DIPLOMAMUNKA

Ráthonyi Tamás

Debrecen

2010

Debreceni Egyetem Informatikai Kar

Java Composite Application Platform Suite

Témavezető:

Dr. Juhász István

egyetemi adjunktus

Készítette:

Ráthonyi Tamás

Programtervező Informatikus M.Sc.

Debrecen

2010

Köszönetnyilvánítás

Szeretnék köszönetet mondani:

- Édesanyámnak, Édesapámnak és családomnak, mert mindig mögöttem állnak, átsegítenek a nehéz időkön és az iránymutatások nélkül nem értem volna el azokat a sikereket, melyek mindig pozitív irányba terelték az életem.
- dr. Juhász István egyetemi adjunktusnak, mert megszeretette velem az információ technológiát, egyetemi éveim alatt végig támogatott és adott egy olyan szemléletmódot, melyet eddig sikerrel alkalmazhattam szakmai karrierem során.
- Balogh Fruzsinának türelméért, segítségéért és hasznos tanácsaiért, melyek segítettek diplomamunkám elkészülésében.
- a Sun Microsystemsnek és a náluk töltött éveim alatt megismert barátaimnak, mert általuk egy addig új, ismeretlen világnak lehettem a részese és az IT iránt tanúsított szenvedélyük megváltoztatta világszemléletemet.
- és végül, de nem utolsó sorban minden barátomnak és ismerősömnek, akik hozzájárultak egyetemi tanulmányaim sikeres befejezéséhez.

Tartalomjegyzék

Előszó .... - 2 -

1. SOA áttekintés .... - 4 -
2. Technológiák .... - 7 -
    1. Open Directory Server .... - 7 -
    2. Open Single Sign-On .... - 9 -
    3. Glassfish .... - 11 -
    4. Open Message Queue .... - 17 -
    5. Open ESB .... - 20 -
    6. Technológiák összegzése .... - 32 -
3. Üzleti folyamat tervezése és implementálása .... - 33 -
    1. A szolgáltatás implementálásának technológia környezete .... - 33 -
    2. Üzleti folyamat bemutatása az ügyfél szemszögéből .... - 35 -
    3. Üzleti folyamat elemzése .... - 36 -
    4. Blue Credit Bank partner szolgáltatásainak implementálása .... - 40 -
    5. A Candy Inventory partner szolgáltatásainak implementálása .... - 46 -
    6. Üzleti folyamat implementálása .... - 46 -
    7. Üzleti folyamatok telepítése és tesztelése .... - 56 -
4. Összegzés .... - 60 -

Irodalomjegyzék .... - 61 -

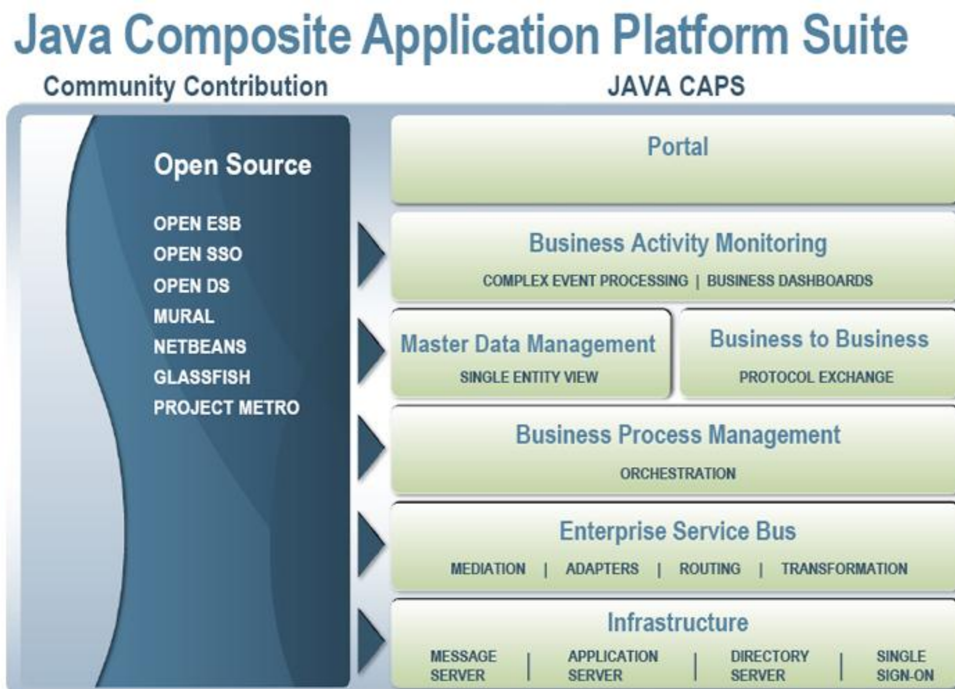
Függelék .... - 62 -

Előszó

Napjainkban egyre inkább elindult a szolgáltatás orientált megközelítések térnyerése az információ technológia területén is. A cél annak elérése, hogy az üzleti világ fogalmait és folyamatait minél gyorsabban, hatékonyabban és rugalmasabban tudjuk leképezni az informatika eszközeire. A eddigi szoftverarchitektúrák nem voltak képesek gyorsan alkalmazkodni az üzleti igények dinamikus változásaihoz, a különböző platformokon implementált megoldások sem működtek együtt mindig zökkenőmentesen. A SOA célkitűzései közé tartozik ezen problémák áthidalása egy szabványosított módon.

Előző Ontológiák és Szolgáltatás Orientált Architektúrák című diplomamunkámban bemutattam az ontológia fogalmát, a SOA-t, mint architektúrális mintát, valamint hogyan lehet ontológiákat alkalmazni SOA architektúrák tervezése során ezzel megkönnyítve a későbbi fejlesztést és minimálisra csökkentve a váratlan kockázatok felbukkanását.

Jelen diplomamunkám célja folytatni az előző dolgozat SOA vonalát és tanulmányozni a ***Sun Microsystems Java Composite Application Platform Suite*** (továbbiakban ***Java CAPS***) nevű terméke alapjául szolgáló technológiákat. Maga a termék egy teljes mértékben szabványokra épülő, nyílt alkalmazásfejlesztési és integrációs platform, mely száz százalékban a szolgáltatás orientált megközelítésre épül. Alapjául a Sun több nyíltforrású projektje szolgál, több-kevesebb kiegészítéssel.



1. ábra Java CAPS felépítése

A dolgozat első felében az *1. ábrán* látható nyílt forrású technológiákat mutatom be, a második felében pedig egy üzleti folyamat tervezését és részleges implementálását mutatom be. A technológiák a szakdolgozat terjedelmi korlátai miatt a teljesség igénye nélkül kerülnek bemutatásra.

1. SOA áttekintés

Első, kvázi nulladik fejezetként szeretnék egy rövid SOA áttekintést adni, hogy az előző diplomamunkában bemutatott fogalmakat alapul vehessem és finomíthassam.

Napjainkban a fejlesztőknek a felhasználótól az erőforrásokig mindent lefedő úgynevezett end-to-end alkalmazásokat kell készíteniük, mely magában foglalja:

- *felhasználói felület készítését (front-end)*
- *üzleti logika készítését (middle-tier)*
- *erőforrások kezelését (back-end)*

Ez első ránézésre túl bonyolultnak tűnhet, de egy jó fejlesztési módszertan és architektúra kiválasztásával ez a folyamat jelentősen leegyszerűsíthető. A már elkészített komponensek újra felhasználhatóak, újak készíthetőek és építhetők be a rendszerbe egyszerűen, valamint létező komponensek kombinálásával újak készíthetőek.

A való világban az alkalmazások…

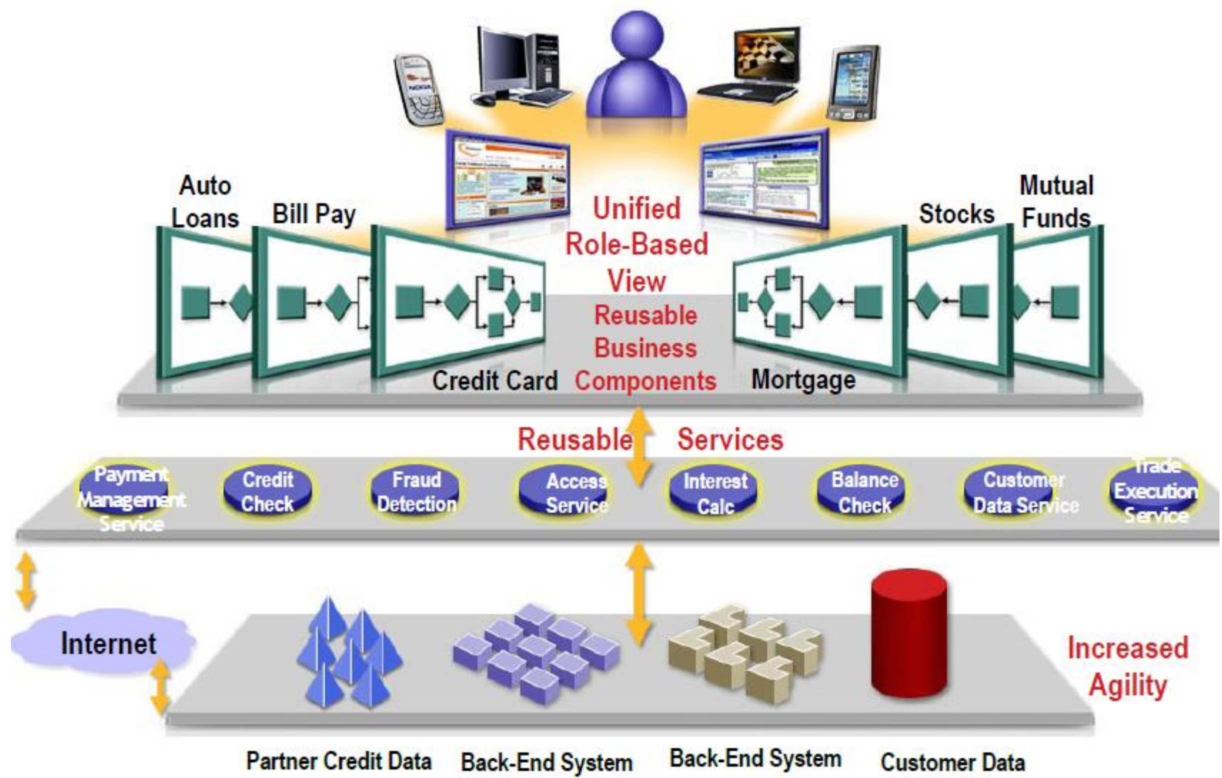
- *nem web alkalmazások*
- *nem Java EE alkalmazások*
- *nem Swing alkalmazások*
- *nem webszolgáltatások*
- *nem relációs adatbázisok*
- *nem szolgáltatásorientált architektúrák*

A való világ alkalmazásai sokkal inkább ezek mind, vagy ezeknek egy jelentős részhalmaza. Hogyan lehet könnyen ilyen összetett alkalmazásokat készíteni? Egy kitűnő megoldás lehet a SOA alapelvek alkalmazása:

- *a funkcionalitás webszolgátatások formájában publikálva van*
- *a szolgáltatások között a kommunikáció szabványosított módon történik*
- *az egész alkalmazás ezeken az alapelveken épül fel.*
- ***De mi is az a SOA…?***

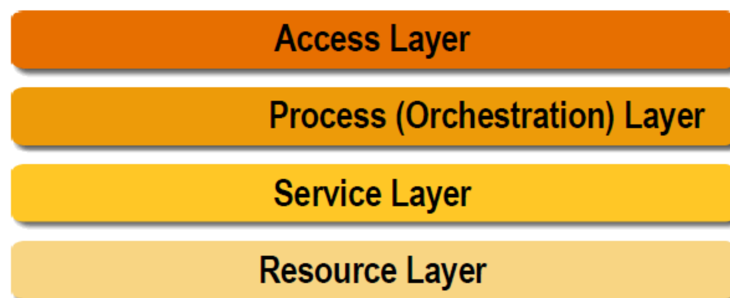
A SOA nem más, mint egy architektúrális minta durva szemcsézettségű szolgáltatások strukturális felépítéséhez. Úgyis tekinthetünk rá, mint egy technológia független best-practice-re. Előnyben részesíti és továbbfejleszti az OO világban kialakult lazán csatoltság fogalmát, a szolgáltatások nem függnek egymástól és semmit nem tudnak a másik implementációjáról.

Akár az is előfordulhat, hogy a szolgáltatás JEE környezetben van implementálva, míg a felhasználó .NET-ben.



2. ábra SOA architektúra

A megfelelő alkalmazáshoz fontos, hogy letisztult szolgáltatásokat tudjunk definiálni, hiszen mindennek ez az alapja. Az új szolgáltatások már létező szolgáltatások újrafelhasználásával és kombinálásával jönnek létre. Ez lehetőséget ad a folyamatok gyors és rugalmas változtatására, akár agilis módon, így amikor egy üzleti igény megváltozik, a szolgáltatásokat újrarendezhetjük megváltoztatva a folyamatot.



3. ábra SOA rétegek

A szolgáltatások a fejlesztők szempontjából egy meghatározott funkció végrehajtására készített fekete dobozok jól definiált interfészekkel, melyek implementációs részleteiről nem tudnak semmit. A szolgáltatásokat leírását WSDL segítségével tehetjük meg egy szolgáltatás

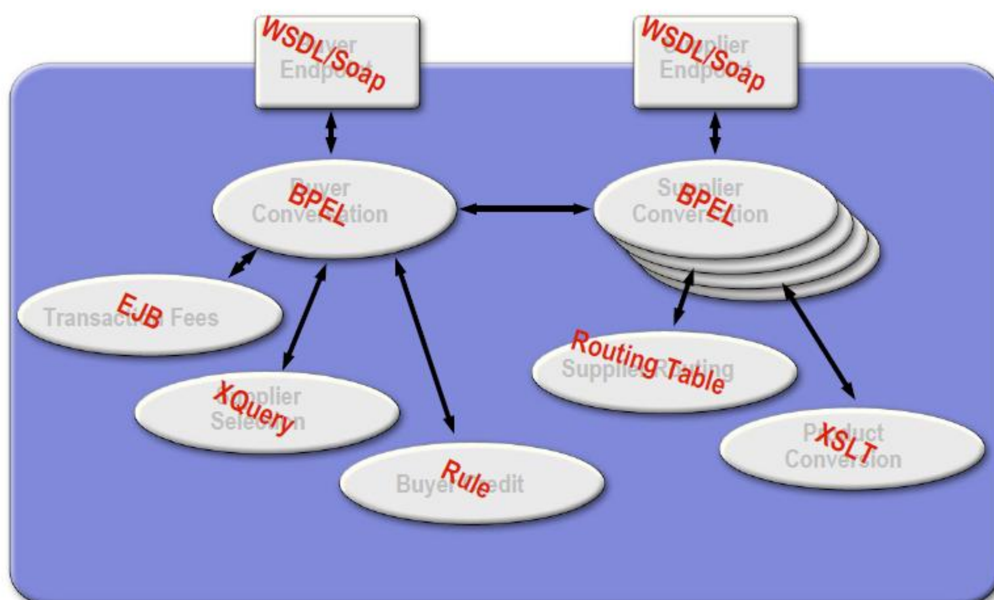
leíróban, ahol a felhasználók a leírás alapján kereshetik meg számukra a megfelelő szolgáltatásokat. A szolgáltatás használó és a szolgáltatás között a kommunikáció XML üzenetek formájában történik.

A szolgáltatások által implementált funkcionalitásra nincs megkötés, szinte bármi lehet:

- *üzleti logika végrehajtása*
- *adatok transzformálása*
- *üzenetek továbbítása*
- *adatbázis lekérdezések végrehajtása*
- *üzleti politika leképezése*
- *üzleti kivételek kezelése*
- *kommunikáció megszervezése több más szolgáltatás között.*

A szolgáltatások implementációs technológiáira sincs megkötés, de a leggyakrabban alkalmazottak a következők:

- *EJB*
- *BPEL*
- *XSLT*
- *Rules*
- *EDI transzformációk*



4. ábra Szolgáltatások készítése

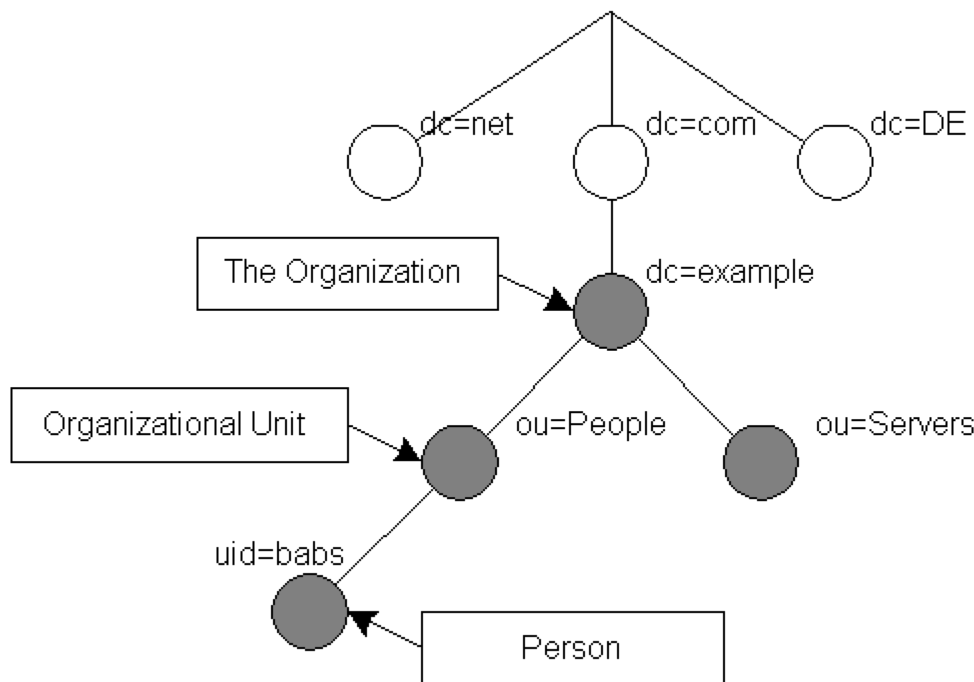
2. Technológiák

1. Open Directory Server

Az ***Open Directory Server*** (***Open DS***) egy ingyenes, LDAP és DSML szabványokon alapuló directory szolgáltatás. Egy directory szolgáltatás egy olyan szolgáltatás, mely lehetővé teszi felhasználók és szoftverrendszerek számára különböző erőforrások azonosítását és elérését a hálózaton. Ezek az erőforrások lehetnek email címek, számítógépek vagy akár periféria eszközök, mint például egy nyomtató. Ideális esetben egy directory szolgáltatás transzparenssé teszi a fizikai hálózati topológiát és protokollokat, így a hálózati felhasználónak nem kell tudnia hol és hogyan vannak az erőforrások fizikailag tárolva.

Ezek az információk egy olyan speciális adatbázisban vannak tárolva, mely keresésre van optimalizálva, ezért olyan esetben érdemes használni, amikor kevés a módosítás, de nagyszámú és gyors lekérdezésekre van szükség. Ennek következtében általában nem alkalmaznak bonyolult tranzakció kezelést vagy roll-back rendszereket, amiket általában az adatbázis-kezelők használnak a nagyméretű, összetett frissítésekhez. A directory aktualizálása tipikusan egyszerű, mindent vagy semmit jellegű változás. Képes arra, hogy széles körben sokszorosítsa az információkat azért, hogy növelje az elérhetőséget és a rendelkezésre állást, miközben a válaszidőt csökkenti. Egyik tipikus alkalmazása authentikáció biztosítása például szerver processzek számára.

Az információ egy faszerű szerkezetben tárolódik (*5. ábra*), és minden csúcsában bejegyzések (entry) szerepelnek. Egy bejegyzésnek van típusa, amely meghatározza, hogy milyen attribútumai lehetnek. Minden egyes ilyen bejegyzésre egyértelműen hivatkozhatunk a bejegyzés DN-jével (Distinguished Name), mely lényegében a fában a csúcshoz vezető utat írja le. A *2. ábrán* látható esetben a dc (domain component) bejegyzések az Internet domain nevek elrendeződését veszik alapul. A példában az example.com site hierarchiájának egy részlete látható. A legmélyebben fekvő csúcs DN - je: *uid = babs ,ou = People, dc = example, dc=com*



5. ábra

Mint látható a directory szolgáltatások érzékeny információkhoz adhatnak hozzáférést, ezért nagyon fontos a megfelelő hozzáférés kontrollálása. Egy lehetséges szabályozás a directory szolgáltatás elérésére a ***Lightweight Directory Access Protocoll*** (***LDAP***). Mint a neve is mutatja ez nem több egy protokollnál, tehát nem egy konkrét termék.

Vegyük a következő példát: Minden email programnak van egy címtára, de hogyan tudjuk megkeresni valakinek a címét, akinek soha nem küldtünk emailt? Hogyan tarthatja a szervezet egy olyan központosított naprakész címtárban az információkat, melyhez mindenki hozzáfér? Ezen kérdések megválaszolására jött létre az LDAP, mint szabvány. Egy LDAP-képes kliens program megkérheti az LDAP szervereket, hogy keressen meg egy neki szükséges bejegyzést. Az LDAP szerverek az adatokat a bejegyzéseikben indexelik, és különböző szűrőket lehet használni, melyek segítségével csak a nekünk szükséges információkat szolgáltatják vissza. Például a következő lehetne egy LDAP kérés magyarra fordított változata: „Keresd meg az összes embert, akinek van email címe, Debrecenben él és a neve tartalmazza a Tamás-t. Szeretném visszakapni ezeknek az embereknek a teljes nevét, email címét és a hozzájuk tartozó leírást”.

Természetesen, mint a fejezet elején említettem az LDAP nincs korlátozva kontakt információkra, vagy információkra emberekről. Kereshetünk titkosított tanúsítványokat, nyomtatókat, hálózati szolgáltatásokat és más szolgáltatásokat a hálózaton. Dolgozatom szempontjából az érdekessége, hogy kombinálhatjuk a következő alfejezetben bemutatandó

single sign-on technológiával, mely lehetővé teszi egy felhasználóhoz tartozó egyetlen jelszó megosztását bármennyi szolgáltatás vagy alkalmazás között.

Az LDAP, mint protokoll nem definiálja hogyan működnek a programok a kliens vagy szerveroldalon. Magát a nyelvet definiálja, mely segítségével a kliens kommunikál a szerverrel (vagy szerver a szerverrel). A kliens oldalon lehet egy email program, egy nyomtató böngésző vagy egy címtár is, míg a szerver oldal használhat csak LDAP-t kommunikációra, vagy lehetnek más metódusok is az adatok fogadására és küldésére – így egy kiegészítő metódusként értelmezve az LDAP-t. A legtöbb kliens általában csak olvas a szerverről. A támogatott keresési lehetőségek nagyon széleskörűek. Előfordulhat néhány kliens, mely ír vagy frissít információkat, de ilyenkor nekünk kell megteremteni a megfelelő biztonsági körülményeket és titkosítást, ezért az frissítésekhez gyakran valamilyen kiegészítő védelmet alkalmaznak, mint például egy titkosított SSL kapcsolat a szerverhez.

A protokoll a még következő két fogalmat definiálja: Engedélyek és Séma. Engedélyeket az adminisztrátor állíthatja be, hogy megengedje bizonyos embereknek az LDAP adatbázishoz való hozzáférést, valamint opcionálisan bizonyos adatokat priváttá tehet. A szerveren tárolt adatok attribútumainak és formátumának leírásához pedig a Séma használható.

2. Open Single Sign-On

Az ***Open Single Sign-On*** (röviden ***Open SSO***) azért jött létre, hogy egy átfogó megoldást kínáljon a single sign-On (SSO) problémák megoldására belső, külső alkalmazások, webalkalmazások és webszolgáltatások esetén. Maga a projekt biztosítja az alap azonosítási szolgáltatásokat, melyek egyszerűsítik a transzparens SSO implementáció, mint biztonsági komponens megvalósítását a hálózatban. Felkínálja a lehetőséget különböző webalkalmazások integrálására, melyek tipikusan különálló azonosításai adattárházakkal dolgoznak és különböző platformokon (webszerverek, alkalmazásszerverek) működnek. A projekt magában foglalja az előző alfejezetben tárgyalt OpenDS projektet, mint konfigurációs tárt.

De mi is az a single sign-on? Több logikailag egymáshoz kapcsolódó, de fizikailag egymástól független szoftverrendszerhez történő hozzáférés biztosító mechanizmus. Aki egyszer jelentkezik be a rendszerbe, kap egy azonosítót, majd ezután hozzáférhet az összes többi rendszerhez is automatikusan anélkül, hogy ismét szükség lenne hitelesítésre. Fordított mechanizmusa a singe sign-off, melynek során egyetlen kijelentkezéssel automatikusan le lesz tiltva a hozzáférés a többi szoftverrendszerhez is. Fontos tényező, hogy a különböző alkalmazások és erőforrások különböző hitelesítő mechanizmusokat támogatnak, így a single sign-on mechanizmusnak tudnia kell lefordítani és tárolnia a különböző hitelesítő adatokat,

melyek össze lesznek hasonlítva az egyetlen kezdeti bejelentkezésnél szerzett hitelesített azonosítóval.

Egy ilyen rendszer számos előnyt biztosíthat a rendszerünkben:

- *leveszi a felhasználó válláról a terhet, hogy sok különböző felhasználónév és jelszó kombinációt kelljen megjegyeznie*
- *csökkenti az eltöltött időt a hitelesített felhasználóhoz tartozó jelszó újra bekérésének elhagyásával*
- *támogatja a hagyományos hitelesítő adatok használatát*
- *csökkenti az IT help deskhez érkező jelszó problémákkal kapcsolatos hívások számát*
- *minden szinten biztonságot ad (belépés/kilépés/hozzáférés), anélkül, hogy újra meg kellene adni a jelszót*

Az előnyök ellenére számos kritika is érte a mechanizmus megjelenése óta. Egyrészt sokkal nagyobb a biztonsági kockázat, ha a felhasználónév és jelszó páros rossz kezekben kerül. Mivel egyetlen kezdeti hitelesítés van, ezért a hitelesítő adatok ellopása esetén hozzáférhetővé válik az összes szoftverrendszer a támadó számára. Ezért az így felépített authentikációhoz érdemes sokkal kifinomultabb hitelesítő adatokkal dolgozni, mint a felhasználónév/jelszó páros. Ilyen megoldás lehet smart card használata, vagy úgynevezett egyszeri jelszó tokenek.

Másrészről az ilyen rendszereknél kritikus kérdés a hiba vagy a rossz rendelkezésre állás. Például hálózati hiba esetén nem lehet elérni a hitelesítő mechanizmus, akkor a hozzáférés meg lesz tagadva az összes többi szoftverrendszerhez is. Ezért SSO alkalmazása olyan rendszerekben, ahol az esetek 100%-ban garantáltnak kell lennie a sikeres belépésnek, például biztonsági rendszerek, nem ajánlott alkalmazni ezt a mechanizmus.

Több tipikus SSO konfiguráció létezik, melyek a következőek:

- *Kerberos alapú*: csak egy kezdeti hitelesítő adat bekérés van, mely létrehoz egy „jegyet”, melyet később felhasznál email kliensek, wikik, verziókezelő rendszerek hitelesítésénél, anélkül, hogy újra azonosítania kellene magát a felhasználónak.
- *Smart card alapú*: kezdeti hitelesítés kéri a felhasználótól a smart card-ot, mely a későbbi azonosítások ennek a smart card-nak a segítségével történnek. A kártyán tárolt adatok lehetnek akár tanúsítványok, akár tárolt jelszavak.

3. Glassfish

A ***Glassfish*** alkalmazásszerver a ***Sun Microsystems Java EE*** platformjának referencia implementációja. Legfrissebb verziója a v3, mely az első olyan alkalmazás szerver, mely teljesen támogatja az új Java EE 6 specifikációt. Magát a specifikációt elsősorban nagyvállalati alkalmazásfejlesztésre hozták létre.

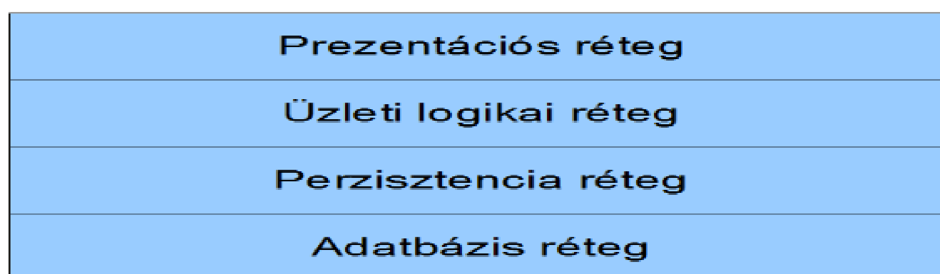
Milyen tulajdonságokkal kell napjainkban egy nagyvállalati alkalmazásnak rendelkeznie?

- *Nagy rendelkezésre állás, hibatűrés*: minden rendszerkomponenst, melyen átmegy az adat legalább dupláznunk kell, így ha az egyik kiesik, akkor a másik átveszi a szerepét. Ennek megvalósításában fontos szerepet játszik az adatok replikálása a komponensek között. Leggyakrabban a négy kilences rendelkezésre állást szokták emlegetni (99.99% - 4.32 perc/hó – évi 52.6 perc), de fontos megjegyezni, hogy nem csak az uptime számít, hanem a Szolgáltatási Szint Szerződésben (SLA – Service Level Agreement) lehet megegyezés egy szolgáltatás (pl.: banki átutalás válaszidejére is)
- *Nagy teljesítmény* (leggyakrabban elosztottak): A rendszernek skálázhatónak kell lennie. Skálázásnak két módja van:
    - *Vertikális skálázás*: minél nagyobb gép, CPU, minél több memória
    - *Horizontális skálázás*: minél több gép, ekkor fontos a gépek közötti gyors kommunikáció biztosítása
- *Tranzakciós*: Nem lehet inkonzisztencia a rendszerben. A tranzakciót egy logikai egységként kell kezelni (egymástól függő, el nem választható műveletek csoportja) és vagy sikeres az összes végrehajtása, vagy sikertelen. Mivel leggyakrabban egyidőben több konkurens felhasználó fér hozzá a rendszerhez, ezért az egyidejű hozzáféréseket is kezelni kell. Erre megfelelő módszer lehet az optimisti locking, pessimistic locking, illetve a two-phase commit.
- *Biztonságos*: Adott erőforrás felhasználójának ellenőrzése három szinten (AAA).
    - *Authentication (Hitelesítés)*: az adott személy beléphet-e a rendszerbe?
    - *Authorization (Jogosultság ellenőrzése)*: Ha felhasználó bejelentkezett, akkor van-e joga egy adott szolgáltatás használatához?
    - *Accounting (Naplózás)*: Szolgáltatások felhasználásának (kezdeti időpontok és vég időpontok rögzítése) naplózása például számlázási, biztonsági célokra.

- *Moduláris*: Konfiguráció megváltoztatása esetén nem szükséges az egész rendszert újraindítani, elég az adott komponenst. Ezzel minimalizálhatjuk a rendszerkiesést, gyorsíthatjuk a fejlesztést. Ezenkívül lehetőséget biztosít a komponensek cseréje, ha például az egyik komponense nem teljesít elég jól, akkor egyszerűen lecseréljük egy másikra.
- *Validálható*: Minden komponens a többitől függetlenül tesztelhető (Unit Testing). Elvárjuk, hogy a helyes bemenetre a helyes kimenetet produkálja a rendszer.
- *Újrafelhasználható*

Ezen szükségletek kielégítése adott esetben nagyon bonyolult is lehet, de a Glassfish alkalmazásszervert használva könnyedén és transzparens módon biztosíthatjuk a szükséges szolgáltatásokat. Az alkalmazásszerver egy speciális része a konténer, mely különböző platformszolgáltatásokat nyújt a benne futó alkalmazásoknak és menedzseli a különböző felhasznált erőforrások életciklusát, elérhetőségét. Ezek a szolgáltatások a JNDI fába regisztrálódnak, és minden Java EE réteg számára elérhetőek. Az alkalmazások ezen konténer egy példányába (instance) deployolódnak. A példány attól függően, hogy milyen szempontok szerint kell felépíteni az alkalmazást kétféle lehet: lehet egy fizikai gépen több példány, vagy egy példány több fizikai gépen is elhelyezkedhet.

Maga a Java EE platform a többrétegű alkalmazás architektúrát támogatja. Általában négy fő rétegre szokás bontani, melyek a *6. ábrán* láthatók.



6. ábra

- A prezentációs réteg a felhasználói interfész megvalósítására szolgál, mely napjainkban leggyakrabban egy webes elérhetőség biztosítása.
- Az üzleti logikai rétegbe kerülnek elhelyezésre az alkalmazás-specifikus kódok, melyek a rendszerből bárhonnan elérhetőek.
- A perzisztencia rétegben történik az állapottér leképezése relációs adatokra és vissza.

- Az adatbázis rétegben pedig leggyakrabban egy relációs adatbáziskezelő helyezkedik el, például MySQL.

A *7. ábrán* az előző 4 rétegű modell kicsit átalakított, Java API-kkal és platformszolgáltatásokkal kiegészített váza látható.

7. ábra

Egy alkalmazás általános fejlesztési lépései a többrétegű modellel és Glassfish alkalmazásszerverrel alulról felfelé a következőek:

Kiválasztunk egy megfelelő erőforrást a perzisztens adatok tárolására. Ez leggyakrabban egy ***JDBC***-n (***Java Database Connectivity***) keresztül elérhető relációs adatbáziskezelő vagy úgynevezett „örökség rendszer". Létre kell hoznunk egy connection pool-t, mely lehetővé teszi, hogy egyszerre sok kapcsolat legyen nyitva az adott erőforrás felé, mégis a kérések felől nézve a kapcsolatok elosztása transzparens, mert a komponensünk szempontjából a pool egyként viselkedik. A Glassfish lehetővé tesz finomhangolási lehetőségeket is, melyek jelentősen befolyásolhatják az alkalmazásunk teljesítményét. Ilyen lehetőségek például a kapcsolatok minimális, maximális száma, illetve pool átméretezési beállítások.

Ha megvan a kapcsolat az erőforrás felé, akkor el kell készítenünk az alkalmazás építőköveit, melyek az entitások (entity bean). Az információs rendszerek leggyakrabban adatokat rögzítenek, transzformálnak és jelenítenek meg. Az entitás fogalom kialakulása előtt jelentős fejtörést, idő- és költségnövekedést okozott a programozóknak a relációs adatbázisok és az objektumorientált programozási nyelvek közötti átjárás megteremtése. A sikeres objektum-relációs leképező keretrendszerek mintájára (***Hibernate, Oracle TopLink***) elkészítették a ***JPA*** (***Java Persistance API***) specifikációt, mely gyors népszerűségre tett szert és hamar a Java SE részeként is bemutatkozott. A JPA lehetővé teszi speciális perzisztálható

osztályok létrehozását, melyek nagyon egyszerűen leképezhetők relációs táblákra. A fejlesztés meggyorsítása érdekében a kódból speciális annotációkkal definiálható a működés, de xml leírókkal később könnyedén felüldefiniálható az eltérő környezeteknek megfelelően. Maga a szemlélet minden adatot objektumként tekint, megengedi a szokásos objektumorientált fogalmak használatát, mint az öröklődés, és az egy-egy, egy-sok, sok-sok kapcsolatok könnyű leképezését. A következő példakód egy Személy entitás reprezentálását példázza:

```
@Entity
@Table(name="persons")
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Integer id;
    @Column(name="firstname")
    private String firstname;
    @Column(name="lastname")
    private String lastname;
    @Version
    private Integer version; ...
}
```

Ha definiáltuk az alkalmazásban használt egyedeket, elkezdhetjük elkészíteni a tényleges üzleti logikát. Ezek elhelyezésére az ***EJB***-k (***Enterprise Java Beans***) lesznek alkalmasak, melyek elosztott, újrafelhasználható, biztonságos, tranzakciós szerveroldali komponensek. Elfedik a programozó elől a többszálúságot, ami jelentősen felgyorsíthatja a fejlesztési folyamatokat. A Java EE 5 specifikáció 3 felé EJB-t definiál:

- *Session Bean*: tipikusan üzleti folyamatok reprezentálására.
- *Message Driven Bean*: valamilyen üzenetre reagál, tipikusan aszinkron módon. A forrás általában egy JMS üzenet. Általában az integrációs réteghez kötődik szorosabban. (Ld.: következő alfejezet)
- *Entity Bean*

Az új Java EE 6 specifikációban, terjedelmi okokból az *Entity Bean*-ek a JPA specifikációval együtt kikerültek egy különálló specifikációba.

A Session Bean-ek tovább bonthatók alcsoportokra:

- *Stateless Session Bean*: ha nincs szükség a hívások közötti információ cserére (conversation state). Egy bean példány akárhány különböző klienst kiszolgálhat úgy, hogy a következő kliensre nincs hatással az előző klienssel végzett művelet. Ezekívül egyik jelentős felhasználásuk, hogy lehetőséget adnak, hogy szolgáltatásaikat webszolgáltatásokként publikájuk.
- *Stateful Session Bean*: ha szükség van a hívások közötti információ cserére. Ekkor egy bean példány csak egyetlen klienssel állhat kapcsolatban, más kliensekkel nem kommunikálhat. Tipikus használati eset például a regisztrációs folyamat, bevásárló kosár.
- *Singleton Session Bean*: a Java EE 6 specifikációval együtt jelent meg. Egy olyan speciális Stateful Session Bean, mely alkalmazás hatókörben található és osztoznak rajta a kliensek az alkalmazás elindulásától, az alkalmazás leállásáig.

EJB-k jellegzetessége, hogy a Stateful Session Bean-ektől eltekintve ezek is pooled service-ek, itt is sok példány dolgozik a háttérben és egyenletes terheléseloszlásnál csökkenthetőek az erőforrásköltségek. Távolról is hívható objektumok, nem muszáj, hogy a hívott és a hívó egy alkalmazáson vagy egy szerveren belül helyezkedjen el.

Egy Stateless Session Bean:

```
@Stateless
public class PersonManagerBean implements PersonManagerBeanLocal {
    @PersistenceContext
    private EntityManager em;
    public void createEntity(String firstname, String lastname) {
        Person person = new Person();
        person.setFirstname(firstname);
        person.setLastname(lastname);
        em.persist(person);
    }

    public String getFullname(Integer id) throws NoSuchPersonException {
        Person person = em.find(Person.class, id);
        if(person == null)
            throw new NoSuchPersonException();
        return person.getFirstname() + "" + person.getLastname();
    }
}
```

Egy Message Driven Bean:

```
@MessageDriven(mappedName = "jndi/pool",
    ActivationConfig =  {
          @ActivationConfigProperty(propertyName =
"acknowledgeMode", propertyValue = "Auto-acknowledge"),
          @ActivationConfigProperty(propertyName =
"destinationType", propertyValue = "javax.jms.Queue")
    })
public class DistributorHandlerBean implements MessageListener {
    public DistributorHandlerBean() {
    }
    public void onMessage(Message message) {
    }
}
```

Ha elkészültünk az üzleti logikai réteggel, akkor már csak a prezentációs réteg megvalósítása van hátra, mely legegyszerűbben egy JSF keretrendszerrel tehető meg. Egy ilyen keretrendszer (például ICEFaces) lehetővé teszi gyors komponensalapú felhasználói felület fejlesztését, mely leggyakrabban csak az üzleti logikai réteggel van kapcsolatban, mert főleg adatok be- és kivitelére szolgál. Ezek a keretrendszerek adott esetben nagyon eltérhetnek egymástól, mert a JSF 1.2 specifikáció nem definiál olyan napjainkban szinte mindenhol használt követelményeket, mint az AJAX. A JSF 1.2 MVC (Model-View-Controller) fejlesztések elősegítésére jött létre, de az igazi előrelépést az Java EE 6-al megjelent JSF 2.0 jelenti, mely már szabványként definiálja az AJAX követelményt és egy új újrafelhasználható programozási modellt hoz be a Faceletek megjelenésével. Másik két lehetőség még UI fejlesztésére a JSP és a Servlet. Ez a két technológia akár egymás inverzeként is tekinthető, ugyanis a JSP HTML kódba ágyazott Java kód, míg a Servlet Java kódba ágyazott HTML kód. A háttérben valójában a JSP oldalak is Servlet-té fordulnak és gyakorlatban nem is használnak ilyen fajta „kód-keveredést”. Ezen két technológia egyre inkább háttérbe szorul napjainkban, mert az előzőhöz képest csak alacsonyszintű oldalgenerálást tesznek lehetővé.

Ezen fejlesztési lépések során igénybe vehetjük a már említett konténerszolgáltatásokat. Ilyen szolgáltatás a tranzakciókezelés. Minden réteg támogatja és követelmény, hogy az integrációs réteg is támogassa. Alapból mindent EJB tranzakciós, de ez tetszőlegesen felüldefiniálható. Kétféle tranzakciókezelést használhatunk:

- *Container Managed Transactions* (CMT): a konténer kezeli számunkra tranzakció indítását, véglegesítését, visszagörgetését és a tranzakciós határokat.

- *Bean Managed Transactions* (BMT): a programozónak kell kezelni a tranzakciók indítását, véglegesítését, visszagörgetését, felfüggesztését és a tranzakciók határait.

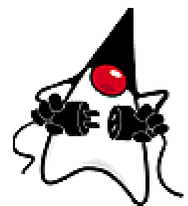
Ezek kezelése a ***JTA***-val (***Java Transaction API***) valósulhat meg. Másik kritikus szolgáltatás a biztonság. A megfelelő biztonság megteremtésére a ***JAAS*** használható (***Java Authentication and Authorization Service***). Előnye, hogy komponens alapú, minden része cserélhető vagy felüldefiniálható. Szerepkörök definiálhatók, melyekhez tartozó identitások tárolására bármilyen JDBC-képes erőforrás használható, illetve egy LDAP szerver is. A hozzáférés szabályozása bárhol definiálható a programban, akár dekleratívan (ekkor a konténer kényszeríti ki a biztonsági előírások betartását) akár programozva (ekkor a programozónak kell biztosítani, hogy ne lehessen áthágni a biztonsági szabályokat). Többféle azonosítást is támogat (Jelszavas, Smar card, Kerberos), melyek jól kombinálhatóak SSO megoldásokkal.

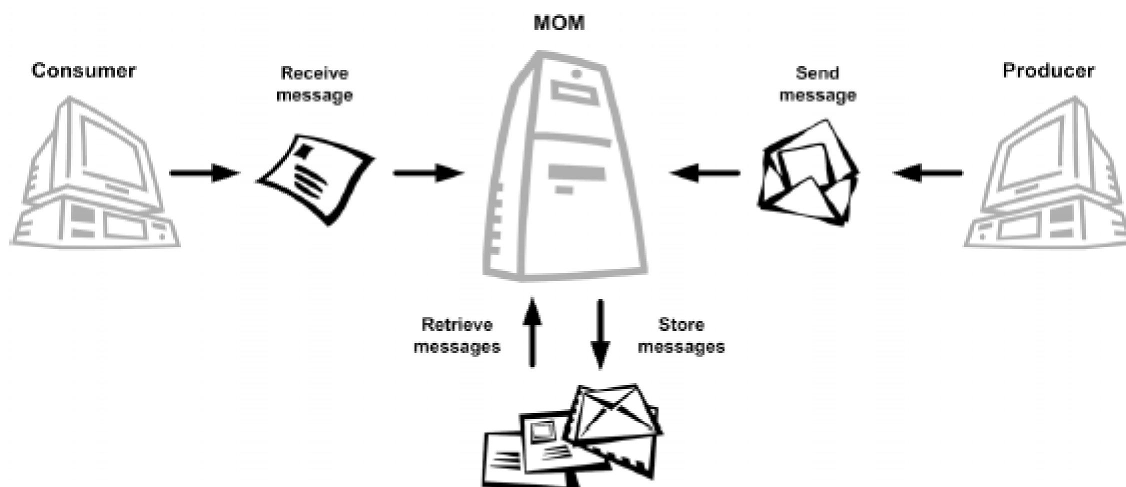
Mint látható a Glassfish kitűnően támogatja a többrétegű alkalmazások elkészítését, hozzájuk tartozó követelmények (terheléselosztás, hibatűrés, biztonság, stb..) kielégítését, valamint az előző két alfejezetben említett technológiával robosztus programok fejlesztését.

4. Open Message Queue

Nagyvállalati alkalmazásokban illetve különböző vállalatok alkalmazásai közötti kommunikációban gyakran alkalmazott eljárás az üzenetküldés (messaging). Amikor üzenetküldés fogalmáról beszélünk, akkor rendszerek, rendszerkomponensek közötti lazán csatolt, aszinkron kommunikációra gondolunk.

Az Open MQ projekt nem más, mint egy skálázható, nagyteljesítményű nyílt forrású üzenetkezelő szerver implementáció. Lehetőséget biztosít üzenet-orientált rendszerintegrációhoz a ***JMS*** (***Java Message Service***) teljes implementálásával, mellyel alkalmazásaink között a megbízható együttműködés biztosítható anélkül, hogy szinkron kommunikációra támaszkodnának. Az ezt biztosító szoftvereket gyakran ***Message Oriented Middleware***-nek is (***MOM***) nevezik. Maga a projekt része a Glassfish alkalmazásszervernek, de önállóan is használható.





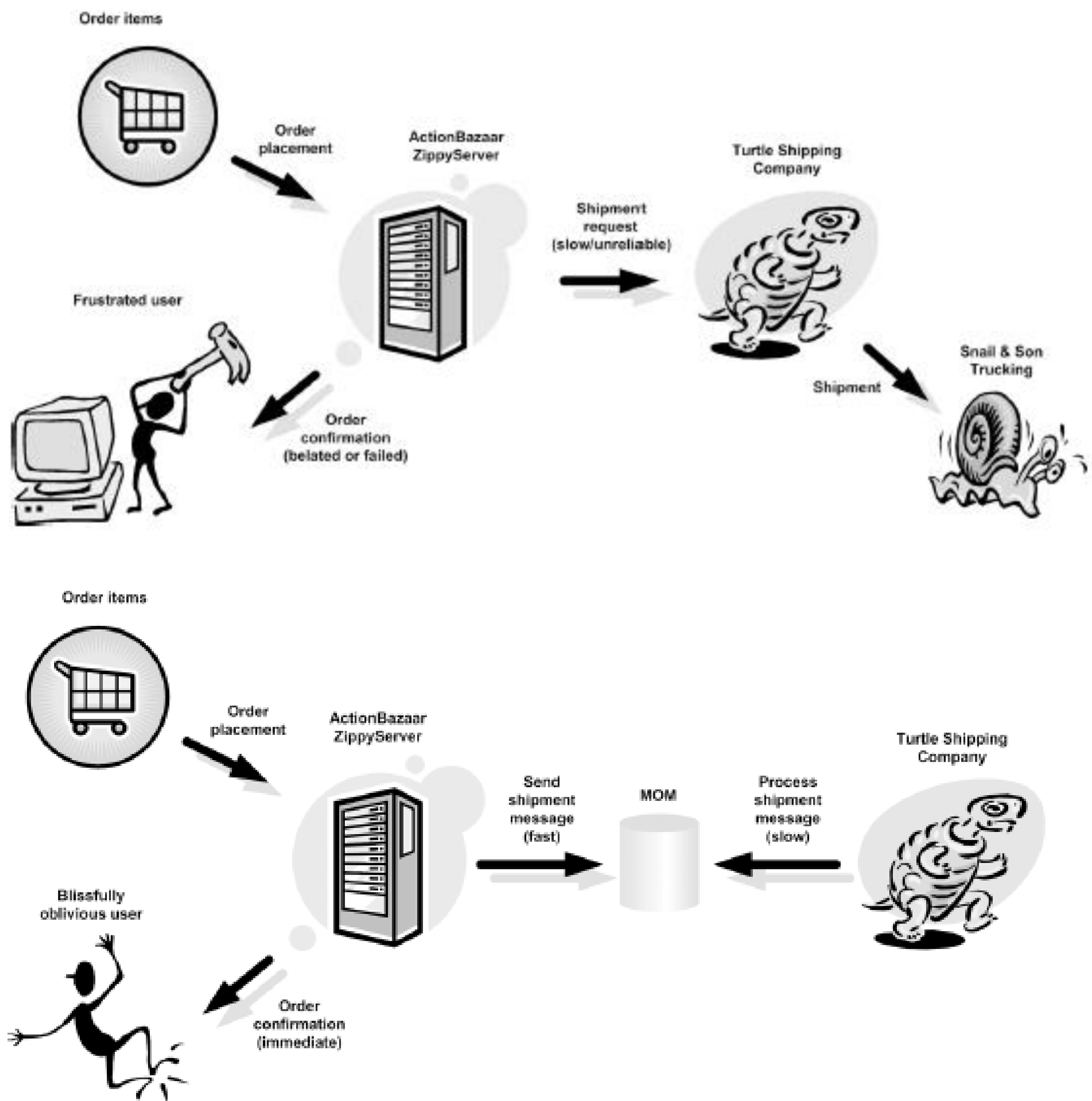
8. ábra Tipikus MOM felépítés

Egy tipikus MOM felépítésben az üzenetszerveren kívül részt vesz még legalább egy küldő (Producer), aki az üzenetet szeretne küldeni egy vagy több másik komponensnek és egy vagy több fogyasztó (Consumer), aki érdekelt az üzenet tartalmában. Nagyon kritikus pont a felépítésben, hogy az kommunikációnak megbízhatónak kell lennie. Az aszinkron kommunikáció és akár különböző szervereken lévő komponensek jellegéből adódóan előfordulhat olyan eset, hogy az üzenet küldésének pillanatában a küldő online, a fogyasztó pedig offline, de az ilyen esetekben is tudni kell biztosítani az üzenetek kézbesítését. Ezt leggyakrabb az üzenetek MOM szerveren történő perzisztens tárolásával oldják meg, számunkra transzparens módon. Ezek az üzenetek csak akkor törlődhetnek, ha a fogyasztó megerősítést küldött az üzenet feldolgozásáról.

Alapvetően kétféle üzenetkezelési modellt különböztetünk meg:

- *Point-to-Point (PTP):* Ekkor egy küldő küld üzenetet, mely pontosan egy fogyasztóhoz fog eljutni. A küldő elhelyezi az üzenetét az úgynevezett üzenetsorban, melyből egy online fogyasztó kiveszi és feldolgozza az üzenetet. Gyakori eset, hogy a sorhoz több fogyasztó is kapcsolódik, ekkor véletlenszerűen lesz kiválasztva melyik fogyasztó kapja meg az üzenetet. Fontos megjegyezni, hogy az sor csak névlegesen sor, tehát nincs garantálva, hogy az először elhelyezett üzenet fog először fogyasztóhoz kerülni.
- *Publish-subscribe (pub-sub):* Egy küldő küld üzeneteket egy úgynevezett témára (topic), mely üzeneteket minden a témára feliratkozott éppen online „előfizető” (subscriber) megkap.

A küldők és fogadók implementálhatóak JMS segítségével is, de a fogadók leggyakoribb megvalósítása Message-Driven Bean-ek (MDB) segítségével történik. Ez az EJB típusjelentősen leegyszerűsíti és meggyorsítja a fejlesztést. A MDB-t be kell regisztrálni egy adott üzenetsor vagy téma figyelésére, és mikor üzenet érkezik az alkalmazásszerver automatikusan aktiválja a megfelelő fogyasztót. A megoldás ereje abban rejlik, hogy a kommunikáció, tranzakciókezelés transzparens módon el van fedve a fejlesztő elől.



9. ábra B2B MOM megvalósítással

MOM tipikus alkalmazása lehet, amikor cégek közötti kommunikációban az egyik cég szolgáltatása lassú, vagy nem megbízható. Ekkor egy szinkron kommunikáció alkalmazás

rontaná a másik cég szolgáltatásának a minőségét, mely hosszútávon piacvesztéshez vezethet. A *9. árbán* egy tipikus Business to Business (B2B) szituáció látható: az ActionBazaar szolgáltatásai megfelelő minőségűek, az ügyfelek könnyen licitálhatnak a termékekre, de a megrendelésnél, mely a Turtle szállítási cégen keresztül történik, könnyen hibába ütközhetnek. Ennek megoldására alkalmaztak egy üzenetsort, mely garantálja a megrendelés megérkezését a Turtle szállítási céghez, akkor is, ha annak szolgáltatása éppen nem elérhető, vagy terhelés miatt olyan válaszidővel rendelkezik, mely az SLA-ban foglaltak többszöröse is lehet.

5. Open ESB

Az előző négy alfejezetben bemutatásra kerültek a szolgáltatásorientált szemléletmód kiépítéséhez szükséges alap építőkövek. Ennek az alfejezetnek a témája az Open ESB, mely már konkrétan a szolgáltatásorientált funkcionalitás megvalósítására fókuszál.

Általánosan elfogadott alapelv SOA berkekben a következő gondolat: a webszolgáltatások nem az egyetlen mód a SOA megvalósítására, de egy egészen kiváló ötlet rá. Két fő webszolgáltatás fajtát különíthetünk el:

- *REST webszolgáltatás (Representational State Transfer)*
  (Megjegyzés: napjainkban egyre inkább térnyerő webszolgáltatás fajta, mely terjedelmi okokból, most nem kerül bemutatásra.)
- *WSDL/SOAP webszolgáltatás*.

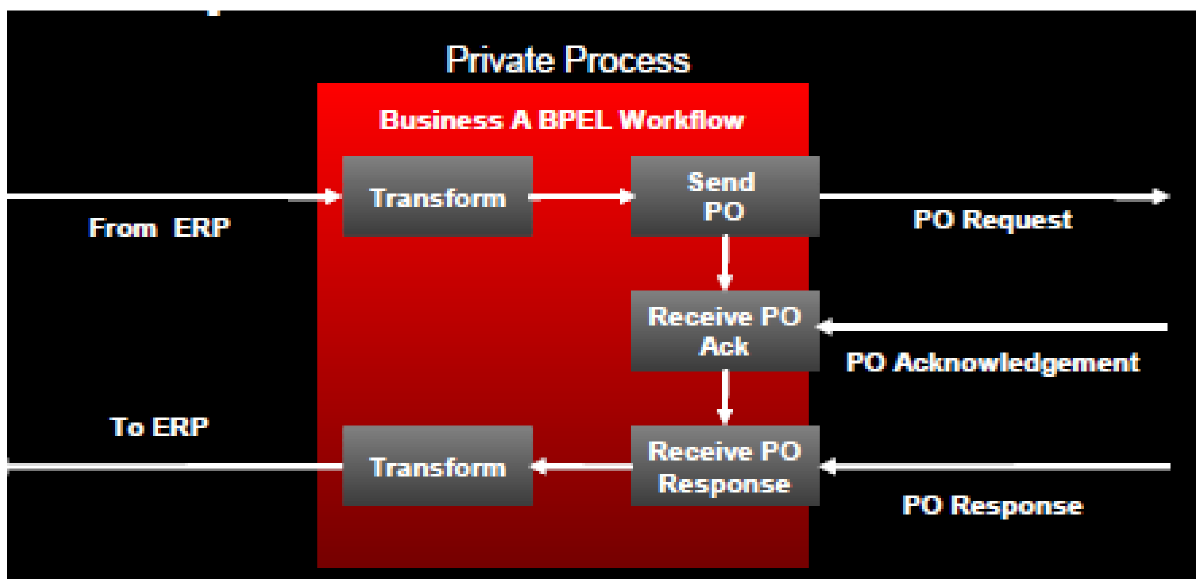
Miért jobb SOA megvalósításához a WSDL alapú megközelítés?

- könnyen automatizálhatóvá teszi a kommunikációt a felek között
  - komponensek tudják értelmezni a WSDL-t, így meghívni egy számukra megfelelő szolgáltatást
- felfedezhető a registry-n keresztül
  - a registry-ben tárolódnak a publikált webszolgáltatások leírására, melyekből kideríthető mire alkalmas a webszolgáltatás, valamint hogyan lehet meghívni.
- validálható szabvány
  - külső cégek validálhatják, hogy a kommunikáció tényleg a szabványnak megfelelően történik
- http protokoll felett működik, így átjuthat a tűzfalakon

- stateless session bean-ek szolgáltatásai egy egyszerű interface definícióval kipublikálhatóak webszolgáltatásként, ezzel meggyorsítva a fejlesztést továbbra is igénybe véve az alkalmazásszerver által kínált transzparens szolgáltatásokat.

Bár a webszolgáltatások jelentős áttörést hoztak, mégsem bizonyultak elegendőnek. A webszolgáltatások kifejlesztése és funkcionalitásának közzététel WSDL segítségével nem bizonyultak elég hatékonynak. Képzeljük el egy koncert jegyrendelés folyamatát. Egy ilyen koncertjegy rendelő szolgáltatásnak legalább 3 művelete van, melyeket meghatározott sorrendben kell végrehajtani:

- ár meghatározása
- jegy megvásárlása
- megerősítés/visszavonás



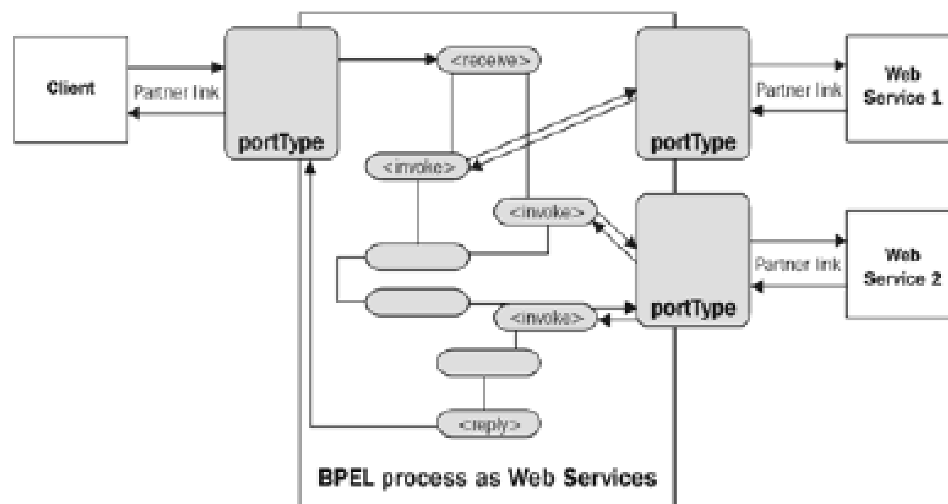
10. ábra Üzleti folyamat több lépésben

Látható, hogy megjelenik az igény a szolgáltatások végrehajtási sorrendjének megadása iránt. Ennek a problémának a megoldására alakul ki az üzleti folyamat (Business Process) fogalom. Az üzleti folyamatok megjelenése az IT-ben több új megoldandó problémát vetett fel:

- *szolgáltatások közötti aszinkron kommunikáció koordinálása*
- *felek közötti üzenetcsere biztosítása*
- *a tevékenységek párhuzamos feldolgozásának implementálása*
- *kompenzáció logika implementálása (visszavonás műveletek)*
- *partner interakciók közötti adatmanipulálás/transzformáció*

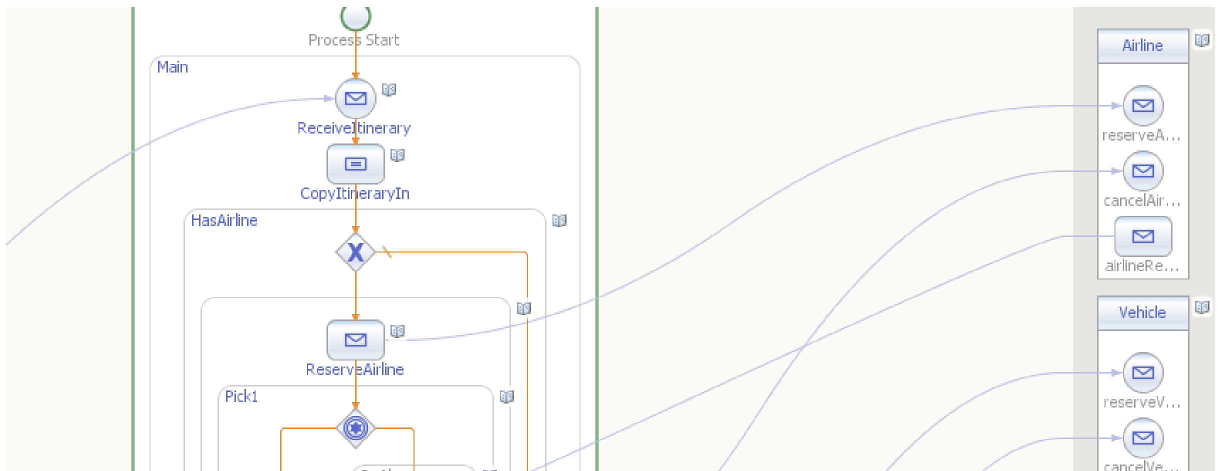
- *hosszan futó üzleti tranzakciók és tevékenységek támogatása*
- *konzisztens hibakezelő biztosítása*
- *univerzális adatmodell az üzenetek cseréjéhez.*

Ezen felmerült problémakörökre válaszul alakult ki a ***BPEL*** (***Web Services Business Process Execution Language***), mellyel leírhatjuk webszolgáltatásokból álló üzleti folyamatainkat. A BPEL úgymond kijavítja a WSDL azon hibáját, hogy nem más, mint üzenetcserére épülő műveletek rendezetlen halmaza. Így a folyamat leírása valójában egy „párbeszéden" alapuló folyamatábrát fog jelenteni a fejlesztők számára, ahol a „párbeszéd" csak webszolgáltatások közötti WSDL-el leírt üzenetcseréket fog tartalmazni. Maga a folyamat példány pedig a folyamatábrát követő „párbeszéd" lesz és a hívó számára nem fog különbözni egy webszolgáltatás hívástól. Elvárás, hogy futtató rendszer képes legyen akárhány konkurens folyamatpéldány végrehajtására.



11. ábra BPEL folyamat, mint webszolgáltatás

Maga a BPEL egy gráf-orientált webszolgáltatás kompozíció-készítő nyelv. Az XML-alapú felépítésből adódóan egy BPEL folyamat könnyen modellezhető és összeállítható egy vizuális modellező eszköz segítségével. Az aciklusos gráf megadására felhasználhatunk tevékenység csomópontokat (üzenetcsere, belső művelet, döntési pont) és megszorításokat (végrehajtási sorrend, konkurens végrehajtás) teljesen elkülönítve a hibakezelő és kompenzációkezelő logikától.



12. ábra Üzleti folyamat vizuális modellezése NetBeans-ben

Az üzleti folyamat leírása során a következőeket tehetjük meg:

- *felhasználhatunk szolgáltatást (invoke)*
- *létrehozhatunk szolgáltatást (recieve/reply)*
- *finom-szemcsézettségű szolgáltatások aggregációja*
- *durva-szemcsézettségű szolgáltatások létrehozása*

A BPEL leíró dokumentum struktúrája:

```
<process>
     <!– Folyamat résztvevőinek és szerepköreiknek definiálása -->
     <partnerLinks> ... </partnerLinks>
     <!– Folyamatban használat adat/állapot -->
     <variables> ... </variables>
     <!– „Párbeszédekhez” szükséges tulajdonságok-->
     <correlationSets> ... </correlationSets>
     <!– Kivételkezelés -->
     <faultHandlers> ... </faultHandlers>
     <!– Kompenzációkezelés -->
     <compensationHandlers> ... </compensationHandlers>
     <!– Konkurens események kezelése -->
     <eventHandlers> ... </eventHandlers>
     <!– Business process flow -->
     (activities)*
</process>
```

Mint a struktúrában is látható az üzleti folyamat tevékenységek megadásával történik. Minden lépés az üzleti folyamatban egy tevékenységnek felel meg. Kétféle tevékenységtípust különböztetünk meg:

- *alap tevékenység*

- <invoke>
  - megengedi, hogy meghívjon egy a partner által ajánlott egyirányú vagy kérés/válasz alapú műveletet az adott port típuson.
- <recieve>
  - lehetővé teszi egy blokkolódó várakozást egy megfelelő üzenet érkezéséig
  - lehet az üzleti folyamat példányosítója.
- <reply>
  - megengedi, az üzleti folyamatnak, hogy válasz üzenetet küldjön egy <recieve>-en keresztül kapott üzenetre
  - a <recieve> és <reply> kombinációja egy kérés-válasz műveletet eredményez az adott WSDL port típuson.
- <assign>
  - a változók értékei frissíthetőek új értékkel.
- <throw>
  - hibát generál a folyamat belsejéből.
- <wait>
  - adott ideig történő várakozást tesz lehetővé.
- <empty>
  - lehetővé tesz egy nincs-művelet instrukciót az üzleti folyamatban.
  - ez például hasznos lehet konkurens tevékenységek szinkronizálásánál.

- *strukturált tevékenység*
  - <sequence>
    - tevékenységek egy kollekcióját lehet definiálni, melyek szekvenciálisan lesznek végrehajtva.
  - <while>
    - egy tevékenység addig lesz ismételve, míg egy bizonyos feltétel teljesül.
  - <pick>
    - lehetővé teszi blokkolódást és várást egy megfelelő üzenet érkezéséig vagy egy bizonyos időkorlát lejárásáig.

        - ha valamelyik trigger aktiválódik, akkor a kapcsolódó tevékenység végrehajtódik.
    - <flow>
        - egy vagy több tevékenység megadása konkurens végrehajtásra.
    - <scope>
        - lehetőséget ad beágyazott tevékenységek definiálására, saját változókkal, kivételkezelőkkel és kompenzáció kezelőkkel.
    - <compensate>
        - kompenzáció meghívása egy belső hatáskörből, mely már hiba nélkül véget ért.
        - egy ilyen konstrukció csak egy kivételkezelőből vagy másik kompenzáció kezelőből hívható meg.
    - <switch>
        - feltételes elágaztatást biztosít
        - a tevékenység case ágak rendezett listáját tartalmazza egy opcionális otherwise ággal.
    - <if>
        - feltétels elágaztatást biztosít.
    - <repeatUntil>
        - tevékenységet ismétel, a feltételnek megfelelően.
    - <foreach>
        - végigmegy egy kollekció összes elemén.

Az előzőeket felhasználva egy példatevékenység BPEL-el leírva:

```
<sequence>
<receive partnerLink="customer" portType="lns:purchaseOrderPT"
     operation="sendPurchaseOrder" variable="PO"
     createInstance="yes" />
<flow>
     <invoke partnerLink="inventoryChecker"
portType="lns:inventoryPT"
     operation="checkINV" inputVariable="inventoryRequest"
     outputVariable="inventoryResponse" />

     <invoke partnerLink="creditChecker" portType="lns:creditPT"
     operation="checkCRED" inputVariable="creditRequest"
     outputVariable="creditResponse" />
</flow>
```

```
...
<reply partnerLink="customer" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="invoice"/>
</sequence>
```

Most, hogy már le tudjuk írni üzleti folyamatinkat tevékenységgel, nézzük meg, milyen módon kommunikálhatnak külső webszolgáltatás erőforrásokkal:

- BPEL folyamat meghív egy műveletet egy másik webszolgáltatáson
    - invoked partnerLink –el van reprezentálva
- a BPEL folyamatot hívja egy kliens
    - client partnerLink –el van reprezentálva
    - a kliens szempontjából az üzleti folyamat nem különbözik egy webszolgáltatástól. Amikor egy BPEL folyamatot hozunk létre, valójában egy webszolgáltatást írunk le.

Tehát azokat a szolgáltatásokat, melyekkel az üzleti folyamat kommunikál partnerLink –ekkel (partner kapcsolat) vannak modellezve. Minden egyes partner kapcsolatot egy partner kapcsolat típussal (partnerLinkType) tudunk leírni. A kommunikáció során mind az üzleti folyamat, mind a partner szolgáltatás valamilyen szerepkörben van. Az előbbi szerepkörért a myRole attribútum írja le, míg utóbbiét a partnerRole attribútum.

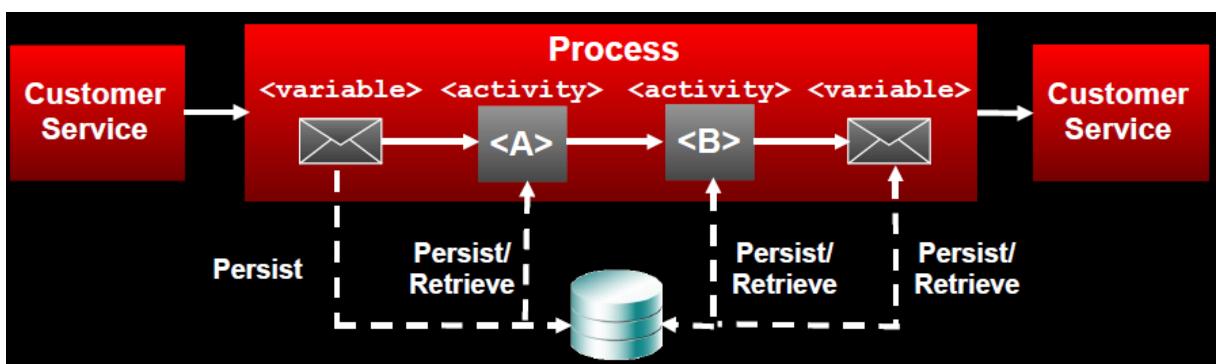
```
<?xml version="1.0" encoding="utf-8"?>
<process name="insuranceSelectionProcess"
      targetNamespace="http://packtpub.com/bpel/example/"
      xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:ins="http://packtpub.com/bpel/insurance/"
      xmlns:com="http://packtpub.com/bpel/company/" >
<partnerLinks>
<partnerLink name="client"
      partnerLinkType="com:selectionLT"
      myRole="insuranceSelectionService"/>
<partnerLink name="insuranceA"
      partnerLinkType="ins:insuranceLT"
      myRole="insuranceRequester"
      partnerRole="insuranceService"/>
<partnerLink name="insuranceB"
      partnerLinkType="ins:insuranceLT"
      myRole="insuranceRequester"
      partnerRole="insuranceService"/>
</partnerLinks>
...
```

Maga a partner link típusa definiálja két szolgáltatás közötti párbeszédben a szolgáltatás által betöltött szerepköröket és mindkét szolgáltatás által biztosított port típust

(portType), melyen keresztül a párbeszéd kontextusában érkező üzeneteket fogadják. Minden egyes szerepkör pontosan egy WSDL port típust specifikál.

```
<partnerLinkType name="BuyerSellerLink"
     xmlns="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">
     <role name="Buyer" portType="buy:BuyerPortType"/>
     <role name="Seller" portType="sell:SellerPortType"/>
</partnerLinkType>
```

Definiálhatunk változókat is, melyek segítségével üzenetek küldhetők és fogadhatók a partnerrel történő kommunikációban. Ezek WSDL típusok vagy üzenetekként vannak definiálva és perzisztálhatók sokáig futó folyamatok számára. (*13. ábra*)



13. ábra

Változók definiálva BPEL-ben:

```
<variables>
     <variable name="PO" messageType="lns:POMessage"/>
     <variable name="Invoice" messageType="lns:InvMessage"/>
     <variable name="POFault" messageType="lns:orderFaultType"/>
</variables>
```

Előző változókhoz tartozó típusok definiálva a folyamathoz tartozó WSDL-ben:

```
<message name="POMessage">
     <part name="customerInfo" type="sns:customerInfo"/>
     <part name="purchaseOrder" type="sns:purchaseOrder"/>
</message>
     <message name="InvMessage">
     <part name="IVC" type="sns:Invoice"/>
</message>
     <message name="orderFaultType">
     <part name="problemInfo" type="xsd:string"/>
</message>
```

A változókban tárolt adatok az <assign> és <copy> használatával manipulálhatóak. A <copy> támogatja az XPath lekérdezéseket rész-adathalmazok lekérdezéséhez.

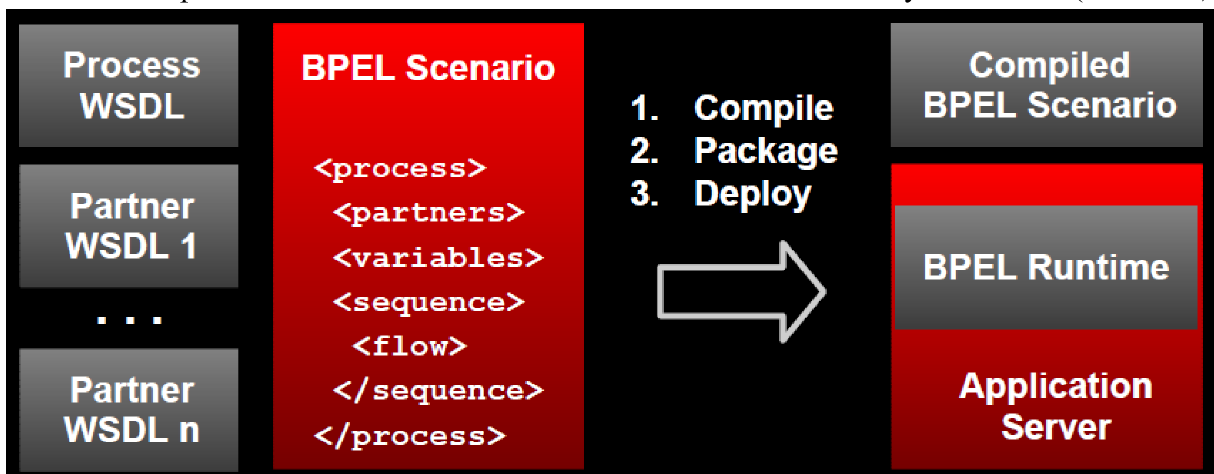
Az utolsó BPEL eszköz, ami bemutatásra kerül az a hatáskör. A hatáskör tevékenységek egy részhalmazához biztosíthat elosztott környezetet. Ez az elosztott környezet tartalmazhat:

- *hibakezelőket*
- *eseménykezelőket*
- *kompenzációkezelő változókat*
- *korrelációs halmazokat*

Ezenkívül lehetőséget biztosít, hogy a változókhoz érkező konkurens kéréseket szerializálhassuk.

```
<scope
variableAccessSerializable="yes|no"
...>
     <variables>
     </variables>
     <correlationSets>? ...
     </correlationSets>
     <faultHandlers>
     </faultHandlers>
     <compensationHandler>? ...
     </compensationHandler>
     <eventHandlers>
     </eventHandlers>
     (activities)*
</scope>
```

A teljesség igénye nélkül az eddig felsorolt eszközöket használhatjuk üzleti folyamataink leírására. Ha elkészültünk a BPEL összeállítással, le kell fordítani, megfelelő struktúrában elhelyezni és BPEL futtató környezettel rendelkező alkalmazásszerverbe deploy-olni. Ezen lépések után bárki számára elérhető lesz az üzleti folyamatunk. (*14. ábra*)



14. ábra BPEL folyamat publikálása

Sajnos a fejlesztők megpróbáltatásai nem érnek véget üzleti folyamataik kipublikálásával. Szükség van a különböző alkalmazások által biztosított folyamatok egységbe integrálására, hogy a megfelelő működést tudjuk biztosítani az ügyfelek számára. Itt jön a képbe a ***JBI*** (***Java Business Integration***). Ez nem más, mint egy szabványos alkalmazásintegrációs keretrendszer. Az JBI az alkalmazás integráció szemszögéből legjobban úgy fogalmazható meg, mint a Java EE fogalma az üzleti alkalmazások területén. Kialakulásának előzménye a SOA fejezetben bemutatott ***EAI*** alkalmazások (***Enterprise Application Integration***) problémái.

15. ábra Java Business Integration

Fizikailag egy szabványos meta-konténer formájában jelenik meg. Egy ilyen szolgáltatás konténer bármilyen típusú szolgáltatást (szolgáltatás egység) tartalmazhat:

- *üzleti logikai szolgáltatás*
- *rendszerszolgáltatás*

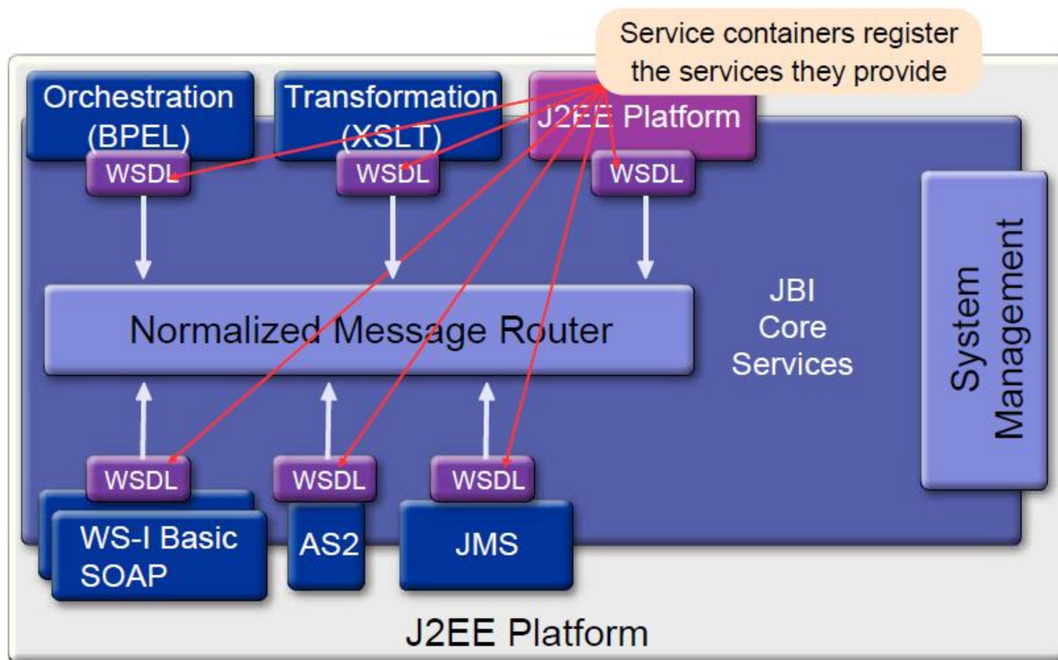
Ezek a szolgáltatások lehetnek lokálisak vagy akár távoli gépen is elhelyezkedhetnek. Az architektúrája moduláris, két fő típusú elemből állhat:

- *Service Engines (SE)* – lokális szolgáltatás vagy fogyasztó.
    - *BPEL SE*
    - *XSLT SE*
    - *JavaEE SE*
    - *IEP SE*
    - *ETL SE*
    - *SQL SE*

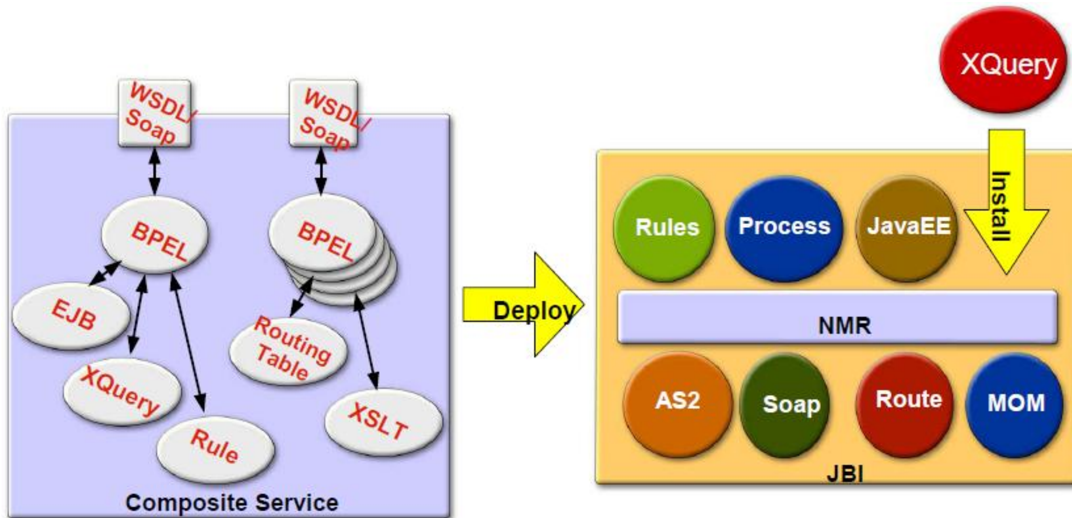
- *Workflow SE*

- *Binding Components (BC)* – távoli szolgáltatás vagy fogyasztó.
  - *MQSeries BC*
  - *HL7 BC*
  - *SAP BC*
  - *SMTP BC*
  - *HTTP BC*
  - *JMS BC*
  - *File BC*
  - *CICS BC*
  - *CORBA BC*

A rendszerszolgáltatásokat biztosító konténerek beregisztrálják az általuk nyújtott szolgáltatásokat a JBI központi szolgáltatási közé, így egyszerűen és könnyen elérhetőek. (*16. ábra*) Majd erre az architektúrára deploy-olódnak a szolgáltatásaink és összeállított BPEL üzleti folyamataink. (*17. ábra*)

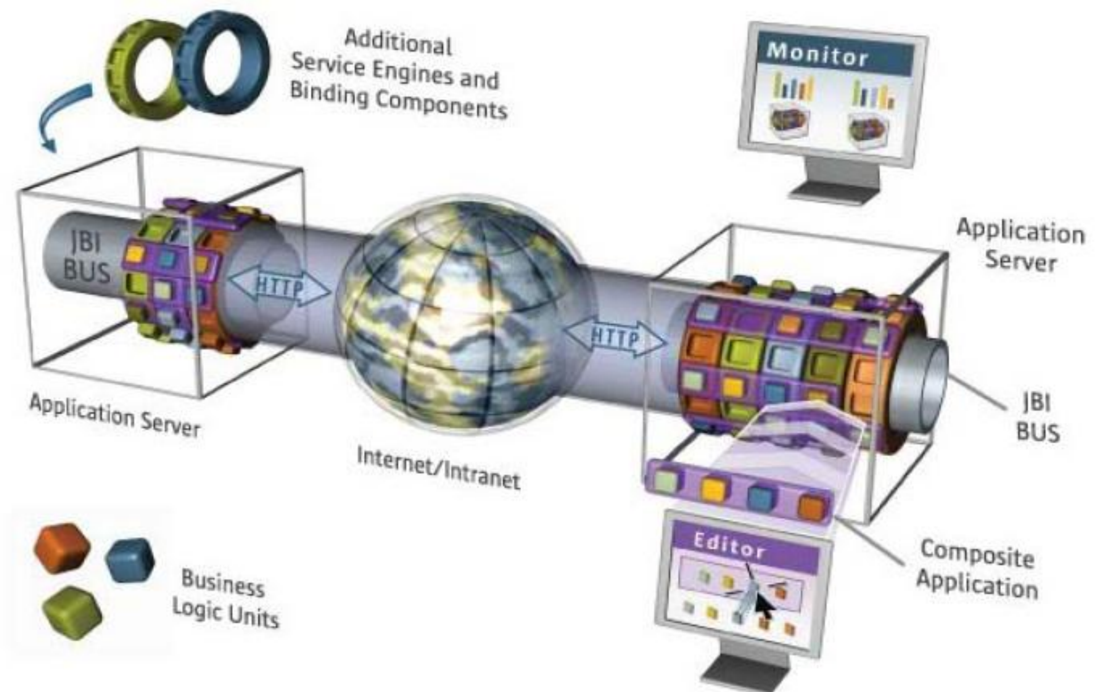


16. ábra Konténerek regisztrálják szolgáltatásaikat



17. ábra

Ezen információk után érkeztünk el ahhoz a ponthoz, amikor megismerhetjük az OpenESB projektet, mely valójában ezen alfejezet címadója volt. A projekt egy Vállalati Szolgáltatás Busz futtató környezetet (***Enterprise Service Bus – ESB***) implementál JBI alapokon. Ez lehetőséget nyújt webszolgáltatások könnyű integrációjára és lazán csatolt üzleti folyamat központú alkalmazások készítésére.



18. ábra OpenESB architektúra

A korábban bemutatott Glassfish alkalmazásszerverben integrálva van a JBI futtató környezet, mely fölé könnyen telepíthető az OpenESB projekt. A futtató környezetben regisztrálva van egy Java EE Service Engine, mely biztosítja az átjárást a Java EE alkalmazások és a JBI szolgáltatások között, így bármely Java EE alkalmazás archív állomány (ear/war/jar) elhelyezhető egy JBI alkalmazásban. Ez azt is jelenti, hogy biztosítva van a tranzakció kezelés, erőforrás pool kezelés, biztonsági megszorítások kezelése és az OO világ egy erőssége a kód újrafelhasználás (EJB/Web alkalmazás elérhető OpenESB komponensekből). Itt jegyezném meg, hogy a való életben leggyakrabban az alkalmazások mögött a terheléselosztás és nagy rendelkezésállás megvalósításhoz több szerver dolgozik a háttérben (clustering). Ekkor a cluster-ben elhelyezkedő minden egyes példánynak lesz saját JBI futtató környezete.

6. Technológiák összegzése

Mielőtt tovább haladnánk, röviden kiemelnék néhány információt emlékeztetőül, melyek ismerete hasznos lehet a második fejezetben:

- a SOA lehetőséget nyújt *rugalmas, magas szintű, agilis szoftver architektúra* kialakítására
- a szolgáltatások implementációja platform független
- szolgáltatások *könnyen készíthetők Java EE segítségével*: Statless Session Bean + Web Service Interface
- szolgáltatások készítése során használhatóak az *alkalmazásszerver által nyújtott rendszerszolgáltatások*
- szolgáltatások bármikor *újraimplementálhatóak más technológia segítségével*, mindaddig, amíg az interfész, melyen keresztül a kommunikáció történik nem változik
- elkészített szolgáltatásainkból BPEL segítségével szervezhetünk *üzleti folyamatokat*
- JBI infrastruktúra teszi lehetővé *a lazán csatolt alkalmazások* készítését OpenESB-vel

Most, hogy már láttuk egy magas absztrakciós nézetből, hogy hogyan tudunk üzleti folyamatokat leírni és ezeket milyen szoftver infrastruktúrára helyezhetjük el, jogosan merül fel a kérdés, hogy ez a valóságban hogyan történik? A 2.fejezetben ezt a kérdést válaszolom meg egy üzleti folyamatot modellezve és implementálva az itt bemutatott technológiákkal.

3. Üzleti folyamat tervezése és implementálása

A diplomamunka második felében egy üzleti folyamat implementálásának lépéseit mutatom be. A leírás feltételezi, hogy az olvasó rendelkezik a NetBeans IDE és a Java programozási nyelv alapszintű ismereteivel. Mivel a cél a szolgáltatásorientált szemlélet bemutatása, ezért a példában a tervezés és implementálás során nem használok bonyolult adatbázis sémákat vagy biztonsági megszorításokat. A következő fő pontokat szeretném szemléltetni a következő alfejezeteken keresztül:

- *szinkron szolgáltatáshívás*
- *aszinkron szolgáltatáshívás*
- *JMS/JDBC Binding*
- *Entity Bean-ek, SessionBean-ek használata*
- *BPEL folyamat tesztelése*

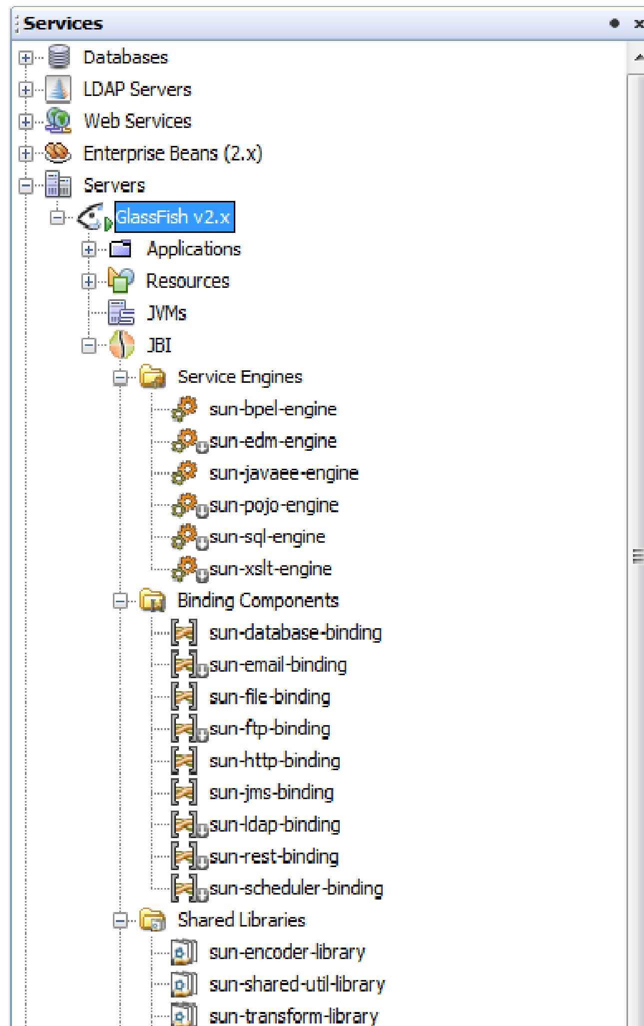
1. A szolgáltatás implementálásának technológia környezete

Az üzleti folyamat implementáláshoz a következő szoftverekre van szükség. (Az összes felhasznált szoftver OpenSource és ingyenesen letölthető.)

- *JDK (Java Development Kit) 6*
- *Glassfish v2.1.1 alkalmazásszerver*: a Java EE 5 platform referencia implementációja. Jelenlegi legfrissebb verzió a v3, mely már a Java EE 6-ot támogatja, de ehhez a verzióhoz még nem készült el a kompatibilis ESB verzió. (Project Fuji néven fejlesztik és a diplomamunka készítésének pillanatában Milestone 9 érhető csak el.) Letöltés: https://glassfish.dev.java.net/
- *NetBeans IDE 6.7.1*: integrált fejlesztői környezet a Java platformhoz. Különböző disztribúciókban érhető el, ha a Java verziót választjuk, akkor tartalmazni fogja a Glassfish alkalmazás szervert is, melynek telepítéséről a telepítő elején opcionálisan dönthetünk. Sajnos a 6.5-ös verzió volt az utolsó, mely tartalmazta a beépített SOA plugint, a későbbi verziókban (6.7, 6.7.1, 6.8, 6.9) ez nincs támogatva, külön kell beszerezni és telepíteni. Minden plusz munka ellenére mégis megéri a 6.7.1-es verziót használni, mert a feltelepített OpenESB pluginnal sokkal rugalmasabb fejlesztési lehetőségét biztosít, mint a korábbi verziók. A Downloads menüpont alatt mindig csak

a legfrisebb verzió van felkínálva letöltésre, de ezen az oldalon az Archive menüpontot választva, elérhetőek a régebbi verziók is.

Letöltés: http://netbeans.org/



19. ábra JBI Runtime megjelenése a Glassfish-ben

- *GlassfishESB v2.2*: Glassfish v2-höz tartozó Enterprise Service Bus. Feltelepíti a szükséges JBI runtime-ot és integrálja NetBeans-be a SOA projektek támogatását. Letöltés: https://open-esb.dev.java.net/
- Az adatok perzisztens tárolásához a Java SE részét képező Java DB –t használom, így nincs szükség külön adatbáziskezelő rendszer telepítésére

A részletes telepítési leírások megtalálhatóak a hivatalos oldalakon. Ajánlott telepítés:

- JDK 6

- NetBeans 6.7.1 Java Release verzió (Glassfisht kiválasztva)
- GlassfishESB v2.2

Az itt bemutatott projekt Windows 7 operációs rendszeren készült, ahol nem megfelelő jogosultságok miatt előfordulhat, hogy a GlassfishESB telepítője nem találja a JDK-t (Java Development Kit). Ezt a problémát a következő parancssori utasítással lehet kiküszöbölni:

```
glassfishesb-v2.2-installer-windows.exe –javahome "c:\Program
Files\Java\jdk1.6.0_18"
```

Ahol a --javahome után megadott elérési út a JDK könyvtárára kell mutasson.

Feltelepített konfiguráció helyességének ellenőrzése:

- NetBeans IDE-ben található **Services** fülön a Glassfish V2 alkalmazásszerver elindítása után megjelent a JBI alpont és tartalmazza a megfelelő Service Engine-eket és Binding Component-eket. (*19. ábra*)

2. Üzleti folyamat bemutatása az ügyfél szemszögéből

Képzeljük el azt a szituációt, hogy képzeletbeli cégünket megkereste a *Candy Company* édességgyártó cég, azzal az igénnyel, hogy szeretné automatizálni az eddig kézzel adminisztrált megrendeléseit, ezzel időt és pénzt takarítva meg.

Az Ügyfél szeretné, ha a folyamat teljesen automatizált lenne és kihasználná a két legnagyobb partnere által ajánlott hasonló jellegű lehetőségeket. Az egyik partner a *Candy Inventory*, mely a gyártó cég leányvállalataként raktárfunkciót lát el, mind a nyers alapanyag, mind az elkészült termék tárolására. Míg a másik partner a *Blue Credit Bank*, ahol az ügyfelek számlát nyithatnak és a megrendelt termékek ára automatikusan levonásra kerül egyenlegükből. A három cég bizonyos ügyfeleinek így extra szolgáltatásként fel tudja ajánlani, hogy ha a megrendelés során kiderül, hogy nincs az ügyfél számláján elegendő összeg a megrendelés teljesítéséhez, akkor az árufoglalás bizonyos ideig fenntartható legyen.

Ezenkívül az Ügyfél azt is szeretné, ha ez a szolgáltatás univerzális lenne oly módon, hogy fel tudják használni ugyanazt a szolgáltatást a hamarosan induló web áruházukban, valamint a korábbi ügyfeleknél már bevezetett **.NET**-ben elkészített alkalmazásokban is, anélkül, hogy a folyamat módosítása esetén módosításokat kellene végezni a másik kettőben is.

3. Üzleti folyamat elemzése

Az Ügyfél által ismertetett információk alapján leszűrhető a három fő kielégítésre szoruló igény:

- *platformfüggetlenség*
- *automatizálás*
- *biztonság*

Egy SOA architektúra kialakításával mindhárom igény nagyon könnyen kielégíthető. A következő részben megvizsgáljuk, hogy milyen szolgáltatásokra lesz szükségünk és ezekből hogyan állíthatunk össze egy szolgáltatás kompozíciót, mely kielégíti a fenti igényeket.

Megjegyzés: Mivel mind a három cég csak a képzelet szüleménye, ezért a fent említett partner cégek szolgáltatásai nincsenek elkészítve, ezért ezek tervezését és implementálását is áttekintjük.

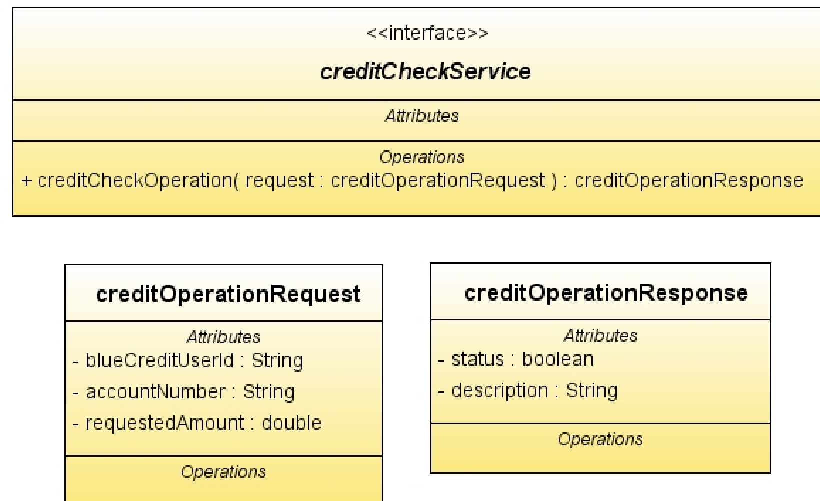
Megjegyzés 2: A tervezés és implementálás során a technológiai lehetőségek bemutatása a cél. Ezt figyelembe véve eltérések lesznek a példa folyamat és egy valóság világban modellezett folyamat között. Például a rendelkezésre álló összeg/árumennyiség ellenőrzése és levonása/lefoglalása a valós világban az elosztott rendszerek kontextusában nem két lépésben történik, mint a jelen példában.

Megjegyzés 3: A példa implementációjának egyszerűsítése miatt a következő feltételezéseket tesszük a tervezés során:

- *A Blue Credit Bank adatbázisa fel van töltve adatokkal és biztosít egy web alkalmazást, melyen keresztül lehet online accountot készíteni.*
- *A raktár műveletek során nem kezelünk ténylegesen mennyiség ellenőrző logikát, feltételezzük, hogy meg lesz a kívánt mennyiség, kivéve, ha valamilyen kivétel következik be.*
- *Ha a rendelkezésre álló összeg ellenőrzése során kiderül, hogy az Ügyfélnek nincs elég fedezete, akkor a rendelése nem törlődik. Ekkor nem tud még egyszer ugyanilyen rendelés azonosítóval rendelést végrehajtani. Feltételezzük, hogy létezik egy másik üzleti folyamat, mely lehetőséget ad fél óráig a szükséges összeg pótlására, ellenkező esetben törli a rendelést.*

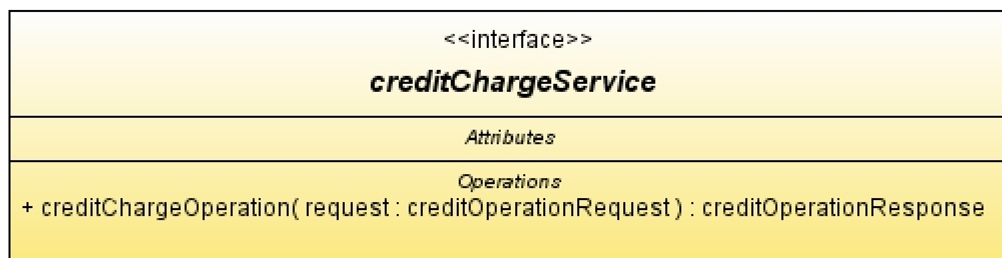
Először tekintsük át a Blue Credit Bank partner által kínált szolgáltatásokat. A bank két szolgáltatást ajánl fel ügyfelei részére:

- *creditCheckService*: ez a szolgáltatás leellenőrzi, hogy az adott ügyfél rendelkezik-e elegendő fedezettel a megrendelés kifizetéséhez. A szolgáltatás először ellenőrzi, hogy létezik-e adott azonosítóval felhasználó és a megadott számlaszám tényleg ahhoz a felhasználóhoz tartozik-e.



20. ábra creditCheckService interface és a hozzátartozó kérés/válasz modellek

- *creditChargeService*: ez a szolgáltatás végrehajtja a megrendelés kiegyenlítését, azaz levonja az ügyfél számlájáról az adott összeget.



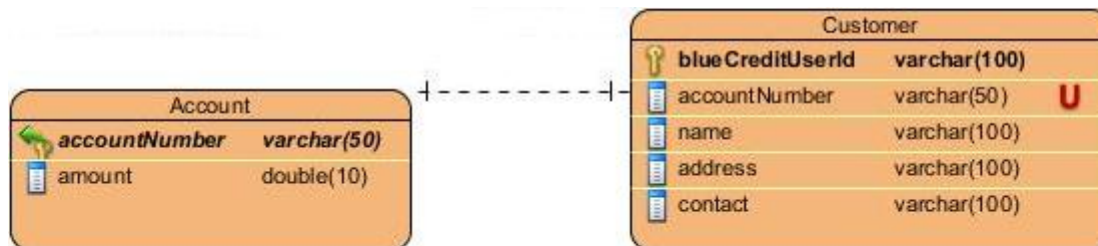
21. ábra creditChargeOperation interface

Mindkét szolgáltatás **hívása szinkron** módon fog történni, ezért nincs szükség semmilyen megkülönböztetésre sem a bemenő, sem a kijövő adatok között.

A szolgáltatások implementációja állapotmentes **Session Bean**-ekkel történik, melyek rendelkeznek egy WebService interfésszel, ami lehetővé teszi számunkra webszolgáltatásként történő elérésüket.

A perzisztens adatok a szolgáltatásokban **Entity Bean**-ekként jelennek meg ORM leképezés alkalmazásával.

A perzisztens adatok tárolását egy **Java DB** adatbázisban végezzük el. Az adatbázis neve Blue Credit Bank Database lesz, míg a séma a bluecredit. A sémában két tábla található, melyek a következőek:



22. ábra bluecredit adatbázis séma

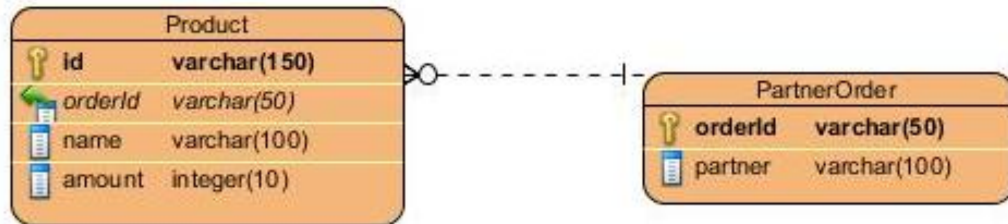
Másik partnerünk a Candy Inventory raktár, mely a következő két szolgáltatást biztosítja:

- *inventoryCheckService*: leellenőrzi, hogy van a raktárban a rendelés teljesítéséhez elegendő áru mennyiség. A valóságban ez a szolgáltatás a megrendelésnek megfelelő mennyiségeket beszúrja a táblába az adott rendelésazonosítóval, mintegy foglalás gyanánt. Ebből következik, hogy mindig lesz elegendő áru, kivéve, ha valamilyen kivétel következik. Például félbeszakadt rendelés újbóli küldése integritási problémákat okozna a táblában, ezért erre nincs lehetőség.
- *fulfillOrderService*: levonja a raktár árukészletéből a lefoglalt árumennyiséget, ami a valóságban az adatok táblából történő törlésének felel meg.

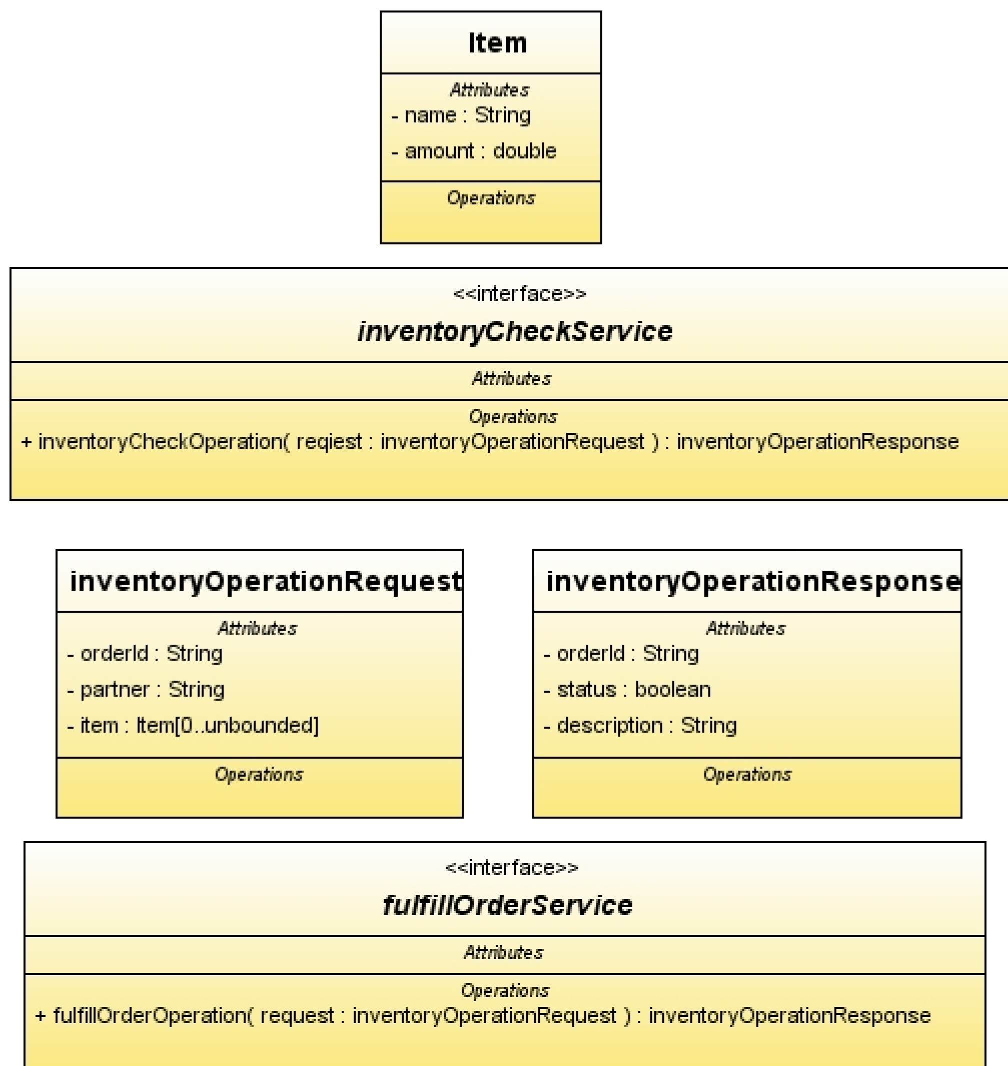
Az inventoryCheckService szolgáltatás **hívása aszinkron** módon fog történni, ezért, mint a 24. ábrán is látható szükség lesz plusz paraméterre a kijövő adatok között. Ez a paraméter az orderId lesz, melynek a program logika szempontjából nincs jelentősége, csak a futtató környezet értelmezi. Erre a paraméterre fogunk **korrelációt** definiálni, ami alapján el környezet eldönti, hogy az aszinkron módon visszaérkező eredmény épp melyik konkurens módon futó üzleti folyamathoz tartozik.

A szolgáltatások implementálása és a perzisztens adatok megjelenése az előzőhöz hasonló módon történik.

A perzisztens adatok a Candy Inventory Database nevű adatbázis candy nevű sémájában kerülnek tárolásra, mely séma a következő táblákat tartalmazza:

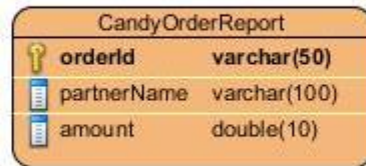


23. ábra candy séma



24. ábra Candy Inventory szolgáltatásai

Magának a Candy Company cégnek is lesz egy saját privát adatbázisa Candy Order Report néven, ahol ez az adatbázis, valamint a séma neve is. A séma csak egyetlen táblát tartalmaz statisztikai adatok rögzítése céljából. A tábla elérése egy **JDBC Binding**-on keresztül fog történni, mely az adatbázis kapcsolatokat egy **Connection Pooling** mechanizmussal transzparens módon elfedi előlünk.



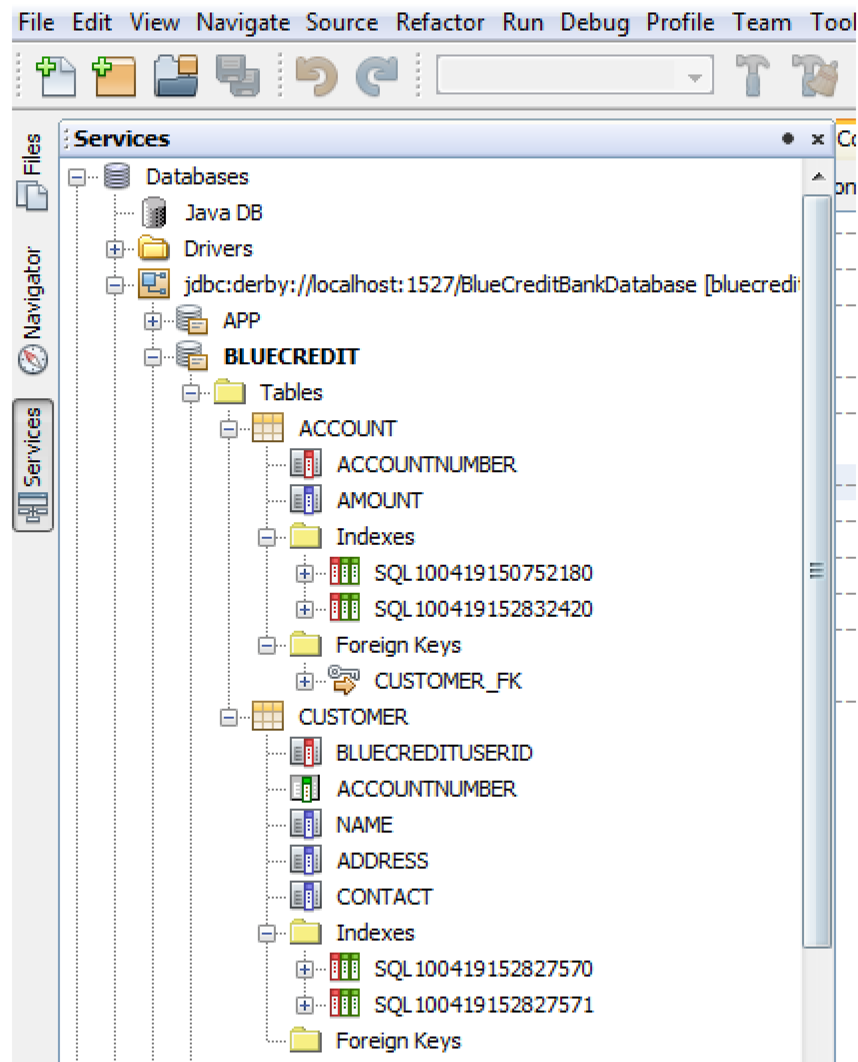
25. ábra Statisztikai kimutatásokhoz szükséges adatokat tároló tábla

Az BPEL folyamat részletes tervezése most nem kerül bemutatásra, a hozzá tartozó Interaction Overview diagram megtekinthető a Függelék részben.

4. Blue Credit Bank partner szolgáltatásainak implementálása

1. Első lépésként készítsük el a szolgáltatásokhoz tartozó adatbázis sémákat. Mint már említésre került a korábbi fejezetben erre a Java DB adatbázisban kerül sor. Az a séma és táblák létrehozása más adatbáziskezelő rendszerekhez hasonlóan DDL utasításokkal történik. Sikeres létrehozás után a *26. ábrá*nak megfelelő képet kell látnunk.
2. Hozzunk létre egy EJB projektet, melyben a WSDL leírók, XML séma leírók, Entity Bean-ek és Stateless Session Bean-ek fognak elhelyezkedni. Érdemes a későbbi könnyebb megkülönböztethetőség érdekében ejb prefix-el kezdeni a projekt nevét. Például: ejbCandyInventoryProject.
3. Hozzunk létre három csomagot a következő nevekkel:
    a. contract: ebben a csomagban kerülnek tárolásra a leíró fájlok (WSDL,XML).
    b. entites: itt helyezzük majd el a perzisztens adatokat a programban reprezentáló Entity implementációkat.
    c. ws: ebben a csomagban pedig maguk a webszolgáltatás implementációk kerülnek elhelyezésre.
4. Generáljuk le a létrehozott adatbázistáblák alapján az Entity Bean-eket az entites csomagba. Ennek működéséhez szükséges a NetBeans-ben előre definiálni az

adatbázis elérést a *Services* fülön. Ezután a *File/New/Persistance/Entity Classes from Database* menüponttal generálhatóak a megfelelő java fájlok.



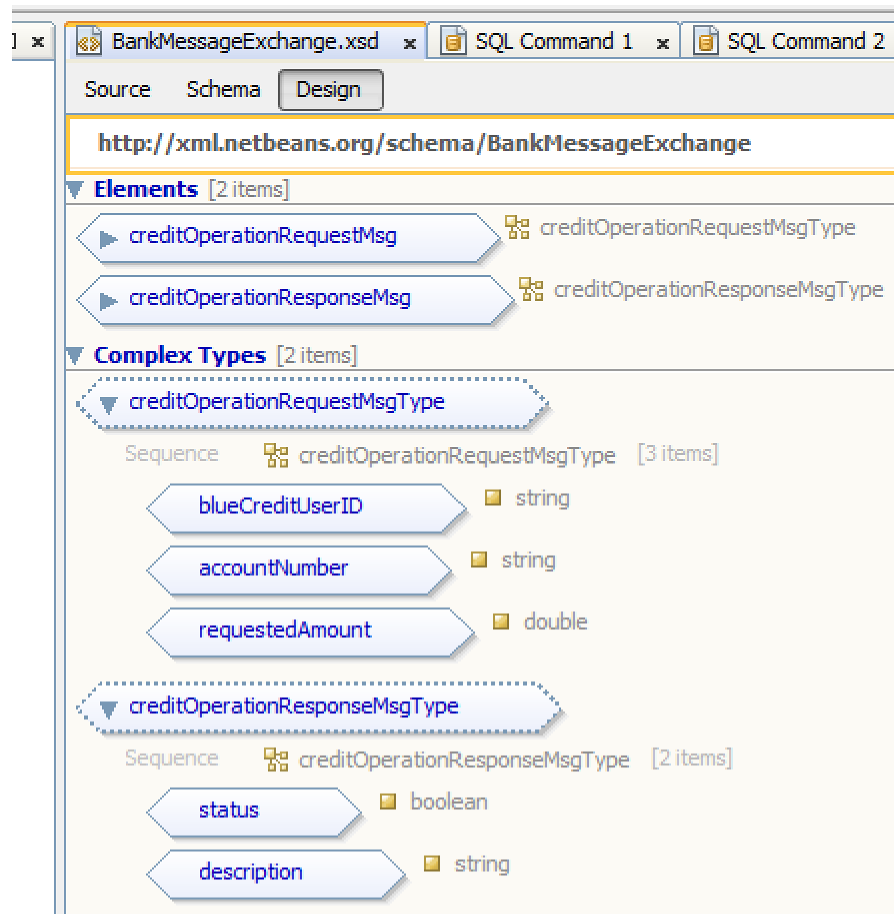
26. ábra A bluecredit adatbázis séma

Válasszuk ki a megfelelő adatbázis kapcsolatot és a generálásban résztvevő táblákat. Az Entity Classes fülön válasszuk a Create Persistance Unit opciót. Minden mást hagyjunk változatlanul. A generálás hatására létrejön két új java osztály az entites csomagban az *Account* és a *Customer*. Az osztályok kódja a JPA API szabvány alapján került generálásra, mely tárgyalása meghaladja jelen diplomamunka kereteit. Egyetlen érdekességként azt emelném ki, hogy az ORM leképezési információk EJB 3.0-tól annotációkkal vannak megadva, és minden XML információ opcionális lett.

```
@Entity
@Table(name = "ACCOUNT")
@NamedQueries({@NamedQuery(name = "Account.findAll", query = "SELECT
a FROM Account a"), @NamedQuery(name =
"Account.findByAccountnumber", query = "SELECT a FROM Account a
WHERE a.accountnumber = :accountnumber"), @NamedQuery(name =
"Account.findByAmount", query = "SELECT a FROM Account a WHERE
a.amount = :amount")})
public class Account implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "ACCOUNTNUMBER")
    private String accountnumber;
    @Basic(optional = false)
    @Column(name = "AMOUNT")
    private double amount;
    @JoinColumn(name = "ACCOUNTNUMBER", referencedColumnName =
"ACCOUNTNUMBER", insertable = false, updatable = false)
    @OneToOne(optional = false)
    private Customer customer;
…
```

27. ábra Részlet az Account entity kódjából

5. Mint a korábbi fejezetekben említettem a webszolgáltatások kommunikációjának alapjai az XML üzenetek. Készítsük el a Blue Credit Bank szolgáltatásai által feldolgozott üzeneteket leíró XML sémát a *contract* csomagban. Ezt a *New/XML/XML Schema* opcióval érhetjük el. Az IDE biztosít számunkra egy vizuális szerkesztőt, ahol drag&drop módszerrel összeállíthatjuk a megfelelő leírót, ami alapján legenerálja a szükséges kódot. A nekünk szükséges BankMessageExchange séma a *28. ábrán* látható. Megfigyelhető, hogy a sémában megjelenő elemek megfelelnek az előző fejezetben *creditOperationRequest* és *creditOperationResponse* UML diagramokkal leírtaknak.

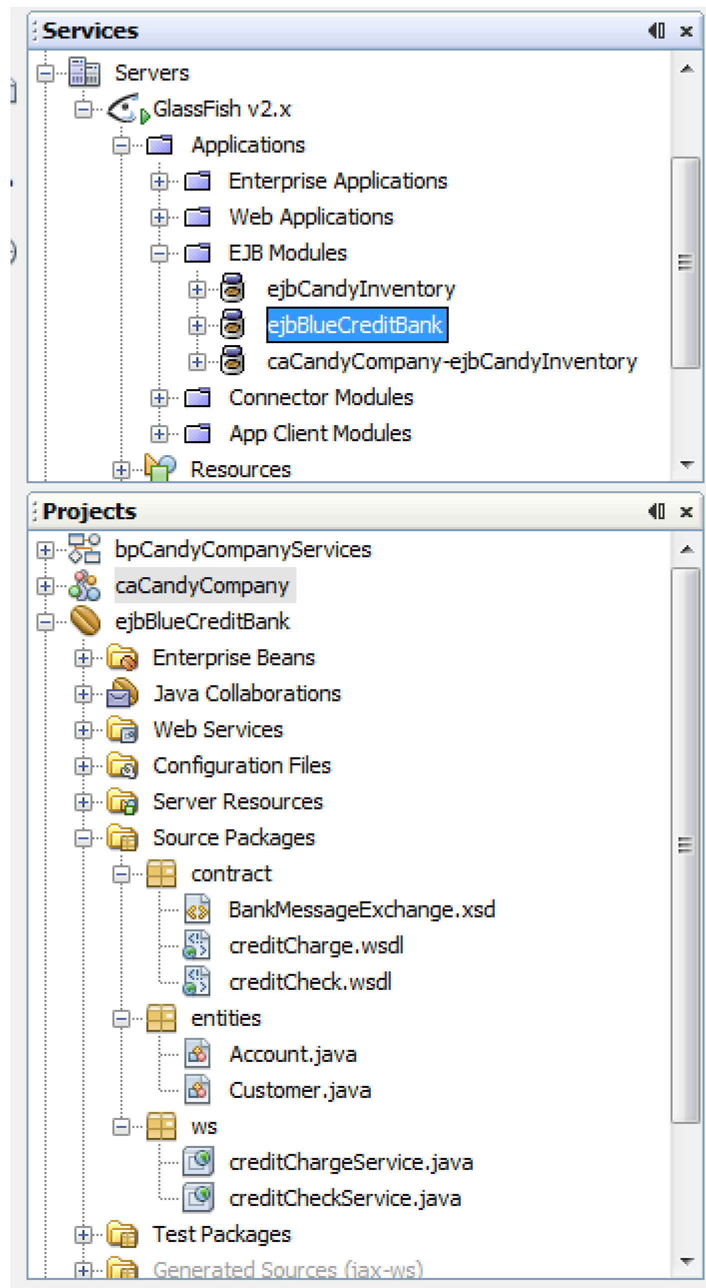


28. ábra BankMessageExchange.xsd XML séma

6. Ha elkészültünk a sémaleírókkal definiálhatjuk magukat a webszolgáltatás leírókat. Ezek szintén a *contract* csomagba kerülnek. Ezt a *New/XML/WSDL Document* opció segítségével tehetjük meg. Definiáljuk az előző fejezetben megadott interfészeket creditCheck és creditCharge WSDL leíróként.
    a. *Name and Location* oldalon a *WSDL Type*-nál az *Abstract WSDL* legyen kiválasztva. Ez azt jelenti, hogy még nem adjuk meg a konkrét leképezést, vagyis azt, hogy a kommunikáció fizikailag milyen formában fog történni.
    b. *Abstract Configuration* oldalon megadhatjuk a logikai kommunikációs portot és a műveletet, melyet a webszolgáltatás kínálni fog. Az Operation Type-nál adható meg, hogy szinkron vagy aszinkron kommunikációban lesz használva a szolgáltatás. Jelen szolgáltatásokat szinkron kommunikációban használjuk, ezért a default Request/Response a számunkra megfelelő.

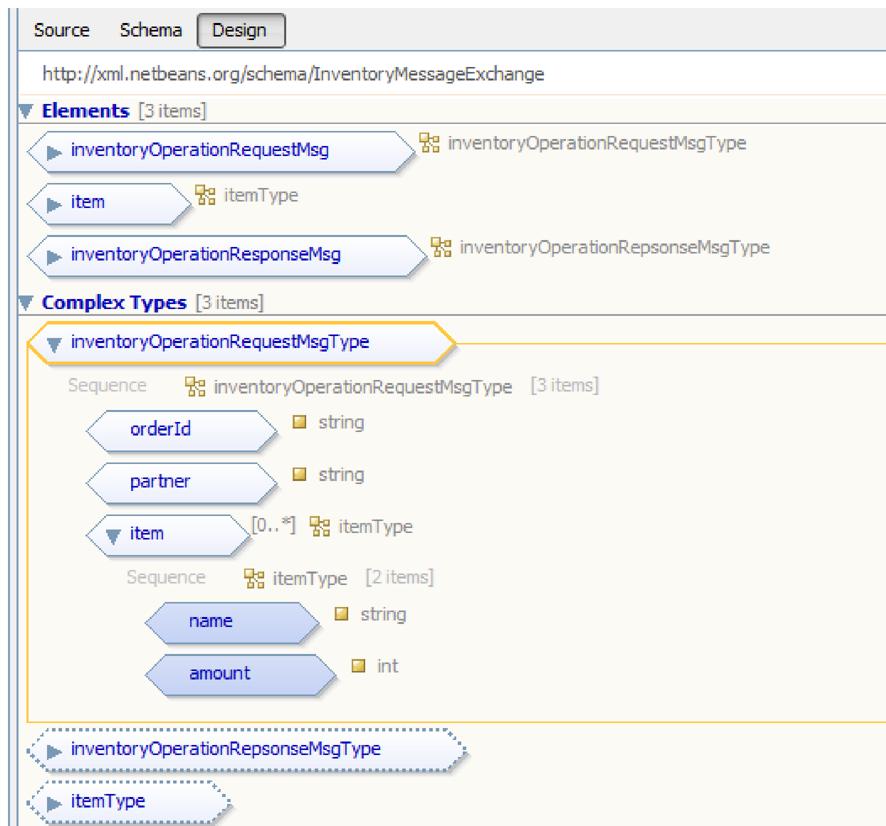
Ezen az oldalon az Input és Output változók típusánál válasszuk ki az XML sémánkban definiált *creditOperationRequestMsg* és *creditOperationResponseMsg* elemeket.

7. Ha megadtuk a WSDL leírókat generáljuk le a webszolgáltatás csonkokat a *New/Web Services/Web Service from WSDL* opcióval a *ws* csomagba. Szolgáltatás leíróknak válasszuk az előző pontban létrehozott két fájlt.



29. ábra Deploy-olt EJB modul és az elkészült projekt

8. A szolgáltatások implementációja itt nem kerül teljes tárgyalásra, kommentezve megtekinthetőek a mellékelt cd-n. Itt két érdekességre hívnám fel a figyelmet:
   a. az egyik az osztály előtt álló ***@WebService*** annotáció, amivel jelezzük, hogy a ***@Stateless*** annotációval ellátott Session Bean elérhető webszolgáltatásként.
   b. a másik, hogy a definiált műveletünk paramétere és visszatérési típusa rendre `org.netbeans.xml.schema.bankmessageexchange.CreditOperationResponseMsgType`, `org.netbeans.xml.schema.bankmessageexchange.CreditOperationRequestMsgType`. Ezek az XML séma alapján az IDE által generált **JavaBeans** konvencióval létrejött osztályok, melyek az XML típusainkat reprezentálják. A kódban később példányosítás, adatelérés céljából ugyanúgy használhatóak, mint bármely más osztály. Megtekinthetők a *Generated Sources* csomagban.
9. Utolsó lépésként fordítsuk a projektet a Build opcióval, majd a Deploy opcióval deployoljuk az alkalmazásszerverbe. (*29. ábra*)



30. ábra item számossága 0..*

5. A Candy Inventory partner szolgáltatásainak implementálása

A partnerszolgáltatások implementálása az előzőekben leírt lépésekkel teljesen megegyezően történik. Egyetlen dologra szeretném felhívni a figyelmet. A *30. ábrán* látható XML séma definícióban az inventoryOperationRequestMsgType-ben az item tagnak a számosságánál beállítottuk, hogy nulla vagy határozatlan elem jelenhet meg belőle. Ennek később az üzleti folyamat modellezése során lesz jelentősége, így erre majd akkor térek ki.

Ettől a különbségtől eltekintve mind a projekt struktúrájának felépítése, mind a fájlok létrehozása és modul deployolása egyezik az előzőekben leírtakkal.

6. Üzleti folyamat implementálása

A partnerszolgáltatások létrehozása és publikálás (deploy) után felhasználhatjuk őket üzleti folyamatok implementálásához. Egy üzleti folyamat implementálásához a SOA kategóriában lévő projekt típusok közül a következő kettőt fogjuk használni:

- *BPEL module*: ebben a modulban helyezhetjük el a *BPEL*-el (*Business Process Execution Language*) leírt üzleti folyamatainkat, a hivatkozott szolgáltatások referenciáit, WSDL és XML leírókat.
- *Composite Application*: ez a projekt magában foglalhat egy vagy több *BPEL* modult és több más *JBI* (*Java Business Integration*) típusú modult. Megteremthetjük a modulok között a kapcsolatokat, egy egységként deployolhatjuk és tesztelhetjük a folyamatokat.

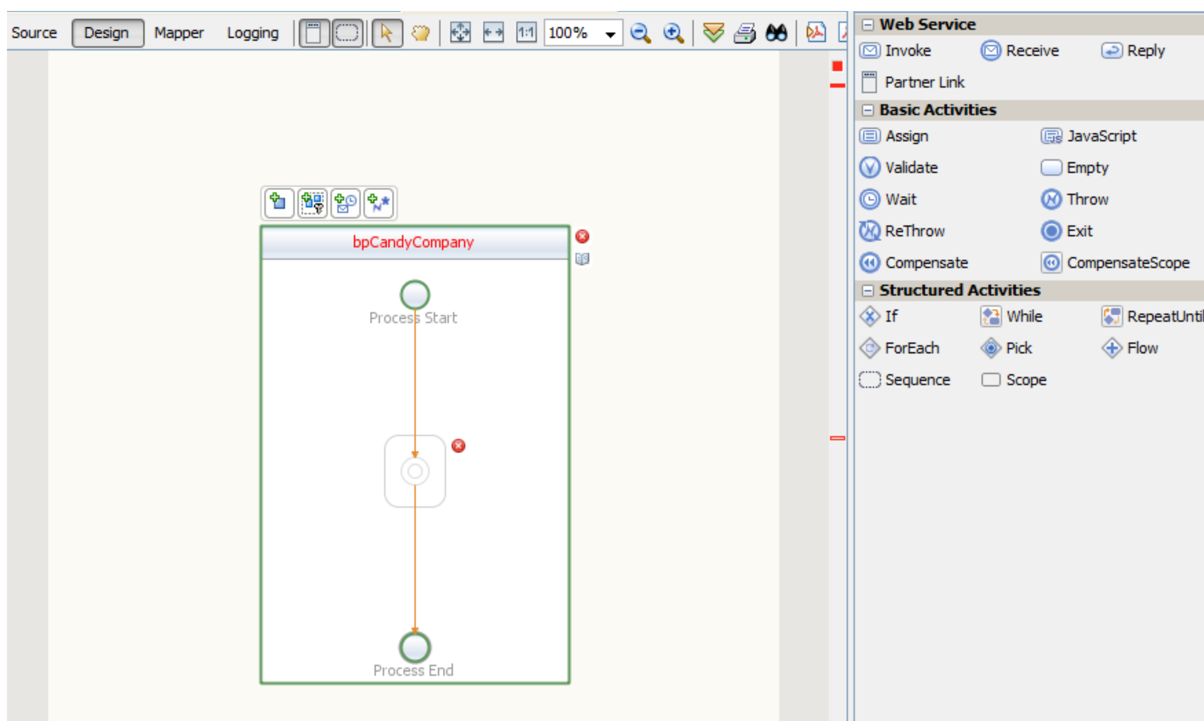
Hozzunk létre egy új BPEL module-t bpCandyCompanyServices néven. A modul teljesen üres, egyetlen BPEL folyamat vázat tartalmaz. *(31. ábra*)

Megnyitva a bpel fájlt megnyílik a *Design* nézet, ahol drag&drop módszerrel összeállítható az üzleti folyamat. A folyamat bal oldalára kerülnek majd azok a partnerszolgáltatások, melyek **hívó szerepkörben** kommunikálnak az üzleti folyamattal, míg a jobb oldalra azok a szolgáltatások, melyek **hívott szerepkörben** működnek együtt. A szerkesztő jobb oldalán, az eszköztárban megfigyelhetőek az Open ESB című fejezetben leírt **alap** BPEL **tevékenységek** vizuális reprezentációi.

A bal felső sarokban választható milyen nézetben szeretnénk szerkeszteni a folyamatunkat:

- *Source*: megmutatja a folyamat forráskódját, amely maga a BPEL-el elkészített XML fájl lesz.

- *Design*: drag&drop módszerrel vizuálisan szerkeszthetővé teszi a folyamatunkat, míg a háttérben legenerálja a szükséges XML kódot.
- *Mapper*: ez a nézet csak akkor választható, ha előtte kiválasztottunk egy olyan elemet a folyamatból, melyhez szükség van bizonyos leképezésekre. Például: változókhoz történő érték hozzárendelés (assign), feltétel kifejezés if tevékenységben.
- *Logging*: ebben a nézetben bizonyos tevékenységeken logolás definiálható.



31. ábra BPEL szerkesztő

A korábbi projektben már elkészített XML sémákra az *External XML Schema element(s)* opcióval hivatkozhatunk, mely beimportálja a szükséges sémákat.

Az elkészített partnerszolgáltatásokat elérhetővé kell tennünk a projektből felhasználás előtt. Ezt az *External WSDL document(s)* opcióval tehetjük meg (*itt a WSDL fájl URL elérését adjuk meg, mert más esetekben nem várt mellékhatások fordulhatnak elő*). Az összes felhasználni kívánt WSDL leírót külső hivatkozással elérhetővé kell tennünk.

Megjegyzés: *A nem SOA projektekben működik a WSDL fájl áthúzása egyik projektből a másikba, de a SOA projektekben ezt egy BUG megakadályozza. Megkerülő megoldás, a fenti opció használata az URL elérés megadásával.*

Miután legeneráltattuk a referenciákat a WSDL leírókra, hozzuk létre az üzleti folyamathoz tartozó XML sémát és WSDL leírót. Emlékezzünk, hogy az üzleti folyamat a hívó szempontjából nem különbözik egy webszolgáltatástól.

CandyOrderMessageExchange XML sémaleíró forrása:

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

      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://xml.netbeans.org/schema/CandyOrderMessageExchange"

xmlns:tns="http://xml.netbeans.org/schema/CandyOrderMessageExchange"
          elementFormDefault="qualified">
          <xsd:complexType name="CandyOrderRequestMsgType">
              <xsd:sequence>
                  <xsd:element name="orderId" type="xsd:string"/>
                  <xsd:element name="blueCreditUserId" type="xsd:string"/>
                  <xsd:element name="partnerName" type="xsd:string"/>
                  <xsd:element name="accountNumber" type="xsd:string"/>
                  <xsd:element name="amount" type="xsd:double"/>
                  <xsd:element name="items" type="tns:itemType"
maxOccurs="unbounded" form="qualified"/>
              </xsd:sequence>
          </xsd:complexType>
          <xsd:complexType name="CandyOrderResponseMsgType">
              <xsd:sequence>
                  <xsd:element name="orderId" type="xsd:string"/>
                  <xsd:element name="status" type="xsd:boolean"/>
                  <xsd:element name="description" type="xsd:string"/>
              </xsd:sequence>
          </xsd:complexType>
          <xsd:complexType name="itemType">
              <xsd:sequence>
                  <xsd:element name="name" type="xsd:string"
form="qualified"/>
                  <xsd:element name="amount" type="xsd:int"
form="qualified"/>
              </xsd:sequence>
          </xsd:complexType>
          <xsd:element name="CandyOrderRequestMsg"
type="tns:CandyOrderRequestMsgType"/>
          <xsd:element name="CandyOrderRepsonseMsg"
type="tns:CandyOrderResponseMsgType"/>
          <xsd:element name="item" type="tns:itemType"/>
      </xsd:schema>
```

A legenerált és újonnan létrehozott WSDL leírók használhatóak partnerekként a BPEL folyamat felépítése során. A WSDL leírókat be lehet húzni a folyamat bal illetve jobb oldalára, az előbbiekben ismertetett szempontok szerint.

A példánkban két BPEL folyamatot modellezünk:

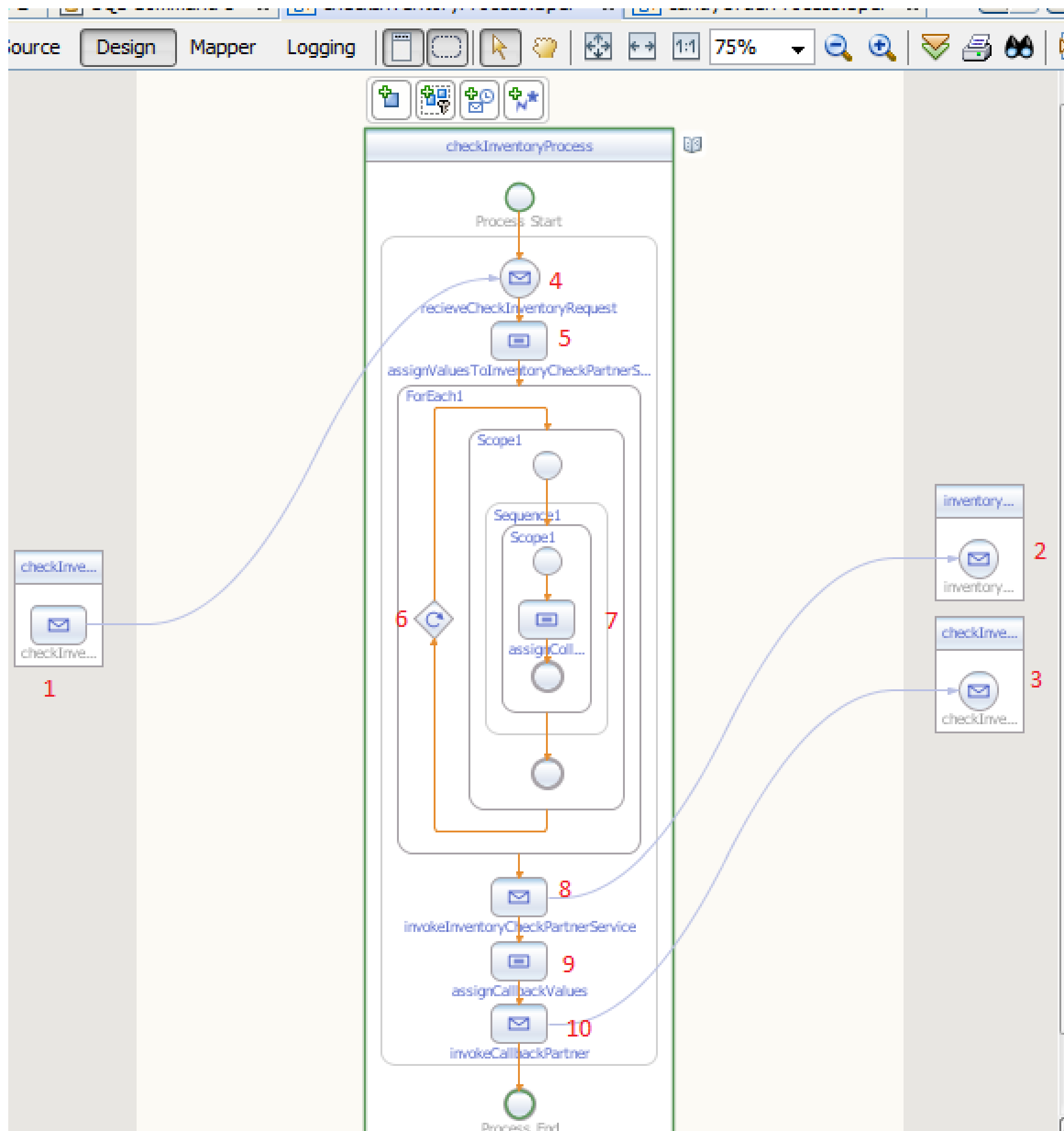
- *checkInventoryProcess*: aszinkron üzleti folyamat. Meghívja a Candy Inventory raktár ellenőrző szolgáltatását és annak visszatérése után aszinkron módon visszaadja az üzenetet a hívó folyamatnak (mely a CandyOrderProcess lesz).
- *candyOrderProcess*: szinkron üzleti folyamat. Aszinkron módon kommunikál a checkInventoryProcess folyamattal, valamint szinkron módon a többi partnerszolgáltatással.

Készítsük el a checnInventoryProcess üzleti folyamatot. Első lépésként készítsünk hozzá két darab WSDL leírót a következő paraméterekkel:

- Név: *checkInventoryPartner*
  Operation Type: **One-Way Operation** (*jelezve, hogy egyirányú, azaz aszinkron lesz a kommunikáció*)
  Input (Message Type): inventoryOperationRequestMsg
  Feladat: a hívó partner ezen keresztül indítja el a folyamatot
- Név: *checkInventoryCallbackPartner*
  Operation Type: **One-Way Operation** (*jelezve, hogy egyirányú, azaz aszinkron lesz a kommunikáció*)
  Input (Message Type): inventoryOperationResponseMsg
  Feladat: a folyamat ezen keresztül értesíti a hívó partnert a folyamat eredményéről

Vegyük a checkInventoryProcess *32. ábrán* látható implementációját. A megjelölt pontok rendre a következők:

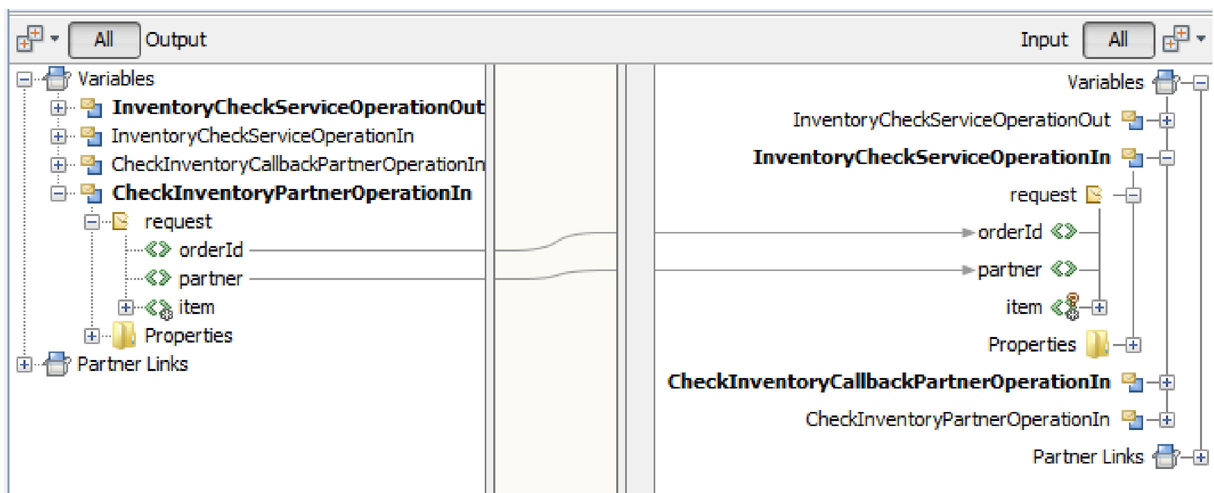
1. *checkInventoryPartner WSDL leíró*: a hívó ezen az interfészen keresztül hívja meg az adott BPEL processt.
2. *inventoryCheckServicePartner WSDL leíró*: az adott BPEL process ezen az interfészen keresztül kommunikál a Java EE-ben implementált raktárt ellenőrző szolgáltatással. Ezt az Open ESB fejezetben már említett *JavaEE Service Engine* teszi lehetővé. (Az implementáció lépései a korábbi fejezetben megtalálhatóak.)
3. *checkInventoryCallbackPartner WSDL leíró*: az adott folyamat ezen az interfészen keresztül juttatja vissza az aszinkron módon meghatározott választ a hívó folyamatnak.



32. ábra checkInventoryProcess üzleti folyamat implementációja

4. *Recieve activity*: a folyamat fogad egy XML üzenetet, ezzel inicializálva és elindítva a feldolgozást.
5. *Assign activity*: általánosan elmondható, hogy minden szinkron híváshoz létre kell hoznunk egy **input és egy output változót**, amin keresztül a szolgáltatás kommunikál a hívó környezettel. (Aszinkron hívás esetén értelemszerűen csak input változó van.) Az input változókhoz a hívás tényleges végrehajtása előtt értékeket kell rendelnünk, amit az Assign activity-vel tehetünk meg. Ez valójában

változók közötti értékmásolás lesz, vagy konstansok/kifejezések változókhoz rendelése. Az output változók értékeit pedig a hívás végrehajtása után újabb másik input változókhoz rendelhetjük vagy visszaadhatjuk a hívó folyamatnak. A leképezés a Mapper szerkesztő segítségével történik, ahogy a *33. ábrán* is látható. Az ábrán megfigyelhető, hogy egy adattag, az *item*, kimarad. Ennek oka, mint ahogy korábban is említésre került 0..unbounded számosságú elemek speciális kezelése.



33. ábra Mapper használata az Assign tevékenység összerendeléseinek megadásához

6. *Foreach activity*: az item kollekció minden elemén végiglépked és hozzárendeli a inventorCheckServiceOperation bemenő paraméterében lévő kollekcióhoz. Ha nem lenne for ciklus, akkor csak az utolsó elem lenne elérhető.

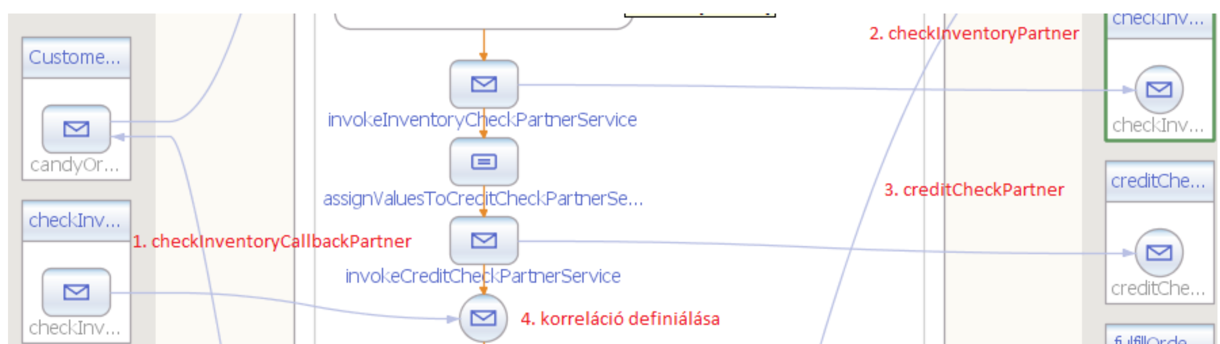
   Megjegyzés: Előfordulhat, hogy a for ciklussal sem működik rendesen az összerendelés, mely egy BUG-nak köszönhető. Ilyenkor érdemes egy Sequence-be rakni, ami megoldja a problémát.
7. *Assign activity*: végrehajtja a kollekció elemek összerendelését a for ciklus magjában.
8. *Invoke activity*: egy másik webszolgáltatás/üzleti folyamat invoke tevékenységgel hívható meg. A 8. pontban lévő tevékenység egy szinkron webszolgáltatás hívás, mely meghívja a Java EE-ben implementált raktár ellenőrző szolgáltatást.
9. *Assign activity*: az előzőekben visszaadott értékét hozzárendeli a Callback interfész input változójához, mely visszaszolgáltatja a hívó felé az értéket.

10. *Invoke activity*: meghívjuk a Callback szolgáltatást (mely definíció szerint egy One-Way Operation), ezért nem várunk vissza semmilyen értéket, ezzel elérve az üzleti folyamat végét.

A candyOrderProcess elkészítése teljesen hasonló dallamra történik (partnerszolgáltatások elérhetővé tétele, XML sémaleíró elkészítése, WSDL leíró elkészítése), ezért csak az eltéréseket, illetve eddig nem említett lehetőségeket emelném ki ennek a folyamatnak az implementációjából. *A teljes implementáció megtalálható a mellékletben*.

Az első, amit ki szeretnék emelni az előbb elkészített üzleti folyamat aszinkron meghívása. A hívás nem fog eltérni egy szinkron szolgáltatás hívásától, tehát ugyanúgy, ahogy az előző esetben itt is invoke activity-vel fog történni. A különbség, ahogy a *34. ábrán* is látható, hogy ezután a végrehajtás tovább folytatódik és nem várunk eredményére. Az eredményt majd egy Recieve activity-vel kaphatjuk vissza.

- a 2. pontban üzenetet küldünk az előzőekben definiált checkInventoryPartner WSDL leírón keresztül, mely, mint említettünk az implementálásnál, elindítja a folyamatot.
- az üzenet küldése után rögtön továbblépünk, nem várunk eredményt. A következő lépésben történik egy szinkron szolgáltatás a szükséges fedezet ellenőrzésére a Blue Credit Bankhoz. Ezzel párhuzamosan a másik üzleti folyamatban történik a raktárkészlet ellenőrzése.

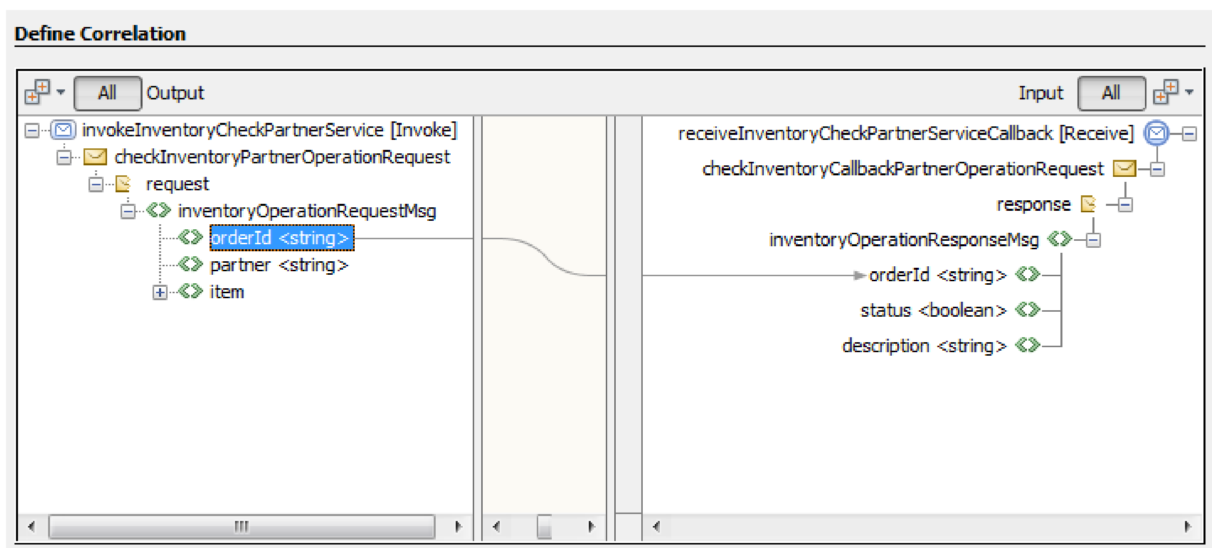


34. ábra Aszinkron és szinkron hívás

- a következő lépés egy Recieve activity, mely a checkInventoryCallbackPartner WSDL leírón keresztül vár bejövő üzenetet a partnertől. *Emlékezzünk, hogy az aszinkron raktárellenőrző BPEL folyamat a végrehajtásának befejeződése előtt ezen az interfészen keresztül küld üzenetet*. Ez a tevékenység addig **blokkolódik**, amíg meg nem kapja a folytatáshoz szükséges megfelelő üzenetet. Honnan tudhatja, hogy melyik

a megfelelő üzenet? Korábbi fejezetben említettük, hogy az *inventoryOperationRepsonseMsgType* típusban fel kell vennünk egy orderId-t, amit mi explicit módon nem fogunk használni, csak a futtató környezet. Ennek az Id-nak minden egyes üzenet esetén egyedinek kell lennie, mert ez alapján dönti el a futtató környezet, hogy melyik aszinkron szolgáltatáshívásra érkezett válasz. *Ne felejtsük el, hogy **elosztott környezetben** vagyunk, ezért akárhány üzleti folyamat hajtódhat végre konkurensen, melyeket meg kell különböztetnünk.*

A korreláció tehát nem lesz más, mint az üzenetek megfelelő folyamathoz rendelése ez alapján az egyedi azonosító alapján. A definiálásakor csak annyit kell megadnunk, hogy az aszinkron hívással elküldött üzenetben melyik tag tartalmazza azt az egyedi azonosítót, amely alapján a visszakapott üzenetekről eldönthető melyik folyamathoz tartoznak. (*35. ábra*)



35. ábra Korreláció definiálása

További érdekességként szolgálhat még az erőforrások használata a BPEL folyamatból. A mellékletben található implementációban a következő két binding található:

- *JMS Binding*: az Open MQ Glassfihben található implementációjában létrehozunk egy üzenetsort, melybe elküldjük a megrendelés adatait a szállító cégnek. Ebből a sorból később a szállító cég kiveszi az üzenetünket és feldolgozza, ezzel akár egy másik BPEL folyamatot elindítva a saját rendszerében, mely vezérli és nyomon követi a szállítást.

- *JDBC Binding*: bármely adatbázishoz kipublikálhatunk egy webszolgáltatás interfészt, melyen keresztül a hagyományos *CRUD* (*Create-Read-Update-Delete*) funkciók transzparansen elérhetőek. Ennek segítségével lehetőségünk van például adatok perzisztálására a BPEL folyamatból.

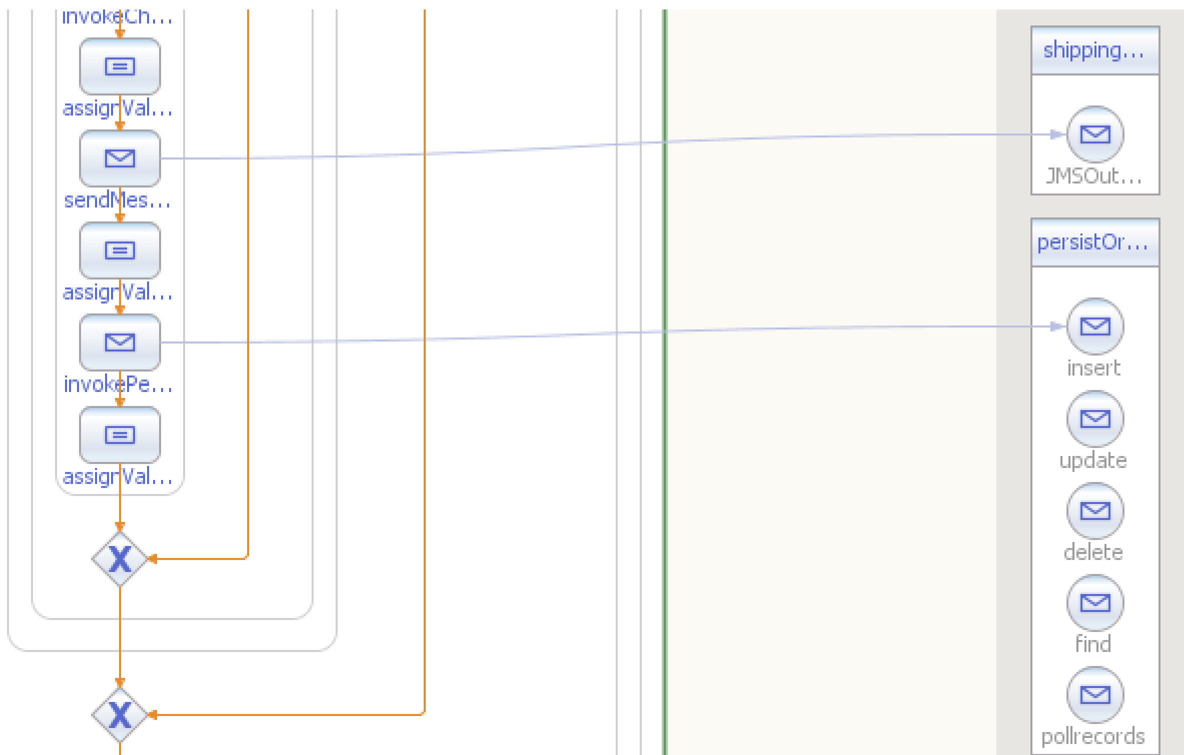
JDBC Binding létrehozásának lépései a következőek:

1. El kell készítenünk egy szolgáltatás WSDL leíróját. Ez az eddig megismert módon fog történni, a következő apró módosítással: a WSDL Type-nál ne az Abstract WSDL Document-t használjuk, mint eddig, hanem válasszuk a **Concrete WSDL Document** opciót. Ennek a kiválasztása azt jelenti, hogy a dokumentum létrehozásakor megmondjuk, hogy fizikailag melyik szolgáltatásra fog leképeződni a beérkező kérés. A felugró két új opció közül válasszuk a következő opciókat.
    a. *Binding*: **JMS**
    b. *Type*: **Send**
2. A konfiguráció beállításokat töltsük ki a *36. ábrának* megfelelően. Látható, hogy konkrétan megadjuk, hogy hol található az üzenet sor, ahová az üzeneteket szeretnénk küldeni, bejelentkezési információkkal együtt, valamint ahogy eddig is, definiálnunk kell az üzenet típusát, mely a candyOrderRequestMsg-nek megfelelő XML üzenet lesz. (A megadott értékek az Open MQ default elérési adati. A default Password a guest.)

Request Configuration
JMS Connection
Connection URL: mq://localhost:7676
User Name: guest
Password: •••••
Payload Processing
Message Type: xml
XSD Element/Type: ns:CandyOrderRequestMsg
encoded type:

36. ábra Open MQ Binding konfigurálása

3. Az összes többi beállítás maradhat a megadott értéken. Ezek a beállítások a tranzakció kezelésre, valamint az üzenet élettartamára vonatkoznak.
4. Ezután a WSDL leírót csakúgy, mint az összes többi szolgáltatás esetében, behúzhatjuk a BPEL tervező nézetbe partnerszolgáltatásnak. Meghívása nem különbözik az eddig leírtaktól. (*37. ábra*)



37. ábra JMS Binding és JDBC Binding hívás

JDBC Binding létrehozása és meghívása teljesen hasonló módon történik néhány szükséges extra feltétellel:

- A NetBeans IDE-n belül a Service-s fülön létre kell hoznunk előtte egy adatbázis kapcsolatot, mely az elérni kívánt adatbázishoz csatlakozik. A tervező ebből a kapcsolatból fogja kiolvasni az elérési információit.
- Létre kell hoznunk egy **Connection Pool**t az alkalmazásszerverben, melynek JNDI nevét utolsó lépésben át kell adnunk a varázslónak. Az alkalmazás futása során az alkalmazásszerver transzparens módon ebből a poolból fog elkérni egy szabad adatbázis kapcsolatot, melyen keresztül végre tudja hajtani a műveleteket. Ezt a beállítást az alkalmazásszerverhez tartozó Admin console-ban tehetjük meg a

Resources/JDBC menüpont alatt. A Connection Poolt hozzuk létre az alábbi kapcsolódási paraméterekkel, majd egy **JDBC Resource** létrehozása után a fenti lépésekhez hasonlóan elkészíthetjük a hozzá tartozó WSDL leírót.

Additional Properties (5)

Add Property | Delete Properties

Name	Value
User	candyorderreport
DatabaseName	CandyOrderReport
Password	candyorderreport
ServerName	localhost
PortNumber	1527

38. ábra CandyOrderReport adatbázishoz kapcsolódó pool paraméterei

7. Üzleti folyamatok telepítése és tesztelése

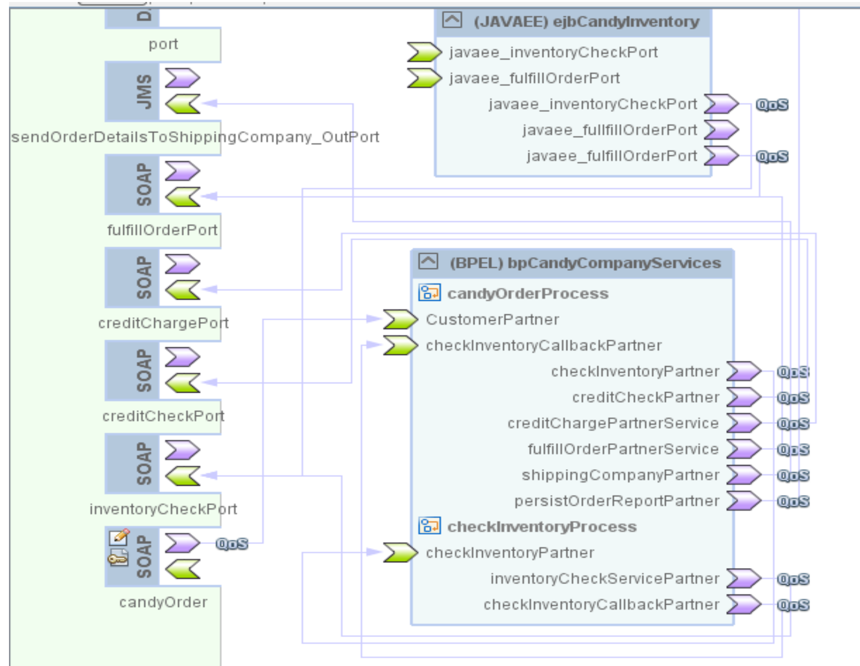
Az elkészült BPEL és JBI modulokat egy egységes és szabványos csomagszerkezetbe kell elhelyezni, mielőtt telepítésre kerülnének az alkalmazásszerverbe. Egy ilyen telepítési egység létrehozására lesz alkalmas a már említett *New Project/SOA/**Composite Application*** projekt.

Az egy egységként kezelni kívánt projekteket nagyon egyszerűen hozzáadhatjuk a projekthez: *jobb klikk/Add JBI module…*

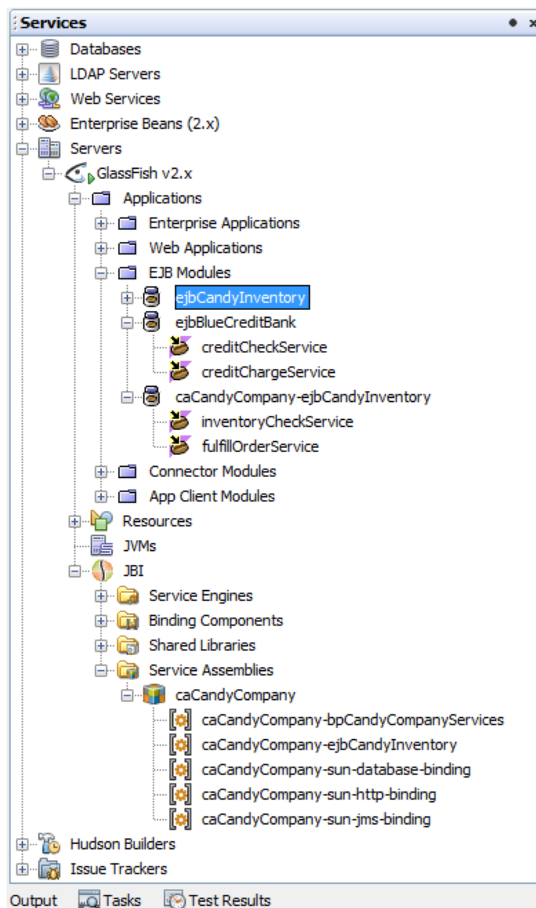
Az összes JBI module hozzáadása után egyszerűen lefordítjuk a projektet (*Build*), aminek hatására legenerálja az IDE WSDL leírók által definiált kapcsolatokat és szolgáltatási egységeket, melyek meg is jelennek a design nézetben. A *39. ábrán* az adott üzleti folyamatunkat egységbe fogó projekt fordítás utáni design nézetének egy részlete látható.

Az ábrán megfigyelhető, hogy megjelenik név szerint az összes általunk WSDL leíróval definiált szolgáltatás és egymáshoz való viszonyuk is. A bal felső sarokban látható a korábban létrehozott kimenet JMS Binding is, melyen keresztül üzenetet küldünk a szállító cégnek.

Fordítás után telepíthetjük az alkalmazást az alkalmazásszerverbe (*Deploy*). Ettől a pillanattól kezdve, ha mindent helyesen csináltunk, elérhető vált partnereink számára az üzleti folyamatunk. Ahogy a 40. ábrán látható, az EJB modulok között megjelentek a kipublikált szolgáltatásaink, a JBI modul részben pedig a telepített caCandyCompany **service assembly**. Ha lenyitjuk, akkor láthatóvá válnak a modul által használt, Open ESB fejezetben ismertetett Service Engine-ek és Binding-ok.



39. ábra Candy Company szolgáltatásai és köztük lévő kapcsolatok



40. ábra EJB modulok és Service Assembly

Utolsó lépésként nézzük meg hogyan tesztelhetjük szolgáltatásunkat. A Composite Application Test könyvtárában hozhatunk létre új teszteseteket. Itt megadhatjuk XML üzenet formájában mi legyen az input, majd az első futtatás után az adott kimenetet később használhatjuk regressziós teszteléshez. Tegyük fel, hogy a Blue Credit Bank adatbázisában megtalálható az alábbi két rekord:

Customer

BCId	AccountNumber	Name	Address	Contact
rathonyit	11223344-5667788	Ráthonyi Tamás	Debrecen	rathonyit@bluecredit.com

Account

AccountNumber	Amount
11223344-5667788	12345678

Ebben az esetben a következő input XML üzenetre az alábbi Output XML üzenetet kell kapnunk.

Input:

```
   <soapenv:Envelope
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:can="http://xml.netbeans.org/schema/CandyOrderMessageExchange"
>
     <soapenv:Body>
       <can:CandyOrderRequestMsg>
         <can:orderId>order1234567810</can:orderId>
         <can:blueCreditUserId>rathonyit</can:blueCreditUserId>
         <can:partnerName>Ráhonyi Tamás</can:partnerName>
         <can:accountNumber>11223344-55667788</can:accountNumber>
         <can:amount>400</can:amount>
         <!--1 or more repetitions:-->
         <can:items>
           <can:name>Rágógumi</can:name>
           <can:amount>600</can:amount>
         </can:items>
         <can:items>
           <can:name>Csoki</can:name>
           <can:amount>10</can:amount>
         </can:items>
```

```
      </can:CandyOrderRequestMsg>
    </soapenv:Body>
  </soapenv:Envelope>
```

Output:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <CandyOrderRepsonseMsg
xmlns="http://xml.netbeans.org/schema/CandyOrderMessageExchange"
xmlns:msgns="http://j2ee.netbeans.org/wsdl/bpCandyCompanyServices/ca
ndyOrder">
      <orderId>order1234567810</orderId>
      <status>true</status>
      <description/>
    </CandyOrderRepsonseMsg>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ezzel a példával nemcsak a fejezet, de a második rész végére is értünk. Ebben a részben megnéztük hogyan lehet az első részben megismert fogalmakat gyakorlatban kipróbálni, milyen eszközök és lehetőségek állnak a fejlesztők rendelkezésére.

Megnéztük hogyan kell XML és WSDL leírót készíteni, Session Bean-t és Entity Bean-t használni, hogyan történhet BPEL folyamatok között szinkron, aszinkron kommunikáció, valamint hogyan lehet erőforrásokat (JMS, JDBC) használni a fejlesztés során. A következő utolsó fejezetben szeretnék egy összegzést írni az itt tapasztaltakról és megfogalmazni a technológiáról alkotott véleményem.

4. Összegzés

Elérkezve a diplomamunka végéhez néhány gondolatban szeretném összegezni tapasztalataimat és a fentebb bemutatott technológiákról kialakult véleményemet.

Úgy gondolom, hogy a webszolgáltatások előnyei egyelőre megkérdőjelezhetetlenek. Bár még napjainkban nem ezek képezik a fő irányvonalat, de egyre több nagy cég (Google, Facebook, Amazon, Yahoo) lát fantáziát a felhasználási lehetőségekben és publikál ki saját szolgáltatásokat partnerek számára. Ha megmarad ez a növekedési tendencia és támogatottság a nagy cégek által, akkor véleményem szerint ez később oda vezethet, hogy az üzleti világban is egyre nagyobb számú lesz a SOA projektek száma, mely megkönnyíti mind az IT-ben dolgozók, mind az üzleti életben dolgozók életét.

A Java CAPS, mint open source projektekből készült kereskedelmi szoftver, technológiailag számomra egységesnek és kiforrottnak tűnt. Az Oracle-Sun tranzakció után sorsa egyelőre kérdéses, például az Open SSO projekt támogatása már meg is szűnt.

Számos pozitív tulajdonság mellett két negatívat említenék meg:

- az egyik, hogy nincs (vagy nagyon nehezen található) hozzá akár összességében, akár a projekteket külön-külön nézve minden témakört lefedő, egységes dokumentáció vagy leírás. Bevezető jellegű leírás sok helyen található, de bonyolultabb szerkezetek modellezése nehézkesebb lehet.
- másik számomra negatív dolog, hogy a NetBeans 6.5-től megszűnt a hivatalos támogatottsága az IDE-ben. Külön pluginként letölthető és telepíthető, de ez a szétválás több BUG-ot eredményezett, melyek megkerülő megoldásai esetenként csak több órás keresés után kerülnek elő.

Összességében úgy gondolom, hogy a technológia már itt van, készen áll a használatra, de szükség lenne az eszközök egységes, teljes körű támogatására és egy mindent lefedő dokumentációra. Véleményem szerint nem lenne jó döntés a projektet támogatásának megszüntetése és a technológiai egység megbontása annak tükrében, hogy a SOA egyre inkább teret hódít napjainkban, sokkal inkább több erőforrást kellene fordítani a hibalehetőségek kiküszöbölésére és az eszközök, dokumentációk, technológia finomítására, továbbfejlesztésére.

Persze ettől a néhány negatívumtól eltekintve, mindenkinek ajánlanám, hogy ismerkedjen meg a SOA elvekkel és megvalósíthatósági lehetőségekkel, mert elképzelhető, hogy ez lesz a *jövő technológiája…*

Irodalomjegyzék

[1] The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/

[2] Open DS wiki, https://www.opends.org/wiki/page/Main

[3] Open ESB wiki, http://wiki.open-esb.java.net/Wiki.jsp

[4] Open ESB intorduction, http://wiki.open-esb.java.net/Wiki.jsp?page=OpenESBIntroductionTutorial

[5] Loan Processing Application tutorial, http://netbeans.org/kb/61/soa/loanprocessing.html

[6] Sangh Shin – JavaPassion.com http://www.javapassion.com/soaprogramming/

[7] Java CAPS wiki, http://wikis.sun.com/display/JavaCAPS/Home

[8] Debu Panda, Reza Rahman, Derek Lane – EJB 3 in action

[9] BPEL developer guide, http://netbeans.org/kb/61/soa/bpel-guide-overview.html

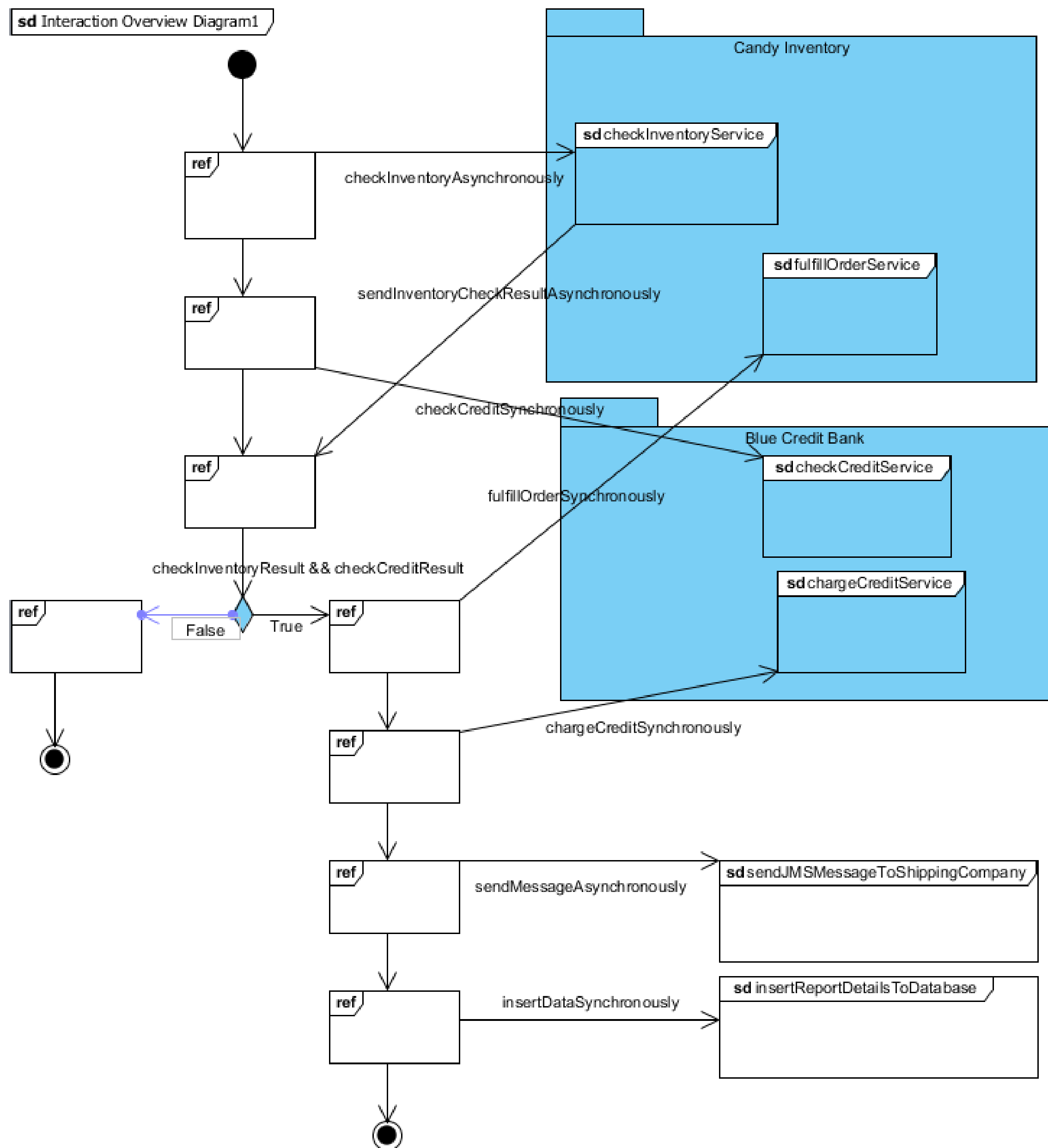
[10] HUP wiki, http://wiki.hup.hu/index.php/LDAP

[11] Wikipedia (LDAP), http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[12] http://www.gracion.com/server/whatldap.html

[13] Wikipedia (SSO), http://en.wikipedia.org/wiki/Single_sign-on

Függelék



41. ábra BPEL folyamat modellezése Interaction Overview diagrammal