SZAKDOLGOZAT

Bencze László

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

Webes információs rendszerek modellezése, fejlesztése
(UML, Java, XML)

Témavezető:
Dr. Adamkó Attila
egyetemi tanársegéd

Készítette:
Bencze László
Programtervező informatikus (Bsc)

Debrecen
2009

Tartalomjegyzék

Bevezetés

Napjainkban az informatika életünk számos területén megjelenik. Nemcsak, megjelent, de már elképzelni sem tudjuk életünket nélküle. Amit eddig, utazással, sorban állással, idegeskedéssel tudtunk csak elintézni azt most már otthonról internet segítségével néhány kattintással el tudjuk intézni. Napjainkban, például ha egy szolgáltató cég piacon szeretne maradni, mindenképpen meg kell, hogy jelenjen az Interneten. A megjelenésnek viszont ára van, még pedig az, hogy új technológiába fektesse pénzét. Az új technológiák megjelenésével új irányvonalak jelentek meg, mindegyik másik irányból közelítette meg a fejlesztést. Ez nem csak a megrendelőt készteti versenyre, de a programozókat és a szoftvert gyártó cégeket is.

Már korábban is érdekeltek ezek a technológiák ezért tanulmányaim során mikor szakdolgozati témaválasztásra került a sor, mindenképpen olyat szerettem volna megjelölni, ahol ezekkel tudok majd foglalkozni. Olyan technológiák megismerésébe szeretném fektetni időmet, melyek napjainkban is helytállnak és a jövőben is helyt fognak. A Debreceni Egyetemen folytatott tanulmányaim alatt megismerkedtem a C és Java nyelvekkel, illetve az Oracle SQL-lel. Ezek közül kiemelném a Java-t, melyet tanulmányaim alatt leginkább használtam és a legjobban meg is kedveltem.

A dolgozat megírásának további pozitívuma lehet számomra, hogy nagyobb rálátásom lesz az alkalmazás-modellezésre, illetve a fejlesztésre, így bízom benne, hogy ezen ismeretek birtokában nagyobb lehetőségem lesz e területen elhelyezkedni.

A téma elméleti kidolgozása mellett szeretnék elkészíteni egy rendszert is, ahol bemutathatom, hogy nem csak elméleti síkon működik, hanem gyakorlatban is a tárgyalt eszközök. Az elméleti kidolgozás során nem referencia jellegűen fogom leírni mi, hogyan működik, hanem gyakorlatiasan, és ahol lehet példa kódokat is beillesztek lehetőleg a saját rendszeremből.

A kifejlesztésre kerülő rendszerem egy édességbolt nyilvántartása, illetve vásárlói igények kielégítése. Ezért a rendszert „CandyShop” névre kereszteltem. A dolgozatom egy rendszermodellezést és fejlesztést mutat be, ezért a következőképpen építettem fel.

Egy Web alapú információs rendszer modellezéséről és fejlesztéséről szól ezért úgy gondoltam, hogy először egy kis áttekintést adnék a webes világról, vagyis maga az Internet hogyan alakult ki majd az Internet adta lehetőségeket kihasználva a szerver- kliens architektúrát tekintem át. Majd definiálom, hogy mit is jelent az információs rendszer fogalma. Miután tudjuk, miről lesz szó, magát a modellezést és a hozzátartozó eszközöket ismertetem. Az elkészült modellt felhasználva a fejlesztéssel fogok foglalkozni, ezen belül először a kódírás során felhasznált eszközöket mutatom be, majd áttérek arra, hogy hogyan is lehet ezeket az eszközöket hatékonyan felhasználni a tovább gyors fejlesztés érdekében.

A modellezés és fejlesztés folyamán a NetBeans 6.5-t fogom használni, mely feltelepítve már minden olyan eszközt tartalmaz, amire a dolgozat elkészítése során szükségem lesz.

1. World Wide Web (WWW) rövid áttekintése

Az Internet története az 1960-as évekre nyúlik vissza. 1969-ben az USA Hadügyminisztériuma telefonvonalon egy kísérleti jellegű, csomagkapcsolt hálózatot hozott létre, melyet ARPANET-nek (Advanced Research Projects Agency Network) kereszteltek. A rendszerrel szemben lényeges elvárás volt, hogy a hálózatban a kommunikáció zavartalan működése akkor is biztosított legyen, ha a hálózat egyes elemei üzemképtelenné válnának. A 70-es évek elején több, ezen az elven működő kisebb elszigetelt magánhálózat is létrejött, de hamarosan felmerült az igény, hogy ezeket a hálózatokat összekapcsolják, és egy egységes adatátviteli módot fejlesszenek ki. Az Amerikai Egyesült Államokban még ennek az évtizednek a végére egyre többen kapcsolódtak e hálózathoz majd később létrejött az első nemzetközi kapcsolat is. A 80-as években százával csatlakoztak ehhez a hálózathoz, egyetemek, főiskolák, kutatóintézetek, valamit az állami hivatalok. Még ebben az évtizedben két részre osztották a hálózatot egy katonai, illetve egy polgári rendeltetésűre. Ezzel a hálózat mérete robbanásszerű ütemben növekedésnek indult. Ma ezt a hálózatot nevezzük Internetnek mely az egész világot behálózza.

Az Interneten számos szolgáltatás érhető el, de ezek között kétségtelenül a World Wide Web (vagy röviden WWW, illetve Web) a legnépszerűbb szolgáltatás, ami nem más, mint egy hipertext- és hipermédia-állományok által alkotott, világméretű információs rendszer. A Web alapelveit Tim Berners-Lee, a CERN(Centre Européen de la Recherche Nucléaire) részecskefizikai kutatóközpont munkatársa dolgozta ki 1989-ben. Eredeti célja a különböző intézményekben világszerte dolgozó kutatók közötti automatizált információ-megosztás volt. Az alapötlet egy globális információs hálózat létrehozása volt a számítógépek, a számítógépes hálózat és a hipertext képességeinek ötvözésével. Az elképzelést az NCSA (National Center for Supercomputing Applications), az amerikai felsőoktatási intézmények számítástechnikai támogatását biztosító szervezet felkarolta, munkatársai Marc Andreessen vezetésével 1992-ben megalkották a Mosaic nevű Webböngésző programot, amelyet hamarosan követtek az egyéb ma közismert böngészők (Netscape Navigator, Microsoft Explorer, HotJava, Opera, stb.).

A hypertext hálózati hivatkozásokat tartalmazó dokumentum; a kapcsolattal („link”-kel) ellátott szövegrészre kattintva gyakran másik számítógépeken, másik országban tárolt dokumentumhoz jutunk el. A hipermedia képekkel, hangokkal, mozgóképekkel és/vagy szövegen belül indítható programokkal ellátott hypertext. A hypertextek létrehozására fejlesztették ki a HTML-t (HyperText Markup Language), ami egy leíró nyelv és képes hyperlinkek definiálására, és egyúttal a létrehozott dokumentum kinézetét is tudja szabályozni. A HTML az SGML (Standardized General Markup Language, szabványosított általános jelölőnyelv) egy leegyszerűsített részhalmazaként jött létre, illetve tartalmazza azt a néhány olyan kiegészítő elemet, amelyek segítségével az egyes HTML oldalak képesek egymásra hivatkozni, a már említett hiperlinkek segítségével, valamint képes nem HTML-adatforrásokat, például képállományokat, szkripteket és stíluslapokat is használni. HTML sikerességét mi sem bizonyítja, hogy megalkotása óta számos változat jelent meg. Jelenlegi érvényes változta a 4.01 amelyet 1999-ben adtak ki. Ez az utolsó változat de már elérhető az 5-ös változat is. 2000-ben adták ki XHTML 1.0 specifikációját, amely a HTML átdolgozása volt érvényes XML dokumentumra.

2. Kliens-szerver architektúra

Internet megjelenésekor triviálisan minden központosított volt, azaz volt egy központi gép és ennek az erőforrásait érték el valamilyen külső terminál segítségével. Terminálok egészen minimális lokális intelligenciával rendelkeztek. Mikor elkezdett terjedni a hálózat a terminálok szerepét átvették személyszámítógépek. Kialakult egy erőforrás megosztó modell un. elosztott rendszer architektúra. Az architektúrának két megvalósítása létezik egyik az osztott objektum architektúra, másik a kliens szerver architektúra.

Ezt a megoldást szokás két rétegű modellnek is nevezni, mert áll egy kliensből, aki szolgáltatásokat vesz igénybe és áll egy szerverből, ami szolgáltatást biztosít. Számunkra most a rajtuk futó folyamatok lesznek érdekesek. Az architektúrán belül szükség van:

- egy adatelérési rétegre, ami közvetlenül éri el az adatbázist,
- egy alkalmazás-feldolgozó rétegre, ami későbbiekben üzleti logika lesz,
- illetve a prezentációs rétegre melynek feladata a megjelenítés.

A folyamatok elhelyezése szerint két megközelítés létezik: mindkét esetben a kliens felelős a megjelenésért, a szerver pedig az adatkezelésért. Ha az alkalmazás-feldolgozó réteg a szerveren található, akkor vékony kliens modellről, ha a klienst oldalra helyezzük át, akkor pedig vastag kliens modellről beszélünk.

Vékony kliens esetén nagy a hálózati forgalom, mert az adatokat küldeni kell a szerverhez és az, ott kerül feldolgozásra. Ha több kliens csatlakozik egyszerre, akkor a szerver terheltség nagymértékben megnő. A vastag kliens esetén a szerver terheltség csökken, mert már feldolgozott adatokat küldünk el. Vékony kliens akkor alkalmazható hatékonyan, ha egy rendszert megosztottá szeretnénk tenni, de ez további bővíthetőségi gondokat hordoz magával, mert az üzleti logika a szerveren, az oldalon van kódolva.

Vastag kliens esetén jobban skálázható a rendszer, de ha sok a kliens folyamat, akkor az menedzselési problémákat vethet fel. A két megoldás között van egy olyan lehetőség, hogy szerver mobilkódokat tartalmazhat és ezek a kliens oldalra letölthetőek és futtathatóak. Java esetén ilyen mobilkód az Applet vagy az Internet Explorer esetén az ActiveX.

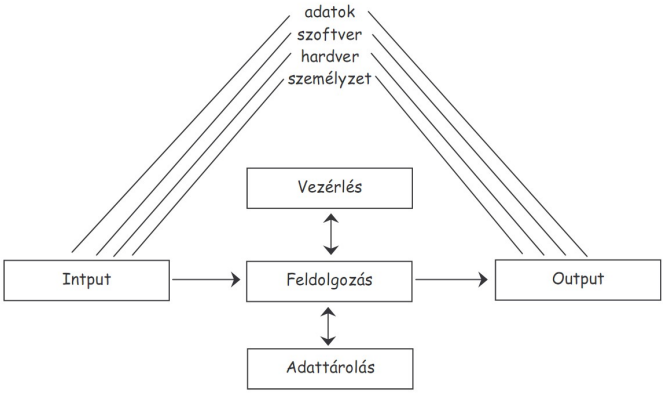
Ezek általában két gépen futnak, de lehetnek szélsőséges esetek, mikor minden folyamat egy gépen fut, de lehet olyan is mikor mind a három kliens szerver folyamat külön gépen fut. Mikor mind három folyamat külön gépet fut akkor három rétegű architektúráról, beszélünk. Ezen elven felépülő rendszerek jobban skálázhatóak, mint a két rétegű rendszerek. A feladatok sokkal optimálisabban eloszthatóak.

3. Információs rendszerek

Adatnak nevezünk minden olyan ismeretet, mely előzőleg már rögzítésre került. Adat lehet egy szám vagy egy karakter sorozat is.
Az információ olyan jelsorozatok által hordozott hír, mely egy rendszer számára új ismeretet hordoz.

Az információs rendszerek célja, hogy a bementi adatokat információvá alakítsák át, és ezzel az információval segítsék a döntéshozók döntéshozatalát Az információs rendszerek nem feltétlenül számítógépes rendszerek, beszélhetünk manuális adattárolásról és adatfeldolgozásról is, bár ezek jelentősége manapság egyre kisebb. Éppen ezért a továbbiakban az információs rendszer alatt, számítógépes rendszert kell érteni.

Az információs rendszer olyan formalizált számítógépes rendszer, mely a különböző forrásokból adatokat gyűjt, azokat feldolgozza, tárolja és az outputjával információt szolgáltat a döntéshozatalhoz.



1. ábra: Információs rendszerek felépítése

Minden információs rendszerben megtalálhatóak az 1. ábrán feltüntetett folyamatok, azaz input, feldolgozás, tárolás, vezérlés és output. A rendszer működéséhez ezeken kívül erőforrásokra – hardver, szoftver személyzetre és adatokra van szükség. Vizsgáljuk meg ezen fogalmakat részletesebben, hogy mit is jelentenek a rendszer szempontjából.

Input: Az input jelenti az adatok bevitelét a rendszerbe és előkészítését a feldolgozásra. Ez a folyamat magában foglalja a tényleges adatfelvitelen kívül, az adatok osztályozását és a beviteli ellenőrzéseket.

Feldolgozás: A feldolgozási folyamatok alakítják át az inputon érkező adatokat a felhasználók számára értékes információkra. A feldolgozás olyan folyamatokból áll, mint az adatok rendezése, összehasonlítása, összegzése, elemzése és karbantartása.

Tárolás: A tárolás során az adatok valamilyen rendezett formában elraktározzuk a későbbi felhasználás céljából. A számítógépes tároláskor általában rekordokba, fájlokba, adatbázisokba szervezzük adatainkat.

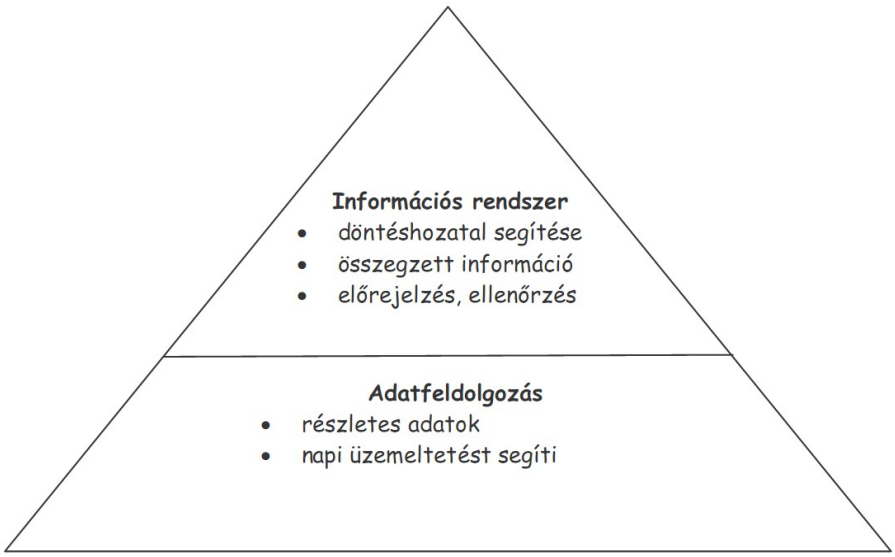
Vezérlés: Egy jó információs rendszernek folyamatosan figyelnie kell a rendszer kimenetelét, és ha annak működése nem a célnak megfelelő, akkor a vezérlő folyamatoknak be kell avatkozniuk.

Output: Az output biztosítja az adatok kivitelét a rendszerből, vagyis az információ közvetítését a rendszerből a felhasználók felé. Fontos, hogy a közölt információ könnyen értelmezhető legyen a felhasználó számára.

3.1. Információs rendszerek rövid történeti áttekintése

Az első számítógépek megjelenése után hamar világossá vált, hogy azokat az üzleti életben is nagy hatékonysággal lehet használni. Első alkalmazások 1950-es években jelentek meg, de a 60-as évekig a számítógépeket kizárólag adatok gyűjtésére tárolására használták. Fókusz az adatokon volt, a mindennapi üzletmenetben használták őket nyilvántartásra, számlázásra, könyvelésre. Ezeket a rendszereket elektronikus adatfeldolgozó rendszereknek hívták (EDP). Ezen rendszerek sokat változtak, mára már ezeket a funkciókat a tranzakció-feldolgozó rendszerek látják el (TPS).

A világ és az információs társadalom (IT) rohamosan fejlődött és felmerült az igény, hogy ha már a dolgozóknál ilyen jól hasznosítható, akkor a felsőbb vezetőknél is lehetne hasznosítani. A vezetőket a döntéshozatalban kellett segíteni, azaz információ feldolgozására és összegzésére volt szükség. Így alakult ki az 1960-as évek közepén a vezetői információs rendszer (MIS). A MIS képes volt előre definiált jelentéseket készíteni, melyek alkalmasak voltak a vezetői döntéshozatalok támogatására. Az adatfeldolgozó rendszerekre épülve jelentek meg az első „igazi" információs rendszerek.



2. ábra: Információs rendszer piramis

A MIS által előre elkészített jelentések egy idő után már nem tudták a menedzserek minden igényeit kielégíteni. Ráadásul ezek a rendszerek egy idő után megnőttek, karbantartásuk nehézkessé vált. Az 1970-es években jelentek meg azok a rendszerek, melyek egy adott problémára koncentrálódtak és lehetőséget adtak az interaktivitásra. Ezek voltak a döntéstámogató rendszerek (DDS). DDS rendszerek továbbfejlesztésének köszönhetően jöttek létre a csoportos döntéstámogató rendszerek (GDSS), melyek fő célja a csoportos döntéstámogatás elősegítése.

A DDS-hez hasonlóan az 1970-es években jelentek meg az irodaautomatizálási rendszerek (OAS). OAS célja, hogy forradalmasítsa az irodai munkát, új lehetőségként bevezette az e-mail küldést, a táblázat kezelő programok használatát és még sok más hasznos dolgot.

MIS és DSS rendszerek nagyon hasznosak voltak, de a legfelső vezetői rétegek igényeit nem elégítették ki. Ezek a menedzserek mindennapi munkájuk során nem használták az információs rendszereket. Másrészt az 1980-as években rendkívüli változásokon ment át az IT világa. Terjedni kezdtek a hálózatok, megjelentek a mikroszámítógépek, irodai programcsomagok és a felhasználók közelebb kerültek a számítógépekhez.
Ezek után mindenképpen kellett egy olyan rendszer, ami a vezetők igényeit is kielégíti. Ezért fejlesztették ki a vezetői információs rendszereket (EIS), melyek összegzett könnyen érthető, grafikus, táblázatos információt szolgáltatnak a vezetők számára.

Nem szorosan ide kötődik, de fontos vonalat képvisel a fejlődésben a mesterséges intelligencia és annak üzleti informatikai felhasználása, a szakértői rendszer (ES), mely egy szűkebb területen képes szakértői tanácsokat adni és döntéseket hozni.

Az 1990-es évek szlogenje egyértelműen az információs rendszerek integrációja és stratégiai szerepe. Az információs rendszer már nem csupán egy plusz szolgáltatás, amely segítséget nyújt az üzleti életben, hanem olyan elengedhetetlen szükséglet, amely nélkül manapság nem lehet érvényesülni.

3.2. Web alapú információs rendszerek

A fentebb tárgyalt információs rendszerek kicsi, de fontos csoportját képzik a Web alapú Információs Rendszerek (Web Information System – WIS). „*Egy WIS olyan információs rendszer, amely a weben keresztül biztosítja az interaktív szolgáltatásait és a komplex adatok elérhetőségét" Ezen rendszerek „az információs rendszerek egy olyan meghatározó alosztályát alkotják, amelyek a szolgáltatásaik révén tipikusan az online információ elérést és a napi feladatok ellátását támogatják igen nagyszámú (ezres vagy milliós) felhasználói körnek, akik távoli helyeken vannak*"

A definíció alapján a web alapú információs rendszereket a következő kategóriákba sorolhatjuk: információs, interaktív, kereskedelmi. A jellemzésére, pedig a következők igazak:

- Információs rendszer,
- elosztott alkalmazás,
- a kliens/szerver alkalmazások speciális esete, melyekben a funkcionalitás a szerver oldalra kerül,
- weben, mint alkalmazás infrastruktúrán alapszik,
- felhasználói kommunikációk webes interfészeken keresztül történnek.

Amiket vizsgálunk Web alapú Információs Rendszer egy számítógép által támogatott rendszer, amely a világháló lehetőségeit használja ki. Ezen rendszerek a felhasználóval folytatott kommunikáció és információ jellege alapján a következők lehetnek:

- Információ szolgáltató
- Hirdetési
- Információs rendszer
- Közösségi

A következő táblázat ezen, rendszerek egy lehetséges osztályozását mutatja be az információ jellegének és kommunikáció irányának függvényében:

	Aszimmetrikus kommunikáció	Szimmetrikus kommunikáció
Objektív információ	Információ szolgáltató	Információs rendszer
Befolyásoló információ	Hirdetési	Közösségi

1. táblázat: A WIS perspektívái

Az információszolgáltatók esetében a rendszertől a felhasználók felé történő egyirányú kommunikációról beszélhetünk. Tipikus példája e rendszereknek a hírportálok. A hirdetési rendszerek ezzel szemben tekinthetők az üzleti kommunikáció csatorjának is, céljuk a promóciós üzenetek eljuttatása a lehetséges vásárlók felé valamilyen weboldalon keresztül. Közösségi rendszerek esetén egy virtuális közösséget tudunk kialakítani valamilyen fórum segítségével. A negyedig perspektíva azonban úgy is ismert, mint adatorientált webalkalmazások.

Tipikusan ebbe a kategóriába olyan nyilvántartó rendszerek tartoznak, mint például online repülőjegy foglaló rendszer, vagy könyvtári nyilvántartási rendszer, vagy a mi rendszerünket is ide sorolhatjuk. E rendszerek középpontjában az adatstruktúra, illetve a felhasználó és a rendszer közti információáramlás áll. Ha az összetett üzleti folyamatok segítségével támogatják a felhasználó munkáját, akkor ez esetben az információáramlás kétirányú kommunikációvá válik.

Ilyen rendszerek kidolgozás kimondottan összetett feladat. A fejlesztők annak érdekékben, hogy időt takarítsanak meg, többnyire mellőzik a szisztematikus módszerek alkalmazását. Ennek eredményeképpen helytelen tervek és egyedi, nem újrafelhasználható modellezési praktikák alakulnak ki. Ezen túlmenően a folyamatosan módosuló igényekre adott gyors reagálást szinte lehetetlen időben megvalósítani, mert az új igények megjelenése a rendszer egyes részeinek vagy akár magának a teljes rendszernek az újratervezését is megkövetelheti. Az újraimplementálás azonban nagyon időigényes munka, amelynek fő oka, hogy a modell és az elkészült implementáció között hiányzik a kapcsolat. A hagyományos szoftverfejlesztési módszerek alkalmazása esetében is gyakori, hogy miután a modell elkészült és első körben implementálásra került, a további lépésekben már nem használják, nem frissítik, egyszerűen „eldobják” ezeket. Így nem alakul ki kapcsolat a modell és a kód között, ezért a modellben történő változás nem látszik a kódban, és ez fordítva is igaz, a kódban történő változások nem kerülnek átvezetésre a modellben. Emiatt a modell nem tudja ellátni azt a feladatát, amire elsődleges céllal létrejött, nevezetesen, hogy a rendszerről meghatározó információkat szolgáltasson.
Következőkben majd látunk olyan módszert, hogy hogyan lehet a modellből konkrét implementációt készíteni, illetve a változásokat a modellre is visszavezetni.

4. Modellezés

Szoftverfejlesztési életciklusban az egyik legfontosabb és leghosszabb szakasz a modellezés. Korában már volt ráutalás, ha nem megfelelően modellezünk egy rendszert, akkor annak későbbi fejlesztése nehézkesé válhat. A megfelelő modellek kialakításában és azok automatizálásban segít a modellvezérelt architektúra.

4.1. Modell-vezérelt architektúra

Az OMG (Object Management Group) által kidolgozott modellvezérelt architektúra (Model Driven Architecture – MDA). MDA egy olyan keretrendszer, amelynek segítségével alkalmazásainkat platform független formában tudjuk megadni. Az MDA-ban a termékek formális modellként jelennek meg az adott rendszert különböző szemszögékből mutatják be. Az MDA az UML nyelvet, mint szabványos modellező nyelvet használja annak érdekében, hogy a résztvevők között egy mindenki által ismert felületet nyújtson.

A modell vezérelt fejlesztés során először a platform független (Platform Independent Model – PIM)) modellek készülnek el. Ezek a modellek elrejtik, az implementációs részleteket, emiatt magas fokú absztrakcióval rendelkeznek. A PIM modellek segítségével tudjuk bemutatni, hogy hogyan is fog majd kinézni a rendszer. Miután elkészültek a PIM modellek, második lépésben a PIM modelleket, transzformáljuk egy vagy több modellé. Ezeket a modelleket nevezzük platform specifikus modelleknek (Platform Specific Model – PSM). A PSM modellek viszont már szorosan kapcsolódnak valamely technológiához, ezért ezekből már viszonylag könnyen tudunk kódot készíteni.

4.2.UML

Az MDA módszertannál már elő jött a diagram tervezés, ezért olyan eszközre van szükségünk, amely segítségével diagramokat tudunk készíteni. Az OMG UML-t mint jelölő nyelvet ajánlja erre a célra.

Az UML (Unified Modeling Language) egy grafikus modellező eszköz, azaz a tervezett rendszer alkalmazás modell szintű megjelenését diagramok segítségével tudjuk megvalósítani, mely segítségével csupán az alkalmazás vázlatát készíthetjük el, a teljes implementációt nem. Az UML széles körű, integrált eszközöket nyújt, amelyek segítségével a szoftver bizonyos fokú definiáltsága érhető el. Az UML olyan eszközrendszert nyújt, amelynek segítségével

- az alkalmazásfejlesztők specifikálhatják, érvényre juttathatják a kifejlesztendő rendszerrel kapcsolatos követelményeket, elvárásokat
- a rendszerkülönböző szemléletű és megközelítésű modelljeit konstruálhatják meg (viselkedés, szolgáltatás)
- szoftver-életciklus szakaszaihoz szükséges dokumentációk elkészíthetők

4.3. UML felépítése

Az UML felépítése egy metamodellezési megközelítésen alapszik. A metamodellezés azt jelenti, hogy minden konkrét rendszer egy modell példánya, minden modell egy metamodell példánya és tovább folytatva minden metamodell egy meta-metamodell példánya (MOF Meta Object Facility). A metamodellezési konstrukció segítségével a nyelv tömör maradhat. Az UML különböző diagramjai a modellezett rendszert különböző aspektusokból szemléltetik. Így előfordulhat, hogy az egyes diagramok között átfedések lehetnek, azaz megtörténhet, hogy a rendszer ugyanazon pontját modellezzük rajtuk, de a diagramok típusától függően ugyanazt a dolgot más megvilágításban láthatjuk. Alapvetően az UML két nagy csoportba osztja a diagramokat:

- Struktúrára vonatkozó diagramok: A strukturális diagramok a rendszer statikus jellemzőit, a statikus elemeket és ezek kapcsolatait, ezáltal a rendszer belső struktúráját modellezik. A legfontosabb diagramtípus az osztálydiagram. Minden más struktúradiagram az osztálydiagram egy speciális változatának tekinthető.

- Viselkedésre vonatkozó diagramok: A viselkedési diagramokkal a strukturális diagramokon modellezett elemek dinamikáját, a rendszer működése közben megvalósított viselkedését modellezhetjük. Ezek a diagramok írják le, hogy az erőforrások hogyan hajtják végre a feladataikat, hogyan működnek együtt a rendszer céljainak megvalósítása közben. Ezek a diagramok lehetnek: használati eset diagram, állapotautomata diagram, tevékenység diagram (aktivitás-diagram), interakció diagramok (szekvencia, kommunikációs, időzítési).

Terjedelmi okok miatt a viselkedés diagramok közül csak a használati eset - aktivitási diagramok alapjait nézem át, majd részletesebben tekinteném át a számunkra legfontosabb struktúra diagramot, az osztály diagramot.

4.3.1. Használati eset diagram – Use Case

Használati eset diagramok, azokat a követelményeket jelenítik meg az absztrakció különböző szintjein, amelyeket a felhasználók elvárnak a leendő rendszertől. A diagramok a háttérben zajló folyamatokról nyújt, viszont a tényleges végrehajtás mikéntjéről nem nyújt információt. Különböző nézőpontokból fejezhetjük ki a rendszer funkcionalitásait. Az, hogy milyen nézőpontból mutatjuk be a rendszert, illetve milyen részletességgel, az attól függ, hogy kinek szánjuk a diagramokat. Ha a megrendelőt szeretnénk informálni, akkor sokkal nagyobb egységekben, közérthető módon kell elkészíteni. Ha a fejlesztőknek, akkor sokkal bonyolultabb és szakmához kötődő is lehet.
A Use Case diagramok a következő építőelemekből állhatnak:

- Szereplők vagy actorok: a rendszeren kívül eső, de a rendszerhez szorosan kötődő elemek. Az actorok nem konkrét személyeket jelölnek, sokkal inkább helyesebb a típus megnevezés. Ezen gondolkodás mellett actor lehet egy felhasználó, adminisztrátor vagy egyéb más alkalmazás is. A diagramon actorok jele a pálcika ember.

- Használati esetek: Use Case-ek azok a funkcionalitások, amiket a fentebb említett actorok igénybe akarnak majd venni. A use case jele egy ellipszis, beleírva a használati eset funkcióját.

- Kapcsolatok: a szereplők és a használati esetek közti létrejött viszony. Jelölése nyilakkal vagy vonalakkal történik.

Használati eset diagram készítése tehát nem más, mint az egyes szereplők és használati esetek felderítése, megtalálása, majd a közöttük lévő kapcsolatok meghatározása.

Actor és use case között csak egy fajta kapcsolt típus lehet, ez pedig az asszociáció. Jele egy sima nyíl, vagy ha nem kívánjuk az irányt megadni, akkor egy sima vonal. Ez annyit jelent, hogy az említett szereplő és használati eset kapcsolatban áll egymással. Ha több helyen azonos funkcionalitást kívánunk használni, akkor nem kell minden egyes alkalommal ugyanazt a use case-t elkészíteni. Szaggatott nyíllal és egy <<include>> sztereotípiával jelölhetjük, hogy az adott use case használ egy másikat. Az <<extends>> sztereotípiával megjelölt kapcsolat annyiban különbözik az előzőtől, hogy az átfogóbb használati esetnek módja van felhasználni a kisebb egységet, de ez nem kötelező a számára. Az elkészített rendszer raktáros szempontjából bemutatató use case megtalálható a függelékeken belül (6. ábra).

4.3.2. Aktivitás diagram

Az aktivitás diagram segítségével a rendszerünk dinamikus jellegét tudjuk leírni, ábrázolni. Aktivitásnak nevezzük a rendszer használata során végrehajtott tevékenységeket. Az adott tevékenységet egy ívelt oldalú téglalappal jelöljük. Az egyes aktivitások közötti időbeli függést az ún. átmenetekkel jelezzük. Az átmeneteket egy sima nyíllal adjuk meg az aktivitások között és azt jelenti, hogy az egyik befejeződött és kezdődhet a következő. Egy aktivitási diagram mindig egy folyamatot ír le, melynek van egy kezdőállapota, amelyet egy körlappal jelölünk, és van egy vagy több végállapota, amit egy körben lévő körlappal adhatunk meg.

Arra is van lehetőség, hogy bizonyos tevékenységek végrehajtását egy előfeltétel teljesüléséhez kössünk. A feltételeket őrszemeknek nevezzük, és aktivitások közötti nyíl mellett szögletes zárójelben jelezzük. Ha a végrehajtás során a feltétel olyan, hogy több irányba is haladhatunk a feltétel teljesülésének függvényében, akkor ezt a feltételt egy rombusszal jelöljük, majd a rombuszból indítjuk az egyes aktivitásokhoz vezető nyilakat. Fontos, hogy az innen kiinduló nyilak végén olyan aktivitások szerepeljenek, amelyek kölcsönösen kizárják egymást. Az elkészített rendszer regisztrációt bemutató aktivitása megtalálható a függelékeken belül (4. ábra).

4.3.3. Osztály diagram

Korábban utaltam arra, hogy fontos szerepe van az osztálydiagramoknak. Miért is tulajdonítunk ennyire fontos szerepet nekik? Mert az osztálydiagramok segítségével tudjuk leírni a rendszer statikus belső szerkezetét. A jól kialakított szerkezet, pedig egy átlátható és könnyen fejleszthető rendszert eredményez.

Az osztály, mint fogalom az objektumorientáltság egyik fontos része. Az OO fő feladata a valós világ objektumainak modellezése, azaz való világban előforduló szakmai „objektumokat” leképezz technikai objektumokra. Az osztálydiagramokat a modellek gerincének tekintik.
A szoftver életciklus fázisai alapján megkülönböztetünk különböző osztálydiagramokat:

- Elemzés: elemzés szinten az osztályok nem mások, mint a szakterület fogalmai. Itt a hangsúly azon, hogy megértsük az adott problémát.

- Tervezés: megjelennek az osztályokban a technikai megvalósításhoz szükséges elemek. Pl. láthatóság vagy visszatérési érték típus.

- Megvalósítás: egy terv konkrét osztályban való megjelenése, ami nem más, mint az osztályok egy implementációs nyelven való megjelenése.

Ezen, három osztálytípus alkalmazásával tudunk eljutni egy kezdeti problémától egy konkrét megvalósításhoz. Ezek a fázisok egymásra történő transzformálásában nyújt segítséget az MDA.

A modellvezérelt fejlesztés szerint az első lépés mindig egy PIM elkészítése, ami nem más, mint az elemzési diagram. Második lépésben az elemzési diagramot átalakítjuk tervezési diagramra, ami a PSM-nek felel meg. Átalakítás során bekerülnek olyan plusz eszközök, amely már valamilyen platformra jellemzőek. Ha a PSM modell már tartalmaz annyi részletet, hogy le tudjuk generálni az implementációt, akkor elkészült a probléma platform specifikus modellje. Az elkészült PSM megtalálható a függelékeken belül (10. ábra). Ha azonban PSM nem tartalmaz annyi információt, akkor manuálisan kell beavatkozni, hogy elkészíthessük az implementációt. Az elkészült rendszerem implementációját nekem is manuálisan kellett kiegészítenem, melynek szükséglete dolgozatom folyamán világossá válik.

Érezhető, hogy az MDA mennyire megkönnyíti a modellezést, hiszen az ábrázolt PIM-ből transzformációk segítségével gyorsan el lehet jutni az implementációhoz. Ezzel a módszerrel nem csak egy új rendszer modelljét lehet gyorsan elkészíteni, hanem esetleges technológiaváltozáskor vagy evolúciókor biztosítható, hogy a PIM modelleken ne kelljen változtatni, mert már eleve platform függetlenül jelenítettük meg. Mindössze annyit kell tenni, hogy egy új platform specifikus modellt készítünk, amelyben leírjuk a platform független részek megjelenési formáit az új platformon, illetve új technológiákban. A platform specifikus modellek könnyebb kialakításának érdekében célszerű egy új platform specifikus metamodellt meghatározni, amely az új platformot képes absztrakt módon leírni, ez által megkönnyítjük a platform specifikus modellek létrehozását. Természetesen, ehhez szükséges megadni azon transzformációs szabályokat is, amelyek PIM metamodell elemeket leképzik a PSM metamodell elemeire. Ezen szabályok egy PIM modellre történő alkalmazásával előállíthatjuk egy új platform PSM modelljét.

4.4. Tartomány alapú tervezés

A szoftverfejlesztés általában a valós világban létező folyamatok automatizálására vagy valamilyen üzleti feladatra nyújtanak megoldást. Legyen bármiről is szó, már a fejlesztés elejétől kezdve figyelembe kell venni, hogy hova készül és milyen kapcsolata lesz az adott szakterülettel. Az adott szakterületre való fejlesztésben nyújt segítséget a tartomány alapú tervezés.

A tartomány alapú tervezés (Domain Driven Design) mindenképpen egy olyan szoftvermodell elkészítését és karbantartását igényli, amely a szakterületről kialakított átfogó ismereteinket tükrözi. Már az elején fontos átlátni annak jelentőségét, hogy a szoftver az adott területhez nagyon szorosan kapcsolódik. A tartomány alapú tervezés segít a probléma megértésében, támogatja egy olyan modell elkészítését, amely a szoftverrel kapcsolatos alapvető koncepciókat tartalmazza. A tartományra vonatkozó ismereteink kialakítása során lehetővé válik a fontos információk kinyerése és azok általánosítása. A tartomány alapú tervezés magának a szakterületnek a meghatározásához a következő építőelemeket használja: entitás, értékobjektum, szolgáltatás és modul. A szolgáltatások biztosítják a felületet a tevékenységek végrehajtásához és céljuk a tartomány funkcionalitásának biztosítása. A modulok segítségével az összetartozó részeket foghatjuk egybe.

4.5. Tartomány specifikus nyelvek

Egy adott problématerület leírására dolgozták ki a tartomány specifikus nyelveket (Domain Specific Languages – DSL). Számos példát ismerünk tartomány specifikus nyelvekre a HTML-től kezdve az SQL-en át az UNIX belső programnyelvéig. Ami a tartomány specifikus nyelvek új ötlete, hogy saját projektünkhöz tudjunk saját specifikus nyelvet készíteni.

A tartomány specifikus nyelvek számos alapvető különbséget mutatnak az általános célú nyelvekkel szemben:

- kevésbé átfogóak,
- sokkal nagyobb kifejező erővel rendelkeznek a saját problématerületükön,
- nagyon kevés felesleges elemet tartalmaznak.

A modellvezérelt fejlesztés során számos tartomány specifikus nyelvvel találkozhatunk. Ilyen többek között az OCL (Object Constraint Language), amely a modellek megszorításainak leírására szolgál vagy a QVT (Query-View-Transform), amely a modellek transzformációjához nyújt támogatást. Mindezekkel szemben, a modellek elkészítéséhez használt UML (Unified Modeling Language) nyelv tipikusan az általános célú nyelvek közé tartozik.

5.Fejlesztés során használt eszközök

5.1. Komponens architektúra

Komponens architektúra 90-es évek második felében jelent meg, melynek alapvetése, hogy jól definiált interfészekkel komponenseket fejlesztünk, amelyeknek nem adjuk ki a belső működését. Azaz ismerjük, hogy mit csinál, de azt nem, hogy miként csinálja.
Ennek több előnye is van:

A fejlesztés során meghatározhatjuk, hogy mire is van szükségünk, és hogy az adott komponens hogyan is kommunikál a többivel. Így a fejlesztés jobban szétoszthtató, mindenki a saját komponensét fejleszti, mivel meghatározott interfészek van így a kommunikáció is megfelelő lesz.
Mivel a rendszerünk több apróbb elemből épül fel, így könnyebb a tesztelés és az esetleges hiba felderítése és kijavítása.

A komponens tipikus példa lehet az újrafelhasználhatóságra. Egyes komponenseket más szoftverekben is felhasználhatjuk nem csak abban amiben eredetileg terveztük. Minél többször kerül egy komponens felhasználásra annál jobb lesz, vagyis ha kiderül, hogy hibát vétettünk akkor azt azonnal javíthatjuk, következőnek mikor felhasználjuk akkor már a javított változatot építhetjük be.

A szerveroldali komponensalapúság a Java nyelvben egy rendkívül erős egységként jelenik meg, ez pedig az Enterprise JavaBeans.

5.2. Enterprise JavaBeans

Az Enterprise JavaBean-ek (EJB) olyan szerveroldali komponensek, amelyeket szabványos módon használhatunk fel elosztott rendszerek fejlesztése során. Önállóan telepíthetők egy elosztott, többrétegű környezetbe. Az EJB-k segítségével valósul meg az üzleti logika. Mivel komponensekről beszélhetünk, így a komponens architektúránál felsorolt tulajdonságok az EJB-re is jellemzőek lesznek. A szabványosításnak köszönhetően a

szoftveriparban eléggé elterjedt technológiáról van szó, tehát lényegesen egyszerűbb hozzá fejlesztőket találni, mint valamely különleges módszerhez.
Az EJB specifikáció határozza meg azokat a feltételeket, szabályokat, amelyeket mind a komponenseknek, mind pedig az őket futtató alkalmazásszervereknek be kell tartaniuk. Valamint definiálja az előbb említettek közötti kommunikációt megvalósító interfészeket. Azért van erre szükség, ha az EJB fejlesztő készít egy komponenst, akkor biztos lehessen benne, hogy futni fog az adott alkalmazásszerveren. A SUN az adott alkalmazásszervert egy szigorú teszt sorozatnak veti alá, és ha a szerver megfelel a teszteken, akkor erről igazolást kap, amelyben igazolják számunkra, hogy a szerver maradéktalanul teljesíti az EJB specifikációban foglaltakat. Ezáltal a komponensfejlesztők biztosak lehetnek benne, hogy az általuk fejlesztett komponens csakugyan hordozhatóak lesznek.
EJB komponenseket csak Java nyelven írhatunk, ami egyfajta elkötelezettséget jelent a Java mellett, de nekünk most ez nem jelent semmilyen akadályt.
Egy EJB alkalmazás életciklusának – fejlesztés, telepítés, futtatás - végrehajtásához több szereplőre is szükség van:

- Bean gyártója: A bean gyártója készíti el az üzleti logikát magát tartalmazó komponenseket, az Enterprise Bean-eket.
- Alkalmazás összeállító: Az elkészült komponenseket egy nagyobb telepíthető alkalmazássá állítja össze.
- EJB telepítő: Telepíti az elkészült alkalmazást a futtató környezetbe.
- Rendszerüzemeltető: Felügyeli a telepített alkalmazás futását.
- Konténer és szervergyártó: A konténergyártó adja azt az EJB konténert, amely az Enterprise Bean-ek futási környezetét biztosítja. Ez a konténer része az alkalmazásszervernek és ez biztosítja az EJB-k számára megfelelő szolgáltatásokat.
- Perzisztencia menedzsergyártó: Az alkalmazások számára az üzleti adatok perzisztens tárolásáért felelős eszközt készíti.
- Fejlesztő eszközgyártó: a komponensek egyszerűbb fejleszthetőségét támogató fejlesztői környezetek és eszközök gyártói.

Az EJB példányok egy úgynevezett EJB konténerhez kötődnek. A konténer hozza létre, futtatja és további menedzselési feladatot lát el az EJB-kel. Az implementációtól külön helyezkednek el a konfigurációs állományok, amelyekben az általános szerveroldali szolgáltatásokra, például tranzakció kezeléssel és biztonsággal kapcsolatosan adhatunk meg információkat. Ezek megváltoztatásával az alkalmazás összeállításakor és telepítésekor az általuk meghatározott tulajdonságot testre szabhatjuk. Ez teszi lehetővé, hogy különböző alkalmazásokat forráskód módosítása vagy újrafordítása nélkül állítsunk össze különböző Enterprise Bean-ekből. Ehhez felhasznált konfigurációs állományokat telepítés leírónak (deployment descriptor) nevezzük. Az EJB-k készítése a következő lépésekből áll:

- Elkészítjük a konkrét üzleti logikát tartalmazó komponenseket.

- Megírjuk egy külön fájlban, hogy a komponensnek milyen szolgáltatásokra van szüksége.

- A leíró fájl alapján egy – az alkalmazásszerverhez tartozó eszközzel – legeneráljuk az ún. kérés közvetítő objektumot.

- A kliens nem közvetlenül a bean-nel kommunikál, hanem ezzel a kérés közvetítő objektummal. A kérés közvetítő objektum végrehajtja a kért szolgáltatást és közben delegálja kérést a bean-hez.

Az EJB-k hívása többféle módon történhet:

- A távoli eljáráshívás (Remote Method Invocation), amely az Internet Inter-ORB Protocol-al (IIOP) kiegészülve biztosítja a távoli EJB-k elérését.
- Webes felületű elérés esetén az EJB-kre támaszkodva a szervletek, JSP oldalak stb. végzik a kiszolgálást. Ha ezek közös alkalmazásszerveren futnak, akkor egyszerű Java metódushívásról beszélhetünk.

5.2.1. EJB-k típusai

Az EJB-k típusai nevezetesen a következők:

- Seeson Bean: üzleti folyamatainkat reprezentálhatjuk velük, mint például árszámítás, hitelkártya-kezelés. Ezt úgy tehetjük meg, hogy az egyes használati eseteknek metódusokat feletetünk meg és az összetartozó metódusokat, pedig egy Sesson Bean-hez rendeljük.
- Message-driven bean: aszinkron üzenetküldést tudunk velük megvalósítani.
- Entity Bean: feladatuk a perzisztencia megvalósítása, azaz adatbázissal való kapcsolattartás, illetve a Sesson Bean által használt adatokat reprezentálni.

5.3. Annotációk

Kis kitérőként említem meg a Java nyelv történetének legsikeresebb újítását az annotációkat és hogy mi is okozta az annotációk átütő sikerét.

Az annotáció egy forráskódba illesztett metaadat, amely nem közvetlenül módosítja a program illetve a komponens jelentését, hanem azt befolyásolja, hogy a különböző eszközök és osztálykönyvtárak hogyan is kezeljék azt. A fejlesztők már a Java 5 előtt is elhelyezhettek különféle metaadatokat a forráskódban kommentek formájában. Bizonyos speciális szintaxisnak megfelelő kommentek feldolgozására eszközök készültek, ilyennek tekinthető a Javadoc és az XDoclet. A Java 5 az annotációk bevezetésével egységes szintaxist határoz meg tetszőleges metaadatok beszúrásához és definiálásához, amelyek így nem szorulnak a kommentek közé. Az annotációk alkalmazásának általános szintaxisa a következő:

@Annotációneve(paraméter1=érték1, paraméter2=érték2…)

Ha csak egy paramétere van, akkor megengedett:

@Annotációneve(érték)

Ha paraméter tömb típus is lehet, ekkor értéke a szokásos tömbszintaxist követi:

@Annotációneve(paraméter1={elem1 elem2,…} paraméter2=érték2…)

Ahhoz, hogy egy annotációt helyezhessek a forráskódban, az annotáció típusát definiálni kell egy speciális interfészben. Ebben adom meg, hogy milyen paramétereket fogad el, milyen alapértelmezett értékekkel. Képzeljünk el, hogy egy @TODO nevű annotáció akarunk bevezetni, amelyben egy szöveget és egy fix készletből választható fontossági-szintet adhatunk meg, például:

```
@TODO(text=''Implementálni kell'' severity=TODO.CRITICAL)
public void myMethod{
      System.out.println(''Not implemented'');
 }
```

1. példa: Annotáció használata

Ahhoz, hogy ez a kód lefordúljon, a következő módon kell definiálni a TODO annotációt:

```
public @interfaceTODO{
   public enum severity{CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION};
      Severity severity() default Severity.IMPORTANT;
      String text();
}
```

2. példa: Annotáció definiálása

Megfigyelhető az, hogy az @interface nyelvi elemet, és a paraméterek neveit és típusát metódusokhoz hasonló szintaxissal kell leírni! A felsorolt típus (enum) definiálására szintén a Java 5 ad lehetőséget (ez független annotációtól). Egy paraméter alapértelmezett értéke a default kulcsszóval adható meg. Ha egy annotáció csak egyetlen nevet vár, annak kötelezően a value() nevet kell adni. A paraméterek típusa primitív típus String, Class Annotation, Enum illetve ezek tömbjei. Egy annotáció használatához importálni kell az őt definiáló @interface-t, így a TODO példa helyesen:

```
import annotacio.TODO;

 @TODO(text=''Implementálni kell'' severity=TODO.CRITICAL)
public void myMethod{
      System.out.println(''Not implemented'');
 }
```

3. példa: Annotáció helyes használata

Az annotáció típusokhoz néhány további részlet definiálható csak az annotáció típusokra alkalmazható ún. metaannotációk segítségével, melyekből négyet definiál a Java 5. A @Target metaannotációval adható meg, hogy az adott típusú annotáció milyen nyelvi elemekre alkalmazhatjuk így a paraméterként a java.lang.annoctation.ElementType enumerációból alkotott tömböt fogad el. Azt, hogy egy annotációt hogyan kezeljen a fordító és a JVM a @Retention metaannotációval szabályozható. Ebben a RententionPolicy.SOURCE értéket megadva a fordító nem vesz tudomást az annotációról, így nem kerül bele a class-fájlba. RententionPolicy.CLASS érték esetén a fordító beleteszi a class-fájlba, de a JVM az osztály betöltésekor nem őrzi meg. A @Documented metaannotációt alkalmazva egy annotációtípus definiálásakor, az annotáció belekerül a javadoc segítségével a dokumentációba. @Inherited metaanotációval megjelölt annotációtípussal annotált osztály a gyermekosztályaira is örökíti az annotációt.

Látható, hogy milyen egyszerűen és könnyen létre hozhatóak és használhatóak az annotációk. E egyszerűségnek és könnyen használhatóságnak köszönheti rohamos terjedését. Többek között használja a Spring és bevezették az EJB 3.0 specifikációban is.

5.4. EJB 3.0

Az EJB 2.1 technológia bár óriási előrelépés a middleware szolgáltatások kihasználása terén, mégis nehézkes használni, nagyon sok elem felhasználása teszi nehézkessé. Ezek az elemek a következők:

- Bonyolult fejlesztés: Mert a sok Java fájl és hozzájuk tartozó XML állomány, amelyek kezelése nem megfelelő fejlesztői eszköz nélkül nehézkes.

- Metódushívás: EJB 2.1 esetében bonyolult bean-elérési procedúrát kellett végrehajtani ahhoz, hogy használhassunk egy komponenst. Új koncepció kell, hogy a kliensek ne közvetlenül érjék el a megfelelő bean-t, hanem egy ún. kérés közvetítő objektum segítségével. Tovább bonyolította a helyzetet a több interfész és a belőlük létrejövő objektumok sokasága, illetve a névszolgáltatásban történő indirekt keresést elősegítő telepítés leíró bejegyezések módosítása.

- Kötelező elemek: A bean osztályban kötelezően implementálni kell a megfelelő bean interfészt, ami azzal jár, hogy olyan metódusokat is meg kell valósítanunk, amiket esetlegesen nem is használunk.

- Tesztelés: Nehézkes, mert más komponenseket is meg kell írni.

Ezen akadályok leküzdésére jött létre az EJB 3.0, amelyet már korábban bevezetett annotációk segítségével kiküszöböli a fent leírt problémákat.
A koncepció tehát az, hogy csökkentsem a kód méretét és a fájlok darabszámát. Továbbá kliens továbbra sem a bean-nel legyen közvetlen kapcsolatba, hanem egy közvetítő objektummal, ennek megvalósítására pedig mindenképpen kell egy interfész. Így kettő fájlt fogok használni az eddig is használt konkrét üzleti folyamatokat – magát az implementációt - tartalmazó bean-osztályt, illetve az ún. business interfészt. Felmerülhet az a probléma, hogy bizonyos framework-ök megkövetelik a saját interfész implementálást ez pedig kevésbé átláthatóvá, szerkeszthetővé teszi a kódot. Akár több olyan metódust is implementálnom kell, amiket lehet, nem is fogok használni, nem beszélve arról, hogy ezáltal framework függővé válik a kód. E probléma kiküszöbölésére találták ki az ún. POJO (Plain Old Java Object – Egyszerű Java Osztály) osztályokat. Ezen osztályok – ahogy a nevük is reprezentálja – megfelelnek a hagyományos Java osztályoknak, amiket például tudunk példányosítani. POJO osztályok tartalmazzák az elvárt funkcionalitásokat, de mégsem kell vesződni olyan metódusok megírásával, amelyeket egyébként sem használnánk fel. Továbbra is fenn szeretném tartani a helyes működést annak ellenére, hogy csökkentsem a fájlok darabszámát. A helyes működés fenntartására pedig a következő a megoldás:

Eddig interfészben adtam meg azokat a metódusokat, amiket publikálni szeretnék kifelé a kliensnek. Most csak egy interfészem van így az implements kulcsszóval összerendeljük az implementációs bean-t az interfésszel. Kérdéses továbbra is, az hogy interfész távoli vagy helyi? Ennek megadására segít a @Local és @Remote annotáció, alapértelmezettként a @Local-t tekintjük, de felülbírálhatjuk a @Remote annotációval. A @Local és @ Remote annotációkat megadhatjuk az interfészben és az implementációban is.

Egy session bean rendelkezhet állapottal, és lehet még állapotmentes is. Ezeket eddig a telepítés leíróban adtam meg a <session-type> elem segítségével. EJB 3.0 esetében egyszerűen annotációkkal is megadhatom a bean típusát. Az állapotmentes session bean-t a @Stateless, az állapottal rendelkezőt, pedig a @Stateful annotációval jelölöm.

2.1 esetén az ejbCreate() és ejbRemove() metódusok segítségével történt az EJB példányok létrehozása illetve megszüntetése. Az EJB 3.0 esetében az előző két metódusnak a @PostConstruct() és a @PreDestroy() annotáció feleltethető meg.

2.1-ben tapasztalt bonyolult bean-elérési procedúra kiküszöbölésére vezették be a függőséginjektálást (dependency injection). Függőséginjektálás azt jelenti, ha egy business interfész típusú változót @EJB annotációval láttak el, akkor a konténer köteles példányt biztosítani a megadott változónéven, tehát nem kell implicit módon névszolgáltatás keresést alkalmazni a példányosításkor.

5.4.1. Session Bean

A session bean elsősorban üzleti tevékenységeket, folyamatokat megvalósító újrafelhasználható komponens. A lényeges különbség a session bean és entity bean között az élettartamuk, és ennek következtében más a két bean felhasználási területe. Még az entity perzisztens objektum, így állapota hosszú távon is megőrződhet valamilyen tárban, ami tipikusan adatbázisban szokott történi. Addig a session bean-ek élettartama az őt hívó kliens kapcsolatának időtartamáig terjed. Amikor egy kliens meghívja a session bean valamilyen metódusát akkor az EJB konténer garantálja, hogy az adott példányhoz egy időben csak egy kliens férhet hozzá ezt úgy oldja meg, hogy a további kérések kiszolgálását átirányítja másik példányhoz vagy akár dinamikusan létre hoz újabb példányokat és azokhoz irányítja a kéréseket.
Minden Enterprise Bean és kliens közötti kommunikáció egyfajta párbeszéd formájában zajlik, ami lényegét tekintve a kliensnek az adott bean-re vonatkozó metódushívásainak egy szekvenciáját jelenti. Egy ilyen párbeszéd felöli a kliens egy adott üzleti folyamatát, ami lehet például egy internetes vásárlás.

A session bean-eknek kétféle altípusa létezik: az állapotmentes és az állapottal rendelkező sesssion bean. Továbbiakban ezzel a két bean típussal ismerkedünk meg kicsit részletesebben.

5.4.1.1. Állapotmentes Session Bean

Most részletesebben ismertetem a session bean-ek egyik fajtáját az állapotmentes session bean-t, illetve, hogy miért is választottam én az alkalmazásom fejlesztése során.

Bizonyos üzleti folyamatok nem igényelik, hogy az egyes metódushívások között megőrződjön a hívott komponens állapota. Az állapotmentes session bean, ahogy az a nevében is szerepel, olyan típusú üzleti folyamatok számára lett tervezve, amelyek során nem kell eltárolni a metódushívásról metódushívásra a session bean állapotát. Egy állapotmentes session bean metódusának meghívásakor a metódus végrehajtásához szükséges összes információt meg kell adni paraméterül a session bean-nek, vagy a session bean az információkat egy adatbázisból meg kell tudnia szerezni.

Példa állapotmentes session bean-re egy felhasználó szolgáltatásokat végző session bean ahol egy metódusa, ami felveszi a felhasználót az adatbázisba, ha még nem volt ilyen felhasználó és kivételt dob, ha már létezik.

```
@Local
public interface FelhasznaloService{
    Felhasznalo hitelesit(String nev, String jelszo);
                Felhasznalo    felvesz(Felhasznalo    uj_felhasznalo)throws
LetezikException;
    Felhasznalo keresByEmail(String email) throws LetezikException;
    Felhasznalo keresByName(String name);
    void modosit(Felhasznalo felhasznalo);
    void torol(String felhasznalo);
    List<Felhasznalo> osszes_felhasznalo();
}
```

Állapotmentes Session Bean interfészének megadása

```
@Stateless
public class FelhasznaloServiceImpl implements FelhasznaloService{

    @EJB
    private FelhasznaloDAO felhasznalodao;

    @EJB
    private PasswordEncrypter encrypter;
```

```
        public Felhasznalo felvesz(Felhasznalo uj_felhasznalo) throws
LetezikException{
      Felhasznalo old = felhasznalodao.
findbyName(uj_felhasznalo.getFelhasz());

        if(old!=null)
             throw new LetezikException(old.getFelhasz()+"nevű felhasználó
már van!");
        uj_felhasznalo.setJelszo(encrypter.encrypt(uj_felhasznalo.getJelsz
o()));
        return felhasznalodao.save(uj_felhasznalo);
    }
...
}
```

4. példa: Állapotmentes Session Bean alkalmazása

1. példában látható, hogy az EJB 3 specifikációban bevezetett annotációk segítségével milyen könnyen és egyszerűen lehet EJB komponenseket írni. Egyszerűsége és könnyen kezelhetősége, illetve jó karbantarthatósága végett választottam az állapotmentes session bean-t a saját rendszerem funkcionalitásának megvalósítására.

5.4.1.2. Állapottal rendelkező Session Bean

Az előzőkben bemutattam az állapotmentes session bean-t, most rátérek arra, hogy mire is alkalmazható az állapottal rendelkező párja és hogy miért is használom én is ezt az alkalmazásomban.

Bizonyos üzleti folyamatok szükségessé teszik, hogy a folyamatok lépesei, azaz az egyes metódushívások között megőrződjön a hívott komponens állapota. Például erre a virtuális bevásárló kosár egy webshop-ban. Mikor a kliens egy új tételt hozzáad a virtuális bevásárló kosarához, akkor természetesen a meglévő állapotot kell módosítania.

Az állapottal rendelkező session bean, ahogy a nevében is szerepel olyan üzleti folyamatok számára lett tervezve, melyek során szükséges, hogy a session bean megőrizze az adott kliens számára az állapotát több metódushívás és tranzakció során is.

Mikor a kliens megszólt egy állapottal rendelkező bean-t, akkor párbeszédet kezd vele, aminek az állapota eltárolódik a bean-ben. Mivel ennek az állapotnak a párbeszéd során mindvégig elérhetőnek kell lennie ugyanannak a kliens következő metódushívásakor is, ezért nehéz példányfarmot megvalósítani állapottal rendelkező bean-ekkel. Ha korlátosak az

erőforrások (memória, adatbázis kapcsolatok stb.), akkor mégis arra kényszerülünk, hogy korlátozzuk az egyszerre a memóriában elhelyezkedő bean-ek számát. A korlátozásra megoldás a passziválás-aktiválás technikája.

Lényege, ha elértem a korlátot, akkor a nem használt bean-t perzisztensen kimentem a tárba ezzel passzív állapotba helyezem és a helyére létre hozok egy úgy példányt. Ha pedig újra szükség van rá, akkor visszatöltöm és aktív állapotba helyezem. Aktiváláskor pedig ugyan abba az állapotba kerül vissza, mint ahogy ki lett mentve.

```
public interface RendelesService {
    Rendeles felvesz(Rendeles rendeles);
    void add(Rendelesi_Tetel rendel);
    void torol(Rendelesi_Tetel rendel);
    Map<Long, Rendelesi_Tetel> getRendeles();
}
```

Állapottal rendelkező Session Bean interfészének megadása

```
@Stateful
@Local(RendelesService.class)
public class RendelesServiceImpl implements RendelesService{
    @EJB
    private RendelesDAO rendelesdao;

    private Map<Long, Rendelesi_Tetel> rendelesek;

    @PostConstruct
    public void init(){
        rendelesek = new HashMap<Long, Rendelesi_Tetel>();
    }
    public void add(Rendelesi_Tetel rendel){
        rendelesek.put(rendel.getId(), rendel);
    }
    public void torol(Rendelesi_Tetel rendel){
        rendelesek.remove(rendel);
    }
    public Rendeles felvesz(Rendeles rendeles) {
        return rendelesdao.save(rendeles);
    }

    public Map<Long, Rendelesi_Tetel> getRendeles() {
        return rendelesek;
    }
}
```

5. példa: Állapottal rendelkező Session Bean alkalmazása

Az én alkalmazásom egy része is egy online bevásárló bolt, azaz a felhasználóknak van egy kosaruk, amibe tehetik a kiválasztott termékeket. Ahhoz, hogy meg tudjam valósítani a vásárlást mindenképpen egy olyan megoldás kell ami meg tudja őrizni a hívások között a kosár állapotát. Erre a célra pontosan megfelel az állapottal rendelkező session bean. A 2. példában a saját alkalmazásom bevásárló kosarát megvalósító session bean látható.

5.4.2. Entity Bean

Az entity bean-ek segítségével lehetőségünk van modellezni az üzleti folyamatok által használt adatokat, méghozzá perzisztens módon. A modellezésre használhatók a következő típusú entity bean-ek.

Entity bean-ek fajtái:

BMP (Bean Managed Persistence):
A BMP esetén a bean gyártója implementálhatja a bean perzisztenciáját. A BMP esetén a programozónak kell implementálnia a perzisztenciakezelést. A perzisztenciakezelés implementálása következtében konkrét JDBC hívásokra van szükségünk. A programozónak kötelezően implementálnia kell néhány metódust, mint például ejbCreate(), ejbRemove(), ejbFind(), ejbLoad(), és még egy néhány metódust.

CMP (Container Managed Persistence):
A konténer által kezelt perzisztencia esetén a konténer valósítja meg a bean leképezését a tárba (általában adatbázisba). Nem szükséges a perzisztenciával foglalkozni, a konténer legenerálja a működéshez szükséges metódusokat. Miképpen valósítja meg a leképezést arra még később visszafogok térni.

5.4.2.1. Perzisztencia és kezelése

Már esett szó az entity bean-ek kapcsán, hogy az adatok tárolhatóak perzisztens módon. Továbbiakban azzal fogok részletesebben foglalkozni mit is jelent a fogalom, hogy perzisztencia és hogyan is valósítható meg az egyes technológiák segítségével.

Alkalmazás használata során adatokat kell feldolgozni illetve létre is lehet hozni új adatokat. Mikor létrehozunk, valamilyen új adatot az csak a memóriában jelenik meg és mikor az alkalmazás befejezi a futást akkor ezek automatikusan megszűnnek. Újból induláskor megint létre kell hozni, hogy használni tudjuk újra. Ezért, hogy ne keljen minden induláskor újra létrehozni ki kell menteni valamilyen módon valamilyen külső tárolóra a már létrehozott adatot. Mikor kimentek egy adott objektumot, akkor perzitáclom azaz az objektum túl fogja élni a programfutását. Adatokat kimenthetünk adatbázisba vagy a fájlrendszer valamely fájljába. A nagy és bonyolult adat struktúrával rendelkező alkalmazásoknál nem érdemes fájlba menti, mert a fájlok kezelése nehézkes lehet ekkor célszerű adatbázisokat alkalmazni. Adatbázis piacon igaz, hogy léteznek objektum orientált adatbázisok, de még mindig a relációs adatbázisokat használnak túlnyomó részben. Így azonban újabb probléma merül fel, hogyan képezzük le az objektumokat relációkra. Ebben nyújt segítséget az ORM (Object Relation Mapping) amely a következő képen oldja meg a problémát: az adatbázis táblái az osztálynak, a táblák oszlopai pedig az osztály attribútumainak felelnek meg. Egy osztály példánya a hozzá tartozó tábla egy sorának feleltethető meg.

Az entity bean az ORM leképezést használják az adatok tárba való mentésére. Az automatikus objektumrelációs leképezés ugyanakkor alapvetően hasznos dolog, ami sok kódolástól szabadítja meg a fejlesztőt, így több alternatív megoldás is született erre a feladatra (pl. JDO, Hibernate, TopLink). Ezek közös jellemzője, hogy az EJB konténertől független megoldások, így Java SE-alkalmazásokban is elérhetők. Azzal, hogy a perzisztens objektumok egyszerű Java- osztályok példányai, így egyszerűbb fejlesztői modellt biztosíthatunk alkalmazásaink számára.

Az EJB 3.0 specifikáció az említett keretrendszerek tapasztalatait felhasználva egy új perisztenciamegoldást tartalmaz a Java Persistence API-t (JPA), amely szintén POJO alapú és akár Java SE-ben is használható. Fontos megjegyezni, hogy bár a JPA is az entitás (entity) megnevezést használja a perzisztens objektumokra, a JPA entitások nem tekinthetők az entity bean technológia új verziójának, mivel az EJB 3.0 specifikáció tartalmazza az entity bean-eket is, a 2.1-es verzióhoz képest változatlan formában. A JPA egyszerűségének és többlet szolgáltatásainak köszönhetően mára már teljesen felváltotta az entity bean technológiát.

5.4.2.2. Objektumrelációs leképezés annotációkkal

Rendszerem készítésekor szembesültem azzal a problémával, hogy számos entitást későbbiek során is elérhetővé kell tennem. Mindenképpen egy rugalmas és könnyen használható eszközt szerettem volna, ami nem igényel plusz programozói munkát és mások számára is könnyen értelmezhető. Ezért a választásom a JPA-ra esett, mert az annotációk segítségével könnyen meg tudom mondani, hogy milyen attribútumokat szeretnék későbbiekben is elérhetővé tenni. Továbbiakban az alkalmazásom egyik fő részének a termékek entitás perzisztenssé tételét néznék meg részletesen.

Első lépésben elkészítettem az egyszerű java osztályt, amiben felsoroltam a termékekre jellemző attribútumokat. JPA egyik alapkövetelménye, hogy minden entitásnak kötelezően kell lennie egy argumentum nélkül konstruktorának. Másik fontos követelmény, hogy az egyes attribútumoknak kell lennie, get/set metódusának. Ahhoz, hogy entitás legyen az elkészült osztályunk az osztály név elé kell írni a @Entity annotációt, ezzel egy perzisztenssé tettük az osztályt. Entitások egy alapkövetelménye az elsődleges kulcs megadása, ezt a @Id annotációval adhatjuk meg. Ahhoz, hogy használni tudjam ezeket az annotációkat mindenképpen importálni kell a javax.persistence csomagot. Az elmondottakat szemlélteti a következő példa:

```
import javax.persistence.*;

@Entity
public class Termek{

     @Id
     private Long id;
```

```
      private String nev;
      private String leiras;
      private String megjegyzes;
      private int ar;
      private Date lejarat;

      public Termek() {
      }
      public Long getId() {
         return id;
      }
      public void setId(Long id) {
         this.id = id;
      }
...
}
```

6. példa: Osztály Entitásnak való megadása

CMP entity bean esetén konténerspecifikus módon kell deklarálni, hogy az egyes attribútumok milyen nevű tábla milyen nevű oszlopára képződjenek le. Példában is látható, hogy semmi ilyesmit nem adtunk meg, JPA alapértelmezettként az osztály névvel és attribútum névvel azonos táblát és oszlopokat hoz létre. Természetesen ezek a megfelelő annotációkkal felül definiálhatók. A @Table annotáció használata esetén az annotációban megadott név lesz az adatbázisbeli tábla neve. Perzisztens attribútumokra alkalmazható a @Colum annotáció, ekkor az oszlop neve az adatbázisban az annotációban megadott név lesz. A néven kívül még más tulajdonságok is megadhatók az oszlopokra, ezekre nem terjedelmi okok miatt nem szeretnék kitérni.

Az entitások rendelkezhetnek tetszőleges nem perzisztens attribútumokkal is, ezeket @Transient annotációval tudjuk megjelölni. A perzisztens attribútumoknál letézik bizonyos megkötés a használható típusokra. Ezekre azért van szükség, hogy megfelelő módon lehessen konvertálni őket az adott SQL-típusokra. A perzisztens attribútumok lehetséges típusait a következők lehetnek:

- Java primitív típusok (int, long, char, stb.)
- primitív típusok csomagoló osztályai (Integer, Long, Char, stb.)
- Java szerializálható típusok
- felhasználó által definiált szerializálható típusok
- felsorolásos típus
- java.lang.String

- java.math.BigInteger, java.math.BigDecimal
- java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time
- java.sql.Timestamp
- byte[], Byte[]
- char[], Character[]

Az adatok betöltődése a tárból, illetve az entitások attribútumainak tárba való kimentése az annotációk elhelyezésétől függ. Ha a beállító és lekérdező metódusok előtt helyezem el, akkor ezek felhasználásával történik meg a mentés és a betöltés. Ha az attribútumok előtt helyezem el, akkor betöltésnél közvetlenül a változókba töltődik be a kért adat és nem a setter metódusok használatával. A kimentés hasonló elven működik, mint a betöltés. Bármelyiket is választom, figyelembe kell venni, hogy egy entitáson belül csak egyféle mód lehetséges, különben a működés definiálatlan lesz.
A perzisztens attribútumok közül van egy, ami különleges jelentőséggel bír, ez nem más, mint az elsődleges kulcs. Az elsődleges kulcs kizárólagosan azonosítja az adott táblát az adatbázisrendszeren belül, és segíti a más táblákkal történő kapcsolat felvételt. Mint ahogy már említettem, az elsődleges kulcs attribútumot a @Id annotációval adhatjuk meg. A többi perzisztens attribútumtól eltérően, erre erősebb megkötések vonatkoznak a típusok tekintetében. Az elsődleges kulcs típusa lehet:

- primitív típus, kivéve a lebegőpontosok
- a primitív típusoknak megfelelő csomagoló osztályok
- String
- java.util.Date
- java.sql.Date

Alkalmazásom használata során eljött, azaz igény, hogy ne nekünk kelljen az egyes egyedek kulcs értékeit megadni, hanem automatikusan történjen a megadás. Az elsődleges kulcs értékek generálására van lehetőségünk, méghozzá a @GeneratedValue annotáció segítségével. Fontos megjegyezni, hogy ilyenkor csak egész típusú lehet az elsődleges kulcs. A generálás többféle módon történhet, a GenerationType felsorolásos típus négy értéke közül az egyiket állíthatjuk be. A négy érték a következő:

- SEQUENCE: egy, az adatbázis által kezelt számláló felhasználásával történik az elsődleges kulcs generálása.

- IDENTITY: egy, az adatbázisbeli tábla autoinkrementálódó oszlopára képződik le az elsődleges kulcs.

- TABLE: a @TableGenerator annotáció és a TABLE használatával egy adatbázisbeli tábla adott sorának adott oszlopában található érték lesz az elsődleges kulcs.

- AUTO: az előző három közül valamelyik.

```
@Entity
public class Termek{
    private Long id;
    ...
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
   ...
}
```

7. példa: Entitás elsődleges kulcsának generálásának megadása

5.4.2.3. Kapcsolatok entitások között

Az előbbiekben bemutatott módon egymástól teljesen független entitások hozhatók létre. Az alkalmazásom entitásai készítése közben, merült fel az probléma, hogy nem egy különálló entitás értékeit szeretném használni, hanem két vagy több entitás által hordozott értékeket. JPA a kapcsolatokat szintén annotációk segítségével oldja meg. Az EJB hétféle kapcsolatot tud kezelni. Alkalmazásom készítése során, nem volt szükség minden a hét fajta alkalmazására, de továbbiakban szeretném bemutatni minden a hét fajtát.

5.4.2.3.1. Egyirányú egy-egy kapcsolat

Erre a kapcsolattípusra jó példa lehet mikor a rendszeremben a felhasználók és a címeket tároló táblák közti kapcsolatot szeretnék megadni. Minden felhasználóhoz egy cím tartozik és minden címhez egy felhasználó fog tartozni ebben a kapcsolat modellben, valamint a címeket tároló táblából nem lehet visszakeresni, hogy az adott cím melyik felhasználóhoz tartozik. Ennek megfelelően csak a Felhasznalo bean-ben kell jelezni a kapcsolatot:

```
@Entity
public class Felhasznalo implements Serializable{
...
     private Cim cim;
...

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="Cim_Id")
    public Cim getCim() {
        return cim;
    }
    public void setCim(Cim cim) {
        this.cim = cim;
    }
...
}
```

8. példa: Egyirányú egy-egy kapcsolat megadása

A forráskód részlet alapján látható, hogy a kapcsolat megadásához a @OneToOne annotációt kell használnom. A @JoinColumn annotációval, pedig a külső kulcsot adtam meg, amely segítségével a felhasználóhoz tartozó címinformációk kérhetők le.

5.4.2.3.2. Kétirányú egy-egy kapcsolat

Alkalmazásom entitásai között akkor hozhatok létre kétirányú kapcsolatot, hogy ha azt szeretném, ha kapcsolatban álló táblák bármelyikéből kiindulva elérhetőek legyenek a másik táblában tárolt adatok. Például, ha a felhasználó eredeti nevére vonatkozó tulajdonságok egy másik táblában vannak tárolva, az alkalmazásban, pedig az eredeti névre keresek rá és látni szeretném a felhasználó egyéb adatait akkor célszerű használni ezt a kapcsolatot. Ekkor mindkét bean-ben jelezni kell a kapcsoltot ezt a következőképpen tehetjük meg:

```
@Entity
public class Nev implements Serializable{

    private String vezeteknev;
    private String keresztnev;
    private Felhasznalo felhasznalo;

...

  @OneToOne(mappedBy="nev")
    public Felhasznalo getFelhasznalo() {
        return felhasznalo;
    }
    public void setFelhasznalo(Felhasznalo felhasznalo) {
        this.felhasznalo = felhasznalo;
    }
...
}
```

Kapcsolat másik irányának megadása

Ami újdonság az egyirányú kapcsolat leírásához képest azaz, hogy az @OneToOne annotációnak a mappedBy attribútumát is beállítottam Az itt megadott érték jelzi, hogy a Felhasznalo bean nev tulajdonsága alapján kétirányú kapcsolatot szeretnénk a két bean között. Ennek megfelelően a Felhasznalo bean a következő képen fog kinézni:

```
@Entity
public class Felhasznalo implements Serializable{
...
private Nev nev;
...
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="Nev_Id")
    public Nev getNev() {
        return nev;
    }
...
}
```

9. példa: Egy-egy kapcsolat megadása

5.4.2.3.3. Egyirányú egy-sok kapcsolat

Alkalmazásomban szerepkörrel írom le, hogy az adott szerepkörben lévő felhasználó milyen mélységben éri el a rendszert. Az egyes szerepkörök, pedig jogosultságokat tartalmaznak, melyik megmondják, hogy az adott szerepkörben lévő felhasználó elérheti-e az adott funkciót. Most számomra nem érdekes az, hogy az adott jogosultság milyen szerepkörökhöz van hozzárendelve, csak az érdekel, hogy az egyes szerepkörök milyen jogosultságokat tartalmaznak. Ennek megfelelően a két bean között sok egy kapcsolatot kell létrehozni a @OneToMany annotáció segítségével a következő képen:

```
@Entity
public class Szerepkor implements Serializable{
...
    private List<Jogosultsag> joglist = new ArrayList<Jogosultsag>();

    @OneToMany(cascade={CascadeType.ALL}) {
    public List<Jogosultsag> getJogList(){
     return joglist;
    }
    public void setJogList(List<Jogosultsag> joglist){
     this.joglist=joglist;
    }
...
}
```

10. példa: Egyirányú egy-sok kapcsolat megadása

A Szerepkor bean-ből tehát a jogosultságok egy listája érhető el, azaz @OneToMany annotációval jeleztem, hogy a kapcsolat egy-sok jellegű. Az Jogosultsag bean-ben nem kell semmilyen módon jelezni a kapcsolatot, így a kapcsolat egy irányú lesz.

5.4.2.3.4. Egyirányú sok-egy kapcsolat

Rendszeremben van egy raktár tábla, amiben termékek vannak nyilvántartva, a termék szempontjából most nem érdekes, hogy még milyen termékek vannak raktáron, az fontos azonban, hogy a raktárban milyen termékek vannak. Így a megfelelő kapcsolat modell kialakításához a @ManyToOne kapcsolatot kell használnunk:

```
@Entity
public class Raktar implements Serializable{

    private Termek termek;
    private int darab;
...
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="Termek_Id")
    public Termek getTermek() {
        return termek_id;
    }
    public void setTermek(Termek termek) {
        this.termek = termek;
    }
...
}
```

11. példa: Egyirányú sok-egy kapcsolat megadása

Ebben az esetben a sok-egy kapcsolatot a @ManyToOne annotációval lehet megadni, míg a Termek entity-ben semmilyen módon nem kell jeleznem a kapcsolatot.

5.4.2.3.5. Kétirányú egy-sok, vagy sok-egy kapcsolat

Előző példán kicsit tovább haladva, felmerült az igény arra, hogy a termékek szemszögéből is szeretném tudni, hogy még milyen termékek vannak még raktáron. Így, hogy mindkét entitásban megadjam a kapcsolatot már csak nézőpont kérdése, hogy az most sok-egy, vagy egy-sok kapcsolat ezért kétirányú esetben nem is különböztetjük meg.

```
@Entity
public class Termek implements Serializable{
...
    private List<Raktar> raktar = new ArrayList<Raktar>();
...
    @OneToMany(mappedBy = "termek")
    public List<Raktar> getRaktar() {
        return raktar;
    }
    public void setRaktar(List<Raktar> raktar) {
        this.raktar = raktar;
    }
...
}
```

12. példa: Kétirányú egy-sok, vagy sok-egy kapcsolat megadása

A Raktar bean ugyanaz lesz, mint egyirányú egy-sok esetben. Azáltal, hogy mindkét bean-ben meg adtuk a kapcsolatot, így kapcsolat kétirányú lesz.

5.4.2.3.6. Kétirányú sok-sok kapcsolat

Sok-sok kapcsolat esetén már mindenképpen szükség van egy kapcsolótáblára relációs modellben, míg objektumorientált modellben ez szükségtelen, ennek megfelelően még egy entitást sem kell létrehozni a kapcsolat reprezentálására, az EJB gondoskodik annak relációs modellbeli létrehozásáról.

Korábban volt egy példa a szerepkör jogosultságokra. Akkor nem volt lényeges a jogosultság szempontjából, hogy melyik szerepkörökhöz tartozik, most viszont felmerült az igény, hogy szeretném tudni melyekhez is tartozik. Ennek megfelelően a kapcsolat így alakul át:

```
@Entity
public class Jogosultsag implements Serializable{
...
   private List<Szerepkor> szerep_list;
...
    @ManyToMany(mappedBy="joglist")
    public List<Szerepkor> getSzerep_list() {
        return szerep_list;
    }
...
}
```

Sok-sok kapcsolat másik irányának megadása

```
@Entity
public class Szerepkor implements Serializable{
...
    private List<Jogosultsag> joglist = new ArrayList<Jogosultsag>();
...
    @ManyToMany
    @JoinTable(name="Hozzaferes",
    joinColumns={@JoinColumn(name="Szerepkor_Id")},
    inverseJoinColumns={@JoinColumn(name="Jogosultsag_Id")})
    public List<Jogosultsag> getJoglist() {
        return joglist;
    }
...
}
```

13. példa: Sok-sok kapcsolat megadása

A kapcsolat megadásához a @ManyToMany annotációt kell használnom és az egyik entity bean-ben meg kell adni a kapcsolótábla adatait. A kapcsoló tábla adatait a @JoinTable annotáció segítségével adhatjuk meg.

5.4.2.3.7. Egyirányú sok-sok kapcsolat

Ha a kapcsolat egyirányú, akkor csak az egyik entity-ben kell jeleznem a kapcsolatot a @ManyToMany annotációval és ebben az entity-ben kell megadnom a kapcsolótáblát is. Az adatbázisban a táblák szerkezete ugyan az lesz, mint kétirányú esetben, így az egyirányúság csak az objektumorientált modellben lesz értelmezhető.

5.4.2.4. Entity Bean-ek menedzselése

Eddigiek alapján rövid ízelítőt kaptunk az entitások létrehozásáról és a köztük lévő kapcsolatokról. De még nem esett szó arról, hogyan is lehet kezelni őket. Mindenképpen szükségem lesz egy Enterprise Bean-re, amelyen keresztül elérhetem ezeket az entity-ket. Ezek az Enterpise Bean-ek nem mások, mint az a session bean-ek amiknek, a működéséről már volt szó. A session bean-eken kívül még szükség lesz EntityManager objektumra is az entitások kezeléshez. Az EntityManager interfész a javax.persistence csomag része akárcsak az entity bean-eknél használt annotációk. Gyakorlatilag minden szükséges műveletet biztosít, amire csak szükségem lehet az entity bean-ek perzisztens módon való kezeléséhez. Ahhoz, hogy használni tudjam mindenképpen referenciát, kell szereznem egy ilyen objektumra, ehhez használhatom az EntityManagerFactory osztályt, vagy ennél kényelmesebb módon is megtehetem, ha az @PersistenceContext annotációval megkérem az alkalmazásszervert, hogy injektálja be nekem. Az EntityManager által biztosított legfontosabb műveletek:

```
package javax.persistence;

public interface EntityManager {

public void persist(Object entity);
public <T> T find(Class <T> entityClass, Object primaryKey);
public <T> T merge(T entity);
```

```
public void remove(Object entity);
public Query createQuery(String queryString);
public Query createNamedQuery(String name);
public Query createNativeQuery(String sqlString);
}
```

14. példa: EntityManager fontosabb műveletei

Csak a leggyakrabban használt műveletek emeltem ki, amiket én is használtam alkalmazásom fejlesztés során, természetesen, még van pár művelet amit meg lehet tekintetni az API-ban.
Leggyakrabban használt művelet find(Class<T>entityClass,ObjectprimaryKey) metódus, amely segítségével az elsődleges kulcs alapján a tábla egy sorát kérdezhetem le, azaz egy objektum lesz a keresés eredménye. A persist(Objectentity) metódus segítségével gyakorlatilag egy INSERT utasítást tudom végrehajtani, ennek hatására az EntityManager perzisztálja az entity bean-t. A remove(Objectentity) metódus törli az adatbázisból az entity-ben tárolt adatokat, vagyis a tábla egy sorát törli az adatbázisból. A merge(Tentity) metódus segítségével módosíthatom a már letárolt adatokat. Különböző bonyolult lekérdezések futtatására is van lehetőségem ezeket a createQuery, createNamedQuery, illetve a createNativeQuery metódusok segítségével tehetem meg.
Még számos dolog dologról lehetne írni az entity bean-ek és maga az EJB technológia kapcsán. De szerintem ezek alapján már érzékelhető, hogy miért is érdemes ezt a technológiát választani egy-egy alkalmazásfejlesztés kapcsán.

5.5. TopLink

Entitások objektum relációs leképezése kapcsán merült fel a TopLink neve, mint egy alternatív megoldás a leképezésre. Most szeretném megmutatni, hogy valójában mit is takar ez az eszköz.
A TopLink valójában egy lehetséges implementációja az objektumok leképezésének. Miért csak egy lehetséges implementáció? Mert, más gyártók ugyanazt a leképezést feltételezhetően teljesen másképp oldották meg és ezzel egy újabb lehetséges implementációt, illetve eszközt adtak a fejlesztők kezébe. Az egyes implementációk sikerességét mi sem bizonyítja, hogy ezek alapján született meg az egységes specifikáció a Java Persistence API (JPA). JPA megjelenése után, ha perzisztenciát szeretnénk kezelni, akkor nem kell, az egyedi

megoldásokhoz igazodunk, nyugodtan használhatjuk a JPA által nyújtott eszközöket mellyel megmondjuk, mit mire szeretnék leképezni. Majd annyi a dolgunk van még, hogy választunk egy implementációt, ami majd elvégzi a konkrét leképezést.
Én a TopLink-et választottam erre a célra de, bármikor áttérhetek másik implementációra anélkül, hogy a kódban bármilyen változtatást kellene tennem.

A TopLink-nek nem csak ez az egy szolgáltatása van, hanem lehetőséget biztosít objektumok XML-re történő leképezéséhez, illetve elosztott működéshez is nyújt szolgáltatást. Ha sikerült felkeltenem a technológia iránti érdeklődést, akkor az irodalomjegyzék [8] forrásban utána lehet olvasni a technológiának.

5.6 XML – Extensible Markup Language

Az XML a W3C által ajánlott általános célú leíró nyelv, speciális célú leíró nyelvek létrehozására. Már korábban említett SGML nyelv egyszerűsített részhalmaza, mely különböző adattípusok leírására képes. A web áttekintés során tárgyalt HTML is a SGML részhalmaza. Merülhet fel a kérdés bennük, hogy mi a közös a két nyelvben? Mindkét nyelv ’tag’-eket használ, de még a HTML esetén előre beépített tag-eket lehet csak használni addig az XML esetén bármilyen nevű tag-ek használhatók. A tag-ekben plusz információ megadásaként használhatunk attribútumokat. Ekkor a következő képen néz ki egy elem:

<elem_neve attribútum_neve="attribútum értéke">Elem tartalma</elem_neve>

Az XML azzal az elsődleges céllal jött létre, hogy adatszerkezeteket strukturáltan lehessen tárolni, de használható más nyelvek definiálására is.

Egy XML dokumentum önmagában egy fa adatstruktúrát reprezentál. A fa struktúrának köszönhetően az elemek egymáshoz való hierarchikus viszonya rögzített. A hierarchikusságból következik, hogy az elemek elérése gyors és könnyen újra lehet szervezni. Nagy előnye az XML-nek, hogy nem csak gép számára, de ember számára is könnyen olvasható, értelmezhető. Ember számára az olvashatóságot az Unicode karakterkészlet támogatása teszi lehetővé. Az XML nem csak adatok leírására és tárolására alkalmazható, hanem mondjuk interneten történő adattovábbításra.

Természetesen, előnye mellett sajnos hátrányai is vannak. Hierarchikus szerkezetből adódóan nincs mód a dokumentum egyes részeinek közvetlen elérésére és frissítésére. Egymást részben átfedő (nem hierarchikus) adatstruktúrák modellezése külön erőfeszítést igényel. A szintaxisa elég bőbeszédű és részben redundáns. Ez nehezítheti az emberi olvashatóságot és az alkalmazások hatékonyságát, valamint nagyobb tárolási költséggel jár. Ahhoz, hogy egy XML dokumentum helyes legyen, a következő követelményeknek kell megfelelnie:

- Helyesen formázottság. Egy helyesen formázott XML dokumentum megfelel minden XML szintaxis szabálynak. Például ha egy nem üres elem rendelkezik nyitó tag-gel, de nem rendelkezik záró tag-gel, akkor nem helyesen formázott. Az a dokumentum, ami nem helyesen formázott, nem tekinthető XML-nek. Az elemzőnek meg kell tagadnia a feldolgozását.

- Érvényesség. Egy érvényes dokumentum olyan adatot tárol, ami megfelel a felhasználó által definiált tartalmi szabálynak, ami leírja a helyes adat értékeket és helyeket. Például ha a dokumentum egy elemének olyan szöveget kell tartalmaznia, ami egész számként értelmezhető, és ehelyett a szöveg "helló", üres, vagy más elemeket tartalmaz, akkor a dokumentum nem érvényes.

Továbbiakban az XML-t konfigurációs fájlnak fogom használni, de nem csak ennek használható, hanem sok más egyéb célra is. Akinek sikerült felkeltem az érdeklődést XML technológiák iránt, annak érdemes felkeresni az irodalomjegyzék [5] sorszámú forrását.

5.7. Szervlet

Mielőtt bele kezdünk korszerűbb web technológiákkal megismerésébe mindenképpen meg kell ismernünk azt a technológiát, amiből kifejlődtek. Valójában még mindig ott vannak csak nekünk nem kell már vele foglalkoznunk ilyen alacsony szinten. Továbbiakban a szervletekkel foglalkozom kicsit részletesen, hogy miért is alakult ki és milyen előnyei, hátrányai vannak.

A webet kezdetben statikus oldalak letöltésére használták, majd egyre inkább elterjedtek a weboldalak dinamikus generálására képes technológiák (CGI, PHP, ASP). Segítségükkel vált megvalósíthatóvá olyan komplexitású funkcionalitás, amely a hagyományos asztali alkalmazások egy részét is képes volt kiváltani, így az ilyen webes megoldások joggal nevezhetők webalkalmazásnak. A Java kezdetektől fogva erősen kötődik a webhez, a szerverről letölthető és böngészőben kliensoldalon végrehajtható appletek révén. Így nem meglepő, hogy a szerveroldali, vagy is dinamikus weboldal-generálást végző technológiák támogatásában is élen jár a Java. Az első szerveroldali webes Java technológiát a szervletek jelentették.

A Java szervletek olyan Java nyelvű osztályok, amelyeket felhasználva könnyedén és hatékonyan fejleszthetünk dinamikus tartalmakat generáló szerveroldali megoldásokat. Itt fontos kiemelni, hogy fontosabb szerveroldali, webes megoldások a szervlettechnológiára épülve működnek, illetve alakultak ki, így a működés legfőbb alapelveinek és fejlődés irányvonalának meghatározásánál fontos szerepet játszanak a szervletek. Legfrissebb változata a Java EE 5 részeként a Servlet 2.5.

A Servlet API-t a javax.servlet és a javax.servlet.http csomagok alkotják. A javax.servlet csomagban a generikus, protokolloktól független szervleteket támogató osztályok és interfészek találhatók. Ezeket az osztályokat a javax.servlet.http csomagban lévő osztályok bővítik, hogy HTTP-specifikus funkciókkal lássák el az előbbieket. Egy általános protokoltól független szervletnek GenericServlet alosztályt, míg a HTTP-szervletnek HttpServlet alosztály kell bővítenie, amely maga is a GenericServlet alosztály egy HTTP-specifikus tulajdonságokkal felruházott változata.

Az önállóan futtatható Java alkalmazásoktól eltérően a szervleteknek nem kell tartalmazniuk main() metódust. Ehelyett a kiszolgáló a kérések kezelésekor a szervlet más-más metódusait hívja meg. Minden olyan esetben, amikor egy kiszolgáló egy szervlethez továbbít egy kérést, meghívja a szervlet service() metódusát, amely paraméterül egy ServletRequest és ServletRespone típusú objektumot kap. Elsőben értelemszerűen a kérés érhető el, míg a másodikba fogjuk tudni a választ generálni.

Ha egy generikus szervletet készítünk, akkor a szervlet service() metódusát kell felüldefiniálni, hogy a kérést a saját feladatának megfelelően kezelje. Egy HTTP-szervlet esetén a service() metódus úgy felül van már definiálva, hogy a kérésnek megfelelőt, GET kérés esetén doGet(), POST kérés estén doPost() metódust kell felül definiálni.

Szervlet életciklus

Szervleteket egy szervletkonténer vezérli és ő lát el mindenféle menedzselési feladatot. Egy szervleten végzett feladatok, egy életciklus határoznak meg. Az életciklus egy inicializálással (init metódus) kezdődik majd a kiszolgáló (service metódus) és eltávolító (destroy metódus) metódusok következnek. A három metódus megvalósítását a Servlet interfész teszi kötelezővé, amit minden szervletnek kötelezően implementálni kell.
Kérés érkezése estén, ha még nincs példányosítva a szervlet, a webkonténer betölti a szervletosztályt, létrehoz belőle egy példányt, majd meghívja init metódust, amelyben végrehajtható az inicializáció, a kiszolgáláshoz szükséges erőforrások lefoglalása. Minden egyes kérés esetén a szervlet service metódusa hívódik meg, amely a megfelelő kérés és válasz objektumokat kapja paraméterül.

Webalkalmazás létrehozás

Egy webalkalmazás létrehozásnak szabályai vannak. A webalkalmazásnak egy külön könyvtárban kell elhelyezkednie, vagy pedig egy .war kiterjesztésű tömörített állományban. Rendelkezni kell egy WEB-INF könyvtárral, amely a konfigurációs fájlokat tartalmazza. Következőkben egy rövid példán bemutatom, hogy hogyan is működik a szervlet:

```
//kotelező importok

public class HelloServlet extends HttpServlet {
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException {
      String mezo = request.getParameter("mezo");
      response.setContentType("text/html; charset=UTF-8");
      response.setHeader("Content-Language", "hu");
      PrintWriter writer = response.getWriter();
      writer.println("Begépelt üzenet: <b>"+mezo+"</b>");
      }
}
```

HTTPServlet használta

Ezután elkészíthetjük az űrlapot tartalmazó HTML fájlt (index.html):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
      <head>
                        <meta  http-equiv="Content-Type"  content="text/html;
charset=UTF-8">
                  <title>Hello Servlet</title>
      </head>
<body>
      <h1>Hello Servlet</h1>
<form action="hello" method="post">
      <input type="text" name="mezo"><br/>
      <input type="submit" value="Küldés">
</form>
</body>
</html>
```

Adatok bekérésének megadása

És végül a telepítésleíró fájlba (web.xml) következő sorokat kell felvenni:

```
<servlet>
      <servlet-name>hello</servlet-name>
      <servlet-class>HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>hello</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
```

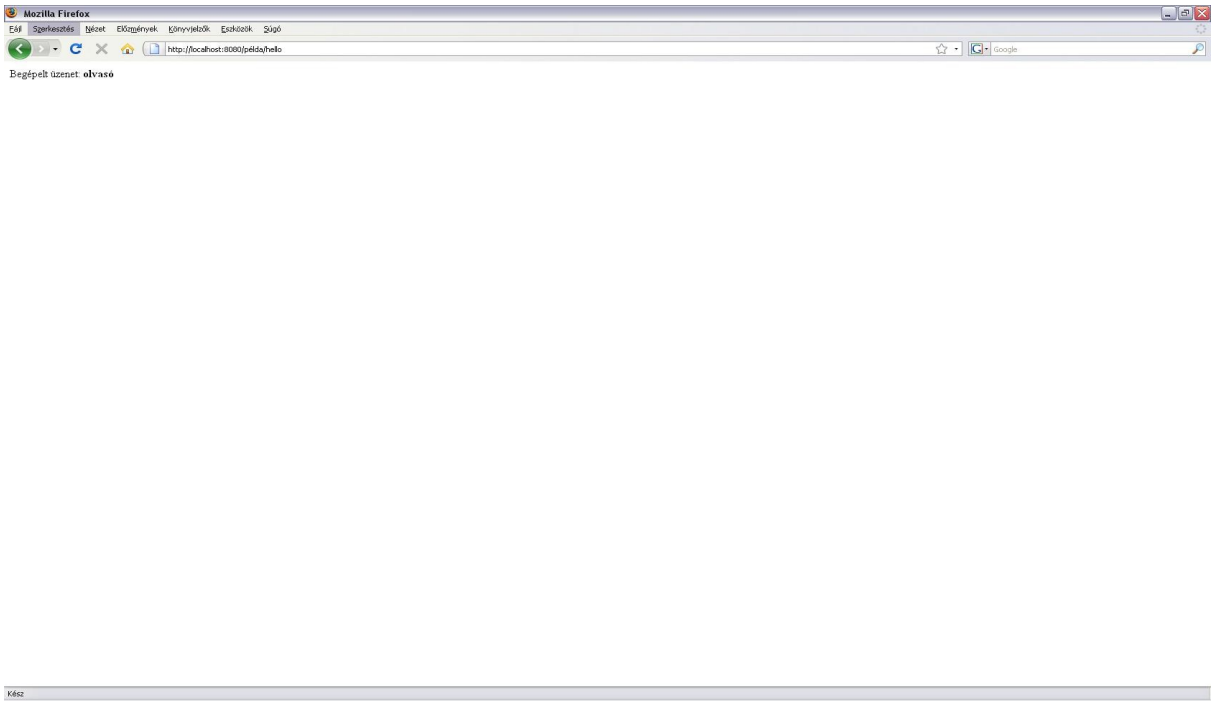
Működéshez szükséges sorok a web.xml fájlban

15. példa: Egy szervletből álló webalkalmazás létrehozása

Ha a webalkalmazás létrehozásának szabályai szerint alakítom ki a megfelelő könyvtár szerkezetet és létrehozom a megfelelő fájlokat, illetve egészítem ki a kötelező tartalmakkal akkor a következő képet kell kapnunk az alkalmazás indítása után:

3. ábra: 15. példa elindítása után

Az input mezőbe tetszőleges szöveget írhatunk - én az „olvasó” szöveget írtam be – az elküld gomb megnyomása utána visszakapjuk amit beírtunk. Én példám szerint ennek a képnek kell megjelennie:



4. ábra: Input mező kitöltése és küld gomb lenyomása után

A 15. példa alapján látható, hogy a szervletek segítségével egyszerűen hozhatam létre webalkalmazásokat, de azt is láthatjuk, hogy csak alacsony szintű eszközöket biztosít a kérések és válaszok kezelésére. A request objektumból egyenként kell kiszedni az elküldött űrlapmezők értékeit, magamnak kell gondoskodni a validálásukról. Ami viszont még rosszabb, a HTML kimenet esetén a HTML tag-eket is a java kódba kell beilleszteni, így például még az a lehetőség sincs, hogy ezt egy HTML szerkesztővel készítsük el. Szerencsére számos Java API létezik már aminek segítségével nem kell ilyen alacsony szintű eszközökkel dolgozni.

5.8. JSP

A JavaServer Pages (továbbiakban röviden JSP) a Java szerveletekhez hasonlóan valamilyen szöveges, leggyakrabban HTML vagy XML formátumú dokumentum kérés alapján történő dinamikus szerveroldali előállítására szolgáló technológia, a Java Enterprise Edition (JEE) platform kötelező része.

A JSP lakok dedikált célja, hogy lehetővé tegye a statikus és dinamikus tartalmak szétválasztását. Fontos azonban megjegyezni, hogy ez csak egy lehetőség a JSP semmilyen módon nem kényszeríti ránk, a statikus és dinamikus tartalmak szétválasztását, így minden lehetőség adott, hogy az oldalon akár üzleti logikát is kódoljunk ezzel végül zavaros és áttekinthetetlen kódot hozva létre.

Egy JSP lap pontosan olyan, mint egy szokásos weboldal, azzal a különbséggel, hogy a HTML kód mellett JSP elemeket is tartalmaz, amelyek a lapnak beérkező kérésektől függően változó tartalmakat állítják elő. Mielőtt a kiszolgáló elküld egy lapot fel kell azt dolgoznia, ami azt jelenti, hogy a JSP lapot először átalakítja szervletté, majd azt hajtja végre. A JSP lapok felépítésének megértéséhez először nézzük meg, hogy milyen elemeket is tartalmazhat egy JSP lap. Egy JSP elemeknek három típusa van: direktívák, akciók és szkript elemek.

- Direktívák: olyan, a lappal kapcsolatos információkat specifikál, amelyek a lap elkérései között nem változnak: mint például a lapon használt szkript nyelv, a munkamenet figyelemmel kísérésének az igénye, stb.

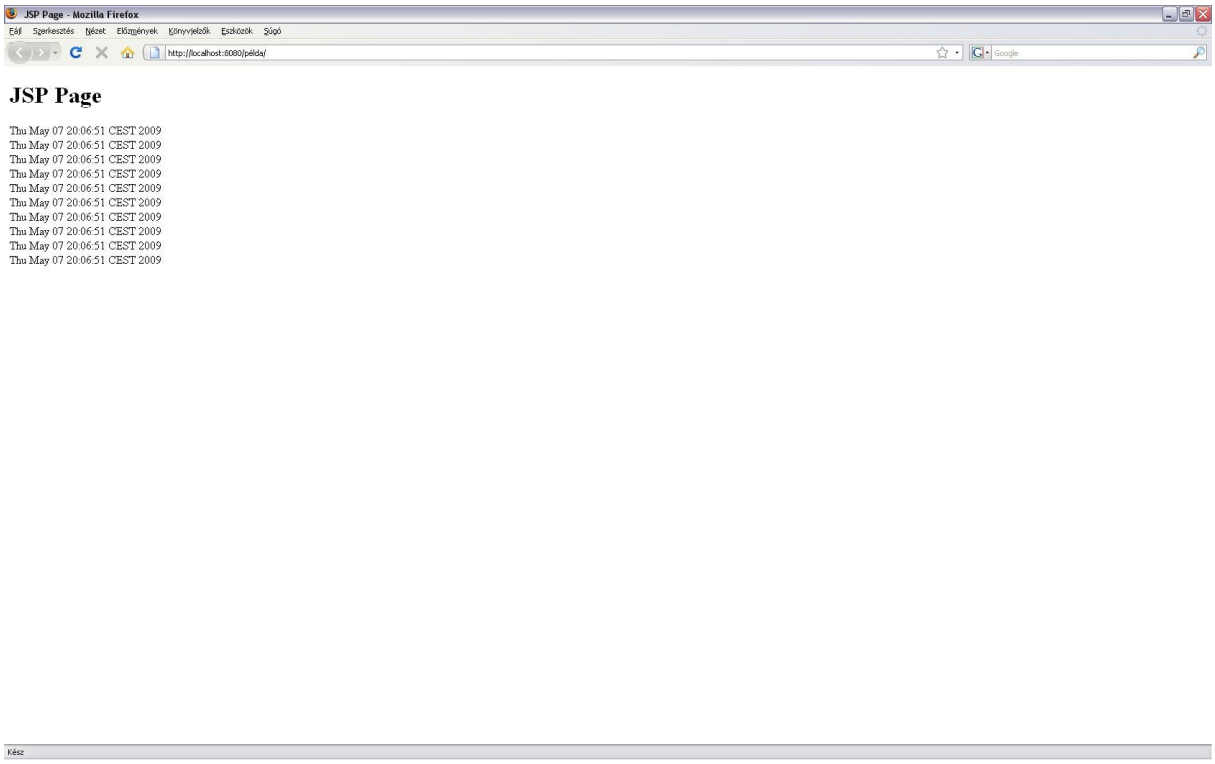
- Akcióelemek: valamilyen akciót hajtanak végre azon információk alapján, amelyek pontosan akkor állnak rendelkezésre, amikor az ügyfél elkéri a JSP lapot. Így például egy akcióelem az ügyféltől kapott paraméterek alapján keresést hajthat végre egy adatbázisban, vagy dinamikusan létrehozhat egy HTML táblázatot, amelyet valamilyen külső rendszerből beolvasott adatokkal tölt ki.
- Szkript elemek: segítségével rövid kódokat vehetünk fel egy JSP lapra, így például egy if utasítást, vagy egy for ciklust, vagy bármilyen más Java kódot.

Terjedelmi okok miatt nem szeretnék kitérni az egyes elemek bemutatására, részletes leírás olvasható az irodalomjegyzék [1],[2] sorszámú forrásaiban.
Helyette nézzünk inkább egy példát, ami nem életszerű, de jól szemlélteti az egyes elemeket. A példa annyit csinál, hogy az aktuális napi dátumot 10-szer kiírja egymás alá formázatlanul.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Page</title>
</head>
<body>
      <h1>JSP Page</h1>
      <jsp:useBean id="date" class="java.util.Date" scope="page"/>
            <% for (int i=0; i<10; i++) { %>
      <%= date %><br/>
      <% } %>
</body>
</html>
```

16. példa: JSP oldalakon szereplő elemek megadása

A 16. példa sikeres lefuttatása után a következő képernyőt kell kapni:



5. ábra: 16. példa sikeres lefuttatása után

A 16. példában látható, hogy a JSP esetében a HTML kód tartalmazza a java kódot a szervletekkel pont fordítva történt és ezzel nehézkesen kezelhető kód keletkezett.

Ha JSP alapokon szeretnénk készítni valamilyen alkalmazást, akkor a fájlszerkezetet hasonlóan kell kialakítani, mint a szervleteknél. A fejlesztés során nem célszerű minden réteget egybe sűríteni, inkább alkalmazzuk az MVC (Model-View- Controller) tervezési mintát, akkor a JSP a view részben fog megjelenni a controller szerepét a szervlet fogja ellátni a modell, pedig lehet egy sima JavaBean. Az alkalmazásban minden kérést célszerű külön szervlethez küldeni nem pedig egy szervletben vizsgálni, hogy melyik kéréshez is tartozik. A kérés címe és a hozzátartozó szervlet összerendelését egy XML leíróban tudjuk megadni. Fejlesztés közben felmerülhet az a gondolat, hogy ne saját magunknak kelljen minden kéréshez tartozó szerveltet példányosítva feltölteni egy kollekcióba, hanem valamilyen konfigurációs állományból automatikusan elvégezhető legyen, illetve miért kell újra és újra létrehozni egy szervletet mikor annak mindig ugyanazt kell csinálnia? Többek között ezek azok a kérdések, amelyek miatt jobb technológia után kell néznünk, még akkor is, ha nem hiányoljuk azt a tengernyi kézre álló funkciót, amit más technológiák kínálnak, és a JSP nem.

5.9. JSF

A JavaServer Faces (JSF) keretrendszer 2004 márciusában jelent meg és egy lényegesen magasabb szintű eszközt adott a webfejlesztők kezébe, mint amilyen a JSP és a szervletek voltak. A JSF bizonyos értelemben a webes alkalmazások fejlesztését az asztali alkalmazások fejlesztéséhez hasonló módon teszi lehetővé. Fejlesztés során bizonyos állapottal rendelkező felületi elemeket, úgynevezett komponenseket használhatunk, amelyekhez az asztali alkalmazásoknál megszokott módon eseménykezelőt rendelhetünk.
Az adatok megjelenítésében és kezelésében számos új lehetőséget biztosít a JSP-hez képest. Adatok tárolására most is használhatunk JavaBean-eket, azonban nem kell a HTTP requestekben keresgélni, hogy az adatokhoz hozzáférjünk. Egy űrlap elemeit képes elmenteni egy JavaBean-ben úgy, hogy automatikusan végrehajtja a szükséges adatkonverziókat, miközben beépített lehetőségeket biztosít az adatok validálására is.
A felületen a megjelenítésnél és a szerveroldalon az adatok feldolgozásnál is egyaránt bean-ekben gondolkozhatunk, ami lényegesen megkönnyíti a fejlesztést, illetve a későbbiekben a program karbantartását.

A JSF használata során nem kell szervletet írnunk ahhoz, hogy megvalósítsuk az MVC tervezési mintát, mivel a JSF már készen tartalmazza azt a szervletet, amely a kéréseket feldolgozza. Fejlesztés során így csupán az a dolgunk, hogy megfelelően konfiguráljuk ezt a készen kapott szervletet és implementáljuk az üzleti logikát. A konfigurációt alapértelmezetten a faces-config.xml-ben végezhetjük el, ahol mindenképpen meg kell adnunk azokat a bean-eket, amelyekkel az alkalmazásnak dolgoznia kell.
Természetesen a JSF nagyon kis részéről esett szó, de látható, hogy a JSP-nél felmerült jó lenne, ha igényekre mindenképpen megadja a megfelelő válaszokat illetve eszközöket. Nem kell szervleteket készíteni, elég csupán konfigurálnunk a JSF által biztosított FacesServlet-et, és így elég közvetlenül üzleti logikára koncentrálnunk.
Bár sok helyen keretrendszerként emlegetik, de így első tanulmányozási körre nem érzem, hogy valamilyen egységes egész keretet adna egy alkalmazás kifejlesztéséhez inkább a megjelenítésben szánnék neki komolyabb szerepet.

5.10. MVC

Már többször esett szó az MVC-ről (Model-View-Control), de eddig nem fejtettem ki, hogy pontosan mi is ez.

Az MVC modellt először a Xerox írta le a Smalltalk nyelvvel kapcsolatos különböző tanulmányaiban. Azóta azonban számos más, népszerű programozási nyelven megírt GUI alkalmazás átvette ezt a modellt. Az MVC minta nem csak fejlesztési szinten értelmezhető, hanem architektúrális szinten is. A minta alap gondolta, hogy az alkalmazás adatait, az üzleti logiját, az adatok megjelenítését, valamint ezek kapcsolatát három önálló egységre bontja, amelyek Modellezés (Model), Megjelenítés (View) és Vezérlés (Controller). Az ilyen fajta szétbontás rugalmassá teszi alkalmazást, mert egy az adatokat többféle módon lehet megjeleníteni és megjelenítésük könnyen módosítható. Az üzleti logika vagy az adatok fizikai megvalósítása (modellezés) pedig anélkül változtatható, hogy hozzá kelljen nyúlni a felhasználói felület kódjához.

6. Spring

Spring keretrendszer egy nyílt forrás kódú alkalmazásfejlesztő keretrendszer a Java platform és a .Net Framework számára. Az első változatot Rod Johnson készítette és publikálta Expert One-on-One J2EE Design and Development című könyvében. 2003-ban adták az első változatot az Apache 2.0 license keretein belül. Az első mérföldkő a Spring életében az 1.0-as kiadás volt 2004 márciusában, majd további kiadások követték 2004 szeptemberében és 2005 márciusában. Majd 2006-ban 1.2.6 elnyerte a Jolt productivity díjat. Jelenlegi aktuális változata a 2.5.6.
A Spring célja leegyszerűsíteni a különböző üzleti alkalmazások kifejlesztését, így számos részből áll, de a teljes Spring ismeretére nincs szükség egy webalkalmazás kifejlesztéséhez.

Spring részei:

- Inversion of Control konténer
- Aspektus orientált programozás
- Adatelérés
- Tranzakció menedzselés
- MVC
- Távoli elérés
- Jogosultságkezelés
- Távoli mendzselés
- Üzenetküldés
- Test

Először Inversion of Control konténer működését nézzük meg részletesen, mert ez a konténer az alapja az összes többi modulnak, hogy miképpen azt majd látni fogjuk a későbbiekben. Következő nagyobb egységben az MVC-vel foglalkoznék, mert nagyobb alkalmazásfejlesztés során biztos, hogy előjön ez a tervezési minta és mindenképpen előnyös lehet, számunkra, hogy a Spring minként is valósítja meg ez a mintát. A végén pedig pár gondolatban az

adatelérést tekintenék át, mert ennek a résznek is fontos szerepe van egy alkalmazásfejlesztése közben.

6.1. Inversion of Control

Az Inversion of Control konténer (röviden IoC konténer) a Spring alap modulja, gyakorlatilag semmit sem tudnék csinálni ennek ismerete nélkül. Az IoC nem Spring sajátosság, hanem egy tervezési minta, amelyet számos más keretrendszer is használ, ezért ismerete mindenképpen hasznos, ha üzleti alkalmazásokat fejlesztünk. Népszerűségét Martin Fowler cikkének köszönheti.

Egy normál program készítése során a fejlesztő számtalan sok könyvtár függvényt hív meg, ezzel szemben az IoC lényege az, hogy a különböző könyvtári függvények hívják meg a fejlesztő által írt függvényeket. Ehhez hasonló működéssel már találkoztunk a JSF-nél, mikor a faces-config.xml-ben megadjuk az alkalmazás közben használt bean-eket. Azáltal, hogy megadjuk a XML konfigurációs állományban a bean-t a JSF képes beállítani az ott megadott paraméternek megfelelően. Hogyan is tudja ezt a műveletet elvégezni? Úgy, hogy a JavaBean konvenció szerint minden adattagnak kötelezően létezni kell set metódusnak és ezt képes meghívni JSF. A paraméterek beállítására a különböző IoC konténerek leggyakrabban a set metódusok segítségével végzik, számos konténer támogatja a konstruktoron keresztüli befecskendezést is.

Az IoC-nek természetesen nem, csak azaz előnye, hogy inicializálhatom vele a bean-jeimet, segítségével konfigurációs állományokon keresztül a kód újrafordítása nélkül befolyásolhatom az alkalmazás működését úgy, hogy kicserélhetem az üzleti logikák implementációját is.

6.1.1. A bean-ek konfigurálása

A JSF esetében az volt látható, hogy egy konfigurációs állományon keresztül megadtuk, hogy milyen objektumokat szeretnénk használni az alkalmazásban és azt is, hogy hogyan kell azokat inicializálni.

A Spring nagyon hasonló lehetőségeket kínál, azzal a különbséggel, hogy a konfigurációs állomány feldolgozására számos megoldást biztosít. Természetesen ezek a

megoldások egy jól meghatározott alapműködés köré csoportosulnak, így mindegyik megoldásban közel azonos dolgok történnek.
Ez az alapműködés az org.springframework.beans.factory. BeanFactory interfészben van deklarálva. A BeanFactory interfész egyik legegyszerűbb és leggyakrabban használt megvalósítása az org.springframework.beans.factory.xml.XmlBeanFactory osztály. Használta rendekül egyszerű:

```
BeanFactory factory = new XmlBeanFactory(new
FileSystemResource("c:/beans.xml"));
```

17. példa: BeanFactory objektum legyártása

A konfigurációs állomány beolvasása után nem történik meg a bean-ek inicializálása, ez csak akkor történik meg mikor egy konkrét bean-re szükségünk van. Egy bean elkérése a Bean-Factory-tól a következőképpen történik:

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

18. példa: Egy bean példányának az elkérése a BeanFactory objektumtól

Egyszerű alkalmazásoknál, az itt leírtaknál többre nincs is szükség az XmlBeanFactory használata éppen megfelelő, ha azonban jobban ki akarjuk használni a Spring Framework által nyújtotta lehetőségeket, akkor érdemes megismerkedni az org.springframework.context. ApplicationContext interfésszel, illetve az azt kiterjesztő osztályokkal. Az ApplicationContext interfész a BeanFactory interfész leszármazottja, így minden olyat tud, amit a BeanFactory de néhány új hasznos lehetőséget is biztosít:

- Eszközt biztosít szöveges üzentek feldolgozására, ezáltal könnyen nemzetközisíthetjük az alkalmazásunkat.
- Egyszerűbb módot biztosít a fájlok eléréséhez.
- Közzétehetünk eseményeket, amelyek figyelésére eseményfigyelőket regisztrálhatunk.

ApplicationContext interfésznek is több megvalósítása létezik, ezek közül a legáltalánosabban használtak a ClassPathXmlApplicationContext, amely a konfigurációs állományt a class path-ból olvassa be, a FileSystemXmlApplicationContext, amely a fájlrendszerből próbálja beolvasni a konfigurációs állományt és a XmlWebApplicationContext, amely a webalkalmazásoknál használható. A konfigurációs állomány beolvasása bármelyik esetben a következő egyszerűséggel megtehető:

```
ApplicationContext context = new
FileSystemXmlApplicationContext("c:/config.xml");
```

19. példa: Egy ApplicationContext objektum legyártása

Egy bean elérése a BeanFactory-nál leírt getBean metódussal lehetséges. A bean-ek inicializálása azonban nem első eléréskor történik, hanem a konfigurációs állomány feldolgozásakor, így az alkalmazásnak már nem kell várnia a beanek legyártására.
A webalkalmazás fejlesztés során nem kell bajlódnunk a konfigurációs állományok feldolgozásával és a bean-ek elérésével, mivel a JSF-hez hasonlóan a Spring is tartalmaz előre elkészített szervletet, amelyben már implementálták a bean-ek kezeléséhez szükséges logikát. A Spring több ilyen szervletet is tartalmaz, de ezek közül a legáltalánosabban használt a DispatcherServlet.

6.1.2. A bean-ek legyártása

A Spring alapértelmezettként a DispatcherServlet-et ajánlja, én nem szeretnék ettől eltérni így a további bemutatás során is ezt fogom használni. Megfelelő szervlet választás utána már csak egy konfigurációs állományra lesz szükség. Ezt a DispatcherServlet alapértelmezettként a dispatcher-servlet.xml fájlból próbálja meg beolvasni a konfigurációhoz szükséges utasításokat, amelyet a WEB-INF könyvtárban keres. A konfigurációs állomány felépítése a következő képen nézhet ki:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                                  http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

<beans>
...
                                       <bean          name="LoginController"
class="controller.Simple.LoginController">
        <property name="formView" value="login" />
        <property name="successView" value="base"/>
        <property name="commandName" value="loginData" />
        <property name="commandClass" value="bean.LoginData" />
        <property name="validator" ref="loginValidator"/>
    </bean>
...
</beans>
```

20. példa: Dispatcher-servlet.xml használata

Mit is jelentenek az ehhez hasonló bejegyzések a konfigurációs állományban? A <beans> gyökér tag-ek között tudjuk felsorolni az összes olyan bean-t, amelyet a Spring IoC konténeren keresztül szeretnék legyártatni. Minden egyes bean megadás a <bean> tag-ekkel történik, amelynek két kötelező attribútuma van az egyik az id, amely azt mondja meg, hogy milyen néven szeretnék létrehozni a bean-t, másik pedig a class azzal azt tudjuk megadni, hogy milyen típusú legyen az a bean. Az így létrehozott objektum példányváltozóit egyrészt a <property> tag-ek segítségével állíthatjuk be, ebben az esetben az IoC konténer a bean set metódusait használja a változók beállítására. Másik lehetőség a konstruktoron keresztüli inicializálás, ekkor <constructor-arg> tag-eket kell használni a <property> tag-ek helyett.

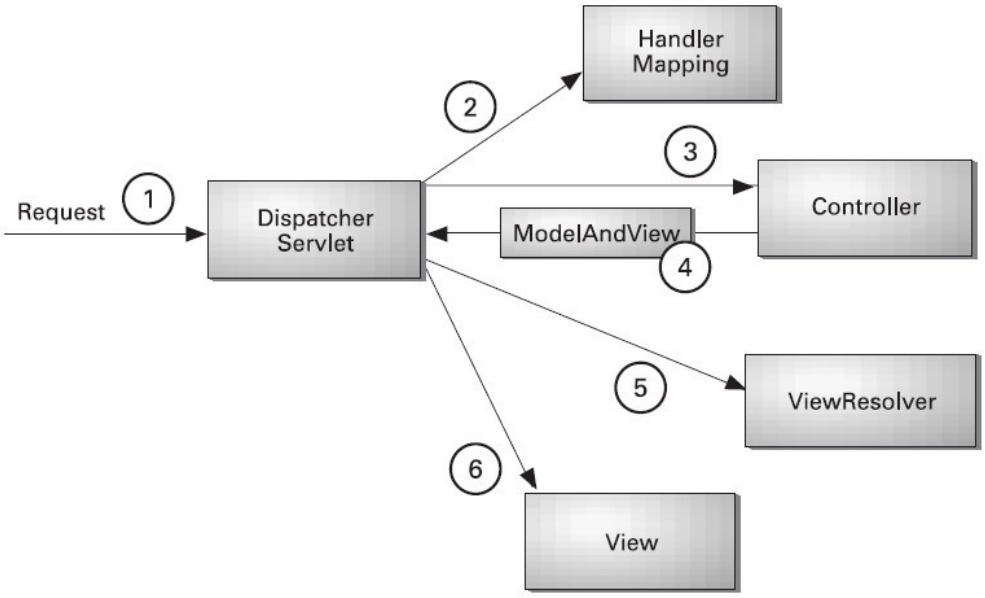
Eddig még semmit sem mondtam arról, hogy az IoC konténer hányszor hozza létre ezeket a beaneket az alkalmazás futása során. Valójában, ha másként nem rendelkezünk a Spring mindig singletonokat hoz létre, ami azt jelenti, hogy a bean-nek csak egyetlen példányát gyártja le és ezt az egy példányt használjuk az alkalmazás működése során. Ez nem mindig hasznos, gondoljunk csak arra példára mikor a felhasználóknak tároljuk a címét egy külön objektumban és nem lenne szerencsés ha minden felhasználóhoz ugyanaz a cím objektum tartozna, ezért a konfigurációs leírásban megadhatjuk, hogy hogyan akarjuk példányosítania bean-jeinket. Ehhez a <bean> tag scope attribútumát kell használni. Az

attribútum a következő értékeket veheti fel: singleton, prototype, request, session, global-session.
Ha a scope attribútumnak a prototype értéket adjuk, akkor a BeanFactory getBean metódusának minden egyes meghívásakor a bean egy új példánya jön létre, vagy is biztosítható, hogy minden felhasználó objektumhoz másik cím objektum tartozzon. A request és session értékek csak akkor adhatóak meg a scope attribútumnak, ha webalkalmazást fejlesztünk, ekkor egy HTTP request, vagy egy HTTP session objektum fog létrejönni a getBean metódus meghívódásakor. A global-session érték csak portletek esetén használható.

6.2.Spring MVC

Korábban már volt szó az MVC tervezési mintáról résztelesen ki is fejtettem, hogy mit is takar valójában ez a rövidítés. Természeten, ha napjainkban akarunk valamilyen alkalmazást fejleszteni célszerű ezt választani a könnyebb fejlesztés és karbantartás végett. De ha nem is akarom így fejleszteni előbb vagy utóbb úgy is valamilyen eszköz rám fogja kényszeríteni a minta alkalmazását. A Spring sem tesz másképen ő is megvalósítja az MVC-t. Továbbiakban azt mutatom be majd meg, milyen módon teszi azt.

A JSP és JSF ismerkedés közben is tapasztaltuk már, hogy a vezérlést egy szervlet látta el. A Spring esetében is így történik, de annyival változik a szervlet feladata, hogy neki csak a megfelelő vezérlőt kell kiválasztania. A megfelelő vezérlő kiválasztásában valamilyen HandlerMapping objektumra van szükség. A kiválasztott vezérlő minden estben egy Controller objektum, amely minden esetben egy request és respone objektumot kap paraméterül. A vezérlő a feldolgozás eredményeképpen előálló információkat, a szükséges megjelenítési információkkal egy ModelAndView objektumba csomagolja és ezt az objektumot visszaküldi a szervletnek. A szervlet a ModelAndView objektumban tárolt megjelenítési információt, ami egy logikai név átadja egy ViewResolver objektumnak, amely előállítja azt a View objektumot, amely a kívánt információkat megjeleníti.



6. ábra: Spring MVC

6.2.1. A vezérlő kiválasztása

A webalkalmazásom készítése közben mindenképpen meg kellett ismerkednem közelebbről a fent említett komponensekkel. Annyit említettem csak az előbb, hogy a megfelelő vezérlő kiválasztásához valamilyen HandlerMapping objektumra van szükségem. A valóságban azonban több HandlerMapping implementáció is létezik következőkben csak párat mutatnék be.

SimpleUrlHandlerMapping: Ez az egyik legkézenfekvőbb megoldás ami létezik. Egyszerűen annyi a dolgom csak, hogy URL-ekhez kötjük, hogy melyik vezérlőt is akarok majd használni. Mit is jelent ez? A webalkalmazásunkban minden link egy URL-re mutat, és minden <form> tag rendelkezik egy action attribútummal, amely szintén egy URL-t tartalmaz. Minden egyes ilyen URL-hez a konfigurációs állományban hozzárendelek egy vezérlőt, így a szervlet tudni fogja, hogy az adott űrlap feldolgozásakor, vagy az adott linkre kattintáskor hová kell továbbítani a kérést a további feldolgozás érdekében. Én szintén ezt a módszert használtam a vezérlők kiválasztásához a webalkalmazás fejlesztése közben, így az én alkalmazásomban a konfigurációs állomány így néz ki:

```
<bean id="urlMapping"class="org.springframework.web.servlet.handler
.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="index.htm">indexController</prop>
                <prop key="login.htm">LoginController</prop>
                <prop key="logout.htm">LogoutController</prop>
                <prop key="registration.htm">RegistrationController</prop>
                <prop key="admin.htm">AdminController</prop>
             ...
                </props>
            </property>
    </bean>
```

21. példa: SimpleUrlHandlerMapping használatának megadása

ControllerClassNameHandlerMapping: Nagyon hasonlóan működik az előzőhöz képest, annyi különbséggel, hogy nem nekem kell megadni, hogy melyik URL-hez melyik controllert akarom hozzárendelni, ő magának kikövetkezteti. Logikája nagyon egyszerű, például, egy login.htm-hez egy loginController-t fog majd keresni, azaz fogja login.htm-t levágja a .htm végződést és hozzáragasztja a Controller végződést. Így a korábbi hosszú konfigurációs bejegyzés lényegesen lerövidül a következőre:

```
<bean id="urlMapping"class="org.springframework.web.servlet.mvc.
ControllerClassNameHandlerMapping"/>
</bean>
```

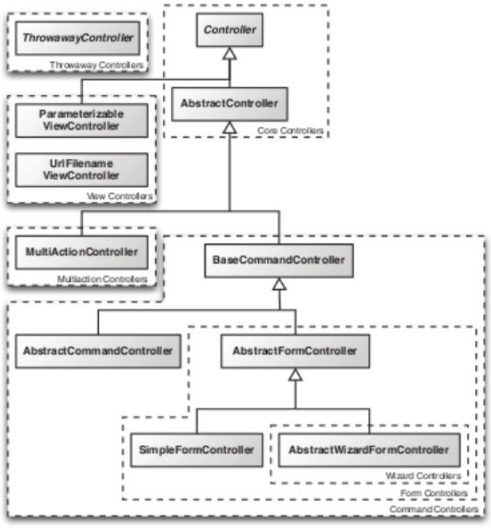
22. példa: ControllerClassNameHandlerMapping használatának megadása

Ha elkezdem tehát valamilyen alkalmazás fejlesztését, akkor első feladatom valamilyen HandlerMapping implementáció kiválasztása. De ha nem tudom eldönteni, melyiket válasszam, akkor sem kell kétségbe esni, mert a Spring ad lehetőséget egyszerre több implementáció keverésére. Ennek mikéntjéről utána lehet nézni az irodalomjegyzék [5] sorszámú forrásában.

6.2.2.Vezérlők készítése

Miután sikerült kiválasztanom, hogy milyen HandlerMappig implementációt szeretnék használni. Következő lépés a megfelelő vezérlők elkészítése. A feladattól függően többféle vezérlőt készíthetek, és ebben a Spring számos absztrakt osztállyal segít. Amint a 4. ábra mutatja ezen osztályok hierarchiája meglehetősen kiterjedt, ezért minden elemének bemutatására most nem is kerül sor, ehelyett megnézzük a két leggyakrabban használtat.

Leggyakoribb eset, mikor egy linkre kattintás nyomán előálló HTTP GET kérést akarunk feldolgozni. Általában ilyen esetben nincs szükség, semmilyen validátor vagy komolyabb eszköz támogatására ezért megfelelő választás számomra, ha a vezérlő osztályokat az AbstractController leszármazottjaként hozzom létre. Az AbstractController absztrakt osztály a handleRequestInternal metódust biztosítja számomra és egyben garantálja, hogy a HTTP GET kérés esetén az osztályunk ezen, metódusa lesz meghívva. A metódus bemenő paraméterei egy HttpServletRequest és egy HttpServletResponse objektum és a visszatérési értéke egy ModelAndView objektum. Használata azonban csak akkor célszerű, ha a GET kérés során nem, vagy csak egy - két request paramétert kell feldolgozni. Amennyiben több kérési paraméter feldolgozására is szükség van, célszerűbb az AbstractCommandController absztrakt osztály kiterjesztése. Ez az osztály a handle metódust biztosítja számomra, melynek az előbbiekhez képest még két plusz bemenő paramétere is van. Egyik az Object bemenő paraméter lehetőséget biztosít az új kérési adatok egy objektumba történő csomagolására, így az adatok ellenőrzése és feldolgozása lényegesen egyszerűbb. A kötés létrehozásához természetesen a konfigurációs bejegyzésben, vagy magában a vezérlő osztály kódjában meg kell adni, hogy mi lesz annak az objektumnak a típusa, amely a kérési paramétereket fogja tartalmazni.



7. ábra: Spring MVC vezérlő hierarchia

Az én alkalmazásomban is kellett olyan linket készítenem, amik különböző szerveroldali műveleteket végeztek el. Ilyen, mikor a felhasználó egy kategóriákba tartozó termékeket szeretné megnézni. Itt lényegében nincs semmilyen bemenő paraméterre szükségem. Nyugodtan lehet választani az AbstractController osztály kiterjesztését, a kérést feldolgozó CookieController osztály elkészítésekor.

```
public class CookieController extends AbstractController {

    @EJB(name="RaktarService")
    private RaktarService raktarService;

          protected ModelAndView handleRequestInternal(HttpServletRequest
request, HttpServletResponse response) throws Exception {
        Map<Long, Raktar> termekek = new HashMap<Long, Raktar>();
       for(Map.Entry e : raktarService.osszes_termek().entrySet()){
             Raktar r = (Raktar)e.getValue();
              if(r.getTermek_id().getKatergoria().getNev().equals("sütemén
yek") && r.getTermek_id().getMegjegyzes().equals(""))
                  termekek.put(r.getId(), r);
       }
        return new ModelAndView("base","model",termekek);
    }
}
```

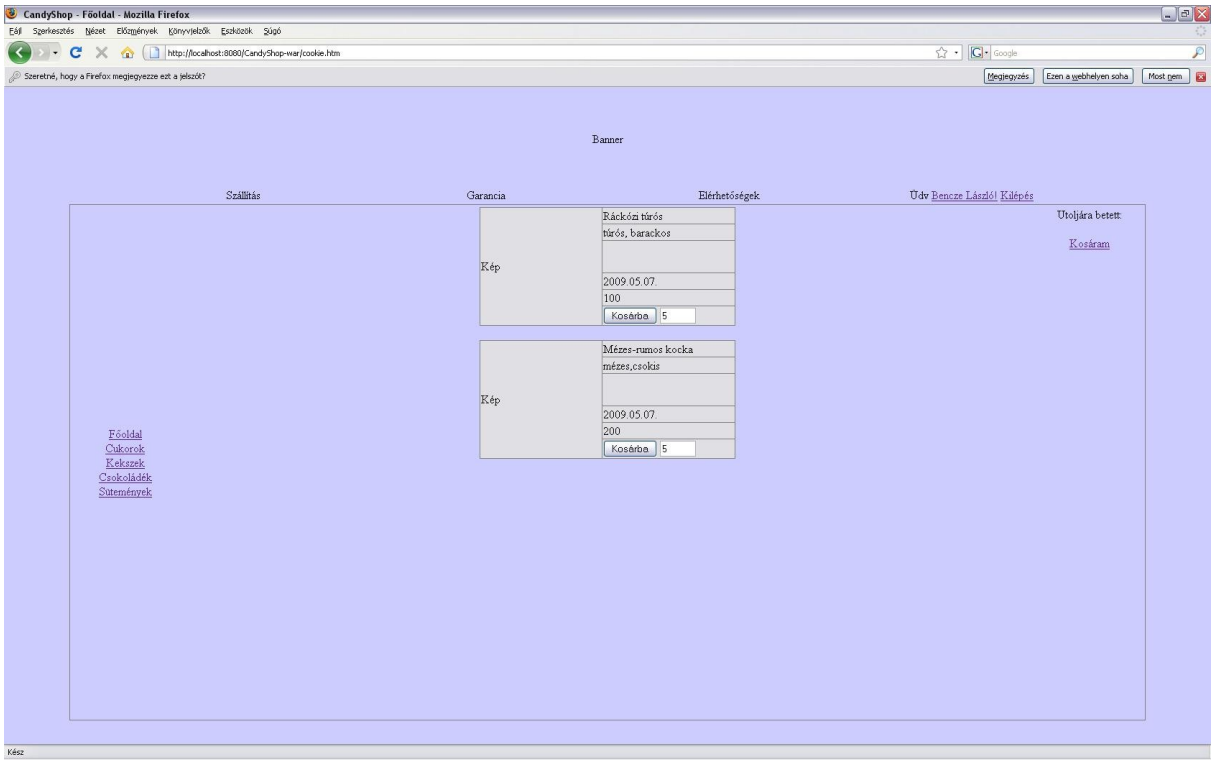
23. példa: Egy AbstractController használata

A vezérlő használatához a következő konfigurációs bejegyzést kellett elkészíteni:

```
<bean name="CookieController"
class="controller.Abstract.CookieController"/>
```

24. példa: Egy AbstractController használata megadása

Ha rákattintunk a sütemények kategóriára akkor a 23. példában látható forrás fog lefutni és a képernyőre kapjuk a következőt:



7. ábra: Sütemények kategória megjelenítése

Ahhoz, hogy megértsük, a példát mindenképpen bővebben meg kell ismeri a ModelAndView osztállyal. Egy ModelAndView osztály osztály konstruktorában, vagy egy view nevet, vagy egy view nevet és egy Map típusú objektumot, vagy pedig egy view nevet, egy modell nevet és egy modell objektumot vár bemenő paraméterként. A View név egy String melynek segítségével a ViewResolver előállítja a megfelelő objektumot, olyan módon, ahogy már az elején is írtam. Ha a betöltendő oldalon semmilyen dinamikus tartalmat nem kell megjeleníteni, akkor a konstruktorban elég egy view nevet megadni.

Ha azonban dinamikus tartalommal kell megjeleníteni, mint az a példában is jól látható, akkor a view név mellett az összetettség függvényében szerepeltetni kell vagy egy Map-et, vagy egy modellnevet és egy modell objektumot. A Map-ben egyébként modell objektumok vannak letárolva, ahol a kulcsok a modell nevek. A modell objektum tetszőleges Java objektum lehet, ezek tárolják a megjelenítendő információkat, az oldalon pedig a modell névvel lehet rájuk hivatkozni. Ezek alapján tegyük fel, hogy a megjelenítendő objektum User típusú, amelynek van String típusú vezetekNev és kerezstNev parameter akkor az oldalon így tudunk rá hivatkozni:

```
...
<body>
${user.vezetekNev} ${user.keresztNev}
</body>
...
```

25. példa: ModelAndView objektum tartalmának megjelenítése

De ez csak akkor igaz, ha user objektumot user névvel tettük elérhetővé a ModelAndView objektumban, azaz az alábbi módon adtuk meg: ModelAndView(viewName,”user”,user);
Webalkalmazás készítése közben előbb vagy utóbb belefutunk egy olyan problémába, hogy űrlapokat kell feldolgoznunk. Az ábrából is kikövetkeztethető, hogy a Spring biztosít erre a feladatra is kész osztályokat. Ezek közül a legcélszerűbb a SimpleFormController használata. Ez az osztály az előzőekben tárgyalt osztályoktól abban tér el, hogy nem absztrakt, hanem már azonnal használható, ha arra van szűkség, hogy a bekért adatokat azonnal visszaküldjük megjelenítésre. Ha ennél komolyabb feladat elvégzését szeretnénk, akkor ki kell terjeszteni és valamelyik onSubmit metódusát felül kell definiálni. Az onSubmit metódusok csupán paraméterezésükben térnek el, legszerényebb verziója csak egy Object típusú objektumot vár bemenő paraméterében, míg legteljesebb verziója ezen felül egy HttpServletRequest, egy HttpServletResponse és egy BindException típusú objektumot is vár. A választásunknak csak attól kell függnie, hogy mire lesz szükségünk implementálás során.

A Spring lehetőség biztosított számomra az adatok ellenőrzésére. A SimpleFormController ezen osztály segítségével fogja elvégezni a beérkező adatok validálását még mielőtt meghívná az onSubmit metódusát. Egy validátor osztály elkészítéséhez a Spring által biztosított Validator interfészt kell implementálni.

Az interfész két metódust biztosít számomra, amelyek közül a validate az érdekesebb, ugyanis a validálást végző osztály ezen metódusát hívja meg a vezérlő osztály. A validátor és SimpleFormControllerre közösen nézzük meg az alkalmazásom bejelentkező űrlapjának feldolgozását:

```
public class LoginController extends SimpleFormController{

protected      ModelAndView      onSubmit(HttpServletRequest      request,
HttpServletResponse response, Object command, BindException errors) throws
NoSuchFieldException {
        HttpSession session = request.getSession();
        LoginData Login = (LoginData)command;

        //login
        Felhasznalo felhasznalo =
felhasznaloService.hitelesit(Login.getLoginName(),
         Login.getPassword());
        if(felhasznalo==null){
            errors.reject("error.login");
            return new ModelAndView(getFormView(),"login",errors);
        }
...
         session.setAttribute("user",felhasznalo);

        return new ModelAndView(getSuccessView(),"model",termekek);
    }
   }
}
```

26. példa: SimpleFormController használata

Az ehhez tartozó konfigurációs bejegyzések a következőképpen néznek ki:

```
                                    <bean          name="LoginController"
class="controller.Simple.LoginController">
        <property name="formView" value="login" />
        <property name="successView" value="base"/>
        <property name="commandName" value="loginData" />
        <property name="commandClass" value="bean.LoginData" />
        <property name="validator" ref="loginValidator"/>
    </bean>

    <bean name="loginValidator" class="validator.LoginValidator"/>
```

27. példa: Egy SimpleFormController használatának megadása

Validátor implementációja:

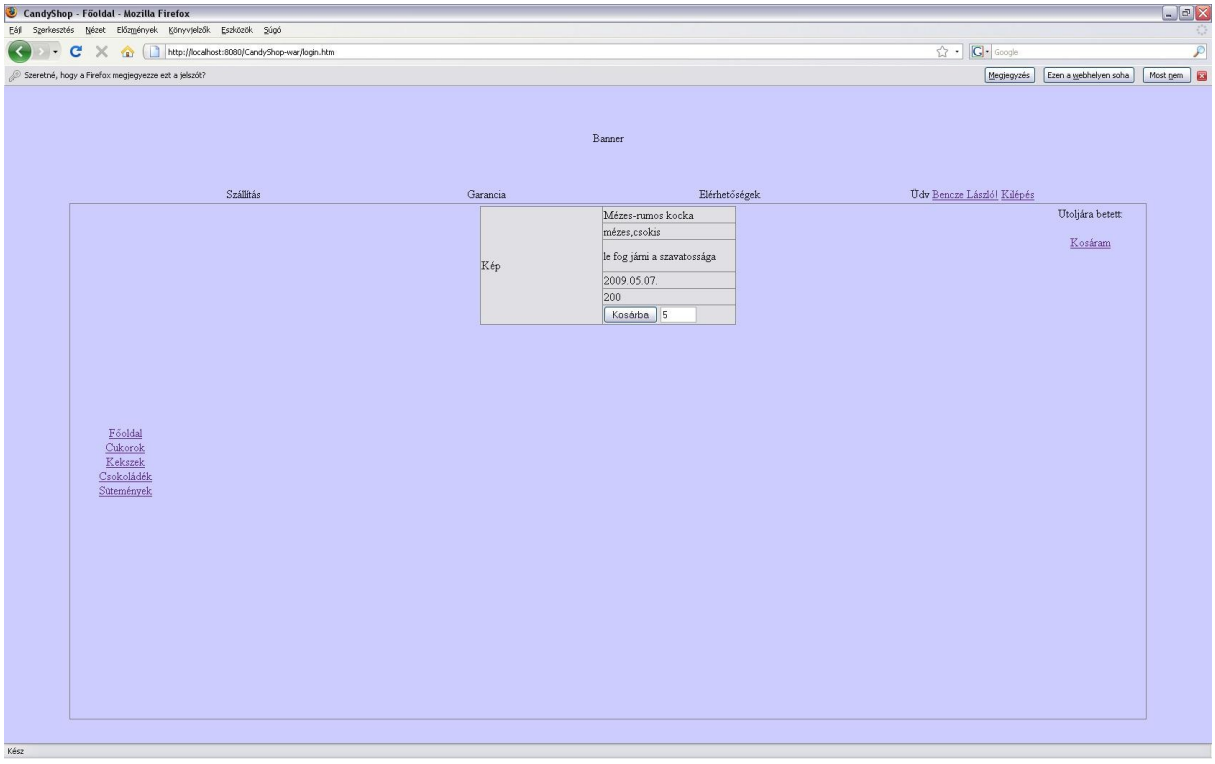
```
public class LoginValidator implements Validator{
```

```
    public boolean supports(Class clazz) {
       return clazz.equals(LoginData.class);
    }
    public void validate(Object obj, Errors errors) {
            ValidationUtils.rejectIfEmptyOrWhitespace(errors, "loginName",
"required", "Field is required.");
             ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
"required", "Field is required.");
    }
}
```

28. példa: Egy validátor használata

A validátor hibát jelez minden esetben, ha a felhasználó név vagy a jelszó űrlap mezők nincsenek kitöltve. Az üres űrlapmezőket a ValidationUtils rejectIfEmptyOrWhitespace metódusával lehet hibaként visszajelezni.

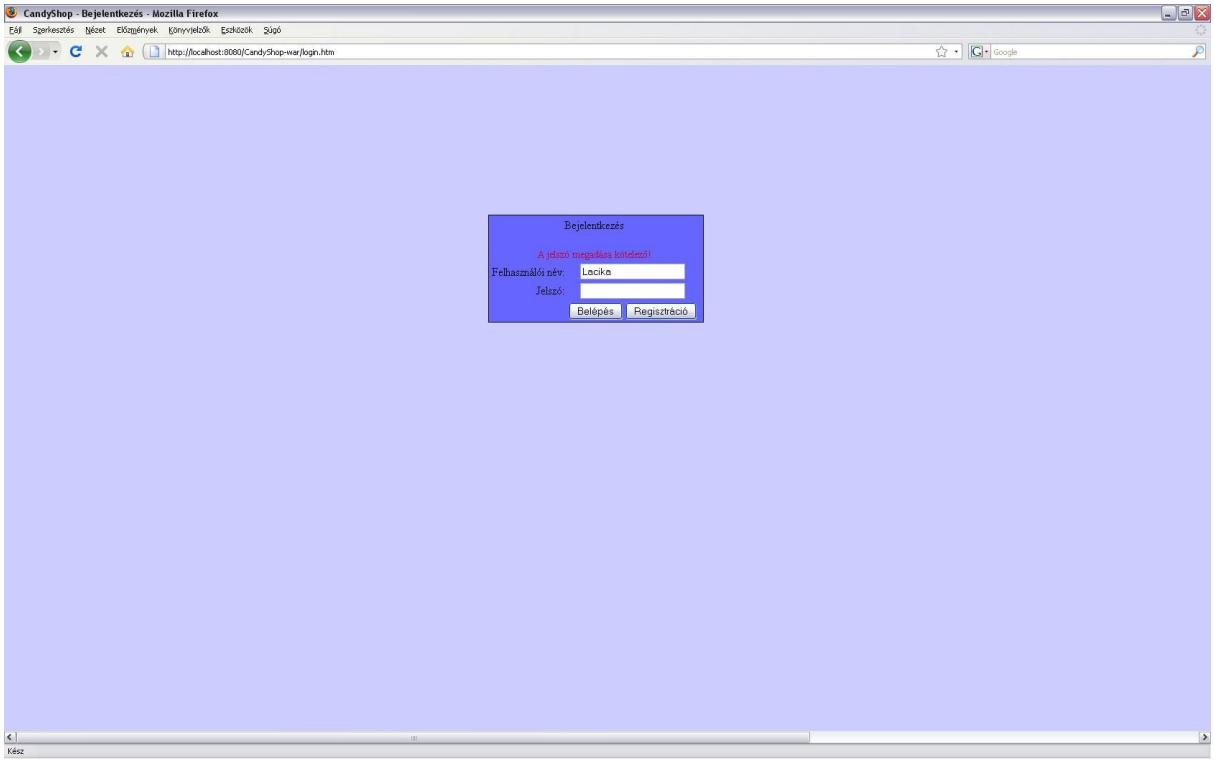
Helyes adatok megadása után a következő képernyőt kell kapnunk:



8. ábra: Belépés utáni képernyő

Ahhoz, hogy a 8. ábrán látható képernyőt kapjuk mindenképpen érvényes adatokat kellet megadni a belépés során. Mikor belépéskor a belép gombra kattintunk akkor a 26-os példában

látható kód fut le. De még mielőtt lefutna LoginController a 28-as példában szereplő validátor fut le és ha valami nincs megadva akkor kapjuk a 9. ábrán látható képernyőt:



9. ábra: Hibás adat megadása után

6.2.3. Megjelenítés

Adatok feldolgozásával meg is vagyunk már csak a megjelenítés maradt. A fejezet elején már esett szó róla, hogy a ModelAndView objektum a vezérlő osztálytól a szervlethez kerül majd innan a ViewResolver-hez. A feladatom tehát az, hogy ismertessem, mi is a ViewResolver és hogyan is működik.

A ViewResolver valójában egy interfész, így ha egy osztály implementája ezt az interfészt, akkor alkalmas lesz, hogy meghatározza egy ModelAndView objektumban visszaadott névből, hogy mi is jelenjen meg a képernyőn. A ViewResolver-ek valójában egy View objektumot állítanak elő, amelynek megjelenítéséről a szervlet gondoskodik. A Spring nem csak a ViewResolver interfészt biztosítja számunkra, hanem számos már előre legyártott implementációt, ezek közül érdemes megismerkedni párra.

Elsőnek bemutatom a legegyszerűbb implementációt ami a InternalResourceViewResolver osztály. Használatához csak annyit kell tennem, hogy be kell állítanom a konfigurációs állományba és továbbá beállítani két fontos példányváltozóját. Mindkét példányváltozó String típusú, az egyik neve prefix, ennek a változónak az értéke határozza meg a megjelenítendő oldal URL-jének előtagját, még a másik a suffix, amelynek értéke az oldal suffixét határozza meg. Az InternalResourceViewResolver a beállított prefix és suffix-ekből elvégzi a következő konkatenációt:

URL = prefix + viewNev + suffix

Az így megkapott URL információt beállítja a View objektum path változójának értékeként, majd ezt az objektumot átadja a szervletnek megjelenítésre. Ahhoz, hogy használhassam a InternalResourceViewResolver-t a következő bejegyzést kell elvégezni a konfigurációs állományban.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />
```

29. példa: Az InternalResourceViewResolver konfigurációja

Az így konfigurált InternalResourceViewResolver, ha azzal a névvel találkozik, hogy login akkor előállítja a következő URL címet: /WEB-INF/jsp/login.jsp, vagyis ez lesz az oldal, amelyik végül a böngészőbe be fog töltődni.
Az InternalResourceViewResolver nyilvánvalóan mindig azonos típusú view objektumokat fog gyártani, hogy pontosan milyet, az beállítható a viewClass példányváltozóján keresztül. Előfordulhat azonban, hogy az alkalmazásnak egy bizonyos ponton nem csak HTML kimenetet kell tudni előállítani, hanem XML-t, PDF-et, vagy bármi mást. Ilyen feladat esetén jól jön egy olyan ViewResolver osztály, amely tetszőlegesen sokféle view-t tud előállítani.
A BeanNameViewResolver pontosan ilyen feladatok ellátására képes. A ModelAndView objektumban kapott view nevet nem egy URL összeállításra fogja felhasználni, hanem a konfigurációs állományban olyan bean definíciót keres, amely bean a view névvel azonos néven lett létrehozva. Ha tehát a view név mondjuk info, akkor a következő bean-t fogja megkeresni a konfigurációs állományban:

```
<bean id="info" class="pelda.InfoView"/>
```

30. példa: BeanNameViewResolver használata

Gyakorlatilag ugyanilyen megfontolásból lehet hasznos a XmlFileViewResolver, amely egy külön megadott XML fájlból olvassa ki a megjelenítendő view objektumot, illetve a ResourceBundleViewResolver, amely ugyanerre egy properties fájlt használ.
Ezeken kívül még létezik számos ViewResolver implementációs osztály, de mi is készíthetünk magunknak sajátot. Amit még célszerű megemlíteni, hogy ennél lehet keverni a különféle megoldásokat, mint a HandlerMapping objektum esetén.
Megjelenítés kérdéséhez tartozhat még, hogy milyen template-et válasszunk a weboldalak megjelenítéséhez. A Spring ebben a kérdésben is elég engedékeny, mert bármilyen template-t választhatunk. Én még korábbról ismereteimből a JSP-t választottam, de most, hogy alkalmam volt kicsit belekóstolni a JSF-be így további tanulmányozás után mindenképpen azt fogom előnyben részesíteni.

6.3. Adatelérés

Mikor elkezdtem tervezni valamilyen alkalmazást mindenképpen tisztázni kellett, hogy hogyan is szeretném elérni majd az adatainak az alkalmazásfejlesztés, illetve használatközben. Mikor nálam is felmerült ez a kérdés megvizsgáltam, hogy a Spring milyen eszközöket biztosít a számomra és hogy azok a későbbi fejlesztéseim során is használhatók-e. Továbbiakban arról írnék, hogyan milyen eszközöket biztosít a Spring és hogy én mit is választottam és hogy miért.

Spring a prezisztenciát nem saját erejéből oldja meg, hanem egy külső eszközre támaszkodik, annyit segít, hogy minden eszközhöz biztosít csomagoló osztályokat. Nekünk annyi a dolgunk, hogy valamelyik eszközt kiválasszuk és hozzá a megfelelő csomagoló osztályt.

Joggal merülhet fel a kérdés, ha a Spring is nyújt adateléréshez segítséget, akkor miért kellett feszegetni az Enterprise JavaBean-eket, miért nem volt jó Spring által nyújtott lehetőségek?
A válasz, pedig abban keresendő, hogy gondolnom kellett arra is, hogy a későbbiek során más alkalmazások számára is könnyen hozzáférhetővé kell tennem az adatbázisban tárolt adatokat, akár olyan alkalmazások számára is, amelyek nem a Spring keretrendszer segítségével lettek létrehozva.
Ha pedig mindezt a lehető legegyszerűbb módon szeretném elérni, akkor kétségtelenül akad erre lényegesen jobb megoldás is, mint azt láttuk az EJB-nél.
EJB egy külső eszköz, ezért közvetlenül nem használhatom az @EJB annotációt, mert az IoC konténer nem fogja számomra megtalálni és beinjektálni megfelelő EJB objektumot. Ha használni szeretém az annotációt, akkor következő bejegyzést, kell elvégezni a konfigurációs állományban:

```
                                                                    <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPostProc
essor">
          <property name="alwaysUseJndiLookup" value="true" />
   </bean>
```

31. példa: EJB annotációk használatának megadása

Ezzel a bejegyzéssel, már létre jönnek a megfelelő objektumok, de a Spring közvetlenül nem tudja megtalálni, hogy nekem milyenre is van szükségem, ezért @EJB annotáció name attribútumára megadására van szükségem. Amellyel megmondhatom, melyik EJB objektumra is van szükségünk.

Ezen rövid leírás alapján leszűrhetjük, hogy a Spring egy kitűnő választás lehet webalkalmazások készítésére, hiszen amint láthattuk beépített eszközeivel világos, jól strukturált webalkalmazások hozhatók létre akkor is, ha kezdő programozóként kevés tapasztalattal rendelkezünk ahhoz, hogy mások számára is jól érthető, megfelelően szervezett kódot készítsünk. Mindezek mellet kiterjedt eszközkészlete azt is lehetővé teszi, hogy úgy hozzunk létre alkalmazásokat, hogy nem kell feltétlenül kilépnünk a Spring világából.

Befejezés

Egy webalkalmazás elkészítése hihetetlenül mélyreható és komplex feladat. Modellezésnél már bemutattam, hogy egy UML diagram elkészítéshez is milyen alapos ismeretekre van szükség, mint például nem mindegy, hogy egy nyíl milyen irányba mutat vagy az adott vonalon milyen címke szerepel. Ezek apró változások, de jelentésük teljesen más és lehet, hogy ezzel befolyásolni tudják a project kimenetelét, majd az eszközök kapcsán csak tovább erősödött ez a tény, igaz részletesen nem mentünk bele, egy EJB komponenshívásba vagy egy entitás leképezésébe. Kisebb projectek esetén lehet nincs is szükség ezek ismeretére, de egy nagyobb és összetettebb rendszer esetén mindenképpen célszerű ismerni mi is történik a háttérben valójában.

A bevezetőben azt a célt tűztem ki magam elé, hogy a dolgozat megírásával nagyobb rálátásom lesz a webalkalmazások modellezésére és fejlesztésére. Én úgy érezem, hogy ezt a kitűzött célt sikerült teljesítenem, de semmiféleképpen nem szabad hátra dőlnöm, mert folyamatosan fejlődnek a technológiák vagy újabb és újabbak jönnek ki, amiket egy fejlesztőnek mindenképpen illik ismerni.

Egy-egy project befejeztével szerintem minden fejlesztő úgy érzi, hogy bizonyos dolgokat másképp kellett volna csinálni, más eszközöket is használhatott volna. Nálam sincs ez másként, úgy érzem, hogy az XML-t jóval több helyen kellett volna használni, nem csak a konfigurációs állománynál. Szerintem az XML technológiában több rejlik, amivel még jobban és gyorsabban lehet alkalmazásokat fejleszteni, illetve gyorsítani lehet a szoftverfejlesztés egyes lépéseit. Továbbiakban, ha lesz rá lehetőségem, akkor mindenképpen törekedni fogok az XML más területen való felhasználására.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Adamkó Attila Tanár úrnak, hogy elvállalta a témavezetésemet és hasznos tanácsokkal és észrevételekkel segítette a szakdolgozatom megírását.

Irodalomjegyzék

[1] Balogh Péter – Berényi Zsolt – Dévai István – Imre Gábor – Soós István – Tóthfalussy Balázs: Szoftverfejlesztés Java EE Platformon, SZAK Kiadó Kft., Budapest, 2007

[2] Nyékyné Gaizler Judit: J2EE útikalauz Java programozóknak, ELTE TTK Hallgatói Alapítvány, Budapest, 2002

[3] Adamkó Attila: Webes Információs Rendszerek Modellezése.Debrecen, Informatika a felsőoktatásban 2008 (http://www.agr.unideb.hu/if2008/kiadvany/papers/C82.pdf)

[4] Harald Störrle: UML2 für Studenten, Pearson Studium, 2005, magyar fordítása Adamkó Attila - Bicskey Simon - Espák Miklós: UML2, Panem, 2007

[5] Rod Johnson - Juergen Hoeller - Alef Arendsen: The Spring Framework - Reference Documentation, 2004-2008

Java EE tutorial: http://java.sun.com/javaee/5/docs/tutorial/doc/

[5] World Wide Web Consortium (W3C) honlapja - http://www.w3.org/

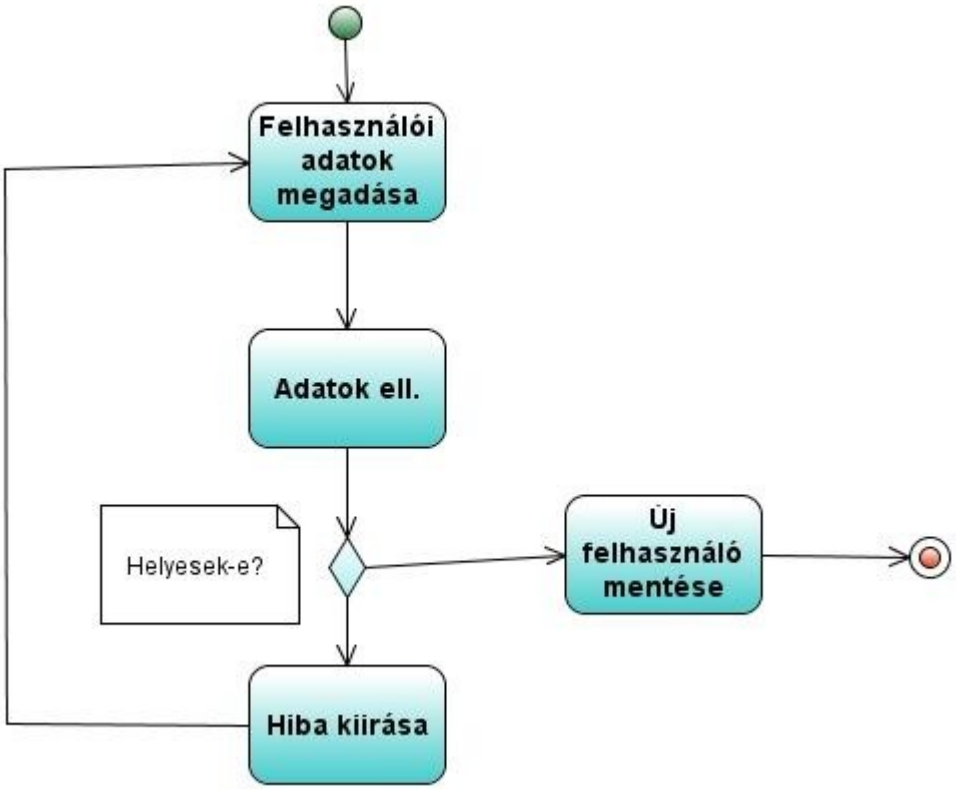
Prog.hu XML cikksorozat: http://www.prog.hu/cikkek/884/Bemutatkozik+az+XML.html

[6] Spring Keretrendszer honlapja - http://www.springsource.org/

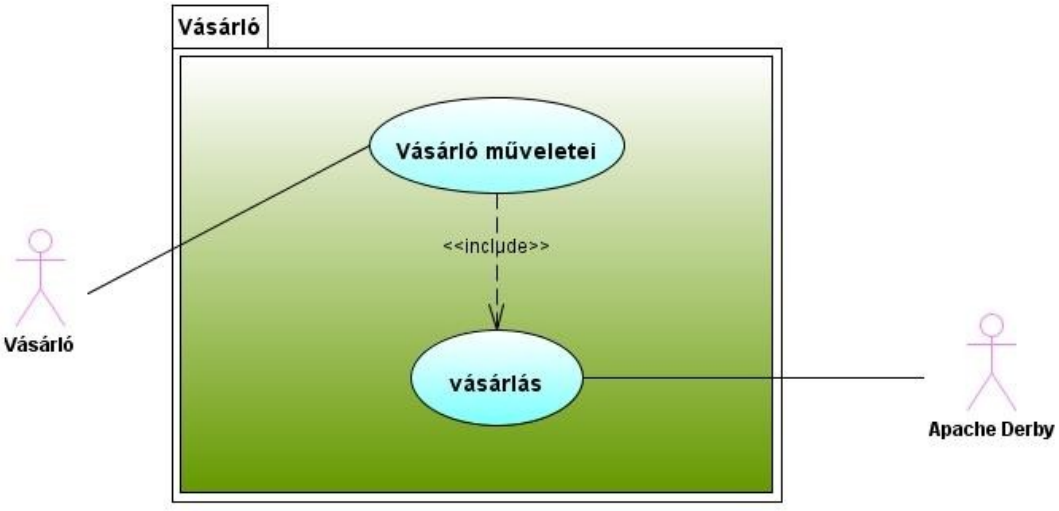
[7] Spring Wikipédia bejegyzése - http://en.wikipedia.org/wiki/Spring_Framework

[8] Oracle Toplink honlapja:
http://www.oracle.com/technology/products/ias/toplink/index.html

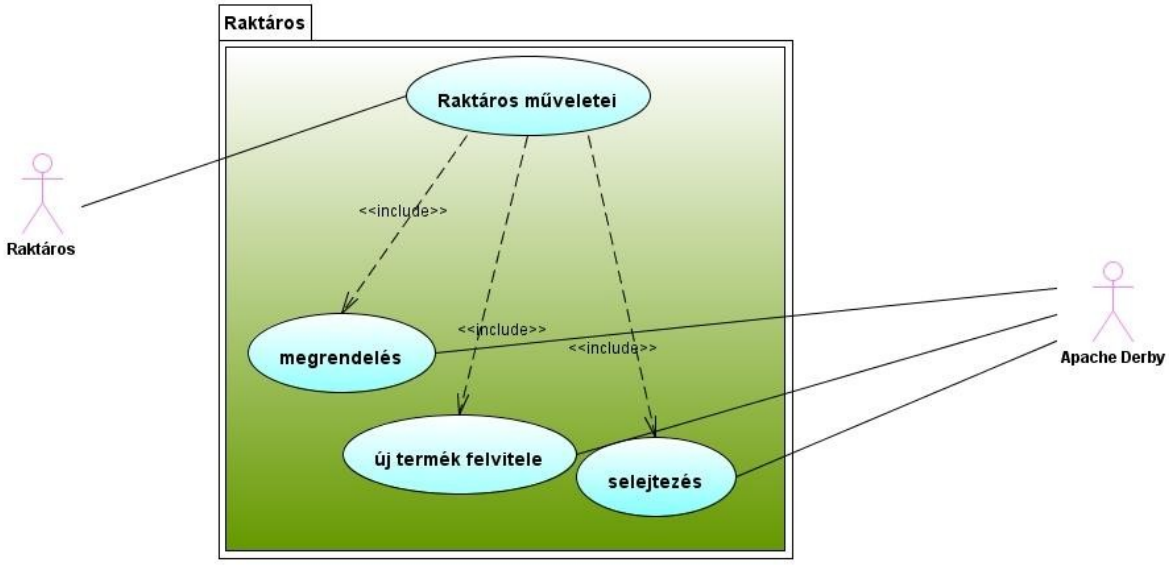
Függelék



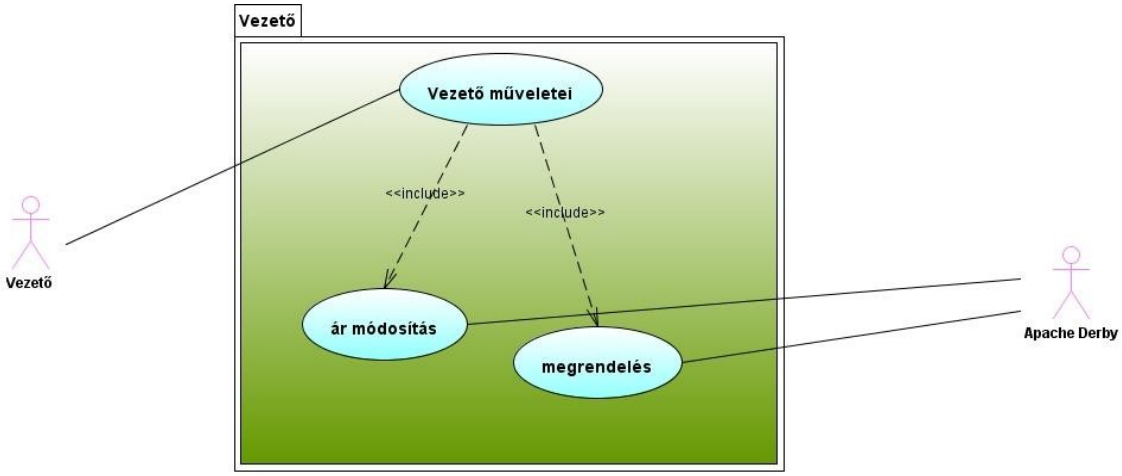
4. ábra: Egy felhasználó regisztrálásának aktivitás diagramja



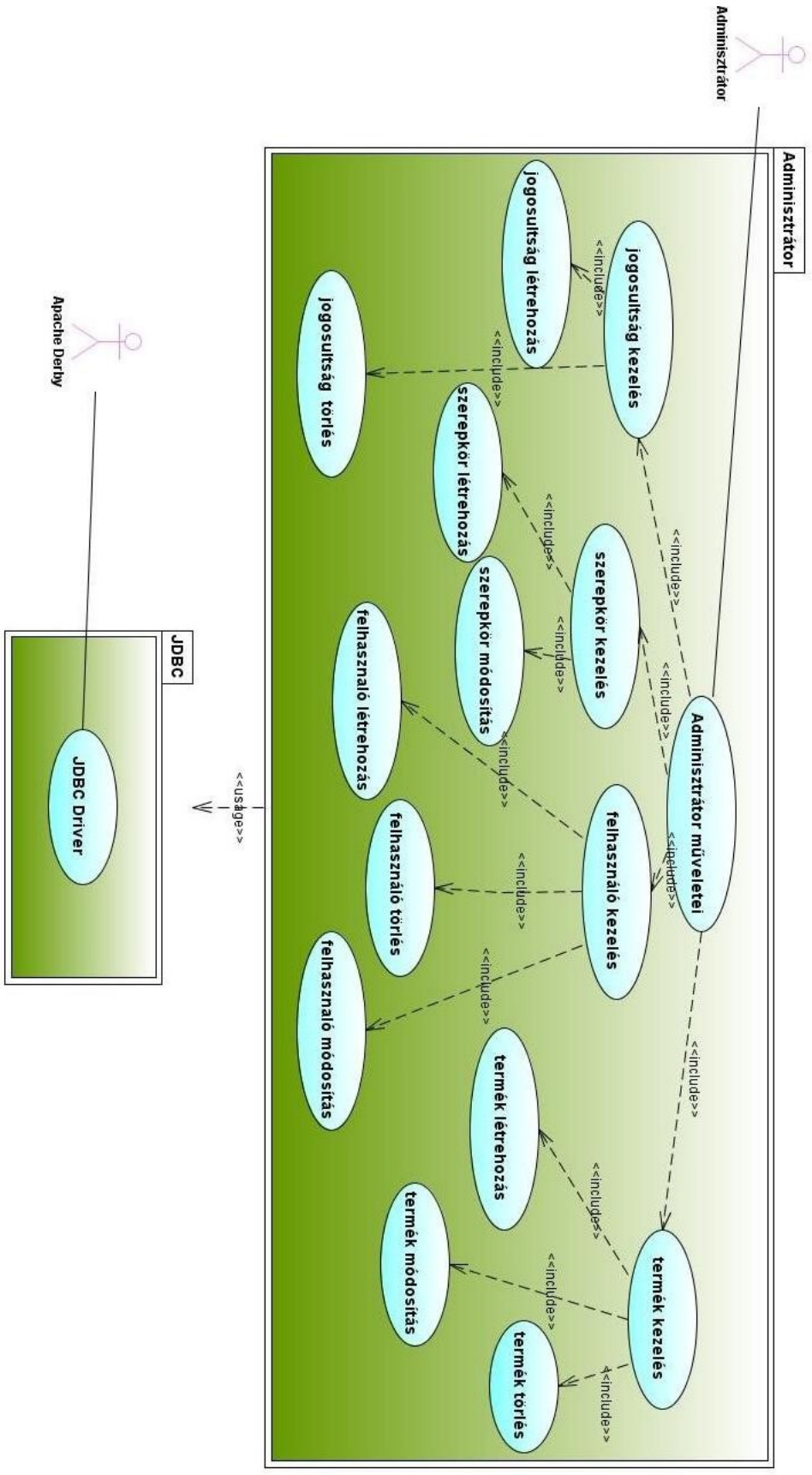
5. ábra: Vásárló típusú felhasználó Use Case diagramja



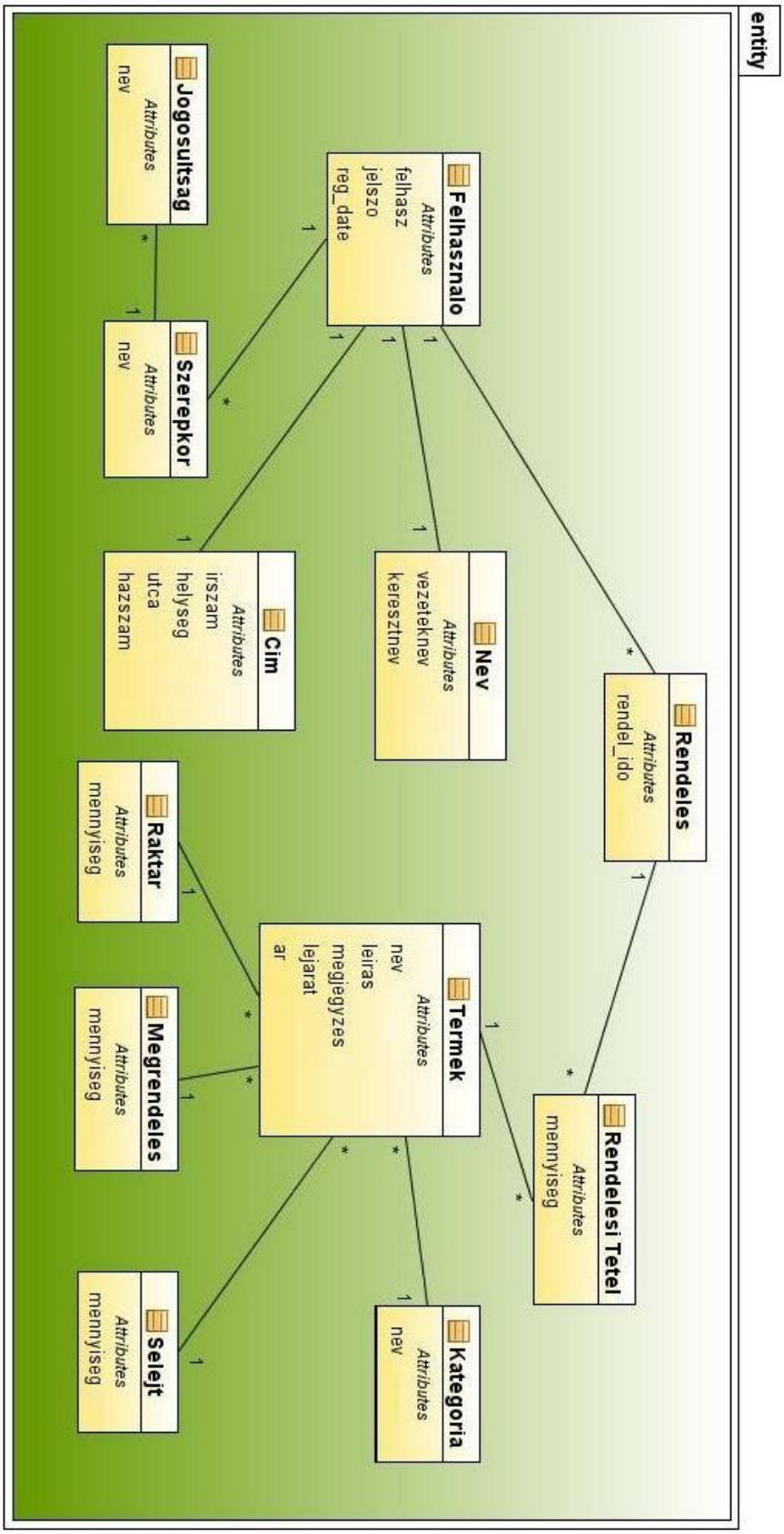
6. ábra: Raktáros típusú felhasználó Use Case diagramja



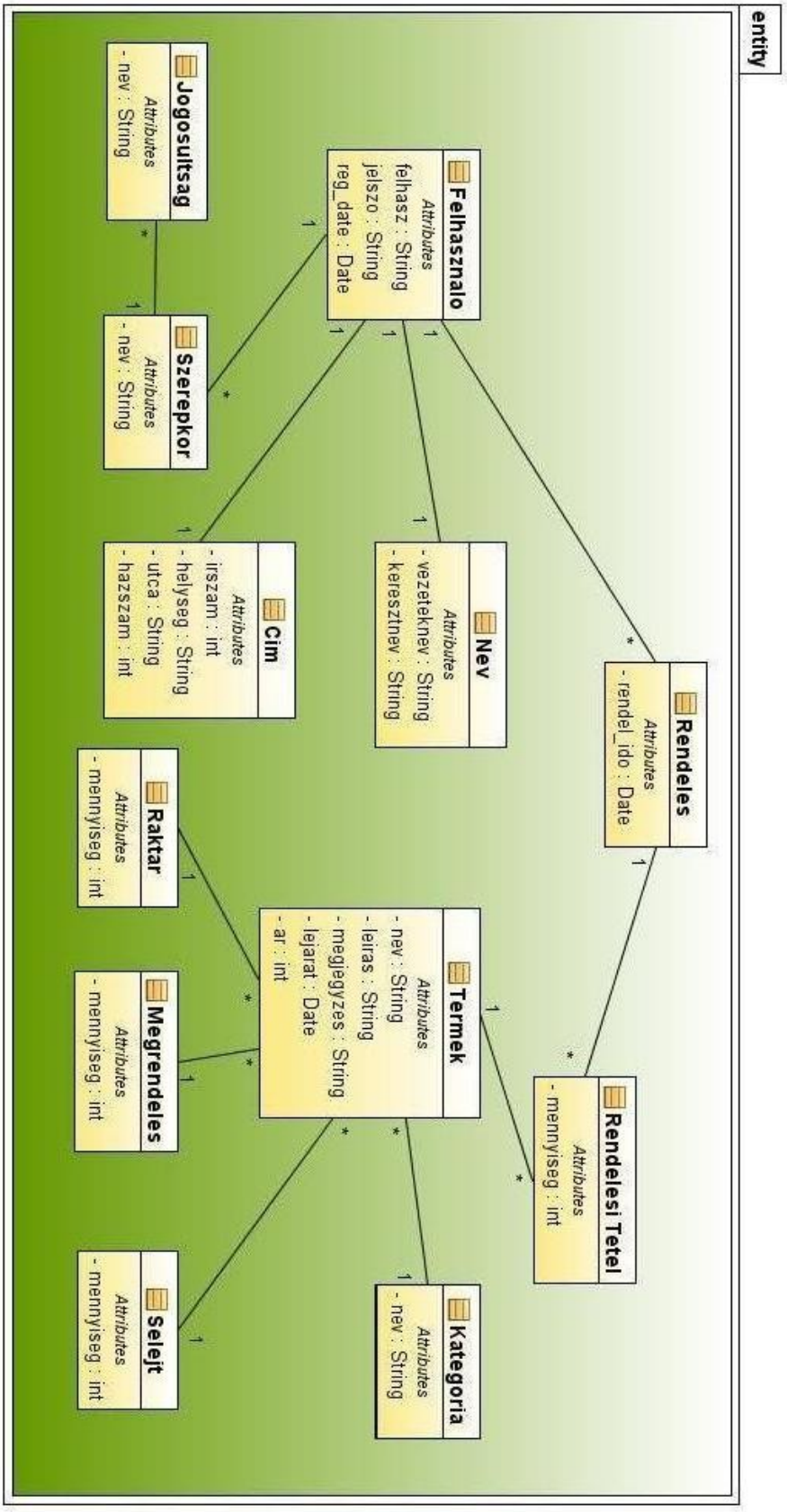
7. ábra: Vezető típusú felhasználó Use Case diagramja



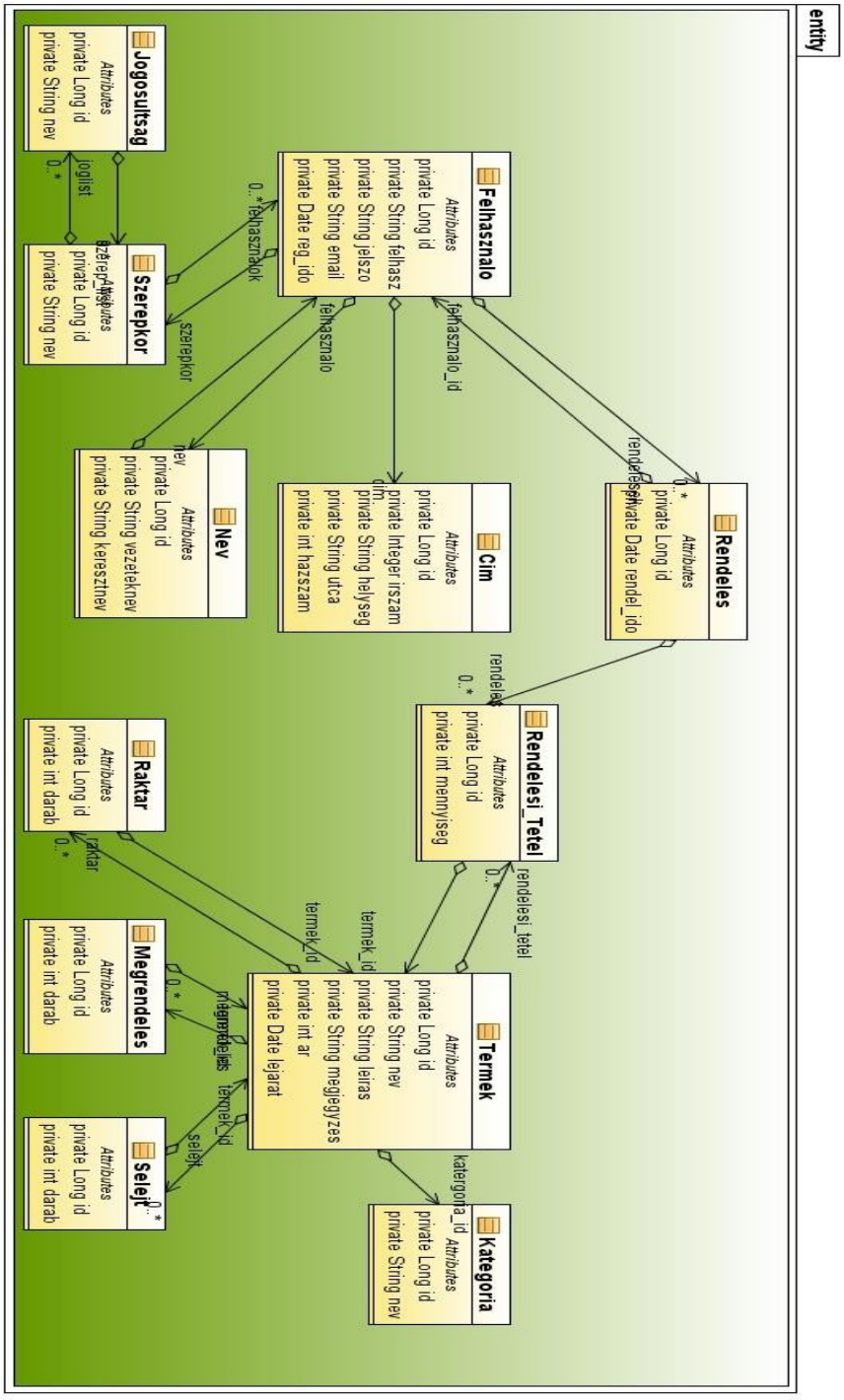
8. ábra: Adminisztrátor típusú felhasználó Use Case diagramja



9. ábra: Platform Független Modell



10. ábra: Platfrom Specifikus Modell



11. ábra: Platform modell