.: statisztika készítés vékonykliensen képgenerálással)

A diplomamunkám várható eredménye egy komplex projekt, amely a telefonközponttal való kommunikáció kiépítésétől az egyenleg lekérdezésen és a tarifakezelésen át a számlakészítésig tart. A köztes folyamatok érintik a legújabb Java SE/EE technológiákat és a relációs adatbázis kezelést.

2. Projekt ismertetése részletesen

2.1. Követelményspecifikáció

- Telefonközpont kimenő jeleit feldolgozó alkalmazás, amely alkalmazás biztonságosan, szabványok szerint perzisztálja az adatokat.
- Több szerepkörös vékonykliens alkalmazás, az adatok megjelenítéséhez.
- Autentikáció és autorizáció megvalósítása.
- Dolgozói szerepkör: Saját hívásrészletező lekérése, statisztikák megtekintése.
- Vezetői szerepkör: Saját, illetve beosztottjai hívásrészletező lekérése, statisztikák megtekintése.
- Adminisztrátori szerepkör: Tarifa beállítások, mellékek kezelése, külsős cégek menedzselése, havi részletes exportálás a könyvelés számára.
- Külföldi tulajdonosok érdekében internacionalizáció megvalósítása (magyar-angol nyelv)
- Automatizált beosztási fa frissítés – nem része az implementációnak –, csak a specifikáció bemutatása.
- Standard tervezési minták használata.
- Magas kódminőség.
- Továbbfejleszthetőség.

2.2. Megvalósíthatósági tanulmány

A követelménytervezés fázisa alatt folyamatos volt a konzultáció az érintett partnerekkel. A szükségletek, és a felhasználói igények tisztán implementálhatóak (4.-5. fejezet). A követelmények teljesítéséhez az eszközök a választott nyelv alapjait képezik (4.1 fejezet). Az ütemterv jól tagolt, könnyen ellenőrizhető (2.3. fejezet). A jelenlegi állapotok vizsgálatakor a hiányosságok észrevehetőek (3.1. fejezet). Ezeket a dokumentumokat kiegészítve egy eredmény összefoglalóval (9. fejezet) a vezetőség a projekt indítása mellett döntött.

2.3. Projekt menedzsment

A projekt életciklusa alatt egyedül töltöttem be több ember szerepét, mint például: manager, architect, developer, tester. A diplomamunkám elvégzésére az egyetem keretein belül 2 szemesztert vettem igénybe. Az előbb említett egyszereplős projekt rugalmasságot enged a projekt menedzselése szempontjából. Adott tehát 9 emberhónap a feladat elvégzésére, amihez külön nem készítettem erőforrás illetve ütemezés szervező dokumentációt, viszont valamilyen korlát szerint szeretném a saját munkám koordinálni. Általánosságban a projektek végrehajtásánál három fő korlátot szoktak felsorolni:

- idő
- költség
- hatókör

Ahhoz, hogy az a bizonyos projektmenedzsment háromszög az esetemben értelmezve legyen, a költséget a ráfordított idő függvényében értelmezem, így a következő eredményre jutottam:

Feladat	Ráfordított erőforrás
Szükséges hardver eszközök kiválasztása	1 emberhét
Kommunikáció kiépítése a központtal	2 emberhét
Rezidens alkalmazás tervezése	2 emberhét
Rezidens alkalmazás fejlesztése	5 emberhét
Rezidens alkalmazás tesztelése	2 emberhét
Web alkalmazás technológiák kiválasztása	2 emberhét
Web alkalmazás architekturális tervezése	2 emberhét
Adatbázis tervezése	1 emberhét
Web alkalmazás tervezése	2 emberhét
Web alkalmazás fejlesztése	10 emberhét
Web alkalmazás tesztelése	3 emberhét
PL/SQL szkript, szkript validálása	1 emberhét
Dokumentáció készítése	4 emberhét
Telefonközponti tarifázó rendszer	**~9 emberhónap**

2.4. Projekt szerkezete: Scrum[1]

A Scrum a szoftverfejlesztés egy fokozatos, iteratív módszere, amit gyakran használnak az agilis szoftverfejlesztés eszközeként. A Scrum főbb szerepkörei a "Scrum Master", aki a folyamatot felügyeli és munkája hasonlít a projekt menedzséréhez, a "Product Owner" aki a projektben érdekelt döntéshozókat képviseli, és a "Csapat" (Team) ami a nagyjából 7 főből áll és lefedi az összes munkafolyamatot. Az előző fejezet gondolatmenetét folytatván, a Scrum szerepköreit is egyedül valósítom meg. (Megjegyzés: utólag találtam az interneten egy cikket, miszerint létezik ilyen egyszereplős fejlesztési technológia, a neve: Solo Scrum[2]) Minden "futam" (sprint) során - amely 2 és 4 hét közötti időtartamot jelent (a „csapat” döntésétől függően) - a „csapat” egy működő szoftver egységet hoz létre. A futam során megvalósítandó funkciók a "Project Backlog"-ból (termék teendő lista) kerülnek ki, ami az elvégzendő munka magas szintű követelményeiből álló, fontossági sorrendbe állított lista (előző fejezetben felsorolt tasklist).

2.5. Fejlesztési módszertan

FDD

Feature Driven Development, vagyis Funkcionalitáson Alapuló Szotfverfejlesztés, ami egy agilis szoftverfejlesztési technika. A fejlesztendő alkalmazás tekinthető funkciók halmazának, a követelmények feltárása után ezen funkciók meghatározása a feladat. A fejlesztő feladata az említett funkciók implementálása, ha a funkció komplexitása túlmutat egy könnyen megoldható probléma összetettségén, akkor a fejlesztő tovább bontja azt. A fejlesztő egy fejlesztési ciklusban egy funkciót fejleszt le.

[1] http://hu.wikipedia.org/wiki/Scrum (2011.04.20)
[2] http://www.pbell.com/index.cfm/2007/6/17/Solo-Scrums (2011.04.20)

BDD

A Behavior Driven Development, vagyis Viselkedésen Alapuló Szoftverfejlesztés, ami szintén egy új agilis szoftverfejlesztési technika. A BDD hasonlóan a TDD-hez (Test Driven Developement), egy kívülről befelé haladó szoftverfejlesztési technika, amely megváltoztatja az elkészült alkalmazások viselkedését. Hosszabb távon valós körülményeket könnyebben lekövető, strukturáltabb, költséghatékonyabb szoftverek elkészítését teszi lehetővé.

Gyakorlatilag ez a módszertan a unit tesztelést elősegítő és megkövetelő fejlesztési metodológia. Segítségével 85-90%-os lefedettség érhető el, így a kódnak szinte minden sora már a fejlesztés során tesztelve van.

Fejlesztés menete:

A fejlesztő első lépésként létrehozza az implementálandó metódus headerjét, majd létrehozza a hozzá tartozó első tesztet. Majd mielőtt implementálni kezdené az üzleti logikát minden egyes követelményre elkészíti a megfelelő tesztet. (when-given-then)

2.6. Build folyamat és dependencia kezelés

A „The Apache Software Foundation” által fejlesztett egyik projektmenedzsment eszköz a MAVEN. Eleinte egyszerű fordítási feladatokat ellátó eszköznek indult (az Ant leváltására), de az igények növekedésével a funkcionalitása sokat bővült. A fő feladata mégis a fejlesztők által elkészített forráskódból, megfelelő konfiguráció alapján a program fordítása, tesztelése. Tapasztalataim alapján a Maven használatánál csak egy rosszabb dolog van, ha nem használjuk.

A fejlesztés során a következő Maven nyújtotta lehetőségeket használom:

- standard plugin (clean, install, test)
- eclipse plugin (eclipse kompatibilis projekt készítése)
- checkstyle plugin (folyamatosan magas kódminőség ellenőrzésre)
- module management (a rendszert funkcionalitás szerint kisebb, úgynevezett module-okra osztottam, parent-child)
- dependency management (a függőségeket kezelése automatikusan történik, verzióütközés nélkül letölti a szükséges dependenciákat egy központi repository-ból a sajátunkba.)

2.7. Naplózás, tesztelés

Naplózás

Egy komolyabb rendszer fejlesztésénél elengedhetetlen a logolás. Nem csak hibakeresési szempontból, hanem a rendszer működésének visszaellenőrzésére is jó célt szolgálhat egy jól megtervezett logolási struktúra. Az egyik legelterjedtebb naplózó keretrendszer a log4j. Kiaknázva a benne rejlő lehetőségeket, használom a Logger, Appender, és Layout osztályait. Alkalmazásonként külön fájlban, 3 különböző szintet használok.

- Debug: Alapvető visszaellenőrzési lépések
- Info: Alkalmazás speciális folyamatainak állapota (pl.: recording)
- Warn: Futás során keletkezett hibák

Teszt

A tesztelés a szoftverfejlesztési folyamat részét képezi, amelynek során vizsgáljuk a specifikáció szerinti helyességét (a specifikációnak megfelelő működés), és teljességét (minden szükséges funkció megvalósításra került-e). A puszta funkcionalitáson túl az elkészített alkalmazással szemben számos egyéb objektív és szubjektív kritériumot is felállíthatunk, pl.: futási sebesség, karbantarthatóság.

Egyik legismertebb tesztelő keretrendszer a JUnit. Ezzel a funkcionalitást könnyen, automatizálva ellenőrizhetjük. Ezen kívül az alábbi tesztelési módszereket használom:

- Teljesítményteszt: életszerű körülményt produkálva terhelem az alkalmazást
- Stressзteszt: életszerűtlen körülményt produkálva vizsgálom az alkalmazás határait.
- Biztonsági teszt: Hozzáférési szintek ellenőrzése az alkalmazáson
- Kompatibilitási teszt: több környezetben az alkalmazás produktumának ellenőrzése
- Használhatósági, felületi teszt: Végfelhasználó szempontjából az alkalmazás ellenőrzése.

3. Kapcsolat kiépítése

3.1. Meglévő rendszer bemutatása

Az intézmény telefonhívásait egy Ericsson MD110-es központ szolgálja ki. A 2 Limes rendszer 600mellékállomás kezelésére alkalmas. Analóg, digitális fővonali és mellékoldali kártyák kezelésére alkalmas. A GSM hívások kezelése GSM Interface segítségével valósul meg. A bejövő hívások az ISDN számok esetében automatikus beválasztás segítségével történik. Az analóg fővonali kártyák és GSM bejövő hívások esetében egy beválasztó berendezés segítségével lehetséges továbbirányítani a hívásokat.

Jelenleg a telefonközponthoz közvetlenül csatlakozik egy PC, amin a ProfiTel által készített TaxaWin nevű program látja el a tarifakezelést. A rendszer az általános feladatokat tökéletesen látja el, viszont a hiányosságok pótlása 1 emberi erőforrást igényel. (pl.: kézi havi elszámolás, egyenleg lekérdezés, üzem csoportosítás, külsős cégek kezelése, költséghely figyelése)

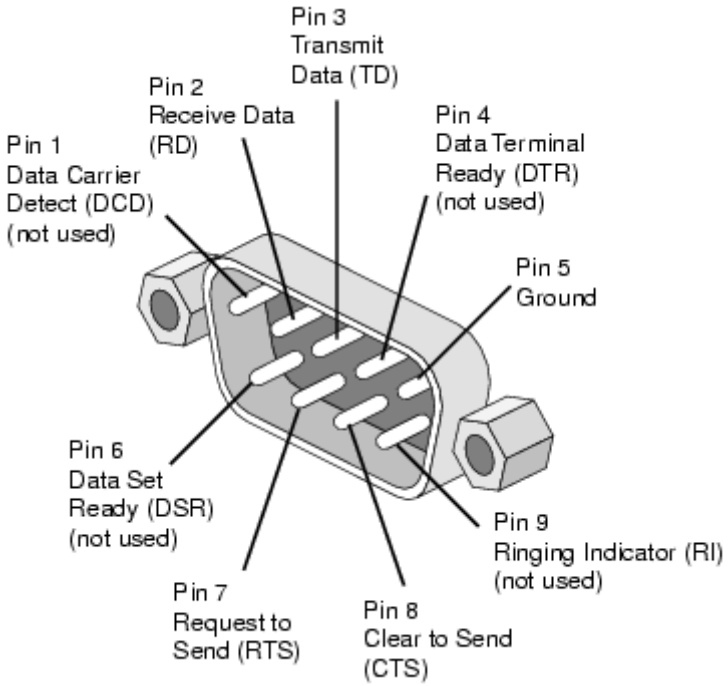
3.2. Szükséges eszközök a kapcsolat kiépítéséhez

Az alkalmazás fejlesztése és tesztelése közben olyan kivételes események következhetnek be, amelyek veszélyeztetik az adatok biztonságát. Ezért egy olyan adatgyűjtő pufferre szükségünk lesz, ami ideiglenesen tárolni tudja az érkező rekordokat. A telefonközponthoz kapott adatgyűjtő erre a célra tökéletesen megfelel, viszont a beüzemelésénél el kell térnünk a specifikációtól. Problémát jelent az is, hogy jelenleg csak hagyományos RS232-es soros porton lehet kommunikálni a telefonközponttal. Azon túl, hogy ez a kommunikációs csatorna már elavult, abból a szempontból sem megfelelő, hogy a telefonközpont és a szerverterem – ahol várhatóan az alkalmazás futni fog – nem egy épületben található.

Tehát szükség volt egy RS232-TCP/IP átalakító modulra. Több cég kínálatát megtekintve végül az adatgyűjtőhöz megfelelő ComToNet eszköz vásárlása mellett döntött a cég, melyben egy a Lantronix cég által gyártott Xport nevű Ethernet-Soros port konverter egység végzi az átalakítást.

3.3. Kapcsolat kiépítése

Az említett specifikációtól való eltérés bemutatásához vizsgáljuk meg az aszinkron soros kommunikáció egyik igen gyakori változatát az RS-232 adatátviteli módot. Aszinkron, tehát az adó tetszőleges időpontban kezdhet egy információcsomag (pl.: byte) továbbításához. Ahhoz, hogy az adó által küldött jelsor egyértelműen fogadható legyen a vételi oldalon meg kell állapodni az időzítés mértékében. A szabvány bizonyos sebességeket enged csak meg, pl.: 110, 300, 1200, 9600 Baud-ot (bit/sec). A tárolóegységünk 9600 Baud értéken üzemel.



1 . ábra: Soros kommunikációs port lábkiosztása[3]

[3] http://image.pinout.net/pinout_9_pin_files/db9.gif

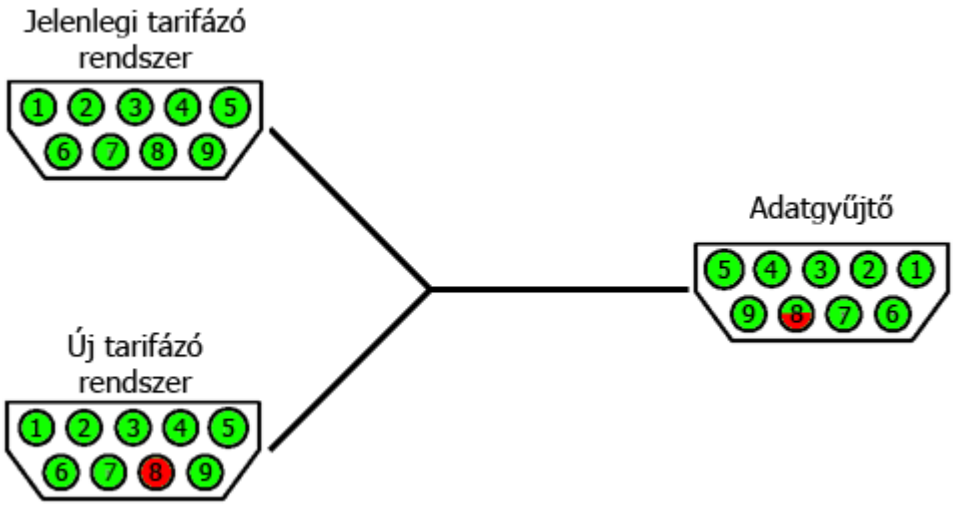
Az adatgyűjtő specifikációja szerint a 8-as lábra érkező jel hatására üríti a puffer tartalmát. Az alkalmazás tesztelése során ezek az adatok nincsenek biztonságban, ezért egy speciális tesztkörnyezetet kell kialakítani.

Igények:

- Az éles adatok ne sérüljenek.
- Az alkalmazás folyamatos push-olásra éles adatokat kapjon az adatgyűjtőtől.
- A puffer rendszeresen (naponta) ürítve legyen.
- A projekt végéig a meglévő tarifázó rendszer biztonságos működése.

Megoldás:

Speciális Y soros kábel készítése a következőképpen:



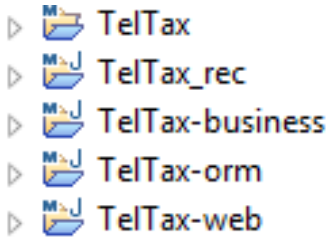
2. ábra: Saját készítésű Y soros kábel

A megoldás a 8-as láb leválasztása a fejlesztendő alkalmazás felől. Ez esetben az adatgyűjtő CTS lábára csak és kizárólag a régi rendszer felől érkezhet jel, így elérjük, hogy a rekordok csak akkor ürülnek a pufferből, ha azt a jelenlegi rendszer már biztonságosan befogadta.

4. A rendszer tervezése

4.1. A rendszer fejlesztői szemmel

A fejlesztés és a későbbi könnyebb karbantartás érdekében megpróbáltam minél granuláltabb módon kialakítani a projekt struktúráját. A logikai és fizikai szinteket figyelembe véve az alábbi architekturális mintát választottam:



3. ábra: Projekt struktúra

- TelTax: Szülő projekt, a modulokat fogja össze
- TelTax_rec: Adatrögzítő modul
- TelTax-business: Web modul üzleti logika
- TelTax-orm: Web modul adatbázis elérés
- TelTax-web: Web modul megjelenítés

4.2. A választott nyelv[4]

A Java egy általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a ’90-es évek elejétől kezdve napjainkig (mára az Oracle tulajdona). A Java alkalmazásokat jellemzően bytecode formátumra alakítják, de közvetlenül natív (gépi) kód is készíthető Java forráskódból. A bytecode futtatása a Java virtuális géppel történik, ami vagy interpretálja a bytecode-ot vagy natív gépi kódot készít belőle és azt futtatja az adott operációs rendszeren.

[4] http://hu.wikipedia.org/wiki/Java_(programozási_nyelv) (2011.04.20)

Négy fontos szempontot tartottak szem előtt, amikor a Javát kifejlesztették:

- **objektum-orientáltság**: a programozási stílusra és a nyelv struktúrájára utal. Az OO fontos szempontja, hogy a szoftvert „dolgok” (objektumok) alapján csoportosítja, nem az elvégzett feladatok a fő szempont. Ennek alapja, hogy az előbbi sokkal kevesebbet változik, mint az utóbbi, így az objektumok (az adatokat tartalmazó entitások) jobb alapot biztosítanak egy szoftverrendszer megtervezéséhez.
- **platformfüggetlenség:** a Javában íródott programok hasonlóan fognak futni különböző hardvereken illetve operációs rendszereken. Ezt úgy lehet megvalósítani, hogy a Java fordítóprogram csak egy úgynevezett Java bájtkódra fordítja le a forráskódot, ami aztán futtatva lesz a virtuális gépben, amely lefordítja az adott hardver gépi kódjára.
- **hálózati kommunikáció támogatása**: a Java hálózati könyvtára (java.net) jelenleg a TCP/IP protokollcsaládra épül, bár ennek részleteit csaknem teljesen elfedi a programozó elől, így elvileg nem kizárt, hogy más alapprotokollokat használó könyvtár-implementációk is megjelenjenek.
- **távoli gépeken is képes legyen biztonságosan futni.**

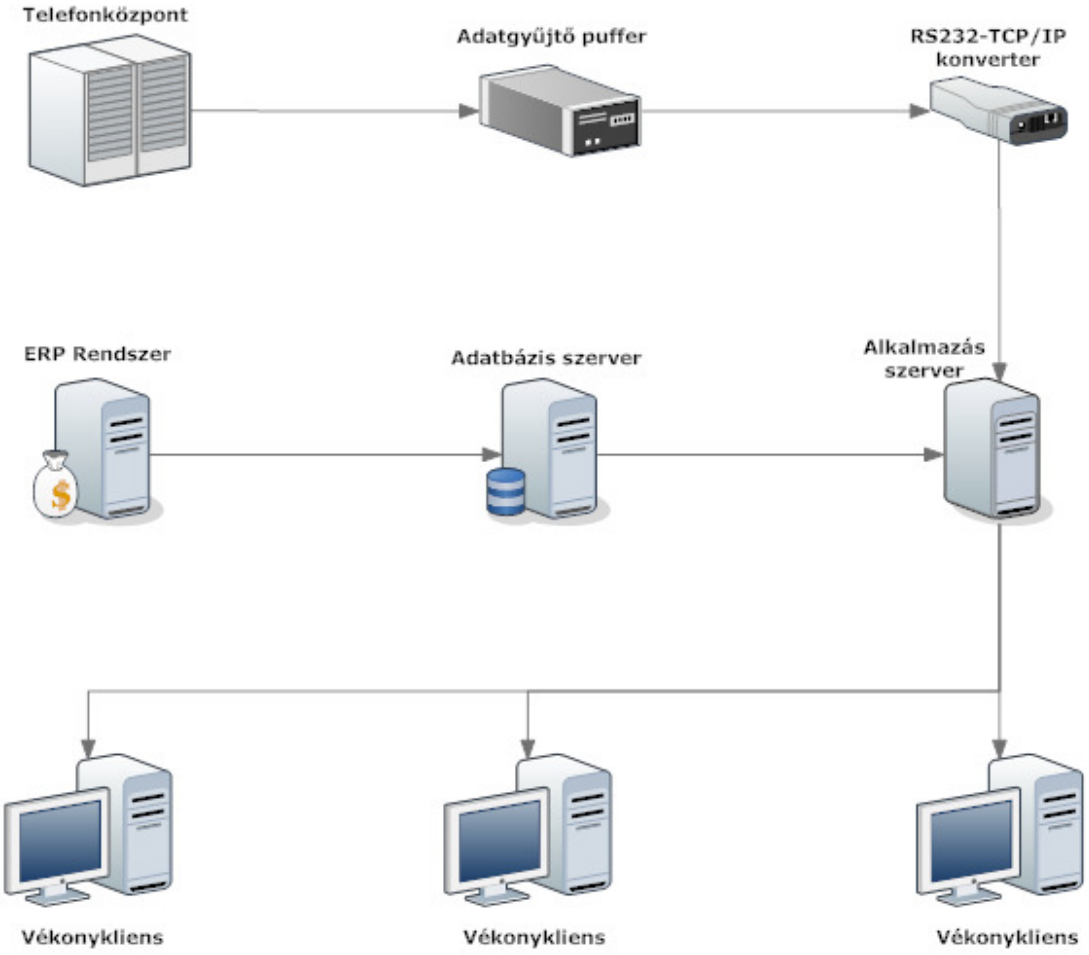
Úgy gondolom, hogy e szempontok szoros párhuzamot mutatnak a projekt céljaival, így a Java nyelv tökéletes választás a rendszer elkészítéséhez.

4.3. A rendszer felépítése

A projekt tervezése során érezhető volt, hogy minden hibaforrást nem tudunk az előkészítés fázisában meghatározni. Ezért a problémákat megpróbáltam minél kisebb részproblémákra osztva megoldani. Így a rendszert 3 fő irányból kezdtem el építeni:

- Rezidens alkalmazás, mely a telefonközpontból érkező rekordokat kezeli.
- Web alkalmazás, mely az eredmény megjelenítéséért felelős.
- PL/SQL-Trigger, mely az ERP rendszer adatait dolgozza fel.

A teljes kiépített rendszert az alábbi séma alapján lehet elképzelni:



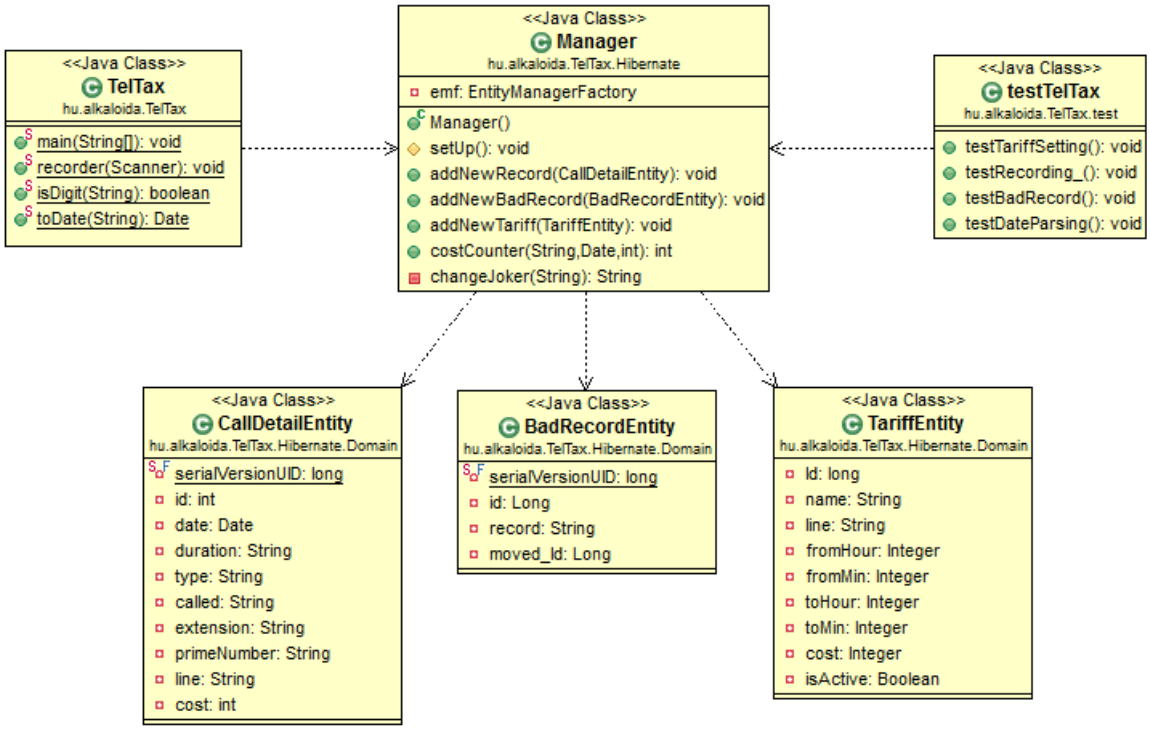
4. ábra: A rendszer felépítése

5. Rezidens alkalmazás fejlesztése

A kommunikáció sikeres kiépítése után elkezdődhetett a rekordok fogadására és feldolgozására alkalmas kód megírása. Az alkalmazás folyamatos valós idejű adatok feldolgozását teszi lehetővé, így lehetőség szerint minden kivételes eseményre fel kell készítenem. Ezért a fejlesztés ezen fázisát további 2 részre bontottam:

- Első célom olyan kód írása volt, mely feltárja a lehetséges hibaforrásokat a feldolgozás közben, illetve megkönnyíti a nyers rekordok elemzését.
- A nyers rekordok hibátlan feldolgozása után a perzisztencia megvalósítása.

Az adatrögzítő alkalmazás minimális adatbázis műveletet (1 select és 2 insert) és minimális üzleti logikát tartalmaz (costCounter), így ezeket egy strukturált Manager osztályba implementáltam. Látható (5. ábra), hogy ezek ellenére is magas „normálformában” maradnak az osztályok.



5. ábra: Adatrögzítő alkalmazás osztály diagram

5.1. Teszt alkalmazás, adattárolás nélkül

Ugyan még nem a végleges verzióját készítem a kódnak, mégis megpróbáltam az igényeknek megfelelően alakítani. Mivel távoli parancssoros környezetben fog futni, ezért könnyen paraméterezhetőnek kell lennie. Az alkalmazás konfigurálható egyszerű parancssori argumentumokkal, és szabványos XML fájlból. A próbaverzióban csak a csatlakozási paraméterek azok, amelyek környezetfüggőek. Várhatóan kevés ilyen konfigurációs adat lesz, ezért az XML értelmezésénél nincs szükség a hierarchia kiépítésre elegendő az egyszerű szekvenciális feldolgozás egy SAX Parser segítségével. A JavaSE Socket és Scanner osztályainak példányával könnyen kezelhetőek a TCP/IP-n érkező bájtfolyamok. A feldolgozás egy összetettebb feladat, vizsgáljunk meg egy néhány óra alatt beérkezett adathalmazt:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
08091445	0011	G	03	31	0036308975	1091	43406	00	080205
08091448	0027	M	01	32	520	1089	32205	00	080206
08091452	0095		03		0036306335	1056		00	090297
08091452	0034		03		0036309787	1092		00	090298
08091458	0027		03		0036309653	1279		00	090297
08091503	0012		00		520	1266	8006	00	080207
08091504	0034		03		0036303714	1793	1793	00	090297
08091513	0074	M	01	32	0648511	1071	20420	00	080211
08091518	0009		03		0036306636	1793	1793	00	090297
08091525	0007	M	01	32	520	1238		00	080212
08091531	0094		03		0036303912	1411	12605	00	090297
08091534	0016	M	01	32	0612829	1279		00	080214
08091536	0024		03		0036302519	1074	44552	00	090297
08091541	0005	M	01	32		1039	37402	00	080215
08091553	0119	M	01	32	1	1039	37402	00	080217
08091555	0076	M	01	32	0688590	1279		00	080218
08091609	0031	M	01	32	814	1279		00	080220
08091612	0007		03		0036309152	1588		00	090297
08091618	0013		03		0036704117	1588	39377	00	090297
08091621	0008		03		0036203389	1071	20420	00	090297
08091628	0369	M	01	32	00912652513	1200		00	080219
08091635	0021		03		0036707021	1200		00	090297
08091636	0003	G	03	31	0036704117	1588	39377	00	080226
08091636	0011	M	01	32	0652203	1071	20420	00	080225
08091641	0064		03		0036302498	1279		00	090298
08091643	0008		03		0036309152	1588		00	090297
08091719	0041	M	01	32	00912652513	1200		00	080227
08091737	0007		03		0036309787	1092		00	090297
08091737	0006		03		0036309787	1092		00	090297
08091751	0023		03		0036705778	1091		00	090297
08091833	0003	M	01	32	400	1276		00	080229
08091915	0015		03		0036309831	1334		00	090297

6. ábra: Hívásforgalom részlet

1. oszlop: A hívás befejezésének dátuma (év nélkül)
2. oszlop: A hívás időtartama egységekben (egy egység 6mp)
3. oszlop: A feladat szempontjából irreleváns adat.
4. oszlop: Hívás irányának értéke:
    a. 00: magán célú
    b. 01: hivatalos célú
    c. 03: hivatalos célú (GSM inteface-en keresztüli hívás)
5. oszlop: A feladat szempontjából irreleváns adat.
6. oszlop: A hívott fél.
7. oszlop: A hívó kódja.
8. oszlop: A hívó törzsszáma.
9. oszlop: A feladat szempontjából irreleváns adat.
10. oszlop: A kimenő fővonalnak a száma.

Jól látható, hogy néhány rekord esetében üres mezőértékek is szerepelnek, ami a szekvenciális feldolgozást megnehezíti. A 3. és 5. oszlop mezői ugyan számunkra jelentéktelen adatokat tartalmaznak, de a 8. oszlop mezőinek értéke már relevánsak. Ezen mező értéke nincs összefüggésben bármelyik másik mező értékeivel. Az viszont kijelenthető, hogy a rekordok mezői kötött formátumúak így a szekvenciális feldolgozást kiegészítettem egy mintaillesztéssel is. Ez az algoritmus megoldást nyújt egy későbbi tesztelés során felfedezett problémára is. Ha az alkalmazást abban az időpillanatban indítjuk el, amikor a telefonközpont már elindította az adatot a gyűjtő felé, de az még nem küldte vissza a jelzőbitet, akkor ez esetben egy hiányos rekordot fog értelmezni az alkalmazás. Ez egy nagyon ritka eset, de kezelése fontos a szál megtartása miatt. A JavaSE Date és Pattern osztályok példányai segítségével a feldolgozás előtt a rekordok meghívódnak egy ellenőrző, boolean típusú metódus paramétereként, mely visszatérési értékétől függően az alkalmazás értelmezi vagy eldobja – a későbbiek folyamán egy gyűjtő táblába – a rekordot.

5.2. Perzisztencia megvalósítása

Kiemelt igény volt a perzisztens tárolás megvalósítása. Fontos kérés volt, hogy lehetőség szerint olyan keretrendszerrel oldjam meg, amely tudja kezelni az összes megvásárolt adatbázis szervert.

A feladat egyszerűségénél fogva nincsenek különösebb elvárások a keretrendszer felé, így az általam jól ismert Hibernate-re esett a választás, méghozzá a 3.5.6-os verzióra.

A Hibernate a JBoss-hoz tartozó ingyenes keretrendszer, amely tulajdonképpen egy köztes réteg az alkalmazás és az adatbázis meghajtó között, kiküszöbölve az SQL parancsok használatát, az objektumok és táblák között kétirányú leképezést végez. A Hibernate egy rendkívül kényelmes JPA alapú ORM rendszer. Ebben a verziójában már a fejlesztőnek csak annyi feladata van, hogy létrehoz néhány felannotált osztályt (entitást), és abba tárolja az adatokat. A keretrendszer pedig szinte láthatatlanul elvégzi az adatbázis leképezést, és a kapcsolódó adatbázis műveleteket.

A keretrendszer alapértelmezett konfigurációját használva a következő módon fog kinézni egy leképezés:

- Entitás osztály → Adattábla
- Entitás osztály attribútum → Adattábla oszlop (mező)
- Entitás osztály példány → Adattábla rekord

A Hibernate ezen verziója a Java SE 5 újdonságát felhasználva könnyíti meg a leképezések beállítását. A @Entity annotáció segítségével értelmezi a keretrendszer, hogy melyik osztály lesz entitás. Minden entitásnak, amit a relációs adatbázisra akarunk leképezni, rendelkeznie kell elsődleges kulccsal. Elsődleges kulcsként használhatóak primitív típusok, primitív típusok osztályai, ezek tömbjei, illetve szöveg, vagy akár dátum típusok. Ezt a @Id metaadat értékével adhatjuk meg, illetve lehetőségünk van a @GeneratedValue annotáció megadására is, amely paramétereként a generálás stratégiájának típusát adhatjuk meg.

Az entitás osztályon túl létrehoztam egy segédosztályt, melyben a tranzakciókhoz tartozó metódusokat implementáltam:

- **open():** Tranzakció megnyitásáért felelős.
- **addRecord(new Record()):** az entitás osztály egy példányának perzisztálása.
- **close():** A tranzakció commitálásáért felelős.

Mivel a tranzakció demarkáció mindig kötelező (tehát akárhányszor módosítjuk az adatbázist mindig új tranzakciót indítunk), ezért egy telefonközpontból érkező rekord feldolgozása után ez a 3 metódus ebben a sorrendben hívódik meg. (A későbbiek folyamán ezt a Spring keretrendszer oldja meg)

6. Web alkalmazás

6.1. Java Enterprise Edition[5]

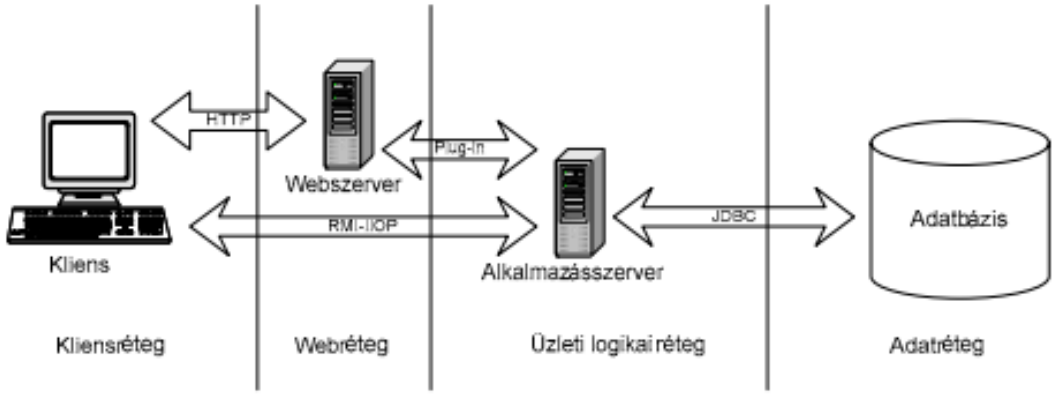
A Java nyelv születése óta a hozzá kapcsolódó technológiák dinamikus fejlődésen mentek át, miközben egyre több feladat megoldására váltak alkalmassá, így használatuk egyre szélesebb körben terjedt el. Mindez indokolttá tette, hogy a Java platformnak 3 kiadása jelenjen meg. Ezen kiadások közül a legnagyobb méretű a Java Enterprise Edition (Java EE, J2EE) elosztott, sok felhasználóval rendelkező, vállalati méretű szoftverrendszerek kialakításakor vethető be. A Java EE-nek részhalmaza a Java SE, vagyis az Enterprise technológiák mellett minden Standard Java Api is rendelkezésre áll.

A Java EE egymondatos meghatározása az alábbi lehetne: architektúra vállalati méretű alkalmazások fejlesztésére, a Java nyelv és az internetes technológiák felhasználásával. A Java EE oly módon támogatja a vállalati méretű rendszerek fejlesztését, hogy gyakori, együtt fellépő problémákra nyújt globális megoldást a futtató környezet. (pl.: perzisztencia, többszálúság, tranzakciókezelés, névszolgáltatás, aszinkron üzenetkezelés, biztonság stb.)

6.2. Az n rétegű architektúra és az alkalmazásszerver[6]

Az előbbi fejezetben hivatkoztam a futtató környezetre, amely middledware-szolgáltatásokat nyújt az alkalmazások számára. Ez a futtatási környezetet Java EE esetében az alkalmazásszerver.

[5] Szoftverfejlesztés Java EE platformon (Szak Kiadó 2007. Imre Gábor) 1.-2. oldal
[6] Szoftverfejlesztés Java EE platformon (Szak Kiadó 2007. Imre Gábor) 6.-7. oldal



7. ábra: Az N rétegű architektúra

Adatréteg: feladata az adatok perzisztens tárolása, és az azokon végezhető elemi műveletek támogatása Leggyakrabban ezt a réteget relációs adatbázis segítségével valósítják meg, bár egyre kiforrottabb megoldást biztosítanak az objektumorientált adatbázisok.

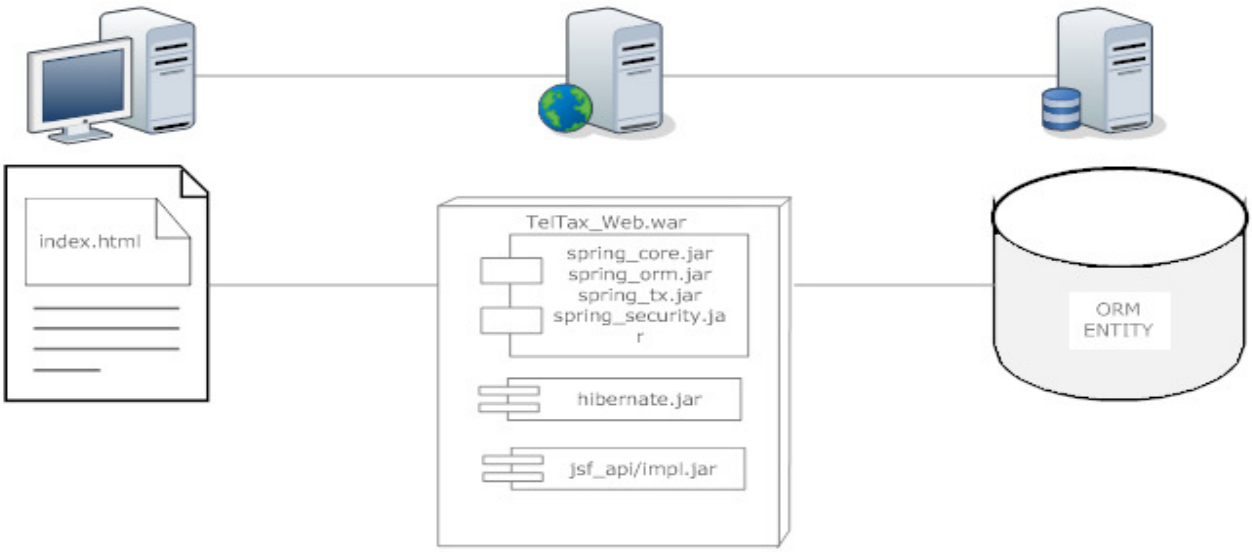
Üzleti Logikai Réteg: az adatrétegre épül, amely a konkrét alkalmazási terület igényeinek megfelelő funkcionalitást biztosítja, oly módon, hogy az üzleti szabályok figyelembevételével hívja meg az adatréteg szolgáltatásait.

Kliensréteg: biztosítja az alkalmazás felhasználói felületét, vagyis a felhasználói beavatkozások hatására meghívja a megfelelő üzleti logikai funkciót, majd a hívás eredményének megfelelően frissít bizonyos felhasználói felületelemeket. A kliens alapvetően két típusú lehet: vékony illetve vastag kliens. A két típus között megfigyelhető a határ elmosódása, a gazdag webes klienseknek köszönhetően. Természetesen lehetséges, hogy az alkalmazásszerveren futó üzleti logikához több kliens-típus is csatlakozzon. Ilyenkor mutatkozik meg a szigorú rétegelt architektúra egyik előnye: ha az üzleti logikát valóban az alkalmazásszerverre telepített rétegbe koncentráltuk, akkor a különféle kliensekben nem kell az üzleti logikát megismételnünk, csupán a megjelenést kell lecserélnünk.

Webréteg: amennyiben vékony klienseket is ki akarunk szolgálni, szükség van egy webrétegre is, amelyik a böngészőktől érkező http-kéréseket értelmezi, meghívja a megfelelő üzleti logikát, majd pedig megfelelő (tipikusan HTML-, de akár XML-, WML-, tetszőleges bináris formátumú) választ generál. A webréteget Java EE esetén

szintén az alkalmazásszerverre telepítjük. Az alkalmazásszerver képes közvetlenül fogadni a böngészőktől érkező kéréseket, de gyakran egy webszervert is beiktatnak az alkalmazásszerver elé, amely például a statikus erőforrásokat képes kiszolgálni, így csak azokat a kéréseket továbbítja az alkalmazásszerver felé, amelyek üzleti logika meghívását igénylik.

6.3. Az architektúra konkrét megvalósítása



8. ábra: Az architektúra konkrét megvalósítása

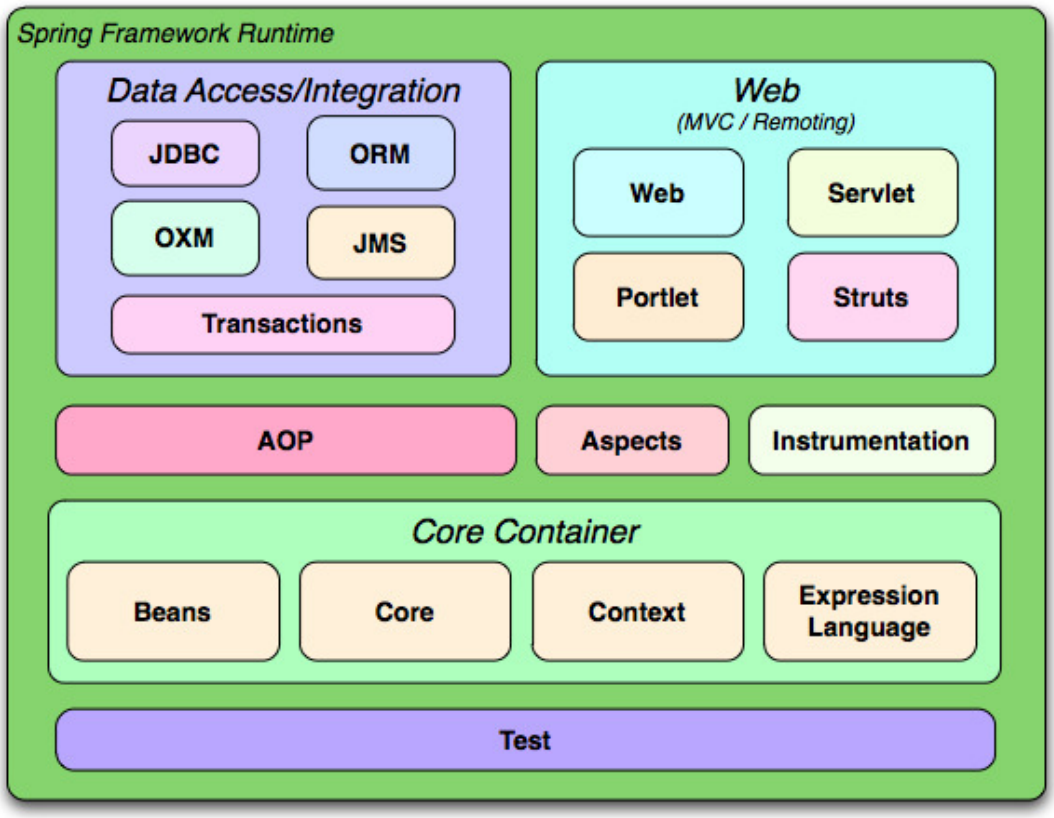
Adatbázis szint: Egyszerű relációs adatbázisra van szükségünk, ami tud kezelni tárolt eljárásokat. A MySQL újabb változatai már képesek erre, így az 5.5.8-as verzióra esett a választás.

Üzleti logikai réteg: A Springnek köszönhetően (lásd: következő fejezet) elegendő egy egyszerű servlet container. Az Apache TomCat 6.0.29-es verziója rendelkezik a szervletek futtatásához szükséges eszközökkel, Jasper előfordítóval, illetve jsf apival. Külön webszerverre nincs szükségünk, ugyanis a TomCat elvégzi a statikus adattartalmak kezelését is.

Megjelenési réteg: Egyszerű vékonykliens a felhasználó oldali megjelenítéshez.

6.4. A rendszer alapja: Spring framework

A Springet a Java eszközök svájci bicskájaként is szokták emlegetni, annyi nem feltétlenül összetartozó, de integrálható eszközt tartalmaz. Fogalmazhatunk úgy is, hogy a korabeli Java EE szabvány pehelysúlyúbb alternatívájaként született. Úgy tekintek a két verzióra, mintha egy OpenSource vs Standard különbséget kellene vizsgálni, azzal az érdekességgel, hogy a Standard fokozatosan átemeli az OpenSource-ban már jól bevált eszközöket. Nem célom bemutatni a teljes keret-rendszert, viszont az általam használt eszközökről néhány szó erejéig kitérek (Beans, ORM, Transactions, Security)



9. ábra: Spring moduláris felépítése[7]

[7] http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/images/spring-overview.png

6.4.1. DI Pattern és a Spring DI

Kezdő fejlesztők is biztosan találkoztak már olyan problémával, hogy példányosítás után egy másik kontextusban is szükség volt az objektumra, tipikus példa egy GUI-s asztali alkalmazás. A másik gyakran előforduló eset, amikor egy magasabb szintű komponensnek egy absztrakt interfésszel definiált szolgáltatást fecskendezünk be, például egy riportkészítő program számára egy adatforrást. Az adatforrás interfészének definiálása után különböző implementációkat készíthetünk, amelyeket bármikor lecserélhetünk. Az adatforrás megválasztása ettől kezdve konfigurációs kérdéssé válik: megadhatjuk XML konfigurációs fájlokban, illetve annotációkkal, hogy hogy melyik adatforrás kerüljön példányosításra és befecskendezésre.

A Spring által felkínált injektálási módok:

- *Setter Injection*: Ezzel a módszerrel a befogadó objektum setter metódusain keresztül állítja be a függőségeket. Előfeltétele, hogy létezzen a befogadó objektumnak paraméter nélküli konstruktora, és a Java Bean szabványnak megfelelő setter metódusa.
- *Constructor Injection*: Ezzel a módszerrel a függőségek injektálása példányosításkor történik, a konstruktor paraméterein keresztül.
- *Interface Injection*: Egy interface helyére annak egy megvalósítását injektálhatjuk. (Több megvalósítás esetén konkretizálni kell)

A beanek hatáskörökkel rendelkeznek: Singleton, Prototype, Request, Session. Alapértelmezettként Singleton objektum keletkezik a kontextusban. Az alkalmazásomban a konstruktoron keresztüli befecskendezést használtam.

6.4.2. Spring ORM

A Spring támogatja a JPA2 szabványt, jól használható együtt Hibernate-tel, TopLink-kel, EclipseLink-kel és mint szabványos JPA megvalósítással is. Azaz használhatunk például hibernates Session-t és JPA-s EntityManager-t is a kódunkban a Spring segítségével. Míg az adatrögzítő alkalmazásban kézzel történt az EntityManager kezelése (pl.: létrehozás, lezárás), addig a webalkalmazásban a Spring megold mindent, így nagyjából minden, ami macera lenne a csupasz JPA használatban, kikerül a képből.

Az adatbázis elérés történhet JNDI LookUp-pal, vagy egyszerűen Persistence Unit konfigurálásával. Mivel volt már egy kész konfigurációm az adatrögzítő alkalmazásról, így felhasználva az utóbbi lehetőséget választottam. Gyakorlatilag nem történik más, mint a kontextus kiépülésekor a Hibernate végigpásztázza a kijelölt csomagokat, és a @Entity annotációval ellátott osztályokból a szabvány szerint adatbázis táblákat képez.

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="${jdbc.driverClass}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"
    p:url="${jdbc.url}">
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"/>

<bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
    id="entityManagerFactory" lazy-init="false" p:dataSource-ref="dataSource">
    <property name="persistenceUnitName" value="TelTax"/>
</bean>
```

10. ábra: Spring ORM beállítása

Az ORM alapú adatbázis elérésünk ezzel el is készült, viszont a tranzakciók kezelése kézzel történik. Az adatrögzítő alkalmazás esetében ez még nem jelentett gondot, de a webalkalmazás esetében ez nemcsak nem kényelmes, de nem is biztonságos. Így nincs más teendőnk, a Springre bízzuk a tranzakciók kezelését.

6.4.3. Tranzakciókezelés

Az ORM konfigurálásához hasonlóan nincs más teendőnk, mint a tranzakciókezelő propertyhez hozzárendelni a megfelelő menedzser interfész implementációját.

```
<bean class="org.springframework.orm.hibernate3.HibernateTransactionManager"
    id="transactionManager" p:sessionFactory-ref="sessionFactory"/>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

11. ábra: Tranzakciómenedzser bekötése

Ezek után, már tényleg egyszerű dolgunk van. Azon függvényeket, eljárásokat, amelyek adatbázis műveletet hajtanak végre (speciális beállítás lehet írásra, olvasásra), „tranzakcionális környezetbe” kell helyeznünk. Ezt a @Transactional annotációval tudjuk a legkönnyebben megtenni. A menedzser elvégzi a tranzakciók commit-olását, illetve a rollback-elést.

A Spring tranzakciókezelője az alábbi módokat támogatja:

- MANDATORY: (kizárólag tranzakcióba kerül)
- NESTED: (párhuzamos tranzakcióba kerül)
- NEVER: (kizárólag tranzakció nélkül)
- NOT_SUPPORTED: (tranzakció nélkül)
- REQUIRED: (meglévő tranzakcióba kerül)
- REQUIRES_NEW: (kizárólag új tranzakcióba kerül)
- SUPPORTS: (új vagy meglévő tranzakcióba kerül)

Az alkalmazásomban minden olyan függvény, eljárás amely módosítást végez az adatbázis bármely elemén az REQUIRED, aminek nincs hatása az adatbázisra, az elegendő SUPPORTS módban.

6.4.4. Spring Security

A probléma egyszerűségénél fogva, ágyúval lőttem verébre. Bármilyen JAAS implementáció, vagy egy saját elképzeléseim szerint implementált filter is tökéletesen ellátta volna a feladatot. Mivel a Spring architektúra már ki volt építve, és kéznél volt a Spring Security, így emellett döntöttem. Néhány szó a Spring Security-ról:

- Környezetfüggetlen (pl.: alkalmazásszerver, ejb-k)
- XML-el és annotációkkal könnyen konfigurálható, implementáció cserélhető
- HTTP Basic, HTTP Digest, OpenID, X.509 autentikáció támgotása
- Beépített password encoder-t tartalmaz
- XML-DB alapú account tárolás támogatása
- Könnyen illeszthető Single Sign ON szolgáltatókhoz.

Webes környezetben nincs más teendőnk, mint bejegyezni a DelegatingFilterProxy filter-t a web.xml-be, és beállítani hozzá a védett url mintát. Ezek után, ha a filter sikeresen elkapta a kéréseket, a feldolgozás beállításai következnek:

```
<authentication-manager alias="authenticationManager">
    <authentication-provider>
        <user-service>
            <user name="admin" password="admin"
                authorities="ROLE_USER, ROLE_ADMIN" />
            <user name="user" password="user"
                authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

12. ábra: Az autentikáció-kezelő bekötése

Az authentication-provider csomópontba egyenlőre XML alapú beégetett szerepkörrel rendelkező felhasználók kerültek (egy felhasználó több szerepkörrel is rendelkezhet) .

Ahhoz, hogy ne csak XML alapú legyen az account kezelés, szükségünk lesz egy USER entitásra, ami kiterjeszti a UserDetails interfészt.

Method Summary	
java.util.Collection<GrantedAuthority>	getAuthorities() Returns the authorities granted to the user.
java.lang.String	getPassword() Returns the password used to authenticate the user.
java.lang.String	getUsername() Returns the username used to authenticate the user.
boolean	isAccountNonExpired() Indicates whether the user's account has expired.
boolean	isAccountNonLocked() Indicates whether the user is locked or unlocked.
boolean	isCredentialsNonExpired() Indicates whether the user's credentials (password) has expired.
boolean	isEnabled() Indicates whether the user is enabled or disabled.

13. ábra: UserDetails interfész metódusai[8]

Valamint definiáljunk egy UserService nevű @Repository osztályt, és a trükk csak annyi, hogy implementálnia kell a UserDetailsService interfészt.

Method Summary	
UserDetails	loadUserByUsername(String username) Locates the user based on the username.

14. ábra: UserDetailsService interfész metódusa[9]

[8] http://static.springsource.org/spring-security/site/docs/3.0.x/apidocs/org/springframework/security/core/userdetails/UserDetails.html (2011.04.20.)
[9] http://static.springsource.org/spring-security/site/docs/3.0.x/apidocs/org/springframework/security/core/userdetails/UserDetailsService.html (2011.04.20)

Ez után már nem az XML-ben szereplő felhasználókat, hanem az adatbázisban szereplő felhasználókat fogja alapul venni, a User entitás alapján. A végleges állapot, amelyben már md5-tel hashelt jelszavak kerülnek így leegyszerűsödik:

```
<authentication-manager alias="authenticationManager">
    <authentication-provider>
        <password-encoder hash="md5"/>
    </authentication-provider>
</authentication-manager>
```

15. ábra: Autentikáció-kezelő bekötése

Ezzel már majdnem készen is volnánk, csak az interceptorok beállítása hiányzik. A saját konfigurációmban a JSF keretrendszer adja a *.jsf oldalakat, melyek mindegyikét védeni kell, kivéve az index.jsf oldalt, (lletve a css és design könyvtár). Ezen kívül a sites/user/*.* oldalak esetén a ROLE_USER szerepkör szükséges (bejelentkezett felhasználó), illetve a sites/admin/*.* oldalak esetén nem elég a ROLE_USER jogkör, hanem rendelkezni kell ROLE_ADMIN szerepkörrel is. Sikertelen bejelentkezés esetén történjen átirányítás a /denied.jsf oldalra, sikeres bejelentkezés esetén a szerepkörhöz tartozó nyitóoldalra. Kijelentkezés után ismét az /index.jsf oldal jöjjön be. A konfiguráció a következőképpen alakul:

```
<http access-denied-page="/denied.jsf" auto-config="false">
    <form-login authentication-failure-url="/index.jsf" default-target-url="/sell.faces"
        login-page="/index.jsf" login-processing-url="/j_spring_security_check"/>
    <logout logout-success-url="/index.jsf" logout-url="/logout.jsf"/>
    <intercept-url pattern="/auth/**" access="ROLE_USER" requires-channel="any"/>
    <intercept-url pattern="/css/**" access="IS_AUTHENTICATED_ANONYMOUSLY" />
    <intercept-url pattern="/design/**" access="IS_AUTHENTICATED_ANONYMOUSLY" />
    <intercept-url pattern="/index.*" access="IS_AUTHENTICATED_ANONYMOUSLY" />
    <intercept-url pattern="/sites/admin/**" access="ROLE_ADMIN" />
    <intercept-url pattern="/sites/user/**" access="ROLE_USER" />
</http>
```

16. ábra: Interceptor konfiguráció

A Spring keretrendszert az előző 3 komponensét felhasználva-, bekonfigurálva egy tökéletes alapot használhatok a rendszer implementálásához.

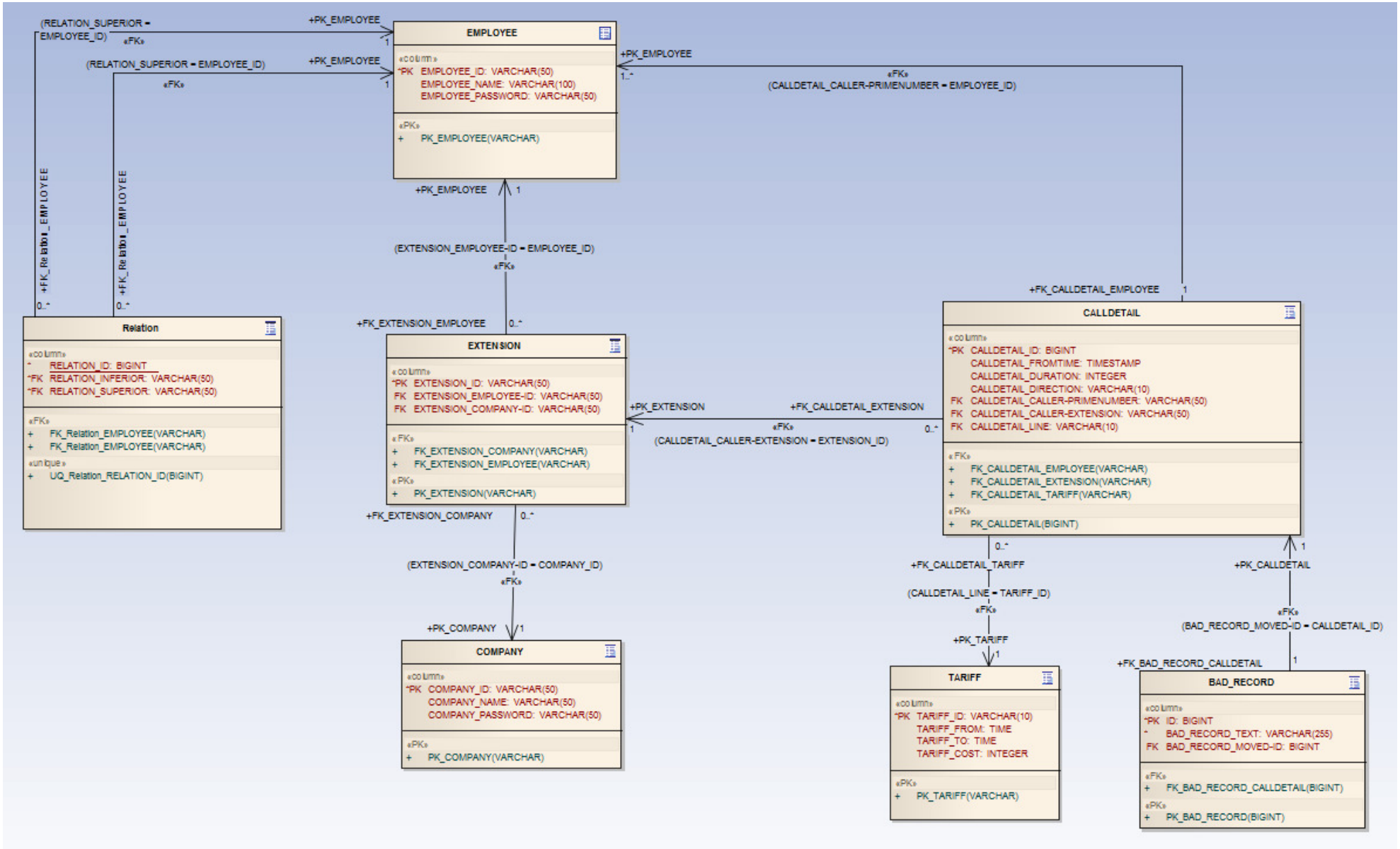
6.5. Adatbázis réteg

A rezidens és a webalkalmazásnak a következő adattáblákkal kell dolgoznia:

- EMPLOYEE: (A vállalat saját dolgozóinak táblája)
- COMPANY: (A vállalat területén működő külsős cégek táblája)
- EXTENSION: (Telefonmellékek táblája)
- CALLDETAILS: (A beérkező hívás rekordoknak a részletei)
- TARIFF: (Tarifa beállításokat tartalmazó tábla)
- RELATION: (Beosztási fának szülő-gyermek csomópontját reprezentáló tábla)
- BAD_RECORD: (Hibás rekordokat reprezentáló tábla)

A táblák közötti kapcsolatok:

- Az adatrögzítő alkalmazás folyamatosan pusholja a CALLDETAIL és a BAD RECORD táblát.
- CALLDETAIL_LINE külső kulcsa a TARIFF táblának, ha null értékű, akkor az adott hívásrekordnak nem lehet kiszámolni a költségét.
- BAD_RECORD_MOVED-ID külső kulcsa a CALLDETAIL táblának, abban az esetben nem null értékű, ha sikerült a hibás rekordot kijavítani.
- A CALLDETAIL_CALLER_PRIMENUMBER és/vagy a CALLDETAIL_CALLER_EXTENSION oszlopokból egyértelműen meghatározható a hívó személye, így ezek külső kulcsként használhatóak.
- A RELATION tábla a viszonyokat reprezentálja, rekurzív módon lekérdezhető adott EMPLOYEE beosztottainak listája.



17. ábra: A rendszer adatbázis diagramja

6.6. Üzleti logikai réteg

Az alkalmazás fejlesztése során törekedtem a minél flexibilisebb kódok készítésére. Gondolok itt arra, hogy ha a fejlesztés során megváltoznak az igények, akkor az lehetőség szerint minél kevesebb újratervezést igényeljen. Ezért a következő architekturális megoldást választottam:

- DAO (adatelérési réteg)
- SOA (logikai réteg)
- DTO (transzport réteg)

DAO

Adatelérési réteg, amely alapvető adatbázis elérési műveleteket lát el. A tervezési minta szigorúan megköveteli, hogy ebben a rétegben a CRUD műveleten kívül más ne legyen implementálva. (Create, Retrieve, Update, Delete). Gyakorlatilag a pattern arról szól, hogy az adatbázist egy programozói interfész mögé rejtjük. Minden egyes entitáshoz készítettem egy-egy DAO-t, amely leszármazottja a GenericDAO ősosztálynak. A Generic osztály elnevezés nem véletlen, mivel így kihasználom a java 1.5 nyújtotta újítás lehetőségét, hogy az osztály tartalma generikusan van értelmezve.

SOA (szűkebb értelemben)

Az architektúra alapeleme a szolgáltatás. Egy szolgáltatásorientált rendszer szolgáltatásokból és a hozzájuk kapcsolódó interfészekből épül fel. A komponenstechnológiákhoz képest ez egy jóval lazábban csatolt architektúrát eredményez. A szolgáltatások autonóm szoftveregységet, egymástól függetlenül léteznek, mindegyiknek megvan a maga feladata, felelőssége.

DTO

Transzport réteg, amely a külvilág felé szolgáltatja a szükséges adatokat. Fontos, hogy csak azt, amire szükség van. Ha nem használnánk DTO-t, akkor minden adatokhoz külön-külön lekérdező függvényt kell írnunk, ráadásul mivel ezeket a kliens oldalra is el kell juttatni, ezért minden lekérés külön hálózati forgalmat generál. Ezeket az overheadeket úgy küszöbölhetjük ki, ha a webrétegnek entity bean referencia helyett egy olyan egyszerű adathordozó objektumot adunk át, amely az attribútumokat getterek segítségével közvetlenül elérhető tagváltozókban tárolja

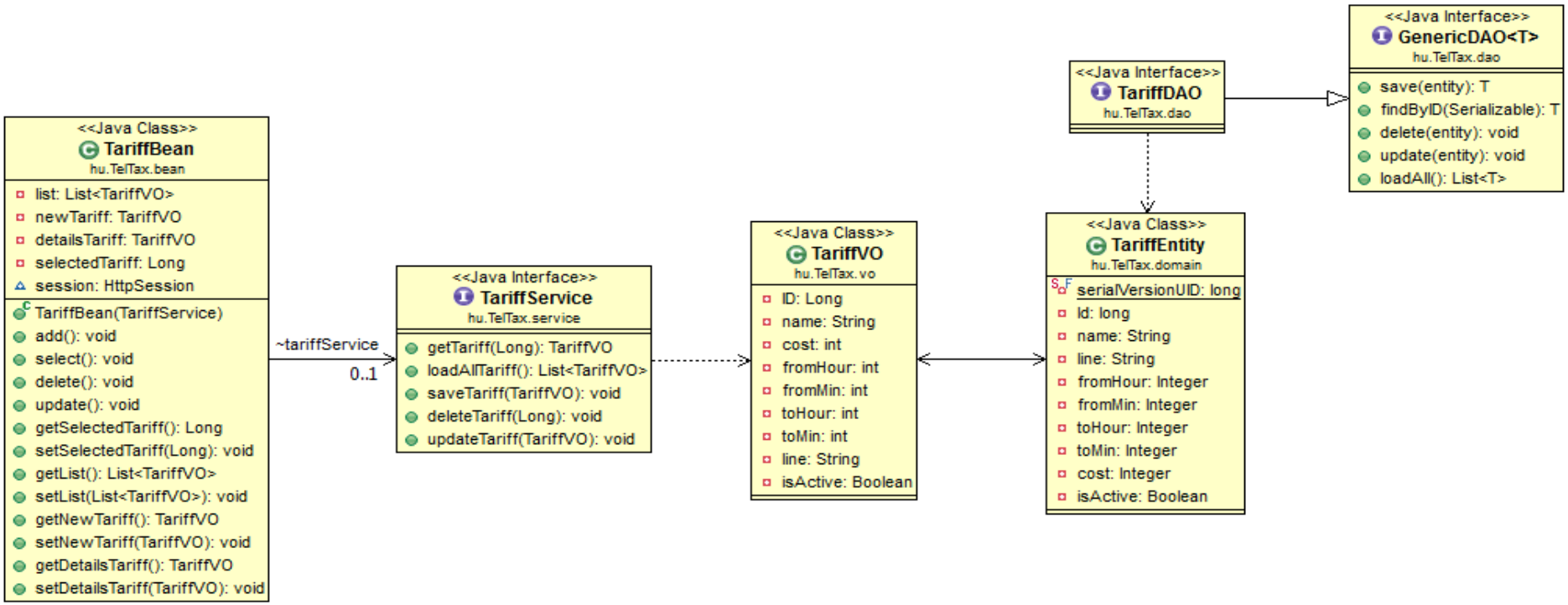
Tehát a transzport réteg VO objektumok gyűjteménye, amely megfelel a Java Bean szabványnak. Ez a későbbiekben tovább egyszerűsíti a dolgom, ugyanis a felületi megjelenítésnél a JSF keretrendszer támogatja a POJO objektumok kezelését.

BL

A web és az adatrögzítő alkalmazás alapvetően tiszta, egyszerű üzleti logikával épül fel, néhány speciális esettől eltekintve:

- *Tarifázás:* Adott hívásrekord megérkezésekor a hívásindítás idejének és kimenő fővonalszámának függvényében kerül számolásra az adott hívás költsége.
- *Beosztási viszonyok kezelése*: adott vezető, (aki ugyanolyan felhasználói szerepkörrel rendelkezik, mint a beosztott) láthatja az ő hatáskörébe eső dolgozók telefonhívásait. (az adott vezető, és a vezető beosztottainak hívásait, az ő felettese is megtekintheti. azonos „szinten” lévő vezetők nem látnak át egymásra)
- *Hibás rekordok kezelése:* Speciális esetekben az adatrögzítő alkalmazás nem tudja megfelelően értelmezni a beérkező rekordot. Ezeket nem törli, hanem eltárolja, mert bizonyos esetekben, ezek még külső segítséggel feldolgozhatóak.
- *Havi összesítés:* Havi riport készítése XLS/PDF formátumban.

6.7. A rendszer fejlesztői szemmel



18. ábra: Webalkalmazás osztálydiagram

6.8. Megjelenítési réteg

A jól megtervezett projekt és tapasztalt fejlesztők ellenére is gyakori hiba a megjelenítési réteg elnagyolása. Gondolok itt arra, hogy általában ez az utolsó fázis, szoros az átadási határidő, ezért kevesebb időt fordítanak rá, mint amennyi szükséges volna. Pedig az egyik, talán legfontosabb réteg, ugyanis ez a legértékelhetőbb a megrendelő számára. Gyakorlatilag mit ér a jól megtervezett alsóbb réteg, ha a megjelenítési réteg átláthatatlan, kezelhetetlen.

Fontos kritériumok, amiket szem előtt tartottam a tervezéskor:

- találó projekt elnevezés, mint brand
- gyorsaság (minél kevesebb adatforgalom)
- egyszerű színek, egyszerű felület
- könnyen értelmezhető űrlapok
- újrafelhasználható templatek

A gyorsaság titka: AJAX[10]

Az Ajax (Asynchronous Javascript and XML) interaktív webalkalmazások létrehozására szolgáló webfejlesztési technika. A weblap kis mennyiségű adatot cserél a szerverrel a háttérben, így a lapot nem kell újratölteni minden egyes alkalommal, amikor a felhasználó módosít valamit. Ez növeli a honlap interaktivitását, sebességét és használhatóságát.

Az Ajax nem egy technológia önmagában, hanem egy kifejezés a következő technikák kombinációjára:

- XHTML (vagy HTML) és CSS a tartalom leírására és formázására.
- DOM kliens oldali script nyelvekkel kezelve a dinamikus megjelenítés és a már megjelenített információ együttműködésének kialakítására.

[10] http://hu.wikipedia.org/wiki/Ajax_(programozás) (2011.04.20)

- XMLHttpRequest objektum az adatok aszinkron kezelésére a kliens és a webszerver között. Néhány Ajax keretrendszer esetén és bizonyos helyzetekben IFrame-et használnak XMLHttpRequest objektum helyett.
- XML formátumot használnak legtöbbször az adattovábbításra a kliens és a szerver között, bár más formátumok is megfelelnek a célnak, mint a formázott HTML vagy a sima szöveg.

Az egyszerűség, nagyszerűség titka: JSF[11]

A java Server Faces technológia az egyik leginkább elterjedt webes keretrendszer, köszönhetően annak, hogy számos Java EE alkalmazásszerver és fejlesztői környezet alapértelmezetten támogatja. Egy ideális webes keretrendszer célja, hogy elegánsan és biztonságosan kapcsolja össze a grafikus és felülettervező által megálmodott felhasználói felületetet a mögöttes üzleti logikával.

A JavaServer Faces egy komponensorientált webprogramozási keretrendszert specifikál, mely része a Sun Java EE 5 platformjának. Fontos kiemelni, hogy a JavaServer Faces csupán egy specifikáció, melynek számos implementációja létezik a Sun által kiadott renferenciaimplementáción túl. Az én választásom a PrimeFaces implementációra esett, ugyanis az elterjedt 1.0, 1.2-es JSF verzió helyett, ez már az új, 2.0-s JSF szabványra épül.

A JSF framework kész megoldást nyújt a következő kérdésekre:

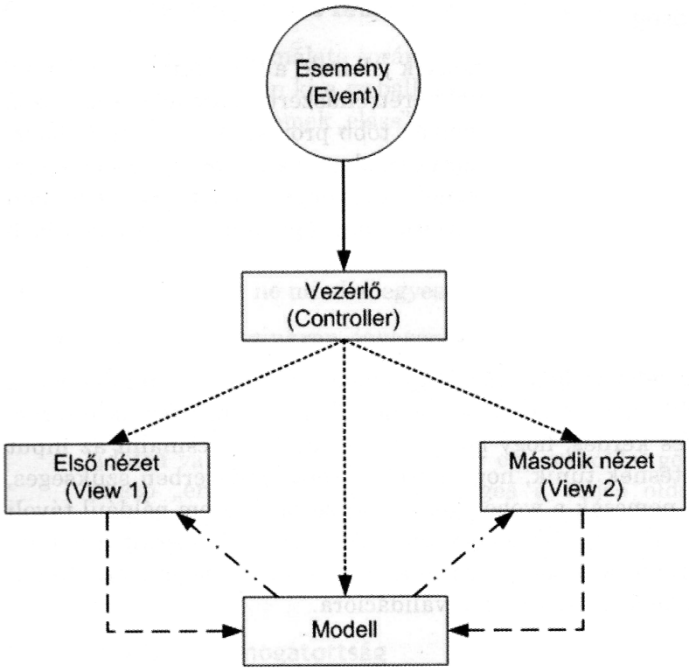
- Konverzió (HTML felületen minden adat String típusú)
- Validáció (Ha lehetőség van rá, már a megjelenítési rétegben történjen szintaktikai és szemantikai vizsgálat)
- Újrafelhasználhatóság (Template oldalak létrehozása, a „statikus” tartalmakhoz)
- Kész komponensek (Magát az oldalt is újrafelhasználható építőelemekből hozhatjuk létre)

[11] Szoftverfejlesztés Java EE platformon (Szak Kiadó 2007. Imre Gábor) 167.-173. oldal

- Navigáció (Folyamatos navigáció, és feltételhez kötött navigáció is engedélyezett)
- Nemzetköziesítés támogatása (Többnyelvű oldalak kezelése properties fájlok segítségével)
- Állapotmentés (Kiküszöbölve a http protokoll állapotmentességét, munkamenetek kezelése)
- MVC támogatása (Nézet és viselkedés szétválasztása)
- Biztonság (A gyakoribb támadási módszerek ellen beépített megoldást nyújt, pl.: sql inject, cross-site scripting)
- Webes technológiák támogatása (CSS, JavaScript, Ajax)
- Bővíthetőség (Saját komponensek, illetve már kész komponens implementációk használata)

Ezek tehát a legfontosabb szolgáltatásai egy modern webes keretrendszernek.

A JSF titka: az MVC pattern[12]



19. ábra: Az MVC tervezési minta

[12] Szoftverfejlesztés Java EE platformon (Szak Kiadó 2007. Imre Gábor) 173.-175. oldal

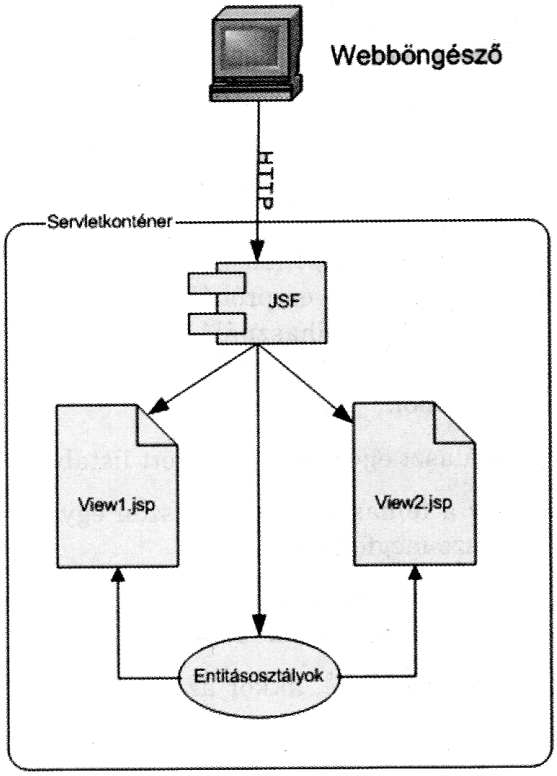
A megvalósításban a Modell alatt egy vagy több osztályt értünk, melyek a megjelenítendő entitásokat írják le. A megjelenítést a View végzi, mely képes arra, hogy az információkat rendezett formában, a megtervezett GUI sémában megjelenítse. A megjelenítési folyamat során a kiszolgáló és a felhasználó közötti párbeszédet pedig a Controller koordinálja.

Vizsgáljuk meg, miképpen történik ez http-alapú technológia esetén:

1. Beérkezik a kérés a felhasználó böngészőjétől. A kérés tartalmazza, hogy melyik oldalt, milyen paraméterekkel szeretnénk megtekinteni, valamint számos kiegészítő információt (pl : cookie-kat)
2. A beérkezett kérést a Controller dolgozza fel, amennyiben a rendszerben több Controller található,, úgy az URL és a paraméterek dekódolása után átadótik a vezérlés az oldalhoz tartozó Controllernek.
3. A Controller feldolgozza a kapott paramétereket, és előállítja (vagy ha már létezik, akkor frissíti) a Modellt.
4. A Modell alapján a View előállítja a felhasználónak elküldendő felületet, majd továbbítja azt a böngésző felé.
5. A felhasználó értelmezi az adatokat, majd új kérést küld, és a folyamat újraindul.

Ha az MVC patternt megpróbáljuk illeszteni a projektemre, akkor a következő eredményre jutunk

- Modell → Value Object
- View → XHTML (JSF felület)
- Controller → Servlet + Bean



20. ábra: A JSF működése

A JSF keretrendszer használatához nincs más teendőnk, mint a web-xml.ben egy újabb filter bekötése, ami egy input (én esetemben XHTML) adatforrásból egy HTML outputot generál.

A dizájn és a használhatóság titka: CSS

A CSS jelentése Cascading Style Sheets, azaz egymásba ágyazott stíluslapok. A HTML oldalaink megjelenését befolyásoló egyszerű nyelvről van szó, mely segítségével meghatározhatjuk, hogy hogyan (és hol) jelenjenek meg az egyes HTML elemek (paragrafusok, címsorok, stb.), többek között befolyásolhatjuk a színüket, méretüket, elhelyezkedésüket, margóikat, stb. Az egymásba ágyazhatóság (kaszkádolás) arra utal, hogy több stíluslapot, meghatározást is megadhatunk egyszerre, illetve egy stílus lehet több elemre is érvényes, amit egy másik stílussal felüldefiniálhatunk. A stílusok öröklődnek az oldal hierarchiája szerint, ha például a gyökér elemre definiálunk egy stílust, akkor az többnyire az oldal összes elemére érvényes (a tulajdonságok örökölhetőségétől függően).

A képi megjelenítésen túl, van még egy másik feladat: A rendszernek egy találóbb elnevezést kell kitalálni, mint a Telefonközponti Tarifázó Rendszer. Azért gondolom így, mert ha a későbbiek folyamán napi rutinként fogják használni a dolgozók, akkor egy mozaikszóval könnyebb együtt élni. A rendszerem így a TelTax fantázianevet kapta. A mozaikszó további ötleteket adott arra, hogyan lehetne ezt még egyedibbé tenni. Így az Alkaloida Vegyészeti Gyár Zrt. mákgubó lógója is bekerült, jelezvén, hogy egy belső rendszerről van szó.

21. ábra: Fantázia név és a logó

7. UpToDate beosztási fa[13]

A fejezet témája már nem tartozik az implementálandó feladataim közé, de a problémára megpróbálok kész megoldást nyújtani. Vegyük azt az esetet, ha hónap közepén történik egy áthelyezés a Vállalaton belül. Ebben az esetben más költséghellyel kell számolni, más költséghely vezető alá fog tartozni az érintett személy. Ugyan ritka az ilyen eset, mégis jobb, ha a rendszer automatikusan tud reagálni rá, főleg akkor, ha van is rá kész eszközünk.

A PL/SQL triggereket elsősorban olyan logikai hibák megakadályozására használnak, amelyeket egyszerű CHECK paranccsal nem lehetne megakadályozni. A triggerek olyan speciális procedúrák egy adatbázisban, amelyeket az INSERT, UPDATE, DELETE parancsok végrehajtások előtt, után vagy helyett hív meg a rendszer. Röviden: a trigger vagy engedélyezi vagy elveti az adott táblán történt módosításokat.

A triggerek fajtái:

- **BEFORE-trigger**: A BEFORE (vagy INSTEAD OF) trigger fő jellemzője, hogy a triggert egy INSERT, UPDATE vagy DELETE parancs helyett hajtják végre. Tehát a parancs a trigger lefutása után sem fog végrehajtódni. Ez azt jelenti, hogy ha mégis úgy döntünk, hogy a parancs nem vezet inkonzisztenciához, akkor azt a triggeren belül nekünk kell "kézzel" újra végrehajtanunk.
- **AFTER-trigger:** A leggyakoribb triggerfajta (AFTER) fő jellemzője, hogy a triggert egy INSERT, UPDATE vagy DELETE parancs futtatása után (tehát amikor az adatok már módosultak) hajtják végre. Ha a triggerben a tranzakciót visszavonják (ROLLBACK), akkor az adatokat az adatbank a ROLLBACK-folyamat keretében törli. Ha a trigger mindent rendben talál, akkor az a táblázatban levő adatokon már nem változtat, hisz az SQL-parancs már a trigger meghívása előtt végrehajtódott.

[13] http://hu.wikipedia.org/wiki/Trigger_(adatbázisok) (2011.04.20.)

Tovább gondolva a triggerek használatát, a következő pszeudó-pl/sql kód segíthet a probléma megoldásában:

1. trigger az ERP táblájára
2. after insert, update, delete
3. begin
4. drop viszony tábla
5. update viszonytábla from ERP tábla
6. end

8. Felhasználói kézikönyv

A fejezet célja a diplomamunkám témájának bemutatásán túl az, hogy segítségét adjon a kevésbé hozzáértő embereknek is eligazodni a rendszerben.

8.1. Rezidens alkalmazás

Az alkalmazás parancssoros környezetben Java Virtuális gép segítségével platformtól függetlenül az alábbi módon futtatható:

```
c:\Work>java -jar TelTax_RA.1.1-SNAPSHOT.jar -help
TelTax DataRecorder Application v1.1
Help menu:
[options] [args]
Available options
        -url ip:port
        -log filename

c:\Work>java -jar TelTax_RA.1.1-SNAPSHOT.jar -url 10.100.103.6:3030 -log recording.dat
```

22. ábra: A rezidens alkalmazás parancssoros környezetben

Paraméterek:

- URL: A telefonközpont (Jelen esetben a TCP/IP konverter) címe (ip:port formátumban).
- LOG: Az alkalmazás naplózásának végpontja. (előtag.kiterjesztés formátumban)

Lehetőség van paraméter nélküli indításra is, ilyen esetben a setup.cfg XML fájlból szedi fel az argumentumokat az alkalmazás. Az előző beállítások XML formátumban:

```
<?xml version="1.0" encoding="UTF-8"?>
<TelTax_cfg>
    <url>10.100.103.6:3030</url>
    <log>recording.dat</log>
</TelTax_cfg>
```

23. ábra: XML paraméterezés

8.2. Web alkalmazás

A webalkalmazás bármilyen webes böngészőből futtatható. Az első használatkor az üdvözlő bejelentkező képernyőn találjuk magunkat.

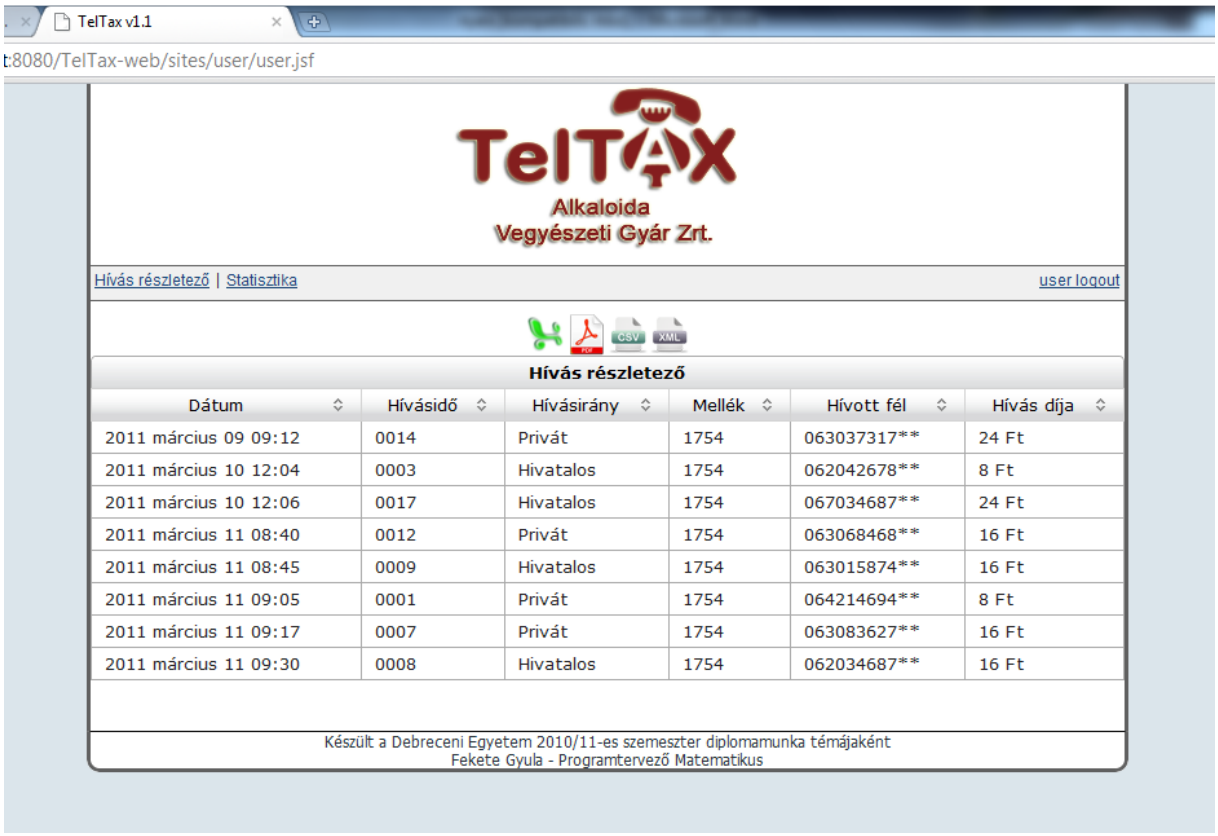
24. ábra: Bejelentkező képernyő

Az oldal mezői és akciói:

- **Törzsszám** / **Cégszám:** Bejelentkezési azonosító felhasználói nevének felel meg, a Vállalat dolgozói a törzsszámukkal tudnak bejelentkezni. A külső cégek pedig a hozzájuk rendelt cégszámmal. Kötelező mező!
- **Jelszó:** Minden felhasználóhoz előre definiált jelszó tartozik. Módosítani csak rendszeradminisztrátori jogkörrel lehet. Kötelező mező!
- **Belépés:** A rendszer használatba vétele a megadott felhasználói azonosítókkal. Valótlan azonosítás esetén hibaüzenet jelenik meg.

8.2.1. Felhasználói szintű lehetőségek

Sikeres Felhasználói szintű bejelentkezés után a felhasználó saját havi indított hívásainak részleteit tekintheti meg:



Dátum	Hívásidő	Hívásirány	Mellék	Hívott fél	Hívás díja
2011 március 09 09:12	0014	Privát	1754	063037317**	24 Ft
2011 március 10 12:04	0003	Hivatalos	1754	062042678**	8 Ft
2011 március 10 12:06	0017	Hivatalos	1754	067034687**	24 Ft
2011 március 11 08:40	0012	Privát	1754	063068468**	16 Ft
2011 március 11 08:45	0009	Hivatalos	1754	063015874**	16 Ft
2011 március 11 09:05	0001	Privát	1754	064214694**	8 Ft
2011 március 11 09:17	0007	Privát	1754	063083627**	16 Ft
2011 március 11 09:30	0008	Hivatalos	1754	062034687**	16 Ft

25. ábra: Hívás részletező képernyő

Hívás részletező tábla oszlopai:

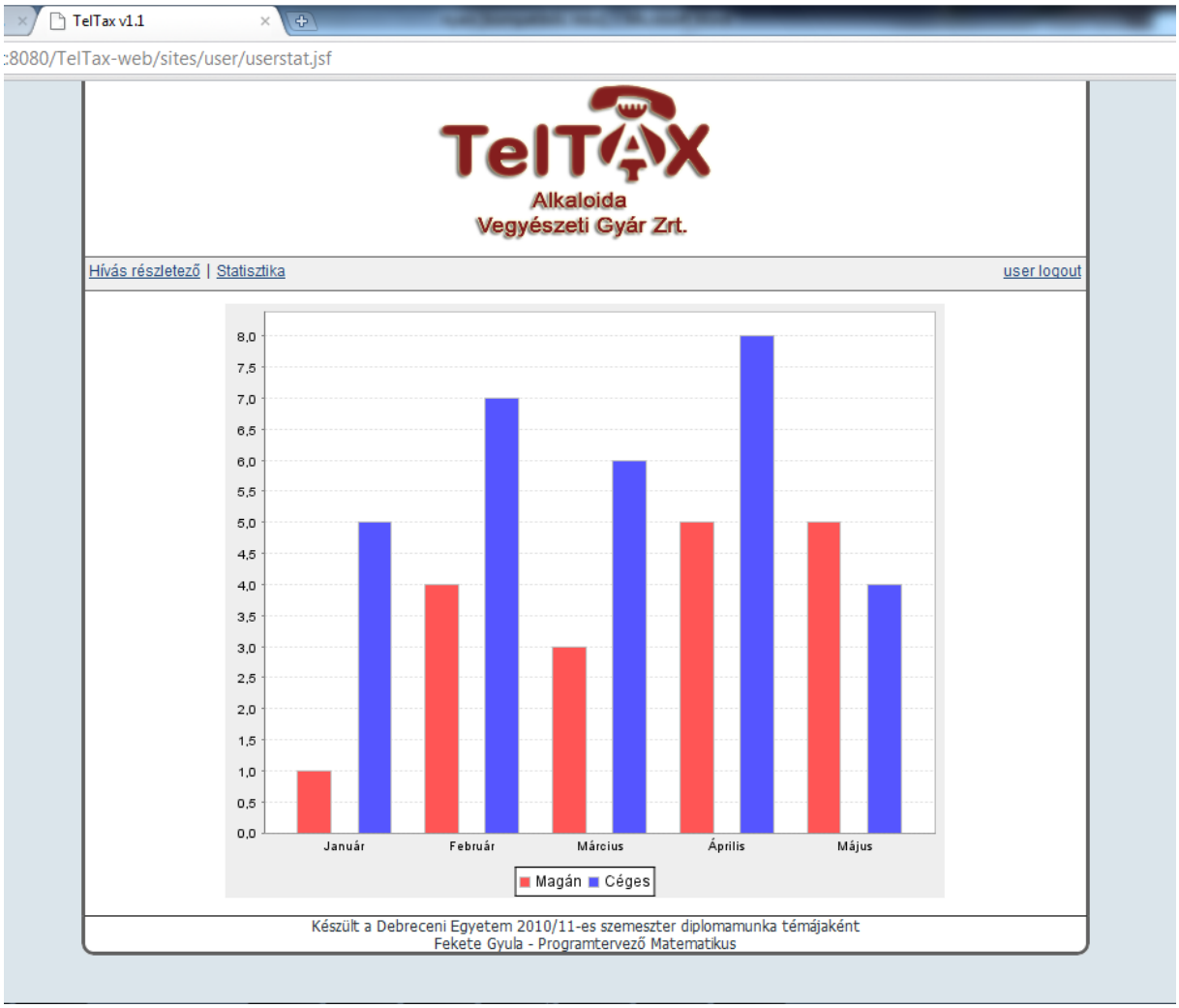
- **Dátum**: A hívás indításának időpontja.
- **Hívásidő**: A hívás időtartama másodperces egységekben.
- **Hívásirány**: Az előhívó alapján kiértékelt érték. (Privát / Hivatalos)
- **Mellék**: A hívás indításának melléke.
- **Hívott fél**: A hívott fél telefonszámának titkos verziója.
- **Hívás díja**: Az aktuális tarifa szerinti hívás költsége.

A tábla összes oszlopa szerint lehetőség van rendezésre.

A táblázatot 4 exportálási módon lehet elmenteni:

-  : XLS fájlformátum
-  : PDF fájlformátum
-  : CSV fájlformátum
-  : XML fájlformátum

A statisztika menüpontban a felhasználó éves telefonhívásainak statisztikáját tekintheti meg:



26. ábra: Statisztika képernyő

A diagram függőleges tengelye a beszélgetési időt (órában), a függőleges tengely az év hónapjait, a kék szín a Hivatalos hívásokat, a piros szín a Privát hívásokat reprezentálja.

8.2.2. Adminisztrátori szintű lehetőségek

Sikeres Adminisztrátori szintű bejelentkezés után a Vállalat mellékeinek karbantartási oldalára jutunk:

27. ábra: Mellékek karbantartása képernyő

Az oldal mezői és akciói:

- **Mellék száma:** Új mellék száma.
- **Hozzáadás:** Mellék listához hozzáadása. Hibaüzenet üres mellékszám esetén!

- **Melléklista:** A már regisztrált mellékek listája, kiválasztásakor kezelhető a mellék adatai.
- **Tulajdonos neve:** A mellék tulajdonosának neve.
- **Üzem neve:** Abban az esetben, ha nincs közvetlen tulajdonosa a melléknek, akkor kitöltendő. Egyébként nem kötelező.
- **Mellék törlése:** Kiválasztott mellék listából való törlése
- **Mentése:** A mellék adatainak mentése.

Adminisztrátori jogkörrel lehetőség van a külsős cégek mellékeinek karbantartására is.

28. ábra: Külsős cégek karbantartása képernyő

Az oldal mezői és akciói:

- **Cég neve:** Új külsős cég neve. (v1.1-ben ez a mező a cégszám)
- **Hozzáadás:** Külsős cég listához hozzáadása. Hibaüzenet üres cégnév esetén!
- **Céglista:** A már regisztrált cégek listája, kiválasztásakor kezelhető a cég mellékei.
- **Cég mellékei:** A céghez tartozó beregisztrált mellékek listája. Egy cégnek több melléke is lehet.
- **Szabad mellékek:** A mellékek karbantartása oldalon hozzáadott új mellékek listája.
- : A cég kijelölt mellékeinek listájának átmozgatása a szabad mellékek listájához.
- : A cég összes mellékének átmozgatása a szabad mellékek listájához.
- : A kijelölt szabad mellékek listájának átmozgatása a cég mellékeinek listájához.
- : Az összes szabad mellék átmozgatása a cég mellékeinek listájához.
- **Mentés:** A mellékek listájának mentése.

A tarifa beállítások menüpontban lehetőség nyílik a tarifák menedzselésére:

29. ábra: Tarifa beállítás képernyő

Az oldal mezői és akciói:

- **Tarifa neve:** Új tarifa neve.
- **Hozzáadás:** Tarifa listához hozzáadása. Hibaüzenet üres cégnév esetén!
- **Tarifalista:** A már regisztrált tarifák, kiválasztásakor kezelhető a tarifa adatai.
- **Tarifa neve:** A kiválasztott tarifa neve.
- **Fővonal:** Fővonalszám, amihez az adott tarifát rendeljük.
- **Időszak kezdete:** A tarifa aktiválásának kezdőidőpontja az adott fővonalon. (óra:perc formátumban)
- **Időszag vége:** A tarifa inaktiválásának időpontja az adott fővonalon. (óra:perc formátumban)
- **Percdíj:** Adott tarifa percdíja.
- **Aktív:** Lehetőség van a tarifa inaktiválásra, törlés nélkül.
- **Tarifa törlése:** Kiválasztott tarifa törlése a listából.
- **Mentés:** Tarifa adatainak mentése.

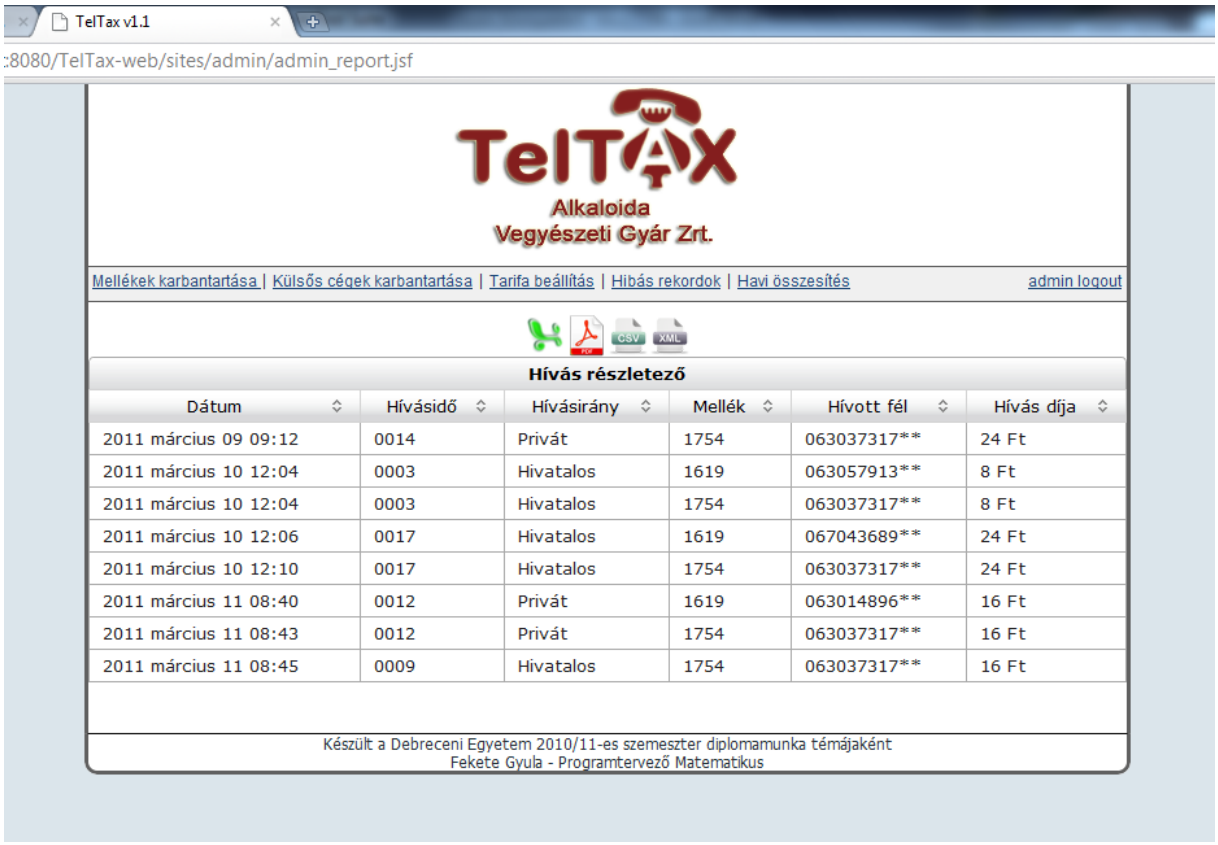
Kivételes esetben előfordulhat, hogy hibás rekordok érkeznek a telefonközponttól. Adminisztrátori jogkörrel lehetőség van ezek kezelésére:

30. ábra: Hibás rekordok képernyő

Hibás rekordokat tartalmazó táblázat oszlopa és akciói:

- **Rekord:** A hibás rekord feldolgozatlanul.
- **:** Hibás rekord törlése a táblából.
- **:** Hibás rekord kézi feldolgozása. Sikeres feldolgozás után a hibás rekord már nem törölhető táblából. Ez esetben automatikusan bekerül a hívásrészletező táblába, mintha gépi feldolgozás történt volna.
- **:** Sikeres kézi feldolgozás után a törlés gomb helyén jelenik meg. Jelöli, hogy melyik azonosítójú hívásrészletet készítették belőle.

Adminisztrátori szerepkörrel a könyvelés számára készíthető a vállalat összes hívásáról riport:



31. ábra: Havi összesítés képernyő

Az oldal mezői és akciói megegyezik a hívás részletező képernyőével, annyi különbséggel, hogy a Vállalat összes kimenő hívását tartalmazza.

9. Összefoglalás

A diplomamunkám végeredménye egy összetett, speciális igényeket kielégítő rendszer az Alkaloida Vegyészeti Gyár Zrt. környezetéhez igazítva. A feladat megoldása közben betekintést kaphattam egy nagyvállalat működésébe, egy személyben megtapasztalhattam a projektmenedzsment és a fejlesztés főbb szerepköreit, mindezen kívül olyan eszközöket használhattam, amik otthoni vagy kisvállalati környezetben nem találhatóak meg. Saját bőrömön tapasztalhattam meg, hogy egy projekt életútján melyek azok a mérföldkövek, amelyeknél újra és újra át kell gondolni a következő lépéseket. Kiemelendő az a tény is, hogy bővült az ismeretem az általam eddig nem ismert Technical Writer szakmában is. A fejlesztés során ügyeltem arra, hogy a lehető legújabb eszközöket/technológiákat használjam, hogy a rendszer életciklusa minél hosszabb időre mutasson.

Azon túl, hogy a diplomamunkám leadása után a rendszer folyamatos supportálásra és bugfixre szorul, kijelenthető az, hogy a rezidens alkalmazás egy jól megkonstruált rendszer, ami a reguláris kifejezések frissítésével bármilyen telefonközponttal használható. A webalkalmazás megfelel az elektronikus hírközlési szolgáltató adatkezelésének különös feltételeiről, az elektronikus hírközlési szolgáltatások adatbiztonságáról, valamint az azonosítókijelzés és hívásátirányítás szabályairól szóló 226/2003-as kormányrendeletnek, illetve az elektronikus hírközlési előfizetői szerződésekre és azok megkötésére vonatkozó részletes szabályairól szóló 16/2003-as IHM rendeletnek.

Rövid távú törekvésem még az alkalmazásokkal kapcsolatban, hogy egy közvetlen bugtracker jelenjen meg a felhasználók és köztem. Mivel nincsenek korlátai a távoli hibajavításnak, így ez a legoptimálisabb megoldás az alkalmazás tökéletesítésére. Hosszabb távon szeretném monitorozni az adatrögzítő működését, és statisztikát készíteni a napszakok forgalmáról. Ezzel le tudnám váltani a folyamatos kapcsolatot a telefonközponttal egy bizonyos intervallum-időszak frissítésre.

Úgy gondolom, hogy a diplomamunkám végeztével, egy komplex alkalmazást adhatok át a helyi üzemnek, amelyet az egyetemi éveim után egy komoly referenciamunkaként tudok felmutatni.

10. Irodalomjegyzék

- Imre Gábor: Szoftverfejlesztés Java EE platformon, Szak Kiadó 2007.
- http://static.springsource.org/spring/docs/3.0.x
- http://hu.wikipedia.org/wiki/Ajax_(programozás)
- http://hu.wikipedia.org/wiki/Trigger_(adatbázisok)
- http://parszab.hu
- http://jtechlog.blogspot.com/
- http://weblabor.hu/cikkek/cssalapjai1

11. Ábrajegyzék

12. Köszönetnyilvánítás

Dr. Juhász István Tanár Úrnak a témavezetői támogatásért, iránymutatásért, és legfőképp az egyetemi évek során átadott tudásért.

Gazdag Editnek a külső konzulensi feladat ellátásáért, és a projecthez való hozzáállásáért.

Krusóczki Zsoltnak a telekommunikációs feladatok megoldásánál nyújtott segítségéért.