DEBRECENI EGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT

CSONGRÁDI TAMÁS

Debrecen
2008

DEBRECENI EGYETEM
INFORMATIKAI KAR

Webszolgáltatások

Webes és elosztott rendszer technológiák Java Enterprise Edition platformon

Témavezető:
Adamkó Attila
egyetemi tanársegéd

Készítette:
Csongrádi Tamás
Programtervező informatikus(B.Sc.)

Tartalomjegyzék

Bevezetés

Rendkívüli átalakulásnak vagyunk részesei. Az elmúlt években az internet - rohamos terjedésével - gyökeresen megváltoztatta életünket. Az eredetileg katonai célokra fejlesztett technológia már szinte minden háztartásban elérhető.
Egyre többen ismerik fel, hogy azokat a dolgokat, amiket a mindennapokban hosszas sorban állás, vagy utazás árán tudnak elvégezni, az internet segítségével pár kattintás alatt meg tudják oldani. Ezzel időt spórolva, és megkímélve magukat az esetleges feszültségektől.
Természetesen ezt látva a különböző cégek is lépésre kényszerülnek, hiszen nem engedhetik meg maguknak, hogy ne haladjanak a korral. Az a cég, amelyik nem fektet elég hangsúlyt arra, hogy megfelelő információ technológiai háttere legyen, az a mai IT központú világban végérvényesen lemarad. Hiszen ki választana egy olyan céget amelynél több oldalas papírmunka, több órás várakozás van, ha létezik mellette olyan szolgáltató is, amelyik mindezeket kiküszöbölve, IT alapokon oldja meg a fenti problémákat.
A cégek közötti verseny pedig a programozókat és a technológiában érdekelteket kényszeríti arra, hogy minél jobb, és minél használhatóbb megoldásokkal álljanak elő. Ahogy az internet népszerűsége nőtt az elvárások is megváltoztak. A kezdetben teljesen statikus honlapoktól kiindulva a mai, adatbázis háttérrel is rendelkező, nagymértékben dinamikus weblapokig hosszú út vezetett.
Ahhoz, hogy minél több embert elérjen egy cég szükség van arra, hogy a különböző eszközökkel rendelkező felhasználók mind egységes módon tudják kezelni és használni a cég által nyújtott szolgáltatásokat. Így alakultak ki a webszolgáltatások, amelyek nem minden esetben jelentenek újdonságot, hanem a már meglévő alkalmazások lesznek egy egységes szabványnak megfelelően átalakítva. Ezek alapján pedig bármely felhasználó elérheti a szolgáltatást, rendelkezzen bármilyen platformmal is.

Szakdolgozatom célja kettős. Egyrészt szeretném bemutatni a Java nyelv lehetőségeit a webes alkalmazások és szolgáltatások terén. Egy konkrét példán keresztül bemutatva egy webalkalmazás készítését, és egyes részeinek webszolgáltatásként való publikálását, mintegy demonstrálva a mai kor elvárásait.
Másrészt pedig szeretném bemutatni egy informatikai projekt egyes lépéseit a kezdetektől egészen az átadásig bezárólag.

Dolgozatom feltételez bizonyos előismereteket a Java servletek, JSP-oldalak és alapvető HTML ismeretek terén. Ezért ezekre külön nem térek ki, viszont a konkrét megvalósítás részben több példát is bemutatok ezen technológiákkal kapcsolatban. Az előbb említett technológiákról bővebben a következő forrásokon lehet olvasni: [1],[2].

Nagyvállalati szoftverek

A *nagyvállalati szoftverek* általában üzleti céllal írt, számítógépes hálózatok feletti elosztott alkalmazások. Az IT technológia egyre nagyobb térhódításának köszönhetően a vállalatok vezetői felismerték, hogy a siker egyik kulcsa lehet a megfelelő vállalati infrastruktúra, melyet egy jól megírt és az elvárásoknak megfelelően működő szoftverrendszer biztosít. Egyre jobb, és egyre komolyabb szoftverekre volt tehát szükség az idő múlásával és ezáltal a programozókra is nagy teher hárult. Egy komoly vállalat nem engedheti meg magának azt, hogy ne tartson lépést a fejlődéssel, hiszen a lemaradás okán keletkező kár adott esetben dollár milliárdokban is mérhető.
Szükség volt tehát egy olyan koncepcióra, ami az ilyen és ehhez hasonló vállalati méretű szoftverek tervezését és elkészítését hivatott menedzselni, és ezáltal a szoftverfejlesztők dolgát megkönnyíteni. Így alakult ki az *elosztott rendszer* technológia.

Elosztott rendszerek

Az elosztott rendszerek alapvetése az, hogy több számítógép vesz részt a feladatok végrehajtásában. Emiatt az erőforrások jobban kihasználhatók, a hibatűrés megnő, és a felhasználók számának növekedésével arányosan a rendszer méretezhető.
Ha egy vállalati méretű szoftvert szeretnénk készíteni több olyan dologra is figyelnünk kell, amik hagyományos desktop alkalmazásoknál nem, vagy csak kis mértékben jelentkeznek. Habár a feladatok szinte mindig egyediek, tehát egy adott cég már meglévő arculatára, berendezkedésére és elérendő céljaira fejlesztünk, vannak olyan elvárások, amelyek minden egyes ilyen méretű szoftver készítésénél kihívást jelentettek / kihívást jelentenek. Mik is ezek a kihívások?

- Perzisztencia (állandóság, adatfennmaradás): alapvető követelmény, hogy a rendszer használata során azok az adatok, amelyekkel dolgozunk – pl. megrendelt könyv adatai, számla egyenleg stb. – a program befejezése, leállítása után is megmaradjanak. Ezt hívják perzisztens adattárolásnak. Leggyakrabban adatbázisokra való leképezést jelent.
- Tranzakció kezelés: ha egyszerre több kliens fér hozzá ugyanazokhoz az adatokhoz, szükséges annak a biztosítása, hogy úgy lássák ezeket, mintha csak

és kizárólag ők használnák, és senki más. Ezen kívül lényeges, hogy az összetartozó tevékenységek vagy mindegyike végrehajtódjon vagy egyik sem – pl. egy hiba okán átutalásnál nem lehet olyan, hogy az egyik számla egyenlege nő, a másik nem változik –. Ezen feladatokat a beépített tranzakció kezelő látja el.

- Magas rendelkezésre állás: a rendszerünktől elvárjuk, hogy minden időben működjön, és a szolgálatunkra álljon. Egy webáruháznál óriási kiesést jelenthet az, ha csak pár órára is, de nem működik. Ilyenkor a vásárlók más áruházak után néznek, amely bevételkiesést jelent a webshopot üzemeltető cégnek. Erre jelenthet megoldást az ún. *klaszterezés*.
- Terheléseloszlás vagy skálázhatóság: ha egy cég egyre több alkalmazottat foglalkoztat, vagy egy webáruházat egyre többen látogatnak, az nyilván megnöveli a rendszerre háruló feladatokat. Ha a rendszerünk olyan, hogy ezen megnövekedett igényeket is el tudja látni a programkód módosítása és látványos teljesítménycsökkenés nélkül (pl. egy oldal betöltésére, vagy egy szolgáltatás végrehajtására fordított idő) akkor *skálázható*nak nevezzük. Ez lehet függőleges skálázhatóság, amikor erősebb hardverre telepítjük a szoftvert, de lehet vízszintes is, amikor több gép együttesen szolgálja ki a klienseket. Ezt hívják a már előző pontban is említett klaszterezésnek.
- Többszálúság: több kliens kiszolgálásánál jelenthet előnyt, az egyszerre érkező igényeket érdemes több szálon lebonyolítani.
- Távoli elérés: tipikus példa, amikor a böngészőnkkel valamilyen szerveren tárolt adatokat jelenítünk meg. A megjelenítés nálunk, a saját gépünkön történik, de az adatok a szerveren vannak. Ez fennáll internetes, és intranetes közegben is, hiszen egy cég hálózatba kapcsolt gépei is bizonyos protokollok segítségével kommunikálnak. Egy jó keretrendszernek sokféle protokollt kell támogatni.
- Névszolgáltatás: ennek segítségével valósulhat meg, hogy az egyes erőforrásokkal történő kapcsolatfelvételhez nem szükséges az erőforrás teljes elérési útját megadnunk – sokszor nem is ismerjük - , hanem elég ezen erőforrást egy névszolgáltatásnál beregisztrálnunk és innentől kezdve csak a nevére kell hivatkoznunk, a konkrét elérés rejtve marad előlünk.
- Aszinkron üzenetküldés: világméretű, de akár kisebb cégeknél is előfordul, hogy az elosztott rendszerekben az egyes erőforrások átmenetileg nem érhetők el. Mert például javítják, fejlesztik vagy épp átszállítják máshová. Aszinkron üzenetküldés támogatása esetén, ha éppen olyan erőforrásnak küldünk üzenetet,

amely nem áll rendelkezésre, az üzenet eltárolódik, és ha az erőforrás ismét elérhető, megkapja az üzenetet.

- Transzparens hibatűrés: ha egy erőforrás meghibásodik, egy másik hasonló, vagy ugyanolyan szolgáltatást nyújtó átveszi az előző feladatát, anélkül, hogy a kliens ebből bármit is észlelne.
- Biztonság: ha egy rendszert többen használnak, mindig felmerül a biztonság kérdése. Beszélhetünk *authorizáció*ról, vagyis arról, hogy az egyes felhasználók mit tehetnek meg a rendszerrel, milyen szolgáltatásokhoz jogosultak hozzáférni. Illetve beszélhetünk *autentikáció*ról, ami a felhasználók azonosítását teszi lehetővé.
- Loggolás: fontos, hogy a rendszerben végrehajtott összes fontos művelet loggolva legyen. Ez nem pusztán biztonsági kérdés, hiszen a logokból készített statisztikákkal akár a szolgáltatás minőségét is javíthatjuk.

Ezeket szokás *middleware* szolgáltatásoknak hívni. Tehát lényegében önálló szolgáltatások, amelyek mindegyikére, vagy némelyikére szükségünk van ahhoz, hogy a szoftverünk a kívánt eredményt produkálja. Célszerű tehát, ha ezeket a szolgáltatásokat egy nagy egységben kapjuk kézhez, és emiatt alakultak ki az **alkalmazásszerver**ek. Az egyes alkalmazásszerverekben a fent említett szolgáltatások nagy része, vagy akár mindegyike megtalálható.

Kezdetben minden cég, minden szoftvergyártó elkészítette a saját alkalmazásszerverét, amin a megírt programokat futtatni lehetett és biztosította az előbb említett szolgáltatásokat. Viszont ez nagy szakértelmet igényelt, és a konkrét üzleti problémán túl az alkalmazásszerver megírását is a programozók nyakába varrta. Így jött az ötlet, hogy ezeket az alkalmazásszervereket ne mi saját magunk írjuk meg, hanem lehessen megvásárolni őket, majd pedig tetszésünk szerint felhasználni. Sok nagy cég gyárt alkalmazásszervert, ezek közül a Java üzleti alkalmazások esetében a legismertebbek: BEA WebLogic, Sun Glassfish, IBM WebSphere, Red Hat JBoss.

Az alkalmazásszerver egy jól definiált *rétegelt architektúrába* illeszthető be, méghozzá a középső rétegbe

Háromrétegű architektúra

A rétegelt architektúra segítségével lehetőségünk van szétválasztani

- az adatok tárolásához, és felhasználásához támogatást nyújtó *adatréteget*,
- erre az adatrétegre épülő, és a tényleges funkcionalitást nyújtó *üzleti logikai réteget*
- valamint a kliens oldalon történő megjelenítést, a *kliensréteget*.

Ezek szorosan kapcsolódnak egymáshoz, és az alattuk lévő rétegre támaszkodnak. A kliens réteg a legfelső.
A *szoftverkrízis* új módszertanok megjelenését váltotta ki. Tipikusan a szoftverkrízisre adott válaszként jelent meg a *komponens architektúra*.

Komponens architektúra

A komponens architektúra alapvetése az, hogy jól definiált interfészekkel rendelkező komponenseket fejlesztünk, amelyeknek nem adjuk ki a belső működését. Tehát ismerjük, hogy mit csinál, de azt nem, hogy hogyan. Ennek több előnye is van.

- A fejlesztés során meghatározhatjuk, hogy mire van szükségünk, és hogy az adott komponensnek milyen módon kell kommunikálnia a többivel. Így a munka jobban beosztható, mindenki csak a meghatározott komponenst fejleszti, és mivel meghatározott az interfész a kommunikáció megfelelő lesz.
- Mivel több apróbb elemből épül fel a rendszer, a hibák jobban felderíthetők, és orvosolhatók.
- Nem csak mi fejleszthetünk komponenseket. Ha szükségünk van egy adott feladatot ellátó elemre, szétnézünk a piacon, és ha megfelel az elvárásainknak, akkor meg is vehetjük.
- A komponensek tipikus példái az újrafelhasználhatóságnak. Ez azt jelenti, hogy az egyes megírt komponenseket más szoftverben is felhasználhatjuk, nemcsak abban a konkrét rendszerben amire eredetileg terveztük. Minél több helyen kerül felhasználásra az elem, egyre jobban csiszolódik, ugyanis ha kiderül egy hiba azt rögtön javíthatjuk, és a következő felhasználásnál már a javított változatot építhetjük be. Ezáltal nő a biztonság, és a későbbi esetleges komponens

értékesítés során a vevő nyugodt lehet, hogy amit meg akar vásárolni az már több helyen bizonyított.

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

Enterprise JavaBeans, avagy az üzleti logika testet ölt

Az *Enterprise JavaBean-ek (EJB)* olyan szerveroldali komponensek, amelyeket szabványos módon használhatunk fel elosztott rendszerek fejlesztése során. Önállóan telepíthetők egy elosztott, többrétegű környezetben. Gyakorlatilag az EJB-k nyújtotta szolgáltatások által valósul meg az üzleti logika. Mivel komponensekről beszélünk, a komponens architektúránál felsorolt tulajdonságok az EJB-re is jellemzőek. Ezen kívül 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. Ezenkívül definiálja az előbb említettek közötti kommunikációt megvalósító interfészeket. Ezekre azért van szükség, hogy ha a fejlesztő készít egy EJB komponenst, biztos lehessen abban, hogy futni fog az adott alkalmazásszerveren. A SUN különböző teszteknek veti alá az alkalmazásszervereket, és ha ezeknek a teszteknek megfelel a szerver, akkor erről egy igazolást kap, amely jelzi nekünk, hogy bátran fejleszthetünk az adott szerverre.

Az EJB-k csak Java nyelven írhatók, ez egyfajta elkötelezettséget jelent a Java irányába, de mivel a Java tökéletesen megfelel céljainknak ez nem jelenthet akadályt.

Egy EJB alkalmazás életciklusának –fejlesztés, telepítés, futtatás - végrehajtásához több szereplőre is szükség van. Ezek röviden a következők:

- Bean gyártója: az üzleti logikát megvalósító komponenst készíti el, magát az Enterprise bean-t.
- Alkalmazás összeállító: az előző lépésben elkészített komponenseket összeépíti, illeszti egymáshoz, hogy azokból telepíthető alkalmazás készüljön.
- EJB telepítő: telepíti az összeillesztett alkalmazást a futtató környezetbe
- Rendszeradminisztrátor: felügyeli a telepített alkalmazás futását.
- Konténer és szerver gyártó: a konténer gyártó adja azt az *EJB konténer*t, amely az Enterprise Bean-ek futási környezetét biztosítja. Ez a konténer része az alkalmazásszervernek, és ez biztosítja az EJB számára a kívánt szolgáltatásokat.
- Perzisztencia menedzser gyártó: az üzleti adatok perzisztens tárolásáért felelős eszközt készíti.

- Fejlesztő eszköz gyártó: a komponensek egyszerűbb fejleszthetőségét támogató fejlesztői környezetek és eszközök gyártói.

Az EJB példányok egy ún. EJB-konténerhez kötődnek. A konténer hozza létre, futtatja, és mindennemű menedzselési feladatot is ellát az EJB-vel kapcsolatban.
Problémát jelentett korábban, hogy az egyes komponensek által megkövetelt szolgáltatásokat magában a komponensek implementációjában írták le. Ez azzal a hátránnyal járt, hogy ha egy másik környezetbe kívántuk telepíteni az adott komponenst – ahol pl. más volt az adatbázis elérése, vagy a biztonságra vonatkozó beállítások - módosítanunk kellett ezeket a bejegyzéseket a forrásban.
Erre találtak ki az ún. **konfigurációs állomány**okat. A konfigurációs állományokban adhatjuk meg tehát, hogy a komponensünk milyen szolgáltatásokat vár el a rendszertől. Az állományok szerkesztéséhez nem szükséges programozói tudás, csak az adott alkalmazásszerver ismerete, így a vevő, akinek eladtuk a komponensünket, anélkül tudja testre szabni, vagyis a saját rendszeréhez igazítani a szoftvert, hogy mi kiadnánk a konkrét forráskódot.
Az EJB-k készítése a következő lépésekből áll:

- Elkészítjük a konkrét üzleti logikát tartalmazó komponensünket.
- 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 ezzel a kérés közvetítő objektummal kommunikál, tehát nem közvetlenül a bean-nel. 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)* az *Internet Inter-ORB Protocol-al (IIOP)* [3] kiegészülve biztosítja a távoli EJB-k elérését.
- Webes felületű elérés esetén servletek, JSP oldalak stb. végzik a kiszolgálást, az EJB-kre támaszkodva. Ha ezek közös alkalmazásszerveren futnak, akkor egyszerű *Java metódushívás*ról van szó

- A platformfüggetlenség megvalósítása érdekében jött létre az *XML webszolgáltatás technológiája*, mely szabvány segítségével akár más programnyelvben írt erőforrásokkal is képes kommunikálni az EJB

EJB-k típusai

Háromféle EJB-ről beszélhetünk:

- Session Bean: az üzleti folyamatok megvalósítására. Metódusokkal valósítjuk meg az elvárásokat, és az egy konkrét feladathoz kötődő metódusokat egy Session Bean fogja össze.
- Entity Bean: a perzisztenciát hivatott megvalósítani, vagyis az adatbázissal való kapcsolattartást, illetve a Session Beanek által használt adatokat reprezentálhatjuk velük.
- Message-driven Bean: az aszinkron üzenetküldést valósíthatjuk meg velük.

Az EJB 2.1 a J2EE 1.4-ben lett bevezetve, legújabb változata az EJB3 ez a Java EE 5 része. Dolgozatomban az EJB 3-as verziójának alapjait kívánom bemutatni.

EJB 3.0

Az EJB 2.1 technológia bár óriási előrelépés a middleware szolgáltatások kihasználása terén, mégis nehézkes benne programozni, egy sor elem teszi a felhasználást nehézkessé. Ezek a következők:

- Karbantarthatóság: a Java és XML fájlok kezelés megfelelő fejlesztőkörnyezet nélkül nehézkes.
- Metódushívás: EJB 2.1 esetében bonyolult bean-elérési procedúrát kellett végrehajtanunk ahhoz, hogy használhassuk a komponensünket. Koncepció, hogy a kliensek ne közvetlenül érjék el a bean-t, hanem egy ún. *kérés közvetítő objektum* segítségével. Bonyolította a helyzetet több interfész és a belőlük

generálódó objektumok sokasága. Nem beszélve a névszolgáltatásban történő indirekt keresést elősegítő telepítés leíró módosításról.

- Kötelező elemek: a bean osztályunkban kötelezően implementálnunk kell a megfelelő bean interfészt, ami azzal jár, hogy olyan metódusokat is meg kell valósítanunk, amiket esetlegesen nem is használunk fel.
- Tesztelés nehézkessége.

Ezen akadályok leküzdésére született meg az EJB 3.0, amely a Java 5-ben bevezetett annotációk segítségével küszöböli ki a fent említett problémákat

Annotációk[4]

Kevés dolog van a programozástechnikában ami ilyen viharos gyorsasággal terjed el, és marad népszerű, mint az annotációk.[5]
Az annotáció egy forráskódba illesztett *metaadat*, amely nem közvetlenül módosítja a program jelentését, hanem azt befolyásolja, hogy a különböző eszközök és osztálykönyvtárak hogyan kezeljék azt.
Már a Java 5 előtt is volt lehetőség metaadatok elhelyezésére az egyes Java osztályokban kommentek formájában, és ezek kezelésére eszközök is születtek. Mégis az annotációk megjelenésével lett valóban formába öntve a metaadatok beszúrása és definiálása, méghozzá a kommentek használata nélkül.
Minden annotációhoz tartozik egy annotációs típus, amelyet deklarálni kell. A típus nevén kívül nulla vagy több név-érték párt tartalmaznak.
Általános szintaxis a következő:

```
@AnnotációNeve(paraméter1=érték, paraméter2=érték2…)
```

Session Bean

A koncepciónk tehát az, hogy csökkentsük a kód méretét, és a fájlok számát. Viszont ahhoz, hogy a klienseink itt sem közvetlenül a bean-nel legyenek kapcsolatban, hanem egy interfészen keresztül kommunikáljanak vele, mindenképpen szükség van egy interfészre. Így tehát 2 fájlt fogunk használni. Az eddig is használt, a konkrét üzleti

folyamatot – magát az implementációt - tartalmazó bean-osztályt, illetve az ún. *business interfész*t.

Nem csak az Enterprise JavaBean-ek esetében, hanem máshol is felmerült az a probléma, hogy bizonyos frameworkök megkövetelik a saját interfészeik implementálását. Ez a kódot kevésbé átláthatóvá, szerkeszthetővé teszi. Akár több olyan metódust is implementálnunk kell, amit lehet, hogy nem is fogunk használni. Nem beszélve arról, hogy ezáltal framework függővé válik a kód. Erre találták ki az ún. *POJO (Plain Old Java Object – „Sima öreg java osztály")* osztályokat. Ezen osztályok – ahogy a nevük is reprezentálja – megfelelnek a hagyományos Java osztályoknak, tehát például tudjuk őket példányosítani. Tartalmazzák az elvárt funkcionalitást, de mégsem kell vesződni azon metódusokkal, amelyeket egyébként sem használnánk.

Az EJB 2.1-hez hasonlóan a működés helyességét megköveteljük, annak ellenére is, hogy a kód méretét és a fájlok számát csökkentjük. Tehát megoldást kell találnunk azokra a problémákra, amelyek a kód rövidsége és a hiányzó fájlok okán keletkeznek. Ezek a problémák a következők:

1. EJB 2.1 esetén a távoli és / vagy a lokális interfészben deklaráltuk azokat a metódusokat amelyeket magában a bean-ben implementáltunk, és amelyeket publikálni szerettünk volna a kliensek felé. A fájlok számának csökkentése miatt most „csak" egy ún. *business-interfész*ünk van, a megoldás magától értetődik: implements kulcsszóval kell összerendelnünk a bean-osztályt és a business interfészt. Egy nyitott kérdés még maradt, honnan tudjuk, hogy az interfészünk távoli, vagy lokális? Alapértelmezettként lokálisnak tekintendő, de ez felülbírálható a *@Remote* annotációval mind a business-interfészben, mind pedig az implementációs osztályban.
2. Egy session bean **rendelkezhet állapottal**, és **lehet állapotmentes**. Állapottal rendelkező esetben a kliens számíthat arra, hogy a következő kérésnél is ugyanazzal a bean-nel fog kommunikálni, tehát tárolhat benne bizonyos adatokat. Állapotmentes esetben ez nem valósulhat meg. Azt, hogy egy session bean állapotmentes vagy állapottal rendelkező EJB 2.1 esetén a telepítés leíróban adhattuk meg, a <*session-type*> elem segítségével. EJB 3 esetén egyszerűen annotációkkal tehetjük ezt meg. Az állapotmentes session bean-t a *@Stateless*, az állapottal rendelkezőt pedig a *@Stateful* annotációval jelöljük.
3. 2.1 esetén az *ejbCreate()* és *ejbRemove()* metódusok segítségével történt az EJB példányok létrehozása illetve megszüntetése. Állapotmentes session bean esetén

nincs is más életciklus. EJB 3.0 esetében az előző két metódusnak a *@PostConstruct()* és a *@PreDestroy()* felel meg. Stateful session bean esetén létezik még az **aktiválás** és **passziválás**, melyek során a bean-nek a hozzá tartozó erőforrásokat lehetősége van elengedni, illetve újra felvenni velük a kapcsolatot, erre szolgálnak a *@PrePassivate* és *@PostActivate* metódusok. Egy annotációt kell még megemlíteni, aminek a segítségével a kliens a hozzá tartozó állapottal rendelkező session bean-ről „lemondhat", ez pedig a *@Remove*.

4. EJB 3.0-ban bevezetésre került egy eszköz, aminek segítségével elkerülhető a 2.1-es verzióban tapasztalható bonyolult bean-elérési procedúra. Ez az ún. *függőséginjektálás* (*dependency injection*). Ennek során, ha egy business interfész típusú változót *@EJB* annotációval látunk el, akkor a konténer köteles példányt biztosítani a megadott változónéven, tehát nem kell implicit módon névszolgáltatás keresést alkalmazni. Ez tulajdonképpen csak fél igazság. Ugyanis a konténeren kívüli klienseknek ezután is JNDI[6] (Java Naming and Directory Interface – a Java-ban, a különböző névszolgáltatások egységes kezeléséért felelős API) keresést kell alkalmazniuk, de a konténerben lévő más EJB-k, és web komponensek – és túlnyomórészt erről van szó – minden további nélkül alkalmazhatják ezt a technikát. Könnyebbség a konténeren kívüli kliensek számára, hogy a keresés során már nem a home objektum – mely az EJB-vel történő kapcsolattartás sarokköve - a kapott eredmény, hanem referencia az adott bean példányra. A bean metódusai által esetlegesen dobott rendszerszintű kivételek a *RuntimeException* leszármazottjaként, *javax.ejb.EJBException*-ként váltódnak ki, tehát nem kötelező elkapni, lekezelni.

Állapotmentes Session Bean - példa

A következőkben egy állapotmentes session bean-re és annak kliensére láthatunk példát. A bean egy egyszerű függvényből áll, amely paraméterül ***mit*** néven egy sztringet és ***hanyszor*** néven egy **int** típusú számot vár és a visszatérési értéke szintén egy sztring, amelyben a ***mit*** értéke annyiszor szerepel, amennyi a ***hanyszor*** értéke:

Bean osztály

```
package EJBpelda;

import javax.ejb.Stateless;

@Stateless
public class sessionPeldaBean implements sessionPeldaRemote {
```

```
    public String tobbszoroz(String mit, int hanyszor) {
        String vissza="";
        for (int i=1;i<=hanyszor;++i) vissza+=mit;

        return vissza;
    }
}
```

Business-interfész

```
package EJBpelda;

import javax.ejb.Remote;

@Remote
public interface sessionPeldaRemote {

    String tobbszoroz(String mit, int hanyszor);

}
```

Kliens

```
package ejbpelda;

import EJBpelda.sessionPeldaRemote;
import javax.ejb.EJB;

public class Main {
    @EJB
    private static sessionPeldaRemote sessionPeldaBean;


    public static void main(String[] args) {
        System.out.println("Eredmény: " + sessionPeldaBean.tobbszoroz("EZT", 3));
    }

}
```

Futtatás után a következő eredményt kapjuk:

```
Eredmény: EZTEZTEZT
```

Session beanek esetében meg kell említenünk egy különleges osztályt, melyet *interceptorosztálynak* hívunk.

Interceptorok

Session- vagy message-driven beanek használata során szükségünk lehet olyan szűrőkre, amelyek a bemenő paramétereket vizsgálják. Ezeket a szűrőket valósíthatjuk meg az interceptorosztályok segítségével, azzal kiegészítve, hogy nemcsak szűrni tudják, hanem módosítani is a paramétereket. Segítségükkel a bean konkrét metódusának meghívása nélkül is jelezhetjük a kliens felé, hogy valami nincs rendben.
Interceptorosztály egy teljesen hagyományos Java osztály, egyetlen **Object** visszatérési értékű interceptor metódust tartalmaz amelyet *@AroundInvoke* annotációval jelölünk.

Ezen kívül szükséges még egy *paraméter nélküli konstruktor*. Az interceptor metódus dobhat kivételt is.

Kérdés, hogy hogyan érhetjük el a meghívott metódus egyes paramétereit? Erre szolgál az interceptor metódus bemenő paramétere, amely egy *InvocationContext* típusú változó, aminek a segítségével nemcsak a paramétereket, hanem a hívott metódust és a bean-példányt is el tudjuk érni.

Azt, hogy egy bean-hez tartozik interceptor, a bean osztályban egy *@Interceptors({Interceptor1.class,Interceptor2.class ...})* annotációval adhatjuk meg. Ez történhet osztály illetve metódus szinten. Előbbi esetben a bean-osztály összes metódusának meghívásakor megszakítódik a kérés, míg az utóbbi esetben csak az adott metódus meghívása esetén. Másik módszer a telepítés leíróban történő hozzárendelés, amely felülbírálja az annotációs megoldást.

Állapotmentes Session Bean Interceptorral - példa

Az előző példa esetén nem vizsgáltuk, hogy mi van akkor, ha paraméterként negatív számot kap a bean. Ezt persze megtehetnénk a bean-osztályban is, de a prezentáció kedvéért most interceptor osztályban fogjuk vizsgálni.

Bean osztály

```
package EJBpelda2;

import javax.ejb.Stateless;
import javax.interceptor.Interceptors;

@Stateless
@Interceptors(interceptorPelda.class)
public class sessionPeldaBean2 implements sessionPeldaBean2Remote {

    public String tobbszoroz(String mit, int hanyszor) {

        String vissza="";
        if (hanyszor != 0)
        for (int i=1;i<=hanyszor;++i) vissza+=mit;

        return vissza;

    }

}
```

Business-interfész

```
package EJBpelda2;

import javax.ejb.Remote;

@Remote
public interface sessionPeldaBean2Remote {

    String tobbszoroz(String mit, int hanyszor) throws
            negativException;
```

```
}
```

Interceptorosztály

```
package EJBpelda2;

import java.lang.reflect.Method;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class interceptorPelda {


    @AroundInvoke
    public Object interceptorPelda (InvocationContext ctx) throws Exception {
        Method method = ctx.getMethod();

        if (method.getName().equals("tobbszoroz")) {
            Integer param = (Integer) (ctx.getParameters()[1]);

            if (param < 0) {

                throw new negativException("Negatív szám nem lehet paraméter: " + param);
            }
        }

        return ctx.proceed();

}
 }
```

Kivételosztály

```
package EJBpelda2;


public class negativException extends Exception {

    public negativException(String string) {
        super(string);
    }

}
```

Kliens

```
package ejbpelda2;

import EJBpelda2.negativException;
import EJBpelda2.sessionPeldaBean2Remote;
import javax.ejb.EJB;

public class Main {
    @EJB
    private static sessionPeldaBean2Remote s;

    public static void main(String[] args) {
        try {
            System.out.println("Eredmény: " + s.tobbszoroz("EZT", -1));
        } catch (negativException ex) {
            System.out.println("Hiba: " + ex.getMessage());
        }
    }
}
```

Ebben az esetben a fordítás és futtatás után a következő eredményt kapjuk:

```
Hiba: Negatív szám nem lehet paraméter: -1
```

Az interceptorosztályban a metódus adatait a *getMethod()* metódus segítségével tudhatjuk meg, mely az *InvocationContext* típusú változóból kéri le a szükséges információkat. Az egyes paramétereket pedig szintén az *InvocationContext* típusú változó *getParameters()* metódusával kapott tömb típusú változó elemei adják.

Entitások

Az entity beanek segítségével lehetőségünk van modellezni az üzleti folyamatok által használt adatokat, méghozzá perzisztens módon. A perzisztencia az esetek döntő többségében adatbázisok formájában valósul meg, ugyanis az adatbázisban kezelt adatokat lényegesen könnyebb kezelni, illetve manipulálni, mint az egyes *szerializált objektumok*at. Bár már régóta léteznek objektum-relációs adatbázisok, még ma is javarészt relációs adatbázisokat használunk. Tehát meg kell oldanunk, hogy a perzisztens módon tárolni kívánt objektumaink az adatbázisban megfelelő módon tárolódjanak. Az entity beanek ezt az ún. *objektumrelációs leképezés*sel (*ORM – Object Relation Mapping*) oldják meg, mely során az adatbázis táblái az osztályoknak, a táblák oszlopai az osztály attribútumainak felelnek meg. Vagyis egy osztály példánya a hozzá tartozó tábla egy sorának felel meg.

Entity bean-t egyszerű módon készíthetünk. Egy hagyományos Java osztályt írunk, majd magát az osztályt ellátjuk egy *@Entity* annotációval. Majd az általunk választott elsődleges kulcsot egy *@Id* annotációval. Ezek után meg kell írnunk az egyes adattagokhoz tartozó beállító és lekérdező metódusokat (*gettereket, settereket*). A *Java Persistence API*[7] alapértelmezett módon az osztályéval megegyező nevű tábla, az attribútuméval megegyező nevű oszlopába képzi le az adatokat. Ha az alapértelmezettől el szeretnénk térni, arra is van lehetőség. A *@Table* annotációval felülbírálhatjuk, hogy melyik táblába, a *@Column* annotációval pedig, hogy az adott tábla melyik oszlopába képződjön le az információ. Sőt, egy entitást akár több táblában is tárolhatunk, erre a *@SecondaryTable* szolgál.
Az entitások – a perzisztencia szempontjából - két fajta attribútummal rendelkezhetnek:
A **nem perzisztens** attribútumok típusára nincs megkötés, csupán annyit kell tennünk, hogy ezeket a *@Transient* annotációval kell jelöljük.

A **perzisztens** attribútumoknál már letézik bizonyos megkötés. Erre 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 helyezzük el, akkor ezek felhasználásával történik meg a mentés és a betöltés. Ha az attribútumok előtt helyezzük el, akkor betöltésnél közvetlenül a változókba töltődik be a kért adat és nem a setter metódusok lesznek felhasználva. A kimentés hasonló elven működik. Bármelyiket is válasszuk, vegyük figyelembe, 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 felvételét. Mint ahogy fentebb olvasható, az elsődleges kulcs attribútumot a *@Id* annotációval választhatjuk ki. A többi perzisztens attribútumtól eltérően, erre erősebb megkötések vonatkoznak a típus tekintetében. Az elsődleges kulcs típusa lehet:

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

Az elsődleges kulcsok generálására is 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 4 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.

Összetett kulcs a *@IdClass* annotációval adható meg.

Entitás-bean - példa

Az alábbi példa egy egyszerű ***vasarlo*** entitást kíván bemutatni, amelyben létezik a vásárló nevét, címét és telefonszámát reprezentáló attribútum.

Entitás bean

```
import javax.persistence.*;

@Entity
public class Vasarlo {
   private int id;
   private String nev;
   private String cim;
   private int telszam;

@Id
@GeneratedValue (strategy = GenerationType.AUTO)

   public int getId( ) {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }

   String getNev( ) {
      return name;
   }

   public void setNev(String nev) {
      this.nev = nev;
   }

String getCim( ) {
      return cím;
   }

   public void setCim(String cim) {
```

```
    this.cim = cím;
  }

int getTelszam( ) {
    return name;
  }

  public void setTelszam(int telszam) {
    this.telszam = telszam;
  }
}
```

Elvárás volt az entitásokkal szemben, hogy egyszerűbben kezelhetők, és ezáltal fejleszthetők lehessenek. Ezért itt is - mint ahogy a session bean-ek esetében - támaszkodhatunk a *POJO* osztályokra. Lehet őket példányosítani, vagyis az előbbi ***vasarlo*** entitást a következőképpen tudjuk használni:

Kliens

```
vasarlo v = new vasarlo();
v.setNev = „vasarloNeve”;
v.setCim = „vasarloCime”;
v.setTelszam = vasarloTelefonSzama;
```

Természetesen attól, hogy egy entitást példányosítunk, még az adatbázisban nem fog megjelenni. Az entitásoknak is vannak életciklusaik.

Entitások életciklusa

Mint azt az előbb említettem, ha *new()* operátorral példányosítunk egy entitást az csak a memóriában fog megjelenni, a neki megfelelő adat nem tárolódik automatikusan az adatbázisban. Az adatbázisban való tároláshoz, és minden más perzisztenciával kapcsolatos szolgáltatás megvalósításához szükségünk van az ún. **perzisztencia provider**-re.
Java fejlesztői az automatikus objektumrelációs leképezés támogatása céljából kifejlesztettek egy technológiát, amely a *Java Persistence API (JPA)* nevet kapta. Ez egy specifikáció, melyet a konkrét megvalósításoknak implementálniuk kell. Azokat az osztálykönyvtárakat, amelyek implementálják a JPA specifikációt perzisztencia providereknek nevezzük. Ezek közül az ismertebbek pl. a Hibernate vagy a TopLink. Ezeknek több előnyük is van, nevezetesen:

- A programozónak nem kell vesződnie a JDBC[8]-kódolással, ezen eszközök automatizáltan megteszik a fejlesztő helyett. A JDBC a különböző adatbáziskezelő rendszerekhez való hozzáférést megvalósító API.

- Mivel ezek az eszközök konténer függetlenek, ezért nemcsak EE környezetben hanem Serial Edition alkalmazásokban is használhatók.

Az egyes entitásokat az ún. *EntityManager* interfész segítségével érhetjük el. Entitások azon halmazát, amelyben minden egyes elsődleges kulccsal rendelkező példányhoz egyetlen memóriabeli példány tartozik **perzisztenciakontextus**nak nevezzük. Az EntityManager perzisztenciakontextussal dolgozik. Java EE környezetben a konténerre van bízva annak biztosítása, hogy ha egy tranzakció több komponenst is magába foglal - pontosabban felhasznál -, akkor egy tranzakcióhoz egy perzisztencia kontextus tartozzon. A session bean-ek esetében *@EJB* annotáció segítségével végezhetjük el a függőséginjektálást, entity bean-ek esetében ezt a *@PersistenceContext* annotációval tehetjük meg. Melynek hatására referenciát kapunk egy EntityManager példányra és a konténer a háttérben hozzárendeli az aktuális tranzakcióhoz tartozó perzisztencia kontextust a megkapott EntityManager példányunkhoz.
Visszatérve az entitások életciklusára, négyféle életciklusról beszélhetünk. A new() operátor segítségével egy entitást **új állapot**ba hozhatunk, ilyenkor az objektum csak a memóriában található meg, nem létezik az őt reprezentáló adatbázis adat. Tehát ha ezen a példányon valamilyen manipulációt hajtunk végre, az adatbázis változatlan marad. Ha viszont meghívjuk az EntityManager.persist(entitás) metódust, akkor az entitásunk **menedzselt állapot**ba kerül, tehát bekerül az EntityManager-hez tartozó perzisztenciakontextusba. Ezután ha valamilyen sikeres tranzakciót hajtunk végre az immár menedzselt példányon, akkor annak látható jele marad az adatbázisban, vagyis a tranzakció végeztével szinkronizálódik a memóriabeli adat és az adatbázisban neki megfelelő adat. **Lecsatolt állapot**ba kerül egy entitás, miután megszűnik az a perzisztenciakontextus, aminek ő része. Lecsatolt állapotban ugyan létezik az entitásnak megfelelő adatbázis rész, de a manipulációk után már nem frissül az adatbázis. **Törölt állapot**ba az EntityManager.remove(entitás) metódussal kerülhet az entitás.

Tervezési minták

Nem csak Enterprise JavaBeanek kapcsán, hanem az EE platform, sőt más programozási nyelvek esetében is léteznek ún. *tervezési minták*. A tervezési minták sokszor felmerülő problémákra nyújtanak megoldást. Ezeket természetesen nem kötelező betartani, de mint a komponensek használatánál említve lett, itt is el kell

mondani, hogy ezek kipróbált és bizonyított dolgok. Nem szükséges tehát egy adott probléma megoldását nekünk kitalálni, hiszen nem biztos, hogy 100%-os megoldást tudunk találni, vagy ha mégis, időt veszíthetünk. Az entity bean-ek esetében egy tervezési mintát szeretnék bemutatni, ez pedig a *session facade*.

Session Facade

Miért is van szükségünk az entity bean-ek esetében tervezési mintára? Ehhez először azt kell végiggondolnunk, hogy milyen probléma jelentkezhet egy vagy több entitás használata során.

Sokszor előfordul, hogy a kliensek távoli metódushívásként csatlakoznak az adott bean-ekhez. Ez plusz költséget jelent, hiszen lokális interfészen keresztül „helyben" elérni egy bean-t mindig „olcsóbb", mint távoli interfészen. Nem beszélve arról, hogy egyes feladatok akár rendkívüli komplexitással is bírhatnak. A komplexitásból adódóan az egyes osztályok szorosan függnek egymástól, nehezebb átlátni a kapcsolatokat, és nehezebb menedzselni is. Erre nyújt megoldást a session facade tervezési minta. Lényege, hogy a kliens nem kérések sorozatát intézi egy entitásnak, ezáltal állandó plusz költséggel terhelve az elérést, hanem egy session bean egy metódusát hívja, és az említett session bean - lokális interfészen kommunikálva - egy egész sor üzleti funkcionalitást végez.

Egy egyszerű entity bean használatára láthatunk példát a következőkben. Egy entitás egy ***felhasználó***t reprezentál, melynek van ***felhasználóneve*** és ***jelszava***, ahol a felhasználónév az elsődleges kulcs. Session facade segítségével történik az entity bean-ekhez történő hozzáférés. A session bean-eket **servlet**ek hívják meg.

Entitás

```
package pelda4;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class felhasznalok implements Serializable {
    @Id private String felhNev;
    private String jelszo;

    public String getFelhNev() {
        return felhNev;
    }

    public void setFelhNev(String felhNev) {
        this.felhNev = felhNev;
    }
```

```
    public String getJelszo() {
        return jelszo;
    }

    public void setJelszo(String jelszo) {
        this.jelszo = jelszo;
    }

    @Override
    public String toString() {
        return "Név: " + felhNev + " Jelszó: "+ jelszo;
    }

}
```

Session facade

```
package pelda4;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless
public class felhasznalokFacade implements felhasznalokFacadeLocal {
    @PersistenceContext
    private EntityManager em;

    public void create(felhasznalok felhasznalok) {
        em.persist(felhasznalok);
    }

    public void edit(felhasznalok felhasznalok) {
        em.merge(felhasznalok);
    }

    public void remove(felhasznalok felhasznalok) {
        em.remove(em.merge(felhasznalok));
    }

    public felhasznalok find(Object id) {
        return em.find(pelda4.felhasznalok.class, id);
    }

    public List<felhasznalok> findAll() {
        return em.createQuery("select object(o) from felhasznalok as o").getResultList();
    }

    public felhasznalok findByFelhNev (String felhNev)
    {
        Query q = em.createQuery("SELECT f from felhasznalok f where f.felhNev=:FelhNev");
        q.setParameter("FelhNev",felhNev);
        felhasznalok f = new felhasznalok();

        try{
            f = (felhasznalok) q.getSingleResult();
        }catch (NoResultException ex){

            return null;
        }

       return  f;    }}
```

Session facade lokális interfész

```
package pelda4;

import java.util.List;
import javax.ejb.Local;

@Local
public interface felhasznalokFacadeLocal {
```

```
    void create(felhasznalok felhasznalok);

    void edit(felhasznalok felhasznalok);

    void remove(felhasznalok felhasznalok);

    felhasznalok find(Object id);

    List<felhasznalok> findAll();

    public pelda4.felhasznalok findByFelhNev(java.lang.String FelhNev);

}
```

Kliens szervletek lényeges részei

Új elem felvétele

```
public class reg extends HttpServlet {
    @EJB
    private felhasznalokFacadeLocal felhasznalokFacade;
…
String felhNev = request.getParameter("felhNev");
        String jelszo = request.getParameter("jelszo");
        felhasznalok f = new felhasznalok();

        f.setFelhNev(felhNev);
        f.setJelszo(jelszo);
        if(felhasznalokFacade.findByFelhNev(felhNev) == null)
        felhasznalokFacade.create(f);
…
```

Keresés

```
Keresés
public class felhKereses extends HttpServlet {
    @EJB
    private felhasznalokFacadeLocal felhasznalokFacade;
…
String felhNev = request.getParameter("felhNev");
        felhasznalok f = new felhasznalok();
        f = felhasznalokFacade.findByFelhNev(felhNev);
…
```

Entitások keresése is az EntityManager-en keresztül történik. A kereséshez szükségünk van egy *Query*-re, mely segítségével az SQL-ben megszokott módon tudunk adatokat keresni az adatbázisban. A kereső metódusok egy **Query objektum**mal térnek vissza, melynek vannak bizonyos metódusai. A fenti példában a Query objektumon a *getSingleResult()* metódust hívtuk meg, ugyanis a felhasználónév az elsődleges kulcs, biztos, hogy csak 1 vagy 0 megoldással térhet vissza a Query. A nulla megoldás kezelése miatt pedig belekerült egy **kivételkezelés** is. Az egyes lekérdezéseket *natív SQL* nyelven is megfogalmazhatjuk, és külön az EJB-hez fejlesztett *EJB-QL* nyelven is. Lekérdezéseket elhelyezhetünk magában az entitás bean osztályban is. Ilyenkor ezeket *@NamedQuery(name=”nev”,query=”MANIPULÁCIÓ”)* annotációval kell jelölnünk, *createNamedQuery(„nev”)* metódussal példányosítani, majd *setParameter(„par.név”,értek)* metódussal inicializálni.

Terjedelmi korlátok miatt az **entitások öröklődésére** és a **tranzakció-kezelésére** nem térek ki.

Webszolgáltatások

A technológia fejlődése, mint ahogy azt korábban említettem, nem hagyja érintetlenül a felhasználók egy csoportját sem. Nincs olyan rétege a társadalomnak, amely ne használná, vagy ne szeretné használni az internet, és a különböző technológiák nyújtotta szolgáltatásokat. Sorra jelennek meg az okostelefonok, mindenféle segédeszközzel felszerelt laptopok, és a bár rendkívül kicsi, de annál többet tudó palmtop készülékek. A nem hagyományos, külön célközönségnek szánt egyéb eszközökről már nem is beszélve. Ezek mind-mind külön kihívást jelentenek a fejlesztők számára, hiszen egy bizonyos alkalmazás másképp fut egy asztali gépen, másként fut egy palmtopon és megint másként egy okostelefonon. Sőt, egy platformon belül, a különböző programnyelvi megvalósítások is leküzdendő akadályt jelentenek. Ezenkívül ugyancsak problémát jelent a különböző komponens alapú rendszerek közötti **platform- és programozási nyelv független** kommunikáció megvalósítása.. Így születtek meg az *XML webszolgáltatások*.

A webszolgáltatások megismerése előtt azonban lényeges, hogy tisztában legyünk azzal, hogy mi is magának a webszolgáltatásnak az alapja. Ez nem más, mint az *XML*.

XML - Extensible Markup Language

A *Word Wide Web Consortium* (*W3C*)[9] hozta létre még 1996-ban. Az *SGML*[10]-ből (*Standard Generalized Markup Language*) származik, mely egy szövegleíró nyelv. **Tag-**eket használhatunk az egyes elemek leírására, de a *HTML*-től eltérően – mely egyébként szintén az SGML-ből származik – saját tag-eket is definiálhatunk. A HTML nyelvben gondot okozott a bonyolultabb felépítésű adatok tárolása, az SGML viszont túlságosan tág feltételeket szabott a saját tag-ek létrehozásának. Ez utóbbi azért volt probléma, mert így nehezebb volt – sőt egyes esetekben lehetetlen – olyan programot írni, amely az SGML-ben tárolt adatokat könnyedén tudta volna olvasni és értelmezni. Nem beszélve arról, hogy az emberek számára is nehezebben olvasható volt. Szükség volt tehát egy olyan nyelvre, mely ezeket a problémákat megoldja. Így született meg az XML.

Webszolgáltatások és az XML

Teljesen platform- és programozási nyelv független eszközt kellett keresni a problémák leküzdésére, és erre kézenfekvő megoldás volt, egy olyan speciális szöveges protokoll, amely minden ma használatos platformon és programozási nyelven működőképes. Itt jött a képbe az XML. Maga a szöveges protokoll és mindennemű leírás, mely a webszolgáltatások technológiáját alkotja az XML-szabványra épül. Ez határozza meg a funkcionalitás leírását, valamint a kliens és szolgáltató közötti kommunikáció mikéntjét. Legfőképpen azt, hogy hogyan is nézzen ki az adott XML-dokumentum szerkezete, milyen elemekből is álljon.

Webszolgáltatások megvalósítása

A következőkre van szükségünk ahhoz, hogy egy szolgáltatást publikálni tudjunk és azt a kliensek fel is tudják használni:

- Először is természetesen magára a **szolgáltatásra**, amely nem más, mint egy funkcionalitás – ez lehet egészen kis méretű, pl. egy valutaátváltó -, melyet az adott vállalat közkincsé kíván tenni a felhasználók számára. Ha ez kész van, akkor XML nyelven **publikálni kell**, hogy a kliensek ténylegesen fel is tudják használni.
- Mivel a szolgáltatások egészen aprók is lehetnek, ezáltal előfordulhat - sőt, ez a jellemzőbb -, hogy több szolgáltatás összessége, illetve együttes használata valósítja meg a ténylegesen elvárt viselkedést és funkcionalitást. Ezért szükségünk van egy olyan rendszerre, egy **tároló**ra, amelyben az egyes elkészített szolgáltatásokat elhelyezzük, és onnan vissza tudjuk keresni őket.
- A klienseknek **képeseknek kell lenniük** arra, hogy a szolgáltatást hívni tudják, vagyis az adott funkcionalitásokat megfelelően fel tudják paraméterezni.

A fent említett három feladatot oldja meg a következő három technológia, az előbbiek sorrendjében: **WSDL**, **UDDI**, **SOAP**.

A ***WSDL (Web Service Description Language)*** dokumentumok biztosítják, hogy a szolgáltató által nyújtani kívánt funkcionalitást szabványos XML-dokumentumokkal le lehessen írni, hogy azt a felhasználók elérhessék. Ezek az adott szolgáltatást távoli eljárásokként határozzák meg, és mint ilyeneknek megadják, hogy milyen paramétereket várnak el a kliensektől. Tartalmazzák még ezen kívül azt az elérési címet (*URL*-t), amely mögött a tényleges szolgáltatás található.

Az ***UDDI (Universal Description, Discovery and Integration)*** szabványos módon biztosítja az előbbi lépésben leírt WSDL-dokumentumok egy speciális adatbázisban történő tárolását, és annak a lehetőségét, hogy onnan az egyes szolgáltatások kikereshetők legyenek. Ezen kívül tartalmazhat még olyan egyéb adatokat, amelyek a szolgáltatások „*járulékos információit*" takarják, mint például a minőség. A szolgáltatástárban való kereséshez az UDDI egy API-t definiál. Az UDDI-re azonban nincs mindig szükség. Ha csak néhány webszolgáltatást kívánunk publikálni, akkor a kliensek az UDDI nélkül is képesek elérni azokat. UDDI-t viszont mindenképpen célszerű használni akkor, ha nagyon sok webszolgáltatást publikálunk – tipikusan együttműködő webszolgáltatásokat – és biztosítani szeretnénk az ezek közötti keresést.

A ***SOAP (Simple Object Acces Protocol)*** az a nyelv, amely segítségével a szolgáltatástárból kikeresett, és onnan megkapott WSDL-dokumentumok alapján, össze tudják állítani a kliensek a szolgáltatás igénybevételéhez szükséges kérésüket. Tehát magának a hívandó metódusnak a paramétereit.

Ezen megoldások a szolgáltatások fejlesztését könnyebbé, egységesebbé tették, és mivel nemcsak kicsi, hanem óriás cégek is a szabvány mögött sorakoztak fel, nagyon gyorsan elterjedtek, és napjainkban élik igazán fénykorukat.

Webszolgáltatások alkalmazása

Webszolgáltatások alkalmazása egyértelműen hasznos, több szempontból is. Egyrészt azáltal, hogy már meglévő alkalmazásokat, funkcionalitásokat teszünk elérhetővé webszolgáltatásként, nő azoknak a száma, akik a szolgáltatást el tudják érni A háttérben azonban nem történt funkcionalitásbeli különbség, csupán egy adott szabvány szerint alakítottuk át már meglévő rendszerünket. Így kihasználtuk az újrafelhasználhatóság nyújtotta előnyöket. Másrészt az egyes szolgáltatások között a függőség minimális, amit **lazán csatoltság**nak neveznek. Ennek az az előnye, hogy a rendszerünk könnyebben menedzselhető, a hibák könnyebben felderíthetők és javíthatók. Nem beszélve az

esetlegesen később megvalósítandó szolgáltatások könnyebb integrációjáról. Érdemes tehát összefoglalni, milyen előnyökkel is jár egy XML webszolgáltatás megvalósítása:

- Függetlenség: az XML mint adatreprezentációs réteg és a SOAP, mint platformfüggetlen szállítási réteg segítségével az eltérő platformok és eltérő programozási nyelvek többé nem jelentenek akadályt.
- Webes megjelenés: a funkcionalitások igénybe vevői lehetnek a világ bármely pontján, az internet segítségével használni tudják az adott szolgáltatásokat.
- Nem saját szolgáltatások igénybevétele: egy már létező, és publikus szolgáltatást nem kell nekünk megírnunk, használhatunk egy másik cég által nyújtott eszközt is. (pl. Google Maps)
- Integráció: ha egy új rendszerre kell átültetnünk már meglévő szolgáltatásunkat, vagy már meglévő rendszerre fejlesztünk, az XML használatával nincsenek integritási gondok, hiszen bárhová is telepítjük, biztosak lehetünk benne, hogy szabványos módon történik a „beépülés".
- Könnyebb fejleszthetőség: egy szabványosított rendszer mindig is könnyebben kezelhető volt, mint egy teljesen egyedi technika, és ez nem csak az IT szektorra igaz. Ha valami szabványosítva van, akkor arra lehet hivatkozni, azt könnyebben lehet tanulni, és könnyebb alkalmazni is, hiszen bárhová is megyünk a világon, a szabvány ugyanaz. Ezáltal a fejlesztőknek is könnyebb a dolguk, hiszen „csak" el kell sajátítaniuk a szabvány által lefektetett szabályokat, és akárhol is dolgozzanak, ugyanazt fogják tőlük számon kérni.
- Több emberhez is eljuthat a szolgáltatás: a függetlenség okán olyan embereket is el tudunk érni, akik korábban a készülékeik sajátosságai miatt képtelenek voltak igénybe venni az adott szolgáltatást.

Simple Object Acces Protocol

A Simple Object Acces Protocol számítógépek közti információcseréhez alkalmazott XML alapú protokoll. Legfontosabb felhasználási módja a HTTP protokollon keresztül történő távoli metódushívás. A kliensek egy ún. **SOAP üzenet** segítségével hívják meg a távoli webszolgáltatást, a fogadó, vagyis a szolgáltató pedig egy **SOAP válasz**ban nyújtja az igényelt funkcionalitást. Közben az átküldött XML üzenet olyan információkat tartalmaz, amelyek alapján a webszolgáltatás az üzenetben megtalálható

adattípusokat le tudja képezni a megfelelő implementációra, melyet az adott platform megvalósít.

A SOAP üzenet

Mind a kliens kérése, mind pedig a szerver válasza egy-egy SOAP üzenetnek felel meg. Egy SOAP üzenet a következő részekből áll vagy állhat:

- Boríték: kötelező elem. Az XML-dokumentum gyökerének tekintendő. Tartalmazza magát az üzenetet, és lehetőség van az üzenettel kapcsolatos egyéb információk elhelyezésére is.
    - Fejléc: opcionális. Az előbb említett egyéb információk elhelyezésére ad lehetőséget. Ilyenek például a biztonsági beállítások.
    - Törzs: kötelező. Itt szerepel például a hivatkozás az adott szolgáltatásra, az egyes paraméterek is itt találhatók meg stb. Hiba esetén pedig egy *Fault* elem segítségével informálódhatunk a bekövetkezett hibáról.

SOAP kódolás

Szükségünk van a különböző adattípusok XML-beli megfeleltetésére és fordítva. Erre a SOAP egy ún. *sémaleírót (XSD)* használ . Segítségével az egyes programozási nyelvekben használatos típusokat nyelvfüggetlen módon tudjuk leírni. Ezen kívül kell, hogy legyen egy eszköz, mely a konkrét alkalmazásokhoz létrehozza a megfelelő XML-sémákat a fent említett sémaleíró használatával. Erre a SOAP tartalmaz egy *alapértelmezett kódolás*t. Az egyes *kódolási eljárások* meghatározzák, hogy az aktuális programozási nyelv típusai hogyan feleltethetők meg az XML szintjén és fordítva. A két folyamatot, melyben a programozási nyelv típusait XML-re, és az XML-elemeket programozási nyelv típusokra alakítjuk, nevezzük *szerializáció*nak és *deszerializáció*nak.
A SOAP törzs leírására a következő lehetőségeink vannak:

- Document: strukturálatlan üzenetek készíthetők ezzel a módszerrel.

- RPC: Remote Procedure Call – Távoli eljáráshívás. Szinkron szolgáltatás. A szolgáltatást, mint távoli eljárást hívjuk meg, mely azonnal le is fut. A törzs tartalmazza a hívandó metódus nevét, és paramétereit.

Ezeket nevezzük *stílus*oknak. Beszélnünk kell még a stíluson kívül az üzenet *kódolás*áról. A törzs leírása esetén nincs semmilyen előírás a struktúrára vonatkozóan, de megadunk egy sémát, amire illeszkednie kell, vagy sem. Ebből a szempontból beszélhetünk:

- Literal: a törzs megfelel az adott XML-sémának
- SOAP-kódolás: a törzsnek nem kell megfelelnie a sémának. Ebben az esetben szabályokat alkalmaz az adatok kódolására.

A stílus és kódolás fent említett változatai alapján négyféle módon adhatjuk meg a törzset:

- RPC-stílus – SOAP kódolás: támogatott, de inkább csak kompatibilitási okokból.
- RPC-stílus – literal kódolás: támogatott.
- Document-stílus – SOAP-kódolás: gyakorlatilag semmilyen eszköz nem támogatja.
- Document-stílus – Literal-kódolás: támogatott, és gyakran használt.

Web Service Description Language

A WSDL a SOAP mellett a másik lényeges része a webszolgáltatások architektúrájának.
Feladata specifikálni, hogy hogyan lehet leírni az adott webszolgáltatást. Kérdés, hogy miket kell tartalmaznia. Mire is van szükségünk ahhoz, hogy egy létező webszolgáltatást igénybe vegyünk? Ahogy az a bevezetőben is kiderült a WSDL XML alapokon, távoli metódusokként határozza meg a webszolgáltatásokat, és mint ilyennek a következő információkat kell tartalmaznia:

- Cím: pontosan hol érhető el az adott szolgáltatás?

- Interfész: a funkció leírása, milyen paramétereket vár el, mit ad vissza stb.
- Adattípusok: a kérések és válaszok során használt adattípusokra vonatkozik.
- Kapcsolattípus: az adott átviteli protokollról információ.

Összességében tehát a webszolgáltatás a konkrét szolgáltatáson kívül annak leírását is tartalmazza, egyfajta **önleíró jellegg**el.

WSDL specifikáció

A WSDL dokumentum a következő részekből áll:

- massage – üzenet: egyirányú üzenet leírása, akár a hívó, akár a hívott felé. Az üzenet mindig egyirányú kapcsolatot jelent. Ha kétirányú kapcsolatra van szükség (ez az általános), akkor legalább két üzenetet kell definiálnunk. A hibákat szintén üzenetként kell kezelnünk.
- types – típusok: az üzenetekben használt típusok leírása az *XML Schema* specifikáció alapján. A típusok leírása a már korábban is említett *XSD*-vel történik, mely nem csak egyszerű, hanem összetett adattípusok leírására is alkalmas.
- portType – porttípus: a webszolgáltatás által végrehajtódó műveletek. A porttípus a programozás nyelvek függvény könyvtárához hasonlítható. Tulajdonképpen a logikailag összetartozó műveleteknek egy névvel ellátott halmaza.
- binding – kötés: az adott portType művelet hálózaton történő átvitelének részleteit írja le. Minden egyes portra meghatározza az üzenet formátumát és a protokoll jellemzőit.
- service – szolgáltatás: a szolgáltatás helyét adja meg.
- operation – művelet: a szolgáltatás egy funkciójának absztrakt definíciója.
- port: egy adott végpont, mely egy hálózati név és egy kötés összeillesztését jelenti. Egy konkrét címet rendel a kötéshez.

WS-I (Web Services Interoperability Organization)

Bármennyire is helyes, és követendő elgondolás volt az egyes szolgáltatások együttműködése platform- és programozási nyelv független módon, néhol „hiba csúszott a gépezetbe”. Nem volt tökéletes a specifikáció, és így az egyes implementációk eltérhettek egymástól. Itt már egy kicsi eltérés is nagyon sokat jelent. Akár a leképezésben, akár az adatátvitel módjában vagy más részletekben megjelenő eltérések meghiúsíthatták az egyes szolgáltatások együttműködését. Mert bár lehet, hogy egyenként működtek az adott funkciók, de ha egymást szerették volna hívni, fel szerették volna használni egy másik egység nyújtotta szolgáltatást, akkor problémákba ütköztek. Viszont a webszolgáltatások létrehozásának egyik alapvető célja pont az volt, hogy az egyes kisebb egységek egy nagyobb, összetettebb egységgé összeállva is képesek legyenek működni.

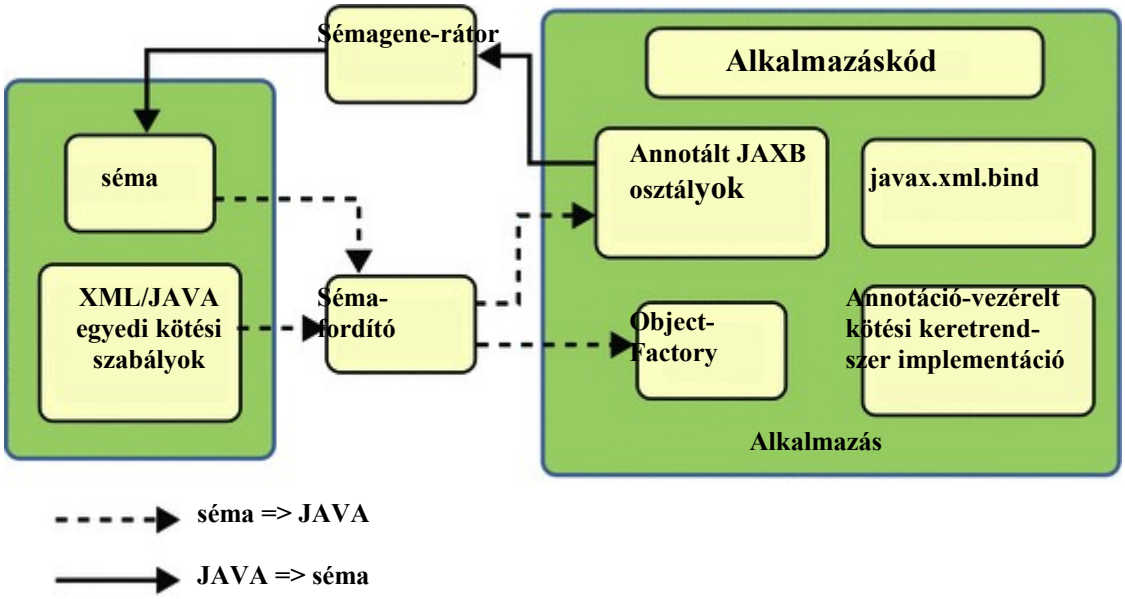
A fent említett problémák megoldására született meg a *WS-Interoperability* szervezet. Ez egy minden cég számára nyitott ipari kezdeményezés, mely irányelveket fogalmaz meg az egyes webszolgáltatás gyártók számára, hogy olyan szolgáltatásokat tudjanak gyártani, amelyek valóban képesek az együttműködésre. Ebből a szempontból hasonló a cél a korábban említett EJB-k és alkalmazásszerverek együttműködését megcélzó szabványosítással.

Olyan nagy cégek sorakoztak fel a kezdeményezés mögött, mint például a Microsoft, a SUN, az IBM, az Oracle corp., a Fujitsu, a BEA Systems Inc. és még sokan mások. Látható, hogy óriási támogatottságnak örvend ezen elgondolás, így az egyes fejlesztőeszközökkel szinte csak a WS-I kompatibilis webszolgáltatásokat lehet készíteni.

A WS-I különféle tesztekkel, szabálygyűjteményekkel stb. segíti a webszolgáltatások fejlesztését. Úgynevezett *profil*okkal dolgozik, amelyek egy specifikáció halmazt határoznak meg. Valamint módszereket mutat be arra, hogy a specifikációk hogyan is működjenek együtt. Ezen kívül példamegoldásokat is ad az egyes problémák megoldására. Például az egyik legismertebb ilyen profile a *Basic Profile*, mely kb. 200 gyakran előforduló problémára adja meg a helyes választ.

Java Architecture for XML Binding

A szoftverfejlesztés során a programozók osztályokban és objektumokban gondolkodnak. A webszolgáltatások pedig XML-üzeneteken alapulnak. Elkerülhetetlen tehát a 2 világ közötti konverzió. A **JAXB** egy specifikáció, mely az XML-üzenetek és Java-osztályok, -objektumok közötti konverziót hivatott megoldani.
Az egyes XML-dokumentumok felépítésének meg kell felelnie az XSD szabványnak. Ekkor tudja ugyanis a JAXB megfelelő módon kezelni. A JAXB felépítése a következőképpen néz ki:



1. ábra: A JAXB felépítése
forrás: [11]

A **sémafordító** a kapott XML-sémából az egyedi kötési szabályok segítségével Java-osztályokat készít. A fordított irányú tevékenységeket, tehát amikor a Java-osztályokból készítünk XML-sémákat a **sémagenerátor** végzi. Mindkét irányban az **annotációk** jelentőssége nagy. A sémafordító többnyire annotált Java-osztályokat készít, a sémagenerátor pedig annotált Java-osztályokból alakítja ki az XML-sémát.
Azt a folyamatot, mely során beolvassuk az XML-dokumentumot, majd az XML-dokumentum elemeiből egy fát,- az ún. **tartalomobjektumokból képzett fát** – építünk, nevezzük *unmarshalling*-nak. Ennek ellenkező irányú végrehajtását *marshalling*-nak hívjuk.

Java API for XML-Based Web Services

J2EE 1.4 verziótól kezdve a Java-nak része 3 API mely a webszolgáltatások készítését hivatott elősegíteni:

- JAXR – Java API for XML Registries: a webszolgáltatás nyilvántartók feladata, hogy az adott szolgáltatásról információkkal lássák el a klienseket. Ilyen információk például, hogy mit nyújt az adott szolgáltatás, hogyan lehet meghívni, stb. A **JAXR** a webszolgáltatások regisztrálását teszi lehetővé ezekben a nyilvántartókban.
- SAAJ – SOAP API with Attachments for Java: a SOAP üzenetek létrehozásához és kezeléséhez szolgáltat osztályokat, interfészeket.
- JAX-RPC – Java API for XML-based RPC: a Java és XML elemek közötti oda-vissza leképezés megvalósítását segíti.

Hasonló problémákkal és nehézségekkel kellett szembenézniük azoknak, akik JAX-RPC webszolgáltatásokat fejlesztettek, mint akik EJB 2.1-ben programoztak. Ugyanis a JAX-RPC esetén is problémát jelentett, hogy sok fájllal kellett dolgozni, ez pedig megfelelő támogatás – pl. egy jó fejlesztőeszköz – nélkül rendkívül nehézkes volt. Ha JAX-RPC-ben kívántunk megvalósítani egy webszolgáltatást, akkor a következőket kellett elkészítenünk:

- SEI - Service Endpoint Interface: a megvalósítani kívánt webszolgáltatás műveleteinek megadása Java-interfészben a java.rmi.Remote kiterjesztettjeként.
- SIB - Service Implementation Bean: az implementáló osztály. A SEI által deklarált műveleteket itt kell implementálni. Nem kötelező elem.
- jaxrpc-mapping fájl: definiálja az oda-vissza történő leképezést.
- webservices.xml: a SEI és a SIB összerendeléséért felelős. Ő felel a WSDL fájl, és az előbb említett jaxrpc-mapping fájl helyének megadásáért.
- Webszolgáltatás végpont: milyen URL segítségével érhetik el a kliensek az adott szolgáltatást. Ha webrétegben fejlesztünk ezt *WEB-INF/web.xml* fájl tartalmazza EJB esetén alkalmazásszervertől függ.

Ezután az elkészült fájlokat – illetve a lefordított .class fájlokat és telepítés leírókat - attól függően, hogy webrétegben vagy EJB rétegben fejlesztettünk, egy .war- vagy .jar-ba kellett tömöríteni.
Ezzel szemben a JAX-WS esetében nincs szükség a webservices.xml és jaxrpc-mapping fájlokra, és még a web.xml-t sem kell módosítani. Mindössze annyi a dolgunk, hogy az egyes osztályokat lefordítjuk és becsomagoljuk a fejlesztéstől függően .war-ba

illetve .jar-ba. Mindezt úgy tudjuk megoldani, hogy – hasonlóan az EJB 3 fejlesztéshez –annotációkkal látjuk el azokat az osztályokat, illetve metódusokat, melyeket webszolgáltatásként szeretnénk publikálni. Sőt, még SEI-t sem kell külön megadnunk. Ha az implementációs osztályunk tartalmaz egy paraméter nélküli konstruktort akkor abból a JAX-WS ún. **implicit SEI**-t definiál. Erről később még lesz szó.

Leképezések

A JAX-WS a Java-XML és az XML-Java leképezésekre már a JAXB-t használja. A következő táblázatok foglalják össze az oda-vissza történő leképezést az egyes típusok között. Szembetűnő, hogy a JAXB többre képes, mint a JAX-RPC 1.1

XML - JAVA

Type	JAX-RPC 1.1	JAXB 2.0
xsd:anySimpleType	java.lang.String	java.lang.Object
xsd:duration	java.lang.String	javax.xml.datatype.Duration
xsd:dateTime	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar
xsd:time	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar
xsd:date	java.util.Calendar	javax.xml.datatype.XMLGregorianCalendar
xsd:gYearMonth	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gYear	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gMonthDay	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gMonth	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:gDay	java.lang.String	javax.xml.datatype.XMLGregorianCalendar
xsd:anyURI	java.net.URI	java.lang.String
xsd:NMTOKENS	java.lang.String[]	java.util.List<java.lang.String>
xsd:IDREF	java.lang.String	java.lang.Object
xsd:IDREFS	java.lang.String[]	java.util.List<java.lang.Object>
xsd:ENTITY	nem támogatott	java.lang.String
xsd:ENTITIES	nem támogatott	java.util.List<java.lang.String>

Forrás: [12]

JAVA – XML

Type	JAX-RPC 1.1	JAXB 2.0
java.lang.String	xsd:string	xsd:string
java.math.BigInteger	xsd:integer	xsd:integer
java.math.BigDecimal	xsd:decimal	xsd:decimal
java.util.Calendar	xsd:dateTime	xsd:dateTime
java.util.Date	xsd:dateTime	xsd:dateTime
javax.xml.namespace.QName	xsd:QName	xsd:QName
java.net.URI	xsd:anyURI	xsd:string
javax.xml.datatype.XMLGregorianCalendar	nincs	xsd:anySimpleType
javax.xml.datatype.Duration	nincs	xsd:duration
java.lang.Object	nincs, esetleg xsd:anyType	xsd:anyType
java.awt.Image	nincs (MIME binding)	xsd:base64Binary
javax.activation.DataHandler	nincs (MIME binding)	xsd:base64Binary
javax.xml.transform.Source	nincs (MIME binding)	xsd:base64Binary

java.util.UUID	nincs	xsd:string

Forrás: [12]

JAVA-WSDL

Ahogy korábban utaltam rá, a JAX-WS képes implicit SEI-t definiálni a paraméter nélküli konstruktorral rendelkező, *és @WebService* annotációval ellátott osztályból. A definiálás során azonban eltérések tapasztalhatók abból a szempontból, hogy melyik metódus legyen megvalósítva webszolgáltatásként. Ezzel kapcsolatban a következőket kell tudni:

- Egy olyan metódus sem lesz publikálva webszolgáltatásként, amelyet az osztály az *object ősosztály*tól örököl.
- Ha egy metódus sincs megjelölve a **@WebMethod** annotációval, akkor az összes publikus metódus publikálva lesz. Kivéve persze az előbb említett object ősosztályból származókat.
- A *@WebMethod* annotációnak létezik egy *exlude* nevű paramétere. Ha ez true értékű - *@WebMethod (exclude=true)* -, akkor az adott annotációval ellátott metódus sem lesz publikálva.
- Ha vannak olyan publikus metódusok, melyeknél a *@WebMethod exlude* paramétere *false* értéket vesz fel – ez az alapértelmezett – akkor csak ezek lesznek publikálva.

A *@WebMethod* annotációban nemcsak az exlude paramétert helyezhetjük el. Itt kaphatnak helyet azok a paraméterek melyek az egyes metódusok WSDL-beli megfelelőjének beállításait adják meg. Például az *operationName* nevezetű a *wsdl:operation name* attribútumát állítja be.
A JAX-RPC esetében létezik az ún. jaxrpc-mapping fájl, amely tulajdonképpen a java – wsdl (XML) leképezést hivatott definiálni (például: wsdl portok, wsdl műveletek stb.). JAX-WS esetében ezeket a beállításokat a *@WebService* annotáció nem kötelező paramétereiben állíthatjuk be. Ilyen paraméterek például: *serviceName*, mely a *wsdl:service* neve, vagy a *wsdlLocation*, mely egy már létező wsdl-fájl eléréséhez szükséges adatokat adja meg.
JAX-WS esetében a SOAP kódolásnál említett négy kombináció közül csak kettőt használunk. A Document/SOAP, mint más eszközök esetében, itt sem támogatott. Amiatt pedig, hogy a SOAP-kódolás nem WS-Interoperability kompatibilis, nem

választhatjuk az RPC/SOAP módszert sem. Viszont megjelenik az ún. *csomagolt dokumentum/*literal stílus.

A leírtakra láthatunk egy példát a következőkben. Ez egy egyszerű webszolgáltatás, amelynek csak annyi a funkciója, hogy a bekért sztringnek visszaadja a hosszát. A webszolgáltatás EJB-rétegben van megvalósítva, és RPC-Literal stílust használ:

Session bean

```
package beans;

import javax.ejb.Stateless;

@Stateless
public class hBean implements hLocal {

    public int h(String s) {
        return s.length();
    }

}
```

Sei

```
package ws;

import beans.hLocal;
import javax.ejb.EJB;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.ejb.Stateless;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;

@WebService()
@SOAPBinding(style=SOAPBinding.Style.RPC)
@Stateless()
public class wServ {
    @EJB
    private hLocal ejbRef;


    @WebMethod(operationName = "h")
    @WebResult(partName="ittAHossz")
    public int h(@WebParam(partName="aString")
            String s) {
        return ejbRef.h(s);
    }

}
```

WSDL (részlet)

```
<message name="h">
<part name="aString" type="xsd:string"/>
</message>
−<message name="hResponse">
<part name="ittAHossz" type="xsd:int"/>
</message>
−<portType name="wServ">
−<operation name="h" parameterOrder="aString">
<input message="tns:h"/>
<output message="tns:hResponse"/>
</operation>
</portType>
−<binding name="wServPortBinding" type="tns:wServ">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
−<operation name="h">
<wsp:PolicyReference URI="#wServPortBinding_h_WSAT_Policy"/>
<soap:operation soapAction=""/>
−<input>
```

```
<soap:body use="literal" namespace="http://ws/"/>
</input>
−<output>
<soap:body use="literal" namespace="http://ws/"/>
</output>
</operation>
</binding>
−<service name="wServService">
−<port name="wServPort" binding="tns:wServPortBinding">
<soap:address location="http://localhost:8080/wServService/wServ"/>
</port>
</service>
```

Látható, hogy az annotációkban megadott módon képződött le. A kérés üzenet paraméterének neve ***aString***, a válasz üzenet paramétere ***ittAHossz***. A típusok szintén beállítódtak, első esetben ***string***, második esetben ***int***. Ezenkívül megjelentek a korábban említett WSDL elemek is, mint például a port, amelyben megtalálható a szolgáltatás konkrét elérési címe. Esetünkben ez *http://localhost: 8080/wServService/wServ*.

SOAP kérés

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ns2:h xmlns:ns2="http://ws/">
         <aString>string</aString>
      </ns2:h>
   </S:Body>
</S:Envelope>
```

SOAP válasz

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:hResponse xmlns:ns2="http://ws/">
         <ittAHossz>6</ittAHossz>
      </ns2:hResponse>
   </S:Body>
</S:Envelope>
```

Szoftverkészítés, mint projekt

Egy nagyobb cégnek készítendő szoftver mindig komplex és adott esetben különleges bonyolultságú lehet. Fontos tehát, hogy még mielőtt a fejlesztéshez hozzákezdünk, pontos képet kapjunk az elkészítendő termék egyes paramétereiről, a megrendelő elvárásairól. A betartandó határidőkről és költségekről nem is beszélve.
Ezen felül, egy komolyabb szoftver elkészítése mindig projektmunkát igényel. Vagyis nem egy ember készíti a teljes rendszert, hanem több, különböző csoportokat alkotva. Ezáltal az egyes programozóknak nem kell túlságosan nagy egységekben gondolkodniuk, elég ha a rájuk rótt feladat megoldásán dolgoznak, az egyes egységek összeillesztése és menedzselése már más munkakörhöz tartozik. Természetesen ezek a munkakörök nagyon sokszor átfedik egymást. Például egy adott programozó dolgozhat a rendszer egy kisebb egységén, és ezen felül segédkezhet az integrációs feladatok megoldásánál is. Amellett, hogy a projektként való felfogás sok előnnyel jár, felmerülnek bizonyos problémák, amelyeket kezelni kell tudni. Ilyenek például az egyes programozók – projekttagok – feladatainak meghatározása, az egyes projekttagok közötti kommunikáció megteremtése, a projekt felügyelete, irányítása és még számos egyéb megoldandó feladat, amelyekkel a projektmenedzsmentnek meg kell birkóznia.

Szoftver életciklus

Egy szoftver életciklusa a következő szakaszokra bomlik:

- Követelmények felállítása (mit várunk el a rendszertől?): azon elvárások összegyűjtése, amelyeket a leendő rendszernek teljesítenie kell.
- Elemzés, analízis (követelményelemzés, absztrakt körbejárás, megvalósítható-e egyáltalán?): a feltárt követelmények alapján eldöntjük, hogy érdemes-e egyáltalán belevágni a projektbe.
- Architekturális elemzés: a rendszer struktúrájának tervezése, platformfüggetlen.
- Tervezés (nem platformfüggetlen): az architekturális elemzés eredményét adott platformra ültetjük át.
- Implementálás: konkrét megvalósítás.

- Tesztelés: az elkészült termék – szoftver - hibáinak és hiányosságainak feltárása.
- Bevezetés: felhasználókkal való megismertetés.
- Üzemeltetés: rendszeres használat.
- Karbantartás (továbbfejlesztés + patch-ek): rendszeres használat során felmerülő problémák javítása, új ötletek megvalósítása.
- Üzemen kívül helyezés: a rendszer felfüggesztése, leállítása.

Dolgozatomban a követelmények elemzésére és a tervezés szintjén az UML-re térek ki részletesen.

Megvalósíthatósági tanulmány

Sokszor a rendszer követelményeinek feltárása előtt készül egy megvalósíthatósági jelentés, amely a rendszer nagyvonalakban történő felvázolását jelenti. Ezért nevezik ezt vázlatnak is. Arra szolgál, hogy eldöntsük egyáltalán van-e értelme belefogni a fejlesztésbe, a kívánt szoftver megvalósítható-e és van-e haszna. A megvalósíthatósági tanulmányban általában a következő kérdésekre keressük a választ:

- A kifejlesztendő szoftver támogatja-e a megrendelő vállalat célkitűzésit, munkáját?
- Az adott rendszer megvalósítható-e az adott időkereten belül, adott költségek betartása mellett, adott technológiával?
- Az elkészítendő rendszer integrálható-e a vállalat már meglévő infrastruktúrájába?

Követelmények feltárása

*Követelmény*eknek nevezzük azokat az igényeket, elképzeléseket, jellemzőket, amelyek körülhatárolják az elkészítendő szoftverrendszer működését. Azt a fázist, ahol ezeket összegyűjtjük, elemezzük, dokumentáljuk és ellenőrizzük *követelmény feltárás*nak, vagy *követelmény elemzésnek* nevezzük.

A követelmények feltárása sosem független a megrendelőtől. Sőt, kifejezetten a megrendelő és a leendő felhasználók igényeit kell figyelembe venni akkor, amikor a követelményeket szeretnénk feltárni. Természetesen a dobozos szoftverek készítésénél más dolgokra is oda kell figyelni, és a tervezés is más módon zajlik. De a dolgozatomban bemutatni kívánt nagyvállalati szoftvertervezésre leginkább az egyediség, és a megrendelő cég irányából jövő különleges igények jellemzők.

Aki készített már programot – legyen szó bármilyen kisméretűről is – az tudja, milyen bosszantó tud lenni, ha időközben derülnek ki olyan elvárások, amelyekről a programozás kezdete előtt szó sem volt, vagy esetleg a már meglévő elvárások módosulnak. Ilyenkor az egész program logikáját át kell gondolni, illeszteni az új elvárásoknak megfelelő funkcionalitást, tesztelni, hogy a már működő kódban nem tett e kárt az újonnan beillesztett rész stb. Ez rendkívül sok vesződséggel jár, ellenben ha már a programozás kezdete előtt tisztán láttunk volna, ezektől megtudtuk volna magunkat kímélni. Kulcsfontosságú tehát a szoftverfejlesztés életciklusai közül a követelmények elemzése, feltárása.

A követelményeket három csoportra oszthatjuk:

- Funkcionális követelmények: mit várunk el a rendszertől a szolgáltatások terén. Milyen feladatokat kell tudnia a rendszernek megoldani, milyen körülmények között, milyen elvárásokkal. (pl. tantárgyakat lehessen meghirdetni, lehessen böngészni a termékek között, lehessen átutalni egyik számláról a másikra stb.)
- Nem-funkcionális követelmények: a rendszer egészére vonatkozó elvárások. Sokkal komplexebb, mint az előző. Ha a funkcionális követelmények közül a rendszer nem teljesíti valamelyiket, attól még működőképes. De ha a rendszerkövetelmények közül akár egy is hiányos, az nagyon sok problémához vezethet. (pl. rendelkezésre állás, biztonság, teljesítmény stb.)

- Lehatárolás: fontos azokkal a dolgokkal is tisztában lenni, amikre nincs szükség a rendszer készítése során. Természetesen itt **releváns követelmények**ről beszélünk, nem olyanokról, amik eleve szóba sem jöhetnének. Például egy tanulmányi rendszernél magától értetődik, hogy nincs szükség olyan funkcionalitásra, hogy termékeket tudjunk megrendelni. De egy webáruháznál fontos megtudni például, hogy fizetésnél milyen valutába történő átváltást támogasson a rendszer, hiszen ha csak adott országok érik el az áruházat, nem fontos, hogy más országok valutája is szerepeljen az átváltandók között. Ha a releváns, nem megkövetelt funkcionalitásokkal is tisztában vagyunk, sok felesleges kódolástól kímélhetjük meg magunkat, illetve a céget, amely a szoftvert fejleszti.

A követelmények felosztásának több módszere is létezik. A fent említett csak egy a sok közül.

Érdemes még megemlíteni, hogy létezik – akár a fentiekkel párhuzamosan is - egy másik csoport, méghozzá a *szakterületi követelmények*. A szakterületi követelmények egy-egy adott szakterülethez kötődnek a megrendelői oldalról. Ezek lehetnek funkcionális követelmények is, de nem az adott rendszerre vonatkozóan, hanem általánosságban, az adott szakterületről jövően. Ezenkívül lehetnek megszorítások is, szintén az adott szakterület vonatkozásában.

A funkcionális követelményeknek **teljes**nek és **ellentmondásmentes**nek kell lenniük. Ezt a kettőt nagyon nehéz biztosítani. Szinte lehetetlen. A teljesség arra vonatkozik, hogy minden elvárással tisztában kell lennünk, az ellentmondás-mentesség pedig azt jelenti, hogy ezek a követelmények ne mondjanak egymásnak ellent.

Kérdés, hogyan kell a követelményeket feltárni? Kiket kell bevonni a folyamatba? Egyáltalán kiket érdemes megkérdezni? Itt jönnek a képbe a **kulcsfigurák**.

A kulcsfigura az a személy, aki valamilyen módon kapcsolatba kerül majd a leendő rendszerrel, vagy már kapcsolatban van azzal a rendszerrel, amelyet a leendő szoftver le kíván váltani. Kulcsfigura lehet pl. maga a megrendelő, leendő felhasználók, rendszergazdák stb. A követelmények feltárását tehát a kulcsfiguráktól beszerzett információk elemzése, rendszerezése és validálása jelenti. Jellemző, hogy már a kezdet kezdetén, a kulcsfigurákkal történő konzultáció során problémák merülnek fel:

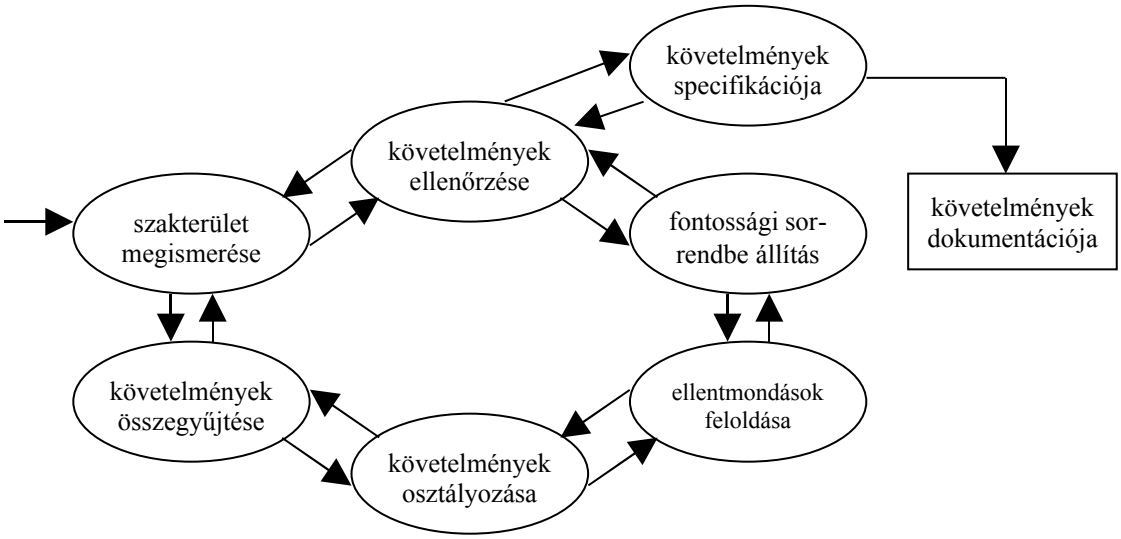
- A kulcsfigurák általában nem tudják, hogy mit várnak el a rendszertől, vagy ha esetleg tisztában is vannak vele, nem tudják megfogalmazni.
- Nincsenek meg a kellő ismereteik a számítógépekkel kapcsolatban. Sokszor túlságosan is irreális, megvalósíthatatlan követelményekkel álnak elő. Ennek a másik oldala, amikor túlságosan triviálisak, tehát gyakorlatilag olyan, mintha nem is mondtak volna semmit.
- Létfontosságú a megfelelő kommunikáció a szoftverfejlesztő cég munkatársai és a kulcsfigurák között. Figyelembe kell venni, hogy a megrendelők nincsenek tisztában a szakmai fogalmakkal, és ez fordítva is igaz. A fejlesztő cég munkatársai nincsenek tisztában a megrendelő oldalán megjelenő szakmaisággal. Természetesen nem várhatjuk el, hogy a megrendelő képezze magát, és tanulja meg a „mi nyelvünket", nekünk, a szoftverfejlesztőknek kell megtanulnunk az adott szakterületre vonatkozó legfontosabb szabályokat, fogalmakat.
- Az egyes kulcsfigurák egymásnak ellentmondó véleményen is lehetnek. A fejlesztők feladata, hogy ezen vélemények közül kiválasszák a valósakat. Emellett az esetlegesen felmerülő ellentmondásokat feloldják.

A megrendelő elvárásai mellett van még egy olyan körülménycsoport, amely döntően befolyásolja a szoftverünk készítését. Ezek pedig a **külső körülmények**. A külső körülmények típusai:

- Politikai: itt lehet érteni az adott cég üzletpolitikáját, de érthetjük alatta a hétköznapi életből vett politikát is.
- Gazdasági: állandóan, dinamikusan változik. Naprakésznek kell lenni nemcsak a megrendelőhöz kapcsolódó gazdasági tényezőkkel, hanem a fejlesztőknek is lépést kell tartaniuk a technológia változásaival, hiszen ha nem teszik, és egy konkurens cég esetleg hamarabb alkalmaz sikerrel egy adott technológiát, az súlyos gazdasági következményekkel járhat a cég szempontjából.

- Figyelembe kell venni ezeken kívül még a törvényi megkötéseket is.

A követelmények feltárásának lépéseit mutatja be a következő ábra:



Ábra 2: Követelmény feltárás lépései

forrás: [Dr. Juhász István – Programozási technológiák - órai jegyzet]

Követelmény feltárási technikák

- Nézőpont orientált módszer: középpontjában az a szemléletmód áll, hogy az egyes kulcsfigurák a rendszert más-más szemszögből látják. Például egy tanulmányi rendszert másként lát egy hallgató, másként egy oktató és megint másként az adminisztrátor.
- Forgatókönyv: A rendszerrel való interakciókat írják le. Egy forgatókönyv jobbára egyet. Általában a következőket szokta tartalmazni:
    - A rendszer állapotának leírása induláskor.
    - A rendszer adott forgatókönyvhöz tartozó normális működésének leírása.
    - Kivételes szituációk kezelése.

- Párhuzamos tevékenységek.
- A rendszer állapotának leírása a forgatókönyv végén.

Manapság rendkívül népszerű, melynek oka, hogy mind a fejlesztőkhöz, mind pedig a felhasználókhoz, megrendelőkhöz közel áll. Sokkal érthetőbb. Általában **használati eset diagramok**kal együtt szokták használni. (A használati eset diagramról még lesz később részletesen szó)

- Etnográfia: megfigyelésen alapuló technika. Megnézzük, hogyan zajlanak a dolgok a való életben. Az egyes folyamatok egymásra épülését remekül meg lehet figyelni. Ezenkívül olyan ismeretekre lehet szert tenni, amelyeket hagyományos módon nem tudhatnánk meg, mert pl. annyira triviálisnak gondolják a kulcsfigurák, hogy el sem mondják. Hátránya, hogy viszonylag sok időbe telik.

Követelmények dokumentálása

Ha összegyűjtöttük a követelményeket, azokat dokumentálni is kell. Egyrészt azért, hogy a szoftverfejlesztés minden egyes fázisában elő lehessen venni, és ellenőrizni, hogy a fejlesztés valóban jó irányba halad-e. Másrészt jó, ha a megbeszélt követelményeket a megrendelővel hitelesíttetjük. Ez sok problémától megkímélhet a jövőben. Ugyanis így a megrendelő a szoftver átadásakor nem tud arra hivatkozni, hogy „nem ez volt megbeszélve”.

A követelmények dokumentálása többféle módon történhet:

- Leírás emberi nyelven: Előnye, hogy mindenki megérti. Hátránya, hogy félreérteni is könnyebb.
- Formanyomtatványok: Természetes nyelven, de strukturáltan. Ez az egyik leggyakoribb módszer.

- Teljesen formalizált nyelv: Szakmabelieknek, informatikusoknak jó, de a felhasználónak nehézkes.
- Diagramok: Ugyanaz mondható el róla, mint az előzőről.
- Matematikai formalizálás: Nagyon absztrakt, nagyon nehéz.

Manapság **hibrid technikákat** alkalmaznak. Általában diagramok és formanyomtatványok együttesét.

Követelmények validálása

A követelmények validálása már egy teljes követelmény dokumentációra vonatkozik. Részeredményeket, kisebb egységeket nem érdemes validálni. Ez a lépés elég sokszor kimarad a fejlesztés során.

A követelmények validálása nem más, mint egy sorozat, melyben a felhasználók/megrendelők és a fejlesztők lépésről-lépésre megnézik az összes követelményt, és a következő tulajdonságokat próbálják meg teljesíteni:

- Ellentmondásmentesség: az egyes elvárások nem mondanak-e egymásnak ellent.
- Teljesség: minden olyan elvárás, feltétel le van-e fektetve, amely a rendszertől elvárt funkcionalitásokat írja körül.
- Megvalósíthatóság: a megvalósíthatósági tanulmány adott egy kezdetleges becslést, hogy érdemes-e egyáltalán belevágni a projektbe, de az csak egy vázlatos leírás alapján becsült. Most már megvannak a konkrét elképzelések, újból fel kell tennünk a kérdést, hogy adott időkeretben, a rendelkezésre álló pénzösszegből, adott technológia mellett, megvalósítható-e a projekt.
- Verifikálás: ez tulajdonképpen egyfajta előre gondolkodás. Ugyanis azt próbáljuk leírni, hogy hogyan tudjuk majd bebizonyítani, a majdan átadásra kerülő szoftverről, hogy teljesíti a dokumentációban foglaltakat.

Különböző validációs technikák léteznek, ezek közül néhány: felülvizsgálat, automatikus validálás, teszteset generálás, prototípuskészítés.

Tervezés Unified Modeling Language segítségével

Az objektum-orientált programozási módszerek egyértelmű előrelépést jelentettek a rendszerfejlesztés során. Segítségével könnyebben módosítható és alapjában véve könnyebben kezelhető szoftvereket írhatunk. Az OO mögött azonban nem volt olyan matematikai modell, amely hűen vissza tudta volna adni annak dinamizmusát. Szükség volt tehát egy olyan egységesített jelölésrendszerre, mely az OO világ elemeit egyszerű ábrákkal tudja modellezni. Fontos szempont volt többek között a könnyű használhatóság, a programozási nyelvektől való függetlenség, a kifejezőképesség és a bővíthetőség. Így alakult ki az Unified Modeling Language vagyis az egységesített modellező nyelv.

A rendszerünket különböző nézőpontokból az UML diagramjaival mutathatjuk be. A diagramok valójában olyan gráfok, amelyeknek csomópontjai a vizsgált terület adott elemeit reprezentálják, az élek pedig az egyes elemek közötti meghatározott összefüggéseket tükrözik.

Vizsgálhatjuk a rendszer abból a szempontból, hogy mik azok az úgymond kötött, statikus elemek, melyek a rendszerünk gerincét adják. Ily módon beszélhetünk strukturális diagramokról. Strukturális diagramok közé soroljuk az osztály-, objektum-, komponens és az alkalmazási diagramot.

Ha a rendszerünk dinamikáját szeretnénk modellezni, vagyis az elérhető funkcionalitást kívánjuk reprezentálni, akkor használjuk a viselkedési diagramokat. Viselkedési diagramok közé tartoznak a következők: használati eset-, aktivitás-, szekvencia-, kommunikációs és időzítési diagram.

A következőkben a használati eset-, aktivitás és szekvencia diagramokat mutatom be részletesebben.

Használati eset diagramok – Use Case

A használati eset diagram azokat a követelményeket jeleníti meg az absztrakció különböző szintjein, melyeket az egyes felhasználók elvárnak a leendő rendszertől. A háttérben lezajló folyamatokról, vagyis a tényleges végrehajtás mikéntjéről viszont nem

nyújt információt. Különböző megközelítésekben fejezhetjük ki a rendszer funkcionalitását. Hogy milyen szemszögből mutatjuk be a rendszert, illetve milyen részletességgel, az mindig attól függ, hogy kinek szánjuk a diagram bemutatását. Ha a megrendelőt kívánjuk informálni, akkor természetesen sokkal nagyobb egységekben, közérthetőbb módon kell elkészítenünk. Azonban ha a fejlesztőknek szánjuk, akkor sokkal bonyolultabb, sokkal inkább szakmához kötődő lehet.

Az egyes use case diagramok a következő építőelemekből állhatnak:

- Szereplők: actorok, a rendszeren kívül eső, de a rendszerhez szorosan kötődő elemek. Az actor nem egy konkrét személyt vagy elemet jelöl, sokkal inkább helyesebb a típus megnevezés. Típusban való gondolkodás alapján actor lehet pl. felhasználó, adminisztrátor, SQL Server vagy más alkalmazás. A diagramon az actor jele egy **pálcika ember**.
- Használati esetek: Use Case-ek, azok a funkciók, melyet az előbb említett actorok majdan igénybe kívánnak venni. Ezeket egyfajta összeköttetésnek, kapcsolódási pontnak lehet tekinteni az actorok és a rendszer között. A use case jele egy **ellipszis**, beleírva a használati eset funkcióját.
- Kapcsolatok: a szereplők és a használati esetek közötti viszony, valamint ezek egymás közötti viszonya (actor-actor, use case – use case). Jelölése **nyilak**kal vagy **vonalak**kal történik.

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

Actor és use case között egyetlen kapcsolattípus lehetséges, ez az **asszociáció**, mely annyit tesz, hogy az említett actor és use case egymással kapcsolatban áll. Jele egy **sima nyíl**, vagy ha nem kívánjuk az irányt megadni akkor egy **sima vonal**. 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 <<include>> sztereotípia azon használati esetek jelzésére is alkalmas, melyek egy nagyobb, átfogóbb használati eset részei, de külön hangsúlyozni szeretnénk őket. Ilyen lehet például a regisztráció

során kötelező felhasználónév és jelszó megadásának különválasztása, majd pedig ezeket egy <<include>> kapcsolattal tartalmazza maga a regisztráció use case. Az **<<extends>>** sztereotípiával jelölt kapcsolat annyiban különbözik az előzőtől, hogy az átfogóbb használati esetnek módja van felhasználni a kisebb egységet, de ez nem kötelező számára. Ilyen például az adatok módosításánál az az eset, amikor csak a címünket változtatjuk meg. Actor és actor között létezik még az ún. **általánosítás/pontosítás** mely során egy nagyobb egységekbe tömöríthetjük az adott szerepköröket, és fordítva. Például a felhasználókon belül lehetnek adminok, sima user-ek és tulajdonosok. Az elkészített rendszer „vásárló" típusú felhasználójának use case diagramja (részlet) megtekinthető a függeléken belül. (5. ábra – 71. oldal)

Aktivitás diagram

Az aktivitás diagram segítségével a rendszerünk dinamikus jellegét tudjuk formalizálni, ábrázolni. Az egyes tevékenységek közötti időbeli függés is általa írható le.

Aktivitásnak nevezzük a rendszer használata során végrehajtott tevékenységeket. Ezeket egy **ívelt oldalú téglalap**pal jelöljük. Az egyes aktivitások közötti időbeli függést az ún. **átmenetek**kel jelölhetjük. Ez egy sima **nyíl** az aktivitások között, mely jelzi, hogy az adott tevékenység befejeződött és kezdődhet a következő. Egy aktivitási diagram mindig egy folyamatot jelöl, melynek van egy **kezdőállapot**a, melyet egy **körlap**pal jelölünk, és van egy vagy több **végállapot**a, melyet egy **körben lévő körlap**pal jelölünk. Arra is van mód, hogy bizonyos tevékenységek végrehajtását **feltétel**ekhez kössük. Ezeket a feltételeket **őrszem**eknek nevezzük és az aktivitások közötti nyíl mellett **szögletes zárójel**ben jelezzük. Ha egy feltétel olyan, hogy **több irány**ba is haladhatunk a végrehajtás során a feltétel teljesülésének függvényében, akkor ezt egy **rombosszal** jelöljük, majd pedig a rombuszból indítjuk az egyes aktivitásokhoz vezető nyilakat. Fontos, hogy az innen kiinduló nyilak végén olyan aktivitások legyenek, melyek kölcsönösen kizárják egymást. **Párhuzamos tevékenység**eket **szinkronizációs vonal**ak beiktatásával jelölhetjük. Az ebből a vonalból kiinduló nyilak végén lévő tevékenységeket párhuzamosan is végrehajthatjuk, de nem kötelező módon. A tevékenységek végül egy másik szinkronizációs vonalba futnak össze, vagyis innentől kezdve ismét egy szálon folyik a végrehajtás. Az

elkészített rendszer „tulajdonos” típusú felhasználójának aktivitás diagramja (részlet) megtekinthető a függeléken belül. (3. ábra – 70. oldal)

Szekvencia diagram

Az egyes tevékenységek végrehajtása során a rendszer objektumai üzeneteket küldenek egymásnak. Ez lehet egy metódushívás vagy akár egy hibaüzenet is. Az **objektum**ot egy **téglalap**pal jelöljük. Ebbe a téglalapba írjuk be aláhúzva az objektum nevét és osztályát kettősponttal elválasztva. Minden egyes – a tevékenységben résztvevő – objektumhoz tartozik egy ún. **életvonal**, amely az objektum időbeli létezését jelöli. Életvonalat az objektum-téglalap alsó élének közepéből kiinduló **függőleges vonal**lal jelölhetjük. Az egyes **üzenetek elküldését** az egyes objektumok életvonalai között húzott **nyilak**kal jelöljük, a nyíl a fogadó felé mutat. Többféle nyilat használhatunk az üzenetek típusának megfelelően. **Szinkron üzenet**ről beszélünk, ha a küldő az elküldött üzenet után nem kezdhet bele új tevékenységbe, hanem meg kell várnia a fogadó válaszát. Ilyen jellegű üzeneteket a küldő életvonalától a fogadó életvonaláig húzott „**telt fejű**” nyíllal ábrázoljuk. A **válasz**t pedig fordított irányban, **szaggatott nyíl**lal jelöljük. **Aszinkron** üzenet esetén a küldőnek nem kell várnia semmiféle válaszra. Jelölése hasonló a szinkron üzenetküldésnél megismerthez, csak „**üres**” a nyíl feje.

Az elkészített rendszer „tulajdonos” típusú felhasználójának szekvencia diagramja (részlet) megtekinthető a függeléken belül. (4. ábra – 70. oldal)

A konkrét rendszerrel kapcsolatos követelmények

Áttekintés dokumentum

Általános leírás

A WebÁruház egy nemzetközi cég által használt számítógépes rendszer, amely nyilvántartja a cég által forgalmazott termékeket, a regisztrált felhasználókat, a rendeléseket és a különböző külsős cégek ajánlatait. A rendszer háromféle felhasználót különböztet meg: vásárlót, tulajdonost és külsős cégeket. A vásárlói tevékenységek regisztrációhoz kötöttek. A tulajdonost a rendszer adminisztrátora regisztrálja a rendszer üzembehelyezésekor. A rendszer webes felületén a vásárlóknak lehetőségük van böngészni az egyes termékek között és akár meg is rendelhetik őket. Adminisztrálhatják saját felhasználói adataikat és nyomon követhetik addigi rendeléseiket. A tulajdonosnak lehetősége van a termékek adatait módosítani, új terméket felvenni, böngészni a céges ajánlatok között és felügyelni a felhasználók rendeléseit. A külsős cégeknek módjukban áll ajánlatot tenni a vezető részére.

Általános követelmények

- Barátságos megjelenítés, színek.
- Konzekvens kezelhetőség. Vagyis a funkcionalitások eléréséhez szükséges tevékenységek menete, és az aktiváláshoz szükséges felületi elemek helyzete oldalról-oldalra azonos legyen.
- A fejlesztés végeztével a forráskód és a teljes dokumentáció tulajdonjoga a megrendelőt illeti meg.
- A rendszer minden művelet eredményéről tájékoztatja a szoftver használóját. Hiba esetén értesítés ad a probléma okáról a felhasználó minőségétől függően.
- Mivel a cég több országból is vásárol árut, és mindig a legjobb áron szeretné beszerezni a termékeket, ezért biztosítani kell egy olyan funkcionalitást, mely

segítségével bármely cég, legyen a világ bármely táján, és rendelkezzen bármilyen platformmal képes legyen a már meglévő termékek közül keresni, és a keresések alapján összeállítani egy ajánlatot. Az ajánlatot el kell tudnia küldeni a tulajdonosnak, válaszként pedig kell kapnia egy ajánlat sorszámot. Az ajánlat sorszám segítségével képesnek kell lennie lekérdezni, hogy elfogadták-e, elutasították-e vagy még meg sem tekintették az ajánlatát. Ezeket a funkcionalitásokat a platformfüggetlenség miatt **webszolgáltatásként** kell megvalósítani. A tulajnak képesnek kell lennie a saját felhasználói felületén ezeket az ajánlatokat böngészni, és egy kattintással eldönteni, hogy elfogadja-e az ajánlatot vagy sem.

Rendszerkövetelmények

Operációs rendszer: Platform-függetlennek kell lennie.
Programozási nyelv: Java technológia.
Célhardver: A szoftver igényeihez fognak igazodni. A szoftver tervezésekor nem kell figyelembe venni.
Várható felhasználók száma: 10000 fő

Telepítés

A program futtatásához GlassFish V2 alkalmazásszerverre van szükség, amely tartalmazza az Apache Derby-t is.

1. A server indítása:

   GlassFish HOME (ez általában a Program Files\glassfish-v2ur1) \bin

   „asadmin start-domain domain1”

2. Apache Derby indítása:

   GlassFish HOME\javadb\bin\startNetworkServer.bat

3. Adatbázis létrehozása:

   GlassFish HOME\javadb\bin\ij.bat

   ij> connect 'jdbc:derby://localhost:1527/shop;create=true';

   ij> disconnect;

   ij> exit;

4. SUN Application Server Admin Console megnyitása:

   http://localhost:4848/login.jsf

   username : admin

   password : adminadmin

5. Adatbázis regisztrálása a JNDI-ben:

   Resources / JDBC / JDBC Resources / NEW

   JNDI Name: jdbc/shop

   POOL NAME : DerbyPool

6. Program feltöltése:

   Applications / Enterprise Applications / DEPLOY

   Type: .EAR

   Packaged file to be uploaded to the server / BROWSE /

   webShop.ear megkeresése

7. Böngészés:

   http://localhost:8080/webShop-war

Konkrét megvalósítás – Implementáció

A követelmények meghatározása és a tervezés után a konkrét megvalósításé a főszerep. A következőkben bemutatom az elkészült rendszer bizonyos részeit, és ahol szükséges ezekről részletesen is írok.
Az adatbázis meglehetősen egyszerű. 4 táblát kell megemlíteni, ezek a következők:

- termek
- vasarlo
- ajanlat
- rendelesek

Mindegyik a nevének megfelelő adatokat tárolja. Maguk az entitások teljesen hasonlóak ahhoz a példához, amelyet az entitások bemutatásánál említettem. Így például a rendelések entitás a következőképpen néz ki:

rendelesek.java (részlet)

```
…
@Entity
public class rendelesek implements Serializable {

    private String vasarloFelhNev;
    private String termekek;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date datum;
    private int osszeErtek;

    public void setId(Long id) {
        this.id = id;
    }
…
```

Egyetlen érdekesség talán a *@Temporal(javax.persistence.TemporalType.DATE)*, mely beállítással csak a dátum tárolódik el a rendelések végrehajtásakor a táblában.

Funkcionalitások – Megjelenítés

Az áruház első oldala az üdvözlő képernyő, melyről a regisztrációs és bejelentkező oldalra tudunk elnavigálni. Ez egy egyszerű JSP oldal, melyben 2 form található. Az egyik a regisztrációhoz, másik a bejelentkezéshez tartozik. A bejelentkezéshez tartozó form a következő:

regisztracio.jsp

```
<form action="regisztracio" method=post>
      <table>

        <tr><td>
        Felhasználónév:</td><td>          <input name="felhNevReg"/></td>
        <tr><td>
        Jelszó:</td> <td>        <input type="password" name="jelszo"/></td>
        <tr><td>
        Jelszó ismét:</td><td>         <input type="password" name="jelszo2"/></td>
        </tr>
         </table>

          <button>Regisztáció</button>
         </form>
```

Ez nem csinál mást, mint elküldi a felhasználónevet és jelszót a *regisztracio* nevezetű servletnek paraméterként. Fontos, hogy **post** metódust alkalmazunk, ugyanis get metódus esetén az URL egészülne ki a megadott paraméterekkel, ami ahhoz vezetne, hogy a jelszó – melyet a beíráskor automatikusan kicsillagoz a böngésző, hála a *type=”password”* beállításnak – elolvasható lenne. A JSP oldal tartalmaz ezen kívül még hibajelzésre szolgáló elemeket is. Ugyanis ha a felhasználó valamit elgépelt, vagy más okból kifolyólag rossz adatokat adott meg, azt jeleznünk kell. Ezt a következőképpen oldottam meg:

regisztracio.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
…
<ul>
      <c:forEach items="${hibaListaReg}" var="hibaListaRegitem">
         <li><font style="color: red">${hibaListaRegitem}</font></li>
      </c:forEach>
      </ul>
```

A **forEach** ciklus a *hibaListaReg* nevezetű gyűjteményen iterál végig, mely a session-ben található meg, melybe értékeket a *regisztracio.java* servlet tehet hiba esetén.

regisztracio.java

```
package web;

// IMPORTOK

public class regisztracio extends HttpServlet {
   @EJB
   private vasarloFacadeLocal vasarloFacade;

   protected void processRequest(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
```

```
String felhNev = request.getParameter("felhNevReg");
String jelszo = request.getParameter("jelszo");
String jelszo2 = request.getParameter("jelszo2");

List<String> hibaListaReg = new ArrayList<String>();

HttpSession session = request.getSession();

if (!Pattern.matches("[\\w]{5,15}",felhNev))
   hibaListaReg.add("Hibás felhasználónév formátum! A felhasználónévnek 5-15 karakternek kell lennie");

if (!Pattern.matches("[\\w]{5,15}",jelszo))
   hibaListaReg.add("Hibás jelszó formátum! A jelszónak 5-15 karakternek kell lennie");

if (!jelszo.equals(jelszo2))
   hibaListaReg.add("A két jelszó nem egyezik");

vasarlo v = vasarloFacade.findByNick(felhNev);
if (v != null) hibaListaReg.add("Már van ilyen felhasználó! Kérem válasszon másik felhasználó nevet!");


if (hibaListaReg.size()!=0)
{
   request.setAttribute("hibaListaReg",hibaListaReg);
   request.getRequestDispatcher("regisztracio.jsp").forward(request,response);
   return;
}
request.getRequestDispatcher("index.jsp").forward(request,response);



} // KÖTELEZŐ SERVLET METÓDUSOK }
```

A regisztráció servlet lekérdezi az egyes paramétereket, majd megnézi, hogy ezek helyesen vannak e megadva. A felhasználónévnek 5-15 karakternek kell lennie, és a két jelszónak meg kell egyeznie, valamint a jelszóra is vonatkozik az előbbi, hosszra való megkötés. Ezenkívül meg kell néznünk, hogy foglalt-e az a név, amelyet a felhasználó épp regisztrálni szeretne. Ha az adatok megfelelnek az elvárásoknak, a hibalistánk üres lesz, akkor sikeres a regisztráció és a böngészőnk automatikusan az index oldalra navigál. Különben visszajutunk a regisztrációs lapra, és megkapjuk azokat a hibaüzeneteket, melyeket a servlet a *hibaListaReg* listába pakolt. Fontos, hogy a regisztrációs oldalra történő navigálás után, a feltétel vizsgálatban legyen egy **return**, ugyanis ha nincs, akkor végrehajtódnak a servlet esetlegesen később következő metódusai is. Egy lehetséges hibaüzenet kombináció a következőképpen néz ki:

A rendszer kétféle státuszú felhasználót ismer. Egyszerű vásárlót és tulajdonost. A külsős cégek esetében nincs biztonsági intézkedés. Az egyszerű vásárlók mind tudnak regisztrálni, a tulajdonost viszont az adminisztrátor hozza létre. E kétféle státuszú felhasználó megkülönböztetése az entitásaik *getTipus()* metódusával lehetséges, mely tulajdonos esetén „*tulaj*”, vásárló esetén „*vasarlo*” értéket szolgáltat. A vásárlók regisztrációjánál automatikusan beállítódik a státusz. A getTipus() metódus visszatérési értékét használja fel a 2 **szűrő** is, mely azon oldalak védelmét hivatott ellátni, melyeket csak regisztrált vásárlóknak, vagy csak a tulajnak kívánunk rendelkezésre bocsátani. Tulaj esetén a szűrő lényegi része így néz ki:

tulajFilter.java

```
public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain)
            throws IOException, ServletException {
.
.
.
HttpSession session = ((HttpServletRequest)request).getSession();
      vasarlo v = (vasarlo) session.getAttribute("vasarlo");

      try {
         if (v!=null && v.getTipus().equals("tulajdonos") ){

            chain.doFilter(request, response);}
         else
            ((HttpServletResponse)response).setStatus(HttpServletResponse.SC_FORBIDDEN);

      }
```

Sikeres bejelentkezéskor automatikusan beállítódik a sessionben egy *vasarlo* nevű rész, melyben az aktuálisan bejelentkezett felhasználó adatait tároljuk. A *web.xml* fájlunkban a következő bejegyzésekkel érjük el, hogy az összes olyan oldal védve legyen a *tulajFilter* által, mely a tulaj mappában van:

web.xml (részlet)

```
<filter>
        <filter-name>tulajFilter</filter-name>
        <filter-class>filters.tulajFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>tulajFilter</filter-name>
        <url-pattern>/tulaj/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
        <dispatcher>ERROR</dispatcher>
    </filter-mapping>
```

A tulaj bejelentkezés után több funkcionalitáshoz is hozzáfér. Ezek közül az egyik a termékek módosítása. Ezen oldal kialakításakor az egyszerűség, a könnyen kezelhetőség és a „tulajdonos keze alá való dolgozás” voltak a kritériumok. Alapvető probléma, hogy ha sok termékünk van, az ne egyszerre jelenjen meg, hanem bizonyos egységekben, például ötösével. Ha pedig csoportonként kezeljük a termékeket, akkor mindenféle manipuláció – pl. törlés vagy módosítás – után elvárás, hogy oda jussunk vissza ahonnan kiindultunk. Ne legyen az, hogy ha például a 31-35 elemek közül törlök egyet, a böngésző a törlést követően az 1-5 elemekre navigál. Elvárás még az is, hogy a módosítandó termék adatai azonos oldalon jelenjenek meg, mint ahol a termékeket böngészem, elkerülve ezzel a több termék módosítása során fellépő ugrálásokat a weblapok között. Illetve a módosítás hatása azonnal jelenjen meg az egyes csoportoknál. Így az oldal kialakítása, és a háttérben zajló tevékenységek valamelyest összetettek. Az előbb említett oldalról pillanatképet a függelék 6. ábráján láthatunk (72. oldal).

Az említetteken kívül több más dolgot is észrevehetünk a screenhoton. Ilyen például, hogy a tulajdonosnak módjában áll keresni a termékek között. Illetve adott kritérium alapján (pl. ár, név, cikkszám stb.) rendezheti is a termékeket. Természetesen a fentebb említett probléma, miszerint más csoporthoz navigál a böngésző a manipuláció után a keresés és rendezés után sem jöhet elő. Nézzük, hogy az egyes feladatok hogyan lettek megvalósítva:

tModositas.jsp (részlet)

```
<%
int tSz = (Integer)session.getAttribute("termekekSzama");
out.write("Termékek száma: " + tSz);
%>
```

A termékek számát egy egyszerű **scriplet** segítségével kérdezzük le a sessionből, és íratjuk ki.

tModositas.jsp (részlet)

```
<form action="termekSzuroTulaj" method=post>
   <table border="3" id="termektabla">

      <thead>
         <th><a href="keres?sorrend=ccs">-</a> Cikkszám <a href="keres?sorrend=cn">+</a></th>
         <th><a href="keres?sorrend=ncs">-</a> Név <a href="keres?sorrend=nn">+</a></th>
```

A keresést a *termekSzuroTulaj* nevű servlet végzi, melynek paraméterként átadásra kerül egy **string**, annak megfelelően, hogy mi alapján és milyen sorrendben kívánjuk rendezni a termékeket. A fenti részletben a '*ccs*' a cikkszám szerinti csökkenő sorrendet jelenti, melyet a *cikkszám* oszlopnév bal oldalán elhelyezett '–' jelre kattintva érhetünk el. Az '*nn*' a név szerinti növekvő sorrendet jelenti stb.

tModositas.jsp (részlet)

```
<c:forEach items="${resz}" var="resz">
         <tr>
            <td><c:out value="${resz.cikkszam}" default=" "/></td>
            <td><c:out value="${resz.nev}" default=" "/></td>
```

A termékeket mindig a *resz* névvel ellátott session-beli kollekcióból íratjuk ki, melyben maximum öt elem található.

tModositas.jsp (részlet)

```
<%
                  termek t2 = (termek) pageContext.getAttribute("resz");
  out.write("<a href=\"tAdatlap?cikkszam="+t2.getCikkszam()+"\">Módosít</a>");
               %>
```

Módosítás és törlésre szolgáló link megjelenítése szintén scriplet segítségével történik. Mely során a lap kontextusából megkeressük a *resz* nevezetűt, és mindig az aktuális cikkszámmal bővítjük a *tAdatlap* illetve *tTorles* servletek *cikkszam* paraméterének értékét.

tModositas.jsp (részlet)

```
<%

      for (int i=1;i<=tSz;++i){
         if (i%5 == 1) out.write("<a href=\"keres?sorszam="+i+"\">"+i+"-"+(i+4)+"</a>");out.write(" ");
      }
               %>
```

Az egyes csoportok elérhetőségét biztosító linkeket is scriplet segítségével írjuk ki. Ez a scriplet felhasználja a korábban definiált *tSz* (termékek száma) attribútumot:

tModositas.jsp (részlet)

```
<jsp:include page="termekAdatlap.jsp" />
```

Egyszerű *include* segítségével van megoldva, hogy a termék adatlapja egy oldalon jelenjen meg a böngészés lappal.

Ezek voltak a megjelenítésért felelős elemek. A háttérben a következő egységek dolgoznak:

Mind közül a legegyszerűbb az a servlet, mely azért felelős, hogy a kiválasztott – módosítás gomb segítségével - termék megjelenjen a lap alján.

tAdatlap.java (részlet)

```
Integer cikkszam = Integer.parseInt(request.getParameter("cikkszam"));
      termek t = termekFacade.findByCikkSzam(cikkszam);
      session.setAttribute("termek",t);
      request.getRequestDispatcher("tulaj/tModositas.jsp").forward(request,response);
```

Cikkszam nevű változóba lekérjük a kérés paraméterét, megkeressük az adott cikkszámú terméket, majd beállítjuk ezt a terméket a session *termek* nevű attribútumának. A kérést továbbítjuk a módosítási lapnak. Így az include által beillesztett *termekAdatlap.jsp* már fog találni termek nevezetű attribútumot a sessionben és megtudja jeleníteni a kívánt terméket.

termekAdatlap.jsp (részlet)

```
jsp:useBean id="termek" scope="session" class="webShop.termek"></jsp:useBean>

<form action="termekModosit" method=post>
<table border="3" id="modosit">
<tr><td>Név:</td>
<td><input type="text" name="nev" value="<jsp:getProperty name="termek" property="nev"/>"/></td>
<td ROWSPAN=4><IMG SRC="kepek/<jsp:getProperty name="termek" property="kep"/>.jpg" WIDTH="70"
HEIGHT="105" UNITS="pixel"></td></tr>
<tr><td>Típus:</td>
<td><input type="text" name="tipus" value="<jsp:getProperty name="termek" property="tipus"/>"/></td></tr>
<tr><td>Ár: </td>
<td><input type="text" name="mennyiseg" value="<jsp:getProperty name="termek"
property="mennyiseg"/>"/></td></tr>
<tr><td>Mennyiség:</td>
<td><input type="text" name="ar" value="<jsp:getProperty name="termek" property="ar"/>"/></td></tr>
<tr><td>Kép:</td>
<td><input type="text" name="kep" value="<jsp:getProperty name="termek" property="kep"/>"/></td>
<td><button>Módosít</button></td>
</tr>
</table>
</form>
```

A szűrő gombra kattintva a *termekSzuroTulaj* servlet, valamelyik termékcsoport linkjére kattintva a *keres* servlet, a törlés gombra kattintva pedig a *tTorles* servlet hívódik meg, a megfelelő paraméterekkel.

Mindhárom servlet esetében találkozunk bizonyos session-beli elemekkel. Ezek a *sorszam* és a *vanEredmeny*. A sorszam segítségével tudjuk tárolni, hogy a servletek végrehajtása előtt, a termékek egyes csoportjai közül melyiken álltunk éppen. A

vanEredmeny pedig azt tárolja, hogy a servletek meghívása előtt történt-e keresés illetve rendezés vagy sem.

A tTorles servlet először elirányít a törlés megerősítése lapra. Ha valóban törölni kívánjuk a terméket – a törlés megerősítést nyert – akkor pedig végleg törli az adatbázisból, és ezenfelül a session keresési eredményeket tároló részéből is törli, ha az tartalmazza az adott terméket, majd átirányítja a kérést a termekSzuroTulaj servletnek.

A termekSzuroTulaj servlet háromféleképpen hívódhat meg:

- Az előbb említett módon, amikor a törlésből jövünk. Ilyenkor nincs semmilyen paraméter, de a session-ben lehet a szűrőfeltételeknek megfelelő termékcsoport.
- Semmilyen szűrési feltétel nincs – üres szűrési feltételek mellett a szűrő gombra kattintva érhetjük el – az összes termékre kíváncsiak vagyunk. Ilyenkor az session-ben esetlegesen meglévő eredmények törlődnek,és törlődik vele együtt a sorszam attribútum is.
- Van szűrési feltétel, és nem a törlésből jövünk. Ilyenkor konkrét paramétereket (pl. név, típus stb.) kap a servlet, ezek alapján keres az adatbázisban. A megtalált termékeket a session vanEredmeny attribútumában tárolja, és a sorszám attribútumot itt is null-ra állítja.

Mindhárom esetben legvégül továbbítja a kérést a keres servletnek.

termekSzuroTulaj.java (részlet)

```
List<termek> listaNevAlapjan = termekFacade.findByNev(nev.toLowerCase());
…
session.setAttribute("vanEredmeny", listaNevAlapjan);
      session.setAttribute("sorszam", null);
      request.getRequestDispatcher("keres").forward(request,response);
```

A sorszám attribútum null-ra állítására azért van szükség, mert mindegy, hogy egy teljesen új keresésről, vagy az összes termék újbóli megjelenítéséről van szó – egy korábbi keresés eredményének megjelenítése után -, mindkét esetben az első 5 terméket kell visszaadnunk elsőként.

A keres servlet-ben többféle vizsgálat is zajlik, annak kiderítése érdekében, hogy honnan érkezett a kérés, és milyen paraméterekkel. Először a sorrendre vonatkozó feltételek kerülnek vizsgálatra:

keres.java (részlet)

```
if (request.getParameter("sorrend") != null) sorrend = true;
…
if (sorrend) session.setAttribute("sorszam", null);
if (request.getParameter("sorszam") != null){
          sorszamByRequest = Integer.parseInt(request.getParameter("sorszam"));
          session.setAttribute("sorszam",sorszamByRequest );}
```

Ha valamilyen sorrendre vonatkozó paramétert kap a keres servlet, akkor a sorszám null értékre állítása fontos. Ugyanis ha új sorrendet kívánunk felállítani mindig a sorrendben elől álló 5 terméket kell visszaadnunk. Ha viszont a kérés tartalmaz a sorszámra vonatkozó paramétert, akkor azzal felül kell írnunk a session-ben lévőt, hiszen egy adott sorrend, másik termékcsoportjára vagyunk kíváncsiak. Ezután következik a termékek listájának feltöltése:

keres.java (részlet)

```
if(session.getAttribute("vanEredmeny") != null ){
            osszesTermek = (List<termek>) session.getAttribute("vanEredmeny");
         //   if (request.getParameter("sorszam") == null)   session.setAttribute("sorszam", null);

        }
        else
         osszesTermek = termekFacade.findAll();
```

Ha a session tartalmaz egy null-tól különböző értékű vanEredmeny nevű attribútumot, akkor biztosak lehetünk benne, hogy a felhasználó valamilyen keresést vagy sorrendbe állítást eszközölt a termékeken, és azt nem is változtatta még meg, tehát ezekkel kell dolgoznunk. Itt lehet látni, hogy miért kellett null-ra állítani a termekSzuroTulaj servlet adott részeiben a sorszam illetve a vanEredmeny attribútumokat. Hiszen amikor az összes termékre kíváncsiak vagyunk, akkor újból le kell kérdeznünk az adatbázist, amit a fent látható kód **else** ágában teszünk meg. Ha ezekkel készen vagyunk, a termékek sorrendbe helyezése következik, persze csak akkor, ha a felhasználó ezt kéri:

keres.java (részlet)

```
sorrend sor = new sorrend();
boolean sorrend = false;
      if (request.getParameter("sorrend") != null) sorrend = true;
      String milyenSorrend = request.getParameter("sorrend");
…
if (sorrend){
            if(milyenSorrend.equals("an")) Collections.sort(osszesTermek,sor.arSorrend);
            if(milyenSorrend.equals("acs")) Collections.sort(osszesTermek,sor.negativArSorrend);

            if(milyenSorrend.equals("cn")) Collections.sort(osszesTermek,sor.cikkszamSorrend);
```

sorrend.java (részlet)

```
public static final Comparator arSorrend = new Comparator(){

      public int compare(Object o1, Object o2) {
         termek t1 = (termek) o1;
         termek t2 = (termek) o2;
         return t1.getAr()-t2.getAr();
      }
   };
```

A megadott sorrend paramétertől függően hívódnak meg az egyes - sorrend.java osztályban lévő - metódusok. Végezetül a megfelelő öt termék kiválasztása, és visszaadása a feladat, az egyes session-beli attribútumok beállítása mellett:

keres.java (részlet)

```
otDbTermek = new LinkedList<termek>();

        if (session.getAttribute("sorszam") == null)
           for (int i=0;i<5 && i<osszesTermek.size();++i)
              otDbTermek.add(osszesTermek.get(i));

        else{

           int sorszam  = (Integer)session.getAttribute("sorszam");

           for (int i = sorszam-1;i<sorszam+4&& i<osszesTermek.size();++i)
              otDbTermek.add(osszesTermek.get(i));

        }
if (sorrend) session.setAttribute("vanEredmeny", osszesTermek);
        session.setAttribute("osszes",osszesTermek);
        session.setAttribute("resz",otDbTermek);
        session.setAttribute("termekekSzama", osszesTermek.size());
        request.getRequestDispatcher("tulaj/tModositas.jsp").forward(request,response);
```

A programban használt többi servlet, jsp oldal és egyéb elemek hasonló módon vannak megvalósítva.

Publikálás webszolgáltatásként

A követelmény feltárás során kiderült, hogy szükségünk van arra, hogy egyes cégek – legyenek bárhol a világon, és rendelkezzenek bármilyen rendszerrel – keresni tudjanak a termékek adatbázisában, és a keresési eredmények alapján össze tudják állítani saját ajánlatukat a cégtulajdonos felé. Mint ahogy a bevezetőben is olvasni lehet, a webszolgáltatások megvalósítása nem mindig jelent új funkcionalitást, mindössze nagyobb célközönség felé történik a publikálás, szabványosított eszközökkel. Ez a mi példánkban is szembetűnő. Ugyanis minek írjuk meg a termékek között kereső eljárásokat, üzleti logikát, ha nekünk ez már készen van. Egyedül a cégek ajánlatainak fogadása, majd pedig az azokra adott válasz megjelenítése jelent új funkcionalitást.
Mivel EJB rétegben fejlesztjük a webszolgáltatást, szükségünk van egy állapotmentes session bean-re, melyben a publikálandó metódusokat helyezzük el:

webszolg.java (részlet)

```
@WebService()
@Stateless()
public class webSzolg {
    @EJB
    private ajanlatFacadeLocal ajanlatFacade;
    @EJB
    private termekFacadeLocal ejbRef;

    @WebMethod(operationName = "findByNev")
    public List<termek> findByNev(String nev) {
        return ejbRef.findByNev(nev.toLowerCase());
    }

    @WebMethod(operationName = "findByNameToredek")
    public List<termek> findByNameToredek(@WebParam(name = "string")
```

```
    String string) {
        String s = "%"+string+"%";
        return ejbRef.findByNev(s.toLowerCase());
    }

    @WebMethod(operationName = "ujAjanlat")
    public Integer ujAjanlat(@WebParam(name = "termekNev")
    String termekNev, @WebParam(name = "mennyiseg")
    int mennyiseg, @WebParam(name = "bruttoOsszAr")
    int bruttoOsszAr, @WebParam(name = "cegNev")
    String cegNev, @WebParam(name = "cim")
    String cim, @WebParam(name = "telefonszam")
    String telefonszam, @WebParam(name = "email")
    String email) {

        ajanlat a = new ajanlat();
        a.setBruttoOsszAr(bruttoOsszAr);
        a.setCegNev(cegNev);
        a.setCim(cim);
        a.setEmail(email);
        a.setMennyiseg(mennyiseg);
        a.setTelefonszam(telefonszam);
        a.setTermekNev(termekNev);
        a.setEgysegAr(bruttoOsszAr/mennyiseg);
        a.setAllapot("Nincs meg megtekintve!");
        ajanlatFacade.create(a);

        return a.getId();
    }
    @WebMethod(operationName = "ajanlatLekerdezes")
    public String ajanlatLekerdezes(@WebParam(name = "ajanlatSzama")
    int ajanlatSzama) {
        ajanlat a = ajanlatFacade.find(ajanlatSzama);

       if (a!=null) return a.getAllapot();
       else return "Nincs ilyen szamu rendeles!";
    }


}
```

Látható, hogy a session bean-t @Webservice annotációval láttuk el, jelezve, hogy webszolgáltatásként kívánjuk publikálni. A @WebMethod és @WebParam annotációk is megtalálhatók az egyes metódusoknál. A @EJB annotációval pedig megvalósítjuk a függőség injektálást, mely segítségével könnyedén tudjuk kezelni a termékek és ajánlatok megfelelő adatbázisbeli részét. A webszolgáltatás egyszerűen csak használja a függőség injektálást és így manipulálja a bean-t. Mint egy átlagos servlet-es megoldásnál.

Sémaleíró (részlet)

```
<xs:element name="findByNev" type="tns:findByNev"/>
<xs:element name="findByNevResponse" type="tns:findByNevResponse"/>
<xs:element name="ujAjanlat" type="tns:ujAjanlat"/>
<xs:element name="ujAjanlatResponse" type="tns:ujAjanlatResponse"/>
–
        <xs:complexType name="findByNev">
–
        <xs:sequence>
<xs:element name="arg0" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
–
        <xs:complexType name="findByNevResponse">
```

```
–
        <xs:sequence>
<xs:element name="return" type="tns:termek" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
–
        <xs:complexType name="termek">
–
        <xs:sequence>
<xs:element name="ar" type="xs:int"/>
<xs:element name="cikkszam" type="xs:int" minOccurs="0"/>
<xs:element name="db" type="xs:int"/>
<xs:element name="id" type="xs:int" minOccurs="0"/>
<xs:element name="kep" type="xs:string" minOccurs="0"/>
<xs:element name="mennyiseg" type="xs:int"/>
<xs:element name="nev" type="xs:string" minOccurs="0"/>
<xs:element name="tipus" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
```

SOAP kérés (példa)

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ns2:ujAjanlat xmlns:ns2="http://webSzolg/">
         <termekNev>termeknev1</termekNev>
         <mennyiseg>1000</mennyiseg>
         <bruttoOsszAr>250</bruttoOsszAr>
         <cegNev>cegnev1</cegNev>
         <cim>cim1</cim>
         <telefonszam>telefonszam1</telefonszam>
         <email>email1</email>
      </ns2:ujAjanlat>
   </S:Body>
</S:Envelope>
```

SOAP válasz (példa)

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:ujAjanlatResponse xmlns:ns2="http://webSzolg/">
         <return>1853</return>
      </ns2:ujAjanlatResponse>
   </S:Body>
</S:Envelope>
```

Befejezés

A nagyvállalati szoftverkészítés hihetetlenül mélyreható és komplex feladat. Dolgozatom, és a program írása közben döbbentem rá, hogy mennyire kifinomult technológiákkal kell dolgoznia egy fejlesztőnek, ha egy komolyabb projekt végrehajtásán fáradozik. Pedig én még csak nagyon messziről súroltam a technológia határait. Hihetetlen szaktudás és tapasztalat összpontosul az egyes fejlesztőeszközökben, amelyeket a munkánk során használunk. Fontos, hogy nem elég a jól begyakorolt módszerekkel alkotni, állandóan naprakésznek kell lenni. Szinte naponta jönnek ki az újabbnál újabb megoldások, amelyek elsajátítása elemi érdeke egy fejlesztőnek. De itt is igaz a mondás, hogy „jó pap is holtig tanul, mégis bután hal meg”!

Elvonatkoztatva a dolgozatomban megvalósított projekttől, és globálisan vizsgálva az internetes szolgáltatásokban rejlő lehetőségeket, le kell vonnunk a következtetést, miszerint még vannak kiaknázatlan területek. Akár egy jó ötlettel is rengeteg mindent elérhet az ember, ha „jókor van, jó helyen”. Remek példákat lehetne erre felhozni a közelmúltból is. Egyes honlapok irreálisan magas áron cserélnek gazdát. Szinte mindenki az internetre próbál valamit fejleszteni, kitalálni. Annak ellenére, hogy egyes cégek rengeteg pénzt fektetnek be bizonyos oldalakba, nem tudnék mondani olyan céget, vagy olyan szolgáltatást, amely hosszútávon képes lenne fennmaradni komolyabb fejlesztések és ötletek nélkül. Jó példa erre az iwiw.hu, melyet a magyar web egyik – ha nem az egyetlen - sikertörténeteként tartanak számon. Mégis, mióta felvásárolták gyakorlatilag semmi nem történt. A felhasználók megszokásból használják, de biztos vagyok benne, ha lenne más alternatíva, mely alternatíva többet nyújt, rögtön átpártolnának. Érdemes tehát keresni a lehetőségeket, gondolkodni az ötleteken, mert akinek sikerül valami nagyon újat és jót alkotni, az történelmet írhat…

Források, hivatkozások, felhasznált irodalom

Internetes források

[1] http://java.sun.com/products/servlet/

[2] http://java.sun.com/products/jsp/

[3] http://www.omg.org/library/iiop4.html

[4] http://www.devx.com/Java/Article/27235

[5] http://www.javaforum.hu/javaforum/0/news/26/show/az_annotacio_sikeres

[6] http://java.sun.com/products/jndi/

[7] http://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html

[8] http://java.sun.com/javase/technologies/database/index.jsp

[9] http://www.w3.org/

[10] http://www.isgmlug.org/sgmlhelp/g-index.htm

[11] http://java.sun.com/javaee/5/docs/tutorial/doc/bnazg.html

[12] http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc2.html

[13] http://www.prog.hu/cikkek/884/Bemutatkozik+az+XML.html

Könyvek

[14] Imre Gábor : Szoftverfejlesztés Java EE Platformon,

[15] Gottdank Tibor : Webszolgáltatások – XML alapú kommunikáció az Interneten

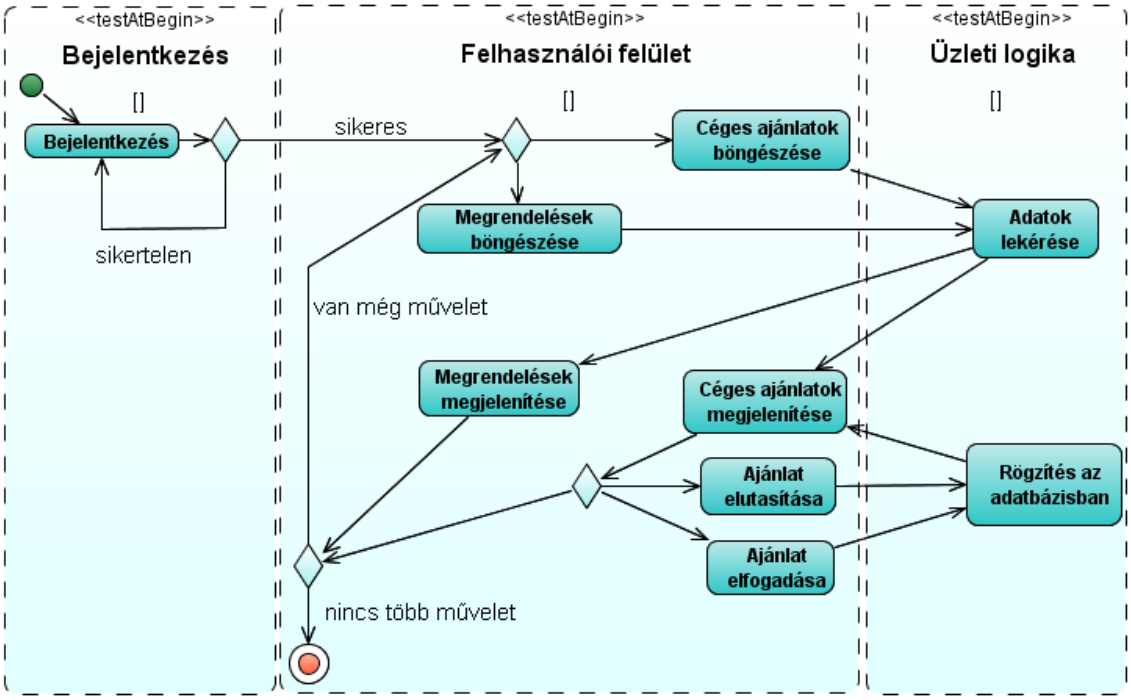
[16] Nyékyné Gaizler Judit : J2EE útikalauz Java programozóknak

[17] Steve Graham… : Java alapú webszolgáltatások

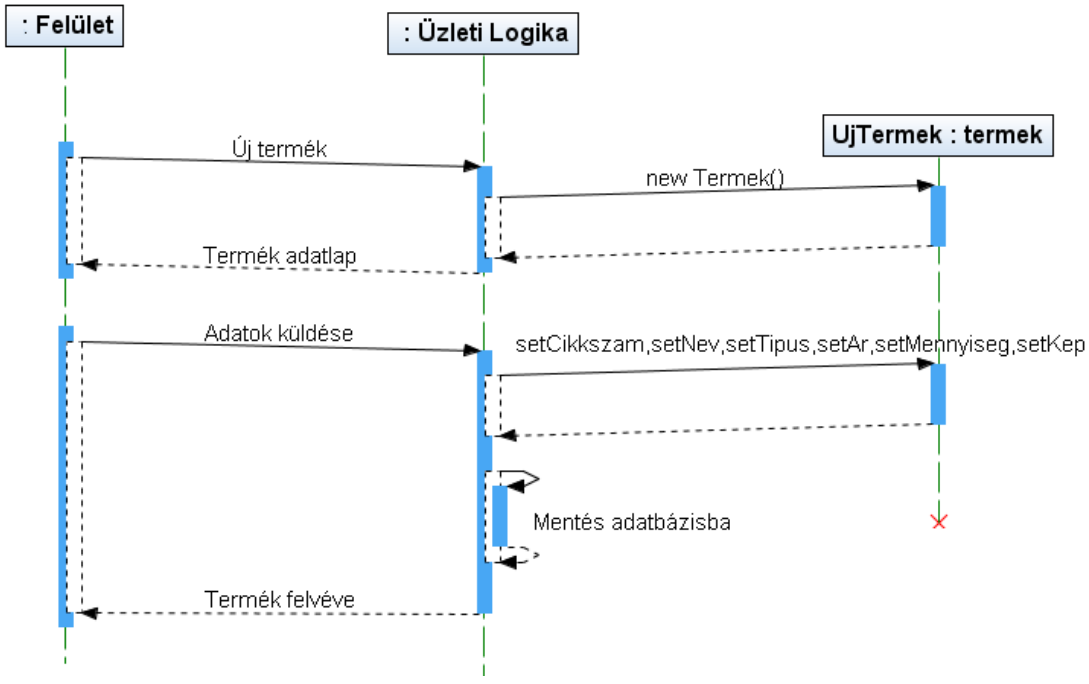
[18] Harald Störrle : UML 2

[19] Raffai Mária : Egységesített megoldások a fejlesztésben

Függelék



3. Ábra – Tulajdonos típusú felhasználó aktivitás diagramja (részlet)



4. Ábra – Tulajdonos típusú felhasználó szekvencia diagramja – Termék felvétel (részlet)

5. Ábra – Vásárló típusú felhasználóra vonatkozó Use Case diagram (részlet)

- tulaj kijelentkezés

Főoldal
Termékek felvétele
Megrendelések adminisztrálása
Céges ajánlatok

Termékek

Termékek száma: 96

- Cikkszám +	- Név +	- Típus +	- Ár +	- Mennyiség +	Kép	Módosítás	Törlés
1	Nev_B-5	Tipus_4	56457	421	kep1	Módosít	Törlés
2	Nev_H-a	Tipus_0	55373	1587	kep2	Módosít	Törlés
3	Nev_H-i	Tipus_1	20774	2392	kep3	Módosít	Törlés
4	Nev_I-a	Tipus_3	26541	1450	kep4	Módosít	Törlés
5	Nev_B-g	Tipus_2	50337	1767	kep5	Módosít	Törlés
						Szűrő	

☐ Csak teljes szót keres

1-5 6-10 11-15 16-20 21-25 26-30 31-35 36-40 41-45 46-50 51-55 56-60 61-65 66-70 71-75 76-80 81-85 86-90 91-95 96-100

A módosítandó termék adatai:

Név:	Nev_B-5	
Típus:	Tipus_4	
Ár:	421	
Mennyiség:	56457	
Kép:	kep1	Módosít

6. Ábra – Tulaj felhasználói felület – Termékmódosítás