Írta:
BILICKI VILMOS

PROGRAMRENDSZEREK FEJLESZTÉSE

Egyetemi tananyag

2011

LEKTORÁLTA: Dr. Charaf Hassan, Budapest Műszaki és Gazdaságtudományi Egyetem Automatizálási és Alkalmazott Informatikai Tanszék

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-492-1

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Benkő Márta

KULCSSZAVAK:

köztesréteg, JEE, CDI, EJB, Servlet, ESB, SOA, SCA/SDO, BPEL4WS

ÖSSZEFOGLALÁS:

A jegyzet egy széles körű áttekintést nyújt az elosztott rendszerek alkalmazása mögött álló motivációkról, az elosztott viselkedésnél követhető paradigmákról és az elosztottságot elfedő szoftver rétegről. Ezen széles repertoárból a klaszter és SOA paradigmákon alapuló alkalmazás és alkalmazásrendszer fejlesztés képezi a jegyzet fajsúlyos részét. Itt a JEE platform szolgáltatásai és a klasszikus 3 rétegű architektúra mentén vizsgáljuk meg az egyes rétegek szerepét, a fellépő problémákat illetve megoldásokat. Túllépve az egyes alkalmazások határait az ESB platform megismerése segítségével valósítjuk meg a SOA koncepció mentén elkészülő elosztott rendszerünket.

1. Tartalomjegyzék

I. rész

Bevezető

2. Bevezető

A szoftverfejlesztés mint tudomány már több mint 30 éves múltra tekint vissza. Ezen életút alatt a módszertanok, megközelítések, eszköztárak számos paradigmaváltáson esetek át. A paradigmaváltások mozgatórugója a legtöbb esetben a szoftverfejlesztők produktivitásának a növelése volt. A különböző paradigmák nem feltétlenül szekvenciálisan, egyértelműen sorbarendezhetően követik egymást, hanem inkább egy fa struktúrában ábrázolhatóak. Adott típusú problémára, adott kontextusban a különböző paradigmák nem nyújtanak azonos teljesítményt sem a szoftver performanciájában, sem a fejlesztők produktivitásában. A szoftverfejlesztőknek így ismernie kell a különböző paradigmák korlátait ahhoz, hogy adott kontextusban legjobban alkalmazható paradigma mentén valósítsák meg a szoftvert.

A szoftverekkel szemben támasztott nem funkcionális – más néven rendszerszintű – követelmények szintén fejlődtek, egyre nagyobb kihívásokat jelentenek. A Julian Browne által kiválasztott 15 követelményből néhány legfontosabb nem funkcionális követelmény:

- *Rendelkezésre állás*: azon idő, amely alatt a komponens képes kielégíteni a funkcionális követelményekben meghatározott funkciókat azon időhöz viszonyítva, amikor ezt nem tudja megtenni
- *Kapacitás/skálázhatóság*: a komponens a terhelés növekedésével is képes ellátni a funkcionális követelményekben megfogalmazott feladatokat
- *Párhuzamosság*: a komponens több, egymással párhuzamos terhelés hatására is képes ellátni a funkcionális követelményekben megfogalmazott feladatokat
- *Fejleszthetőség*: a komponens új funkcionális követelményeket is el tud látni viszonylag szerény fejlesztési ráfordítással
- *Együttműködési képesség*: a komponens képes környezetével és a szükséges komponensekkel aránytalan plusz terhelés okozása nélkül is együttműködni
- *Késleltetés*: a komponens az adott funkcionális követelményben szereplő szolgáltatását adott terhelés mellett adott időtartamon belül kiszolgálja
- *Karbantarthatóság*: a komponensnek támogatnia kell az adott szervezetben definiált karbantartási feladatokat (pl.: biztonsági mentés)
- *Menedzselhetőség*: a komponensnek támogatnia kell a megfelelő konfigurálhatósági szintet
- *Monitorozhatóság*: a komponensnek monitorozhatónak kell lennie
- *Visszaállíthatóság*: hibás adat esetén a komponensnek támogatnia kell a megfelelő állapot visszaállítását
- *Biztonság*: a komponensnek a helyi biztonsági előírásoknak megfelelően kell működnie
- *Konzisztencia*: a komponens az adatot konzisztens állapotban tartja

A produktivitás növelését tehát ezen követelmények betartása mellett kell elérni. Emiatt nem véletlen, hogy nem csak programozási nyelvekről és azok képességeiről kell beszél-

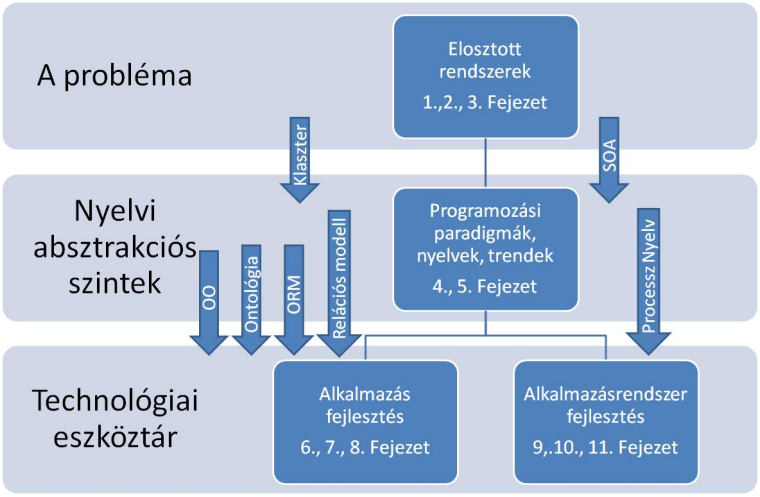
nünk, hanem egyes nyelvekhez köthető vagy független futtató- és fejlesztői környezetekről, amelyek magas szintű szolgáltatásokkal lehetővé teszik ezen nem funkcionális követelmények hatékony, produktív kielégítését. Ezen keretrendszerekből, megközelítésmódokból azonban igen sok van és nehéz közöttük eligazodni. (pl. Melyik a jobb a Spring vagy a JEE?) Ahhoz, hogy egy ilyen kérdésre válaszolni tudjunk (ha egyáltalán lehet), meg kell ismernünk azokat az absztrakt képességeket amelyeket ezek a fejlesztő/futtató környezetek nyújtanak a fejlesztőknek. (pl. Inversion Of Control vagy bijekció) Ezen képességek megértéséhez viszont az alapoktól célszerű kiindulni. Esetünkben ez az úgynevezett *menedzselt kód*. Ilyen a Java és a .NET környezet. A menedzselt kód elsősorban a memóriakezelés kiszervezését jelenti: a programozónak nem kell a memória allokációjával és felszabadításával foglalkoznia, megteszi ezt helyette a szemétgyűjtő. Ezen alapszolgáltatáson túl azonban jóval komplexebb szolgáltatások is megjelennek a menedzselt környezetben (a továbbiakban a Java környezetet értjük menedzselt környezet alatt, a legtöbb esetben a megállapításaink igazak a .NET világra is). Egy ilyen például az alkalmazás-tartományok kezelése, amikor a menedzselt környezet átveszi az operációs rendszertől az alkalmazások elkülönítésének a feladatát, és megvalósítja azt a szálak szintjén. Az alkalmazás-tartományokon belül pedig szerepkör alapú hozzáférés-vezérlés lehetséges, (amennyiben ezt külön beállítjuk és használjuk) amely segítségével adott kódrészek csak adott szerepkör tulajdonosai futtathatnak (Java esetén ezt a JAAS API segítségével tudjuk elérni). Ezen megoldások bár általánosak, nehezen illeszthetőek bele közvetlenül egy vállalat AAA (Azonosítás, Jogosultságkezelés, Naplózás – Authentication, Authorization, Accounting) környezetébe. Egy másik fontos alapszolgáltatás a párhuzamosság támogatása. A Java nyelv és a legtöbb JVM alkalmas párhuzamos futtatásra és az alap problémák (zárolás, megszakíthatatlan kódrész, stb.) kezelésére, bár ez az operációs rendszer képességeitől is függ. Ezzel az eszköztárral azonban a programozóra hárul a különböző versenyhelyzetek (holtpontok) kezelése, a párhuzamos programozás teljes problémaköre. Ez az egyik fő mozgatórugó azon magasabb szintű menedzselt környezetek (ún. vállalati/enterprise környezetek) használata mögött, amelyek elrejtik a párhuzamosság problémáit a programozó elől. A tranzakciók támogatása teljes mértékben hiányzik az alap JVM képességeiből, ezt is különböző vállalati környezetek biztosítják. Az alap nyelv és környezet által biztosított objektum-orientált adat- és algoritmusabsztrakció adott méretig megfelelő, azonban nem teszi lehetővé modulok megfelelő megfogalmazását, ezen modulok megfelelő menedzsmentjét. A sima procedurális alapú logikamegvalósítást szintén nem célszerű alkalmazni magasabb absztrakciós szinten. Az alkalmazás menedzselhetősége, tesztelhetősége szintén gyenge lábakon áll. Ezen és még számos probléma (pl. skálázhatóság, web-alkalmazás, integrálhatóság, stb.) miatt az egyszerű menedzselt környezet nem igazán nyújtja a kívánatos produktivitást azon szoftverfejlsztők részére, akik a fenti problémákkal szembesülnek. Természetesen minden probléma orvosolható sima JVM fölött is, mindent le lehet programozni, a kérdés a produktivitás.

Mint ahogyan már írtuk a megoldást a különböző vállalati környezetek jelentik. Ezen környezetek a menedzselt alapkörnyezet (itt JVM) szolgáltatásaira építve nyújtanak magasabb szintű szolgáltatásokat. Gyakran nevezik ezeket a környezeteket konténereknek. Két nagy csoportot különböztetünk meg: az első csoportba az alkalmazásokat támogató konténerek tartoznak. Ezekre az jellemző, hogy egy alkalmazás működését támogatják akár fizikailag elosztott környezetben is (ilyenek pl. a Spring, az EJB, a web konténerek, részben az OSGi). A másik az alkalmazás-rendszereket támogató konténerek amelyek több alkalmazás együttműködését támogatják (ilyen pl. az ESB, az SCA, részben az OSGi).

A Java nyelvi környezet egyik meghatározó szabványosító mechanizmusa a Java Community Process (JCP), mely JSR-ek (Java Specification Request) segítségével teszi egységessé a különböző területek problémáit kezelő Java API-k és ezeken keresztül a különböző konténerek interfészeit, telepítés-leíróit és az egyéb kapcsolódó, nem szorosan az implementációval összefüggő kérdéseket. Ezen mechanizmuson túl jelentős befolyása van még a Spring mögött álló VMware cégnek és az OASIS, valamint az OSGi testületeknek is. Jegyzetünkben igyekszünk az adott problémánál és az arra adható megoldásoknál megemlíteni a kapcsolódó szabványokat is.

A terjedelmi korlátok nem teszik lehetővé, hogy egy-egy problémakört/megoldást olyan részletességgel ismertessünk, amely az adott megoldás hatékony alkalmazásához szükséges lenne. Ehelyett jegyzetünk célja az, hogy bemutassa azokat a problémákat és megoldásokat, amelyekkel a programozó szembesül, amikor egy úgynevezett vállalati alkalmazást fejleszt.

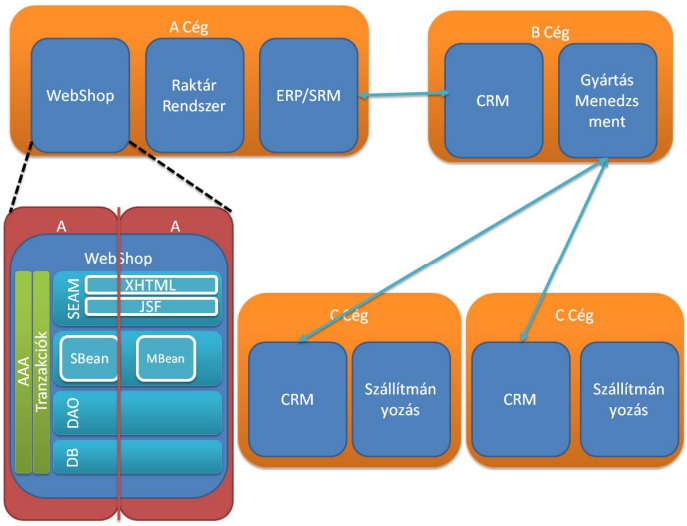
Az első három fejezet egy széleskörű áttekintést nyújt az elosztott rendszerek alkalmazása mögött álló motivációkról, az elosztott viselkedésnél követhető paradigmákról és az elosztottságot elfedő szoftver rétegről. Ezen széles repertoárból a klaszter és SOA alapú elosztott rendszerekre koncentrál a jegyzet. A következő rész (4., 5. fejezet) a nyelvi paradigmák oldaláról közelíti meg a témát és felvillantja azokat a lehetőségeket, melyek adott paradigmák alkalmazásával válnak elérhetővé.



1. ábra

A jegyzet háromnegyedét kitevő 3. és 4. rész pedig a technológia szintjére ereszkedve mutatja be azokat a napjainkban használt megoldásokat, keretrendszereket, melyek az alkalmazások és alkalmazásrendszerek fejlesztése során leggyakrabban előkerülnek.

A 2. ábra egy tipikus elosztott rendszert szemléltet, mely jól demonstrálja a jegyzet megközelítését is. Be szeretnénk egyrészt mutatni a skálázható klaszter alapú alkalmazások fejlesztését (az ábrán a webshop), másrészt a jegyzet célja a SOA alapú rendszerek elemeinek bemutatása is. Ez a képen a különböző meglévő szoftverrendszerek együtteseként látható.



2. ábra

A következő fejezetben az elosztott rendszerek absztrakt problémáival és az azok elrejtésével foglalkozó köztesréteggel ismerkedünk meg.

3. Elosztott rendszerek

Mielőtt belekezdenénk az elosztott rendszerek problematikájának és a lehetséges megoldásoknak, megoldási módszereknek az ismertetésébe, áttekintjük a lehetséges alkalmazás architektúrákat, alkalmazás architektúra paradigmákat. Ezen architektúra mentén tudjuk majd meghatározni azokat a rétegeket, amelyek az elosztottságban szerepet kaphatnak.

A legegyszerűbb, de a legtöbb esetben leginkább kerülendő architektúra a **monolit**. Ezen architektúrát nem szabdalják rétegek, az absztrakció is leginkább a procedúrák (eljárások) szintjén valósul meg. Tipikusan egy gépen futtatott egy darab futtható állományt jelent. A monolit és az egyszerű procedurális paradigma után egy minőségi ugrást jelentett az **objektum-orientált** paradigma úgy a tervezés, mind a nyelvek területén. Ez esetben az absztrakciót az objektumok és a közöttük definiálható tartalmazási, származási viszonyok képviselik. Az objektum-orientált tervezést két egymástól eltérő nézeteket valló iskola határozza meg: a skandináv iskola az adatok reprezentálására koncentrál először (Domain Model), és csak ezután foglalkozik az adatok viselkedésével, folyamatokkal. Az amerikai iskola leginkább a kód újrahasznosíthatóságára fókuszál, kevésbé a tervezésre. Mivel az objektumok már megadják a szoftver alap granularitását ezért az objektumok/objektumcsoportok mentén már lehet rétegekről, modulokról beszélni. Az absztrakció egy magasabb szintjét adja a **komponens-orientált** szoftverfejlesztés, amely az objektum-orinetált paradigma egy természetes kiterjesztése. Ekkor a tervezés a konkrét üzleti funkciók mentén történik, tipikusan sok objektumot magába foglaló komponensek megtervezésével. Az alapvető különbség a két megközelítés között az, hogy amíg az OO tervezés során a tervező a konkrét objektumra, annak a való világbeli tulajdonságaira koncentrálva tervezi meg a valóságot legjobban modellező objektumot, addig a komponens-orientált tervezésnél jóval nagyobb granularitásban és gyakran „hozott anyagból”, azaz meglévő komponensekből állítják össze a rendszert. Egy másik alapvető különbség az, hogy az objektum-orientált tervezésnél nem igazán veszik figyelembe az interakciónál a laza csatolás lehetőségét, (pl. távoli kommunikáció vagy aszinkron kommunikáció) addig ez a komponens-alapú tervezésnél alapvető fontosságú. A komponensek fekete dobozként szerepelnek, és ellentétben az objektum-orientált paradigma esetén alkalmazott láthatósági lehetőségekkel, itt a komponens csakis az általa biztosított interfészén keresztül érhető el.

Egy további különbség az, hogy a komponens gyakran saját perzisztencia (tartós adattárolás) megoldással is bír. A komponens-orientált paradigma egy magasabb szintű absztrakciója a **szolgáltatás-orientált paradigma**. Míg a komponens-orientált megközelítésnél lehetséges volt a kód alapú funkciómegosztás is, addig a szolgáltatás-orientált megközelítésnél a futó kód által biztosított szolgáltatások képezik az alap építőköveket. A komponensek szintjén talán, de a szolgáltatások szintjén mindenképpen érdemes foglalkozni kell az elosztottsággal és az ebből fakadó problémákkal, lehetséges megoldásokkal. Jelen fejezet célja az, hogy a szolgáltatás-orientált architektúrák szintjén megjelenő elosztottsággal kapcsolatos problémákat és megoldásmódokat áttekintse.

3.1. Felmerülő problémák

Leslie Lamport egy igen frappáns definíciót adott az elosztott rendszerekre:

„*A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable*” azaz „*Elosztott rendszer az, ahol egy olyan számítógép hibája teszi használhatatlanná a saját gépünket, amiről azt sem tudtuk, hogy létezik.*”

Egy szabatosabb definíció Wolfgang Emmerichtől: „*Elosztott rendszernek nevezzük az autonóm, egymással hálózaton együttműködő számítógépeket, amelyek közös együttműködésén alapuló szolgáltatását a felhasználó úgy érzékeli, mintha azt egy integrált eszköz szolgáltatta volna.*”

Mielőtt belemerülnénk a részletekbe, tekintsük át az egyik legfontosabb mozgatórugót, amely az elosztott rendszerek mögött áll. Ez a skálázódás képessége, azaz hogy képes legyen nagyobb mennyiségű feladatot is ellátni akkor, ha újabb erőforrásokat teszünk a rendszerbe. Az erőforrásokat két módon tudjuk skálázni:

- *Vertikális skálázás*: ekkor egy fizikai gépen vagyunk és ide teszünk újabb processzorokat, több memóriát, háttértárat. Ennek a módszernek az előnye, hogy egy processzen belüli alkalmazás maradhat, viszont – mint az belátható – a behelyezhető erőforrásoknak fizikai korlátai vannak.
- *Horizontális skálázás*: ekkor nem a futtató gépet bővítjük, hanem újabb futtató gépeket vonunk be. Ez viszont már egy elosztott rendszert eredményez. Igaz a skálázódásnak nincsenek korlátai (elvileg, a lentiekben ismertetett CAP tézis azért ad korlátokat).

Az elosztott rendszerek képességeinek alapvető dimmenzióit és ezek korlátait a 2000-ben előadott *Brewer-féle CAP tétel* határozza meg. A képességek három dimmenziója közül egy-egy konkrét rendszer legfeljebb két képességet tud egyszerre megvalósítani. Az alap képességek az alábbiak:

- *Konzisztenica (Consistency)*: A hagyományos adatbázisok biztosítani tudják, hogy nem konzisztens adat nem kerülhet lementésre (perzisztálásra). Ezen képességre az ACID (Atomicitás, Konzisztencia, Elkülönítés és Tartósság) tulajdonságok megvalósításával tesznek szert. Amennyiben nem elosztott alkalmazásról beszélünk, ez minden további nélkül megtehető.
- *Rendelkezésre állás (Availablity)*: A rendelkezésre állás azt jelenti, hogy a rendszer hibatűrő azaz a szolgáltatás adott százaléknyi eséllyel elérhető. A rendelkezésre állást világméretű trendszerek esetén csak elosztott rendszerrel lehet megoldani.
- *Particioniálás-tűrés (Partition tolerability)*: Gilber és Lynch definíciója szerint „*A teljes rendszer (hálózat) hibáján kívül semmilyen hibakombináció sem okozhatja azt, hogy a rendszer hibásan válaszol*”. A hibák alatt jellemzően nem szoftveres hibákat, hanem hálózati hibákat (például üzenetvesztéseket) értünk.

Mint már említettük, egy egyetlen gépen futó rendszernél a CAP mint kényszer nem jelenik meg, mivel ott például a particionálás nem jelent problémát. Az is igaz viszont, hogy a magas rendelkezésre állást is nehéz ilyen architektúrával megvalósítani, nem beszélve arról az esetről, amennyiben nem egy, hanem több, a világon szétszórt felhasználót/

/helyszínt szeretnénk kiszolgálni. Az alábbi döntéseket hozhatjuk meg (a valóság persze nem bináris, ezen követelmények egy folytonos skálán is ábrázolhatóak, mint azt a BASE esetén is látjuk):

- Enyhítünk a particionálás tűréssel kapcsolatos követelményeken: egy megoldás, hogy minden, az adott tranazakció által érintett adat egy gépen van, ekkor nem léphet fel a particionálás. Ekkor csak a vertikális skálázás a járható út, ez viszont komoly korlátokkal bír. Példák: adatbázisok, klaszter alapú adatbázisok.
- Enyhítünk a rendelkezésre állással kapcsolatos követelményeken: az előző ellentéte, partíció esetén egyszerűen megvárjuk, amíg az megszűnik, és a rendszer megy tovább. Példák: elosztott adatbázisok, elosztott zárolások, többségi protokollok.
- Enyhítünk a konzisztenciával kapcsolatos követelményeken: sok esetben az ACID megközelítés nem kritikus, egy kevésbé friss adat visszaadása is megfelelő lehet. A konzisztenciának az ACID által megvalósított verzióját **erős konzisztenciának** (strong consistency) nevezzük, ekkor csak a legfrissebb jó adat adható ki. A **gyenge konzisztencia** (weak consistency) ezzel szemben megenged egy olyan időtartamot (**inkonzisztencia-ablak** – inconsistency window), amikor nem a legfrissebb adat kerül visszaadásra. A **végső fokon konzisztens** (eventually consistent) megközelítés a gyenge konzisztenica egy speciális esete, amikor hibamentes működés esetén az inkonzisztencia-ablakot a rendszer mérete és aktuális állapota (replikák száma, a rendszer terhelése, hálózati késleltetés, stb.) határozza meg. Ezen konzisztencia megközelítést valósítja meg a BASE (Basically Available Soft-state Eventually consistent) archtektúra. Példák: Web gyorstárak, DNS.

Mint láthattuk, nincs ingyen ebéd, valamilyen kompromisszumot meg kell kötnünk annak érdekében, hogy megvalósíthassuk a rendszerünket. A következő fejezetben áttekintjük az alapvető szinteket, amelyeket egy-egy elosztott rendszer ma szolgáltathat.

3.2. Elosztott rendszer-architektúrák

Az elosztott rendszereket megvalósító szoftver-architektúrák esszenciális típusait architektúra mintaként szokták emlegetni. Ezen architektúra minták nem ortogonálisak, azaz pl. egy megoldás lehet kliens-szerver és lazán vagy szorosan csatolt egyszerre. A következőkben felsorolt architektúra minták sokszor inkább kiegészítik mint kizárják egymást.

3.2.1. Kliens-szerver

A kliens-szerver architektúra minta elsősorban a szerepkörök alapján határozza meg a rendszerben résztvevő entitások helyzetét. A szerver oldal birtokolja az erőforrásokat, míg a kliens oldal használja azokat. Ezen elvek mentén működnek napjaink legnépszerűbb protokolljai (HTTP, SMTP, DNS, stb.). A megközelítés előnyei közé tartozik a könnyebb karbantarthatóság, és a központosítás miatt elvileg könnyebb biztonságos rendszert készíteni. A megközelítés hátránya a skálázhatóság és a robosztusság kérdése, mivel ez csak a szerver oldalon múlik.

3.2.2. P2P

A kliens szerver architektúra ellentétje a P2P architektúra, ahol egyenrangú vagy nem egyenrangú, de az erőforrások megosztásában egyaránt részt vevő entitások alkotják a rendszert. Napjaink legnépszerűbb VoIP alkalmazása, a Skype vagy a legtöbb fájlcserélő protokoll ezen az elven működik. Ami viszont kevésbé látható, hogy a későbbiekben említett felhő architektúrák, de még a klaszter megoldások alatt is gyakran P2P algoritmusokon alapuló replikációs megoldások vannak. A kliens-szerver megoldással szemben a hátránya a komplexitásában és elosztottságában rejlik, viszont cserébe extrém skálázhatóságot nyújt. (Vegyük észre, hogy ezek nem merőleges területek: a JBoss klaszter alatt lehet P2P gyorsítótár, azaz a rendszer egy része megvalósítja a P2P paradigmát, míg a kiszolgált kliensek HTTP protokollon férnek hozzá a tartalomhoz, megvalósítva a kliens-szerver paradigmát).

3.2.3. Többrétegű architektúrák

A szerepkörökön túllépve egy másik rendezőelv a logika lehetséges szeparálásának módja. Erre jó példa a 3- vagy többrétegű architektúra. Ez esetben a konkrét erőforrás megosztásának részleteit vizsgáljuk. Az alkalmazást/rendszert skálázhatóság, robosztusság, menedzselhetőség és sok más szempont miatt is érdemes rétegekre bontani. A leggyakrabban alkalmazott rétegelés a háromrétegű architektúra, melynél az alábbi rétegeket definiáljuk:

- *Megjelenítés*: a társ rendszer felé nyújt interfészt (gyakran ez a felhasználói interfész), és az ehhez kapcsolódó eseményeket kezeli le. Ennek a leggyakoribb megvalósítása a weboldal és az előállító logika.
- *Üzleti logika*: ezen réteg feladata az üzleti folyamatok futtatása, hosszú életű tranzakciók kezelése. Itt kerül sor az adatok szélesebb hatókört, több adattípust átölelő szabályainak kikényszerítése is.
- *Perzisztencia*: az adatok tartós tárolásával foglalkozik. Itt történik meg a szűkebb hatókörrel rendelkező adatszabályok kikényszerítése és gyakran az objektum és relációs adatmodellek közötti leképezés is.

A háromrétegű architektúrát gyakran keverik a megjelenítésben használatos MVC (Model-View-Controller) architektúrával. Fontos megemlíteni, hogy míg az MVC-nél a View és a Controller egyaránt kommunikál a Modellel addig a három rétegű architektúránál ez nem történik meg. A megjelenítés réteg csak az üzleti logikán át érheti el a perzisztenicát. A három réteg jelenti alapesetben azokat a választóvonalakat, amelyek mentén a horizontális skálázás megoldható. Lehet például egy olyan RIA (Rich Internet Application), ahol a megjelenítés rétegen nagy terhelés van de ez az alatta lévő rétegekre nem terjed át, mivel hatékony gyorsítótárazást alkalmaz. Ekkor a megjelenítés réteget kell skálázni több gép bevonásával, míg az üzleti logika és a perzisztencia réteg megmaradhat egy-egy gépen. Amennyiben skálázhatóság, robosztusság vagy egyéb szempontok miatt nem elegendő a három réteg, akkor a három réteget újabb rétegekre lehet bontani. Ennek egy gyakori példája, amikor a perzisztenica réteget két rétegre bontjuk: az egyik az ORM (objektum-relációs leképezés) által megvalósított DAO (Data Access Objects) tervezési minta mentén megvalósított réteg, míg alatta a hagyományos – esetenként heterogén – adatbázis réteg helyezkedik el.

3.2.4. Szorosan csatolt rendszerek

Egy újabb rendezőelv lehet a rendszert alkotó modulok csatolásának módja. A szorosan csatolt rendszerekben a komponensek (vagy ha ilyenek nincsenek, akkor az osztályok, objektumok) aszinkron módon kommunikálnak egymással, szorosan követve a klasszikus függvényhívás szemantikáját. Ebbe a kategóriába tartoznak a távoli eljárásokon alapuló rendszerek is. A megközelítés előnye az, hogy a rendszer a klasszikus, egy processzen belül is megtalálható interkaciós paradigmákra épít, egyszerűvé téve ezzel a megvalósítást. Ezzel a megoldással viszont nehéz robosztus, skálázható rendszereket létrehozni.

3.2.5. Lazán csatolt rendszerek

A lazán csatolt rendszerek a szorosan csatolt rendszerekkel ellentétben üzenet alapú kommunikációra építenek és e kommunikáció mentén az aszinkron interakció az alapértelmezett. Ezen paradigmák mentén jóval nehezebb egy rendszert megvalósítani, mivel az aszinkronitás, többutas végrehajtás végig ott lebeg a fejlesztők szeme előtt, azonban ez a leginkább járható út a robosztus, skálázható rendszerek megvalósítására. Ebbe a kategóriába tartozik még az **eseményalapú szoftver architektúra** is (**EDA – Event Driven Software Arhictecture**), mely az eseményeket tekinti a komponensek közötti interakció alapjainak. Az események feldolgozása az alábbi kategóriákba csoportosítható:

- *Egyszerű események feldolgozása*: ez esetben egy atomi esemény beérkezése indít el egy feldolgozási folyamatot. Egy példa erre a különböző felhasználói interakciókat támogató keretrendszerek eseménykezelése (pl. SWING JSF, stb.)
- *Eseményfolyamok feldolgozása (Event Stream Processing – ESP)*: ez esetben eseményfolyamokat szűrnek egyszerű feltételek alapján, és az érdekes eseményekre feliratkozott vevőknek küldik ki a szűrt eseményeket. Példa lehet erre, amikor a naplózásnál csak adott szintet elérő eseményeket továbbítunk.
- *Komplex esemény feldolgozás (Complex Event Processing - CEP):* ez esetben a valós idejű eseményfolyamon értékelünk ki olyan szabályokat, melyek figyelembe vehetik az események sorrendiségét, bekövetkeztét, számosságát és számos egyéb tulajdonságát. Erre a későbbiekben majd látunk példát a Drools-szal kapcsolatban.

3.2.6. Tér alapú architektúrák (Space Based Architectures - SBA)

A tér alapú elosztott architektúrák lényege az, hogy a rendszert olyan egymástól független egységekre osztják, amelyeket tetszőleges számban lehet hozzáadni, elérve ezzel a lineáris skálázódást. Ezeket az egységeket **feldolgozó egységeknek** (**Processing Unit - PU**) nevezik, és tipikusan egy olyan szoftver rétegen ülnek, amely elrejti előlük az elosztottság problémáit. Ez a réteg – a későbbiekben bővebben kifejtett – köztes réteg. Ennek három alapvető formája van:

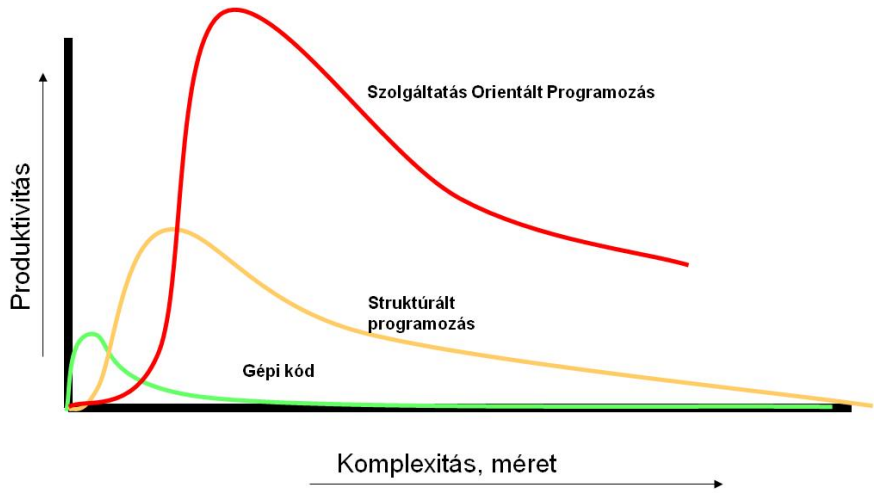
- *Klaszter alapú (Cluster):* a feldolgozásra koncentrál, az adattárolásra különböző tervezési minták vannak, a tipikusan olvasás jellegű alkalmazásoknál igen jó skálázódást tud elérni a CAP kompromisszumai nélkül. Le tudja kezelni az írási ütközést is. Példák erre a különböző alkalmazásszerverek által nyújtott klaszterezési lehetőségek (pl. JBoss Cluster)

- *Felhő alapú (Cloud):* a CAP paradigma mentén a skálázható adattárolást is biztosítja, tipikusan a konzisztencia rovására. Példa erre a Google App Engine.
- *Rács alapú (Grid):* főleg a feldolgozásra koncentrál, nincs igazán írási ütközés, így a feldolgozás tetszőlegesen párhuzamosítható. Ezt leginkább munkafolyamatok (**Job**) formájában szokták megtenni.
- *Elosztott objektum alapú (Distributed Objects):* az elosztott objektumok gyakran a fenti architektúrák alapjait képezik. Ekkor egy-egy objektum vagy replikánsa (**replikált objektum** – replicated object) megjelenik azokon a helyeken, ahol használják, és a háttérben egy keretrendszer (köztes réteg) gondoskodik arról, hogy konzisztens maradjon, vagy pedig egy helyen van tárolva (**élő objektum** – live object), és azokon a helyszíneken, ahol szükséges megfelelő proxy objektumok képviselik az objektumot. A megosztott objektumok gyakran **objektumterekbe (object spaces)** vannak szervezve, melyet valamilyen köztes réteg tart karban, virtualizál (pl. Java Spaces).

A különböző szoftver-architektúrák nem egymást kizárják, hanem igen gyakran egymásra épülnek. Egy-egy komplex rendszerben számos szoftver-architektúra megtalálható. A következő fejezetekben az elosztott rendszerek megvalósítása mögött álló legnépszerűbb paradigmát, a **Szolgáltatás-orientált paradigmát (Service Oriented Architecture Paradigm)** mutatjuk be.

3.3. A SOA-koncepció

A bevezetőben már említett komponens alapú szoftver-architektúra egy speciális formája a szolgáltatás-orientált architektúra, amelyben nem kód, hanem futó kód alapú elemekből állítjuk össze rendszerünket, alkalmazásunkat. Ez esetben nem kell az adott funkcionalitás futtató környezetével, menedzsmentjével foglalkoznunk, ezt megfelelő szolgáltató végzi el helyettünk, mi azt csak felhasználjuk. Persze nem kötelező más által futtatott szolgáltatást igénybe vennünk, a lényeg a szolgáltatás szintű építkezés. Hatékonyságát tekintve a nagyobb, komplexebb problémák esetében egyértelmű az oszd meg és uralkodj SOA szerinti alkalmazásának előnye.



3. ábra: SOA produktivitás

A SOA számos definíciója közül itt az IBM féle definíciót használjuk: „*A szolgáltatás-orientált architektúra egy keretrendszer, mely segítségével az üzleti folyamatokat és a támogató IT infrastruktúrát úgy tudjuk biztonságos, szabványos* ***szolgáltatások*** *formájában integrálni, hogy az újrahasználható és követendő az üzleti igényeket könnyen megváltoztatható*”

Az előző fejezetben bemutatott szoftver-architektúrák közül a SOA leginkább laza csatolásra épít:

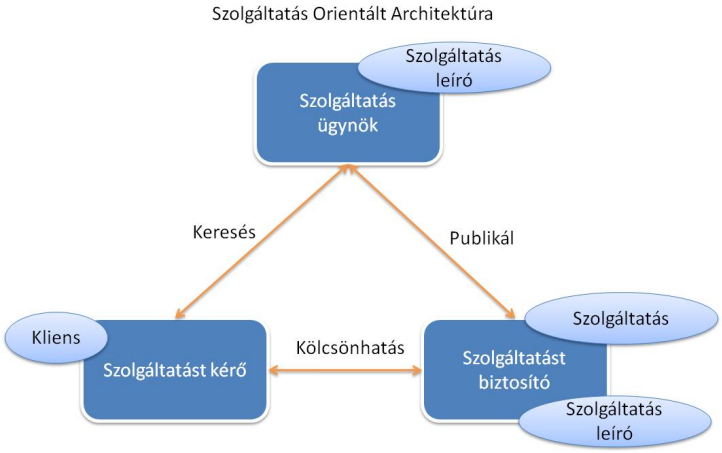
- Az egyes szolgáltatások fekete dobozok, nem látunk bele a belsejükbe, csak a meghirdetett szolgáltatásaikat vehetjük igénybe.
- A szolgáltatások elérése lehet szinkron és aszinkron is. Üzenet vagy metódushívás stílusú egyaránt.

A SOA azonban túlmutat a szolgáltatások definiálásán és ezek valamilyen szintű elérésén: számos olyan keresztülívelő probléma van, amit valamilyen egységes módon át kell vinni, kezelni kell. A meglévő szolgáltatásokból szervezett új szolgáltatás létrehozásánál biztosított szervező képesség (orchestration) segítségével lehetővé válik az üzleti folyamatok változásának könnyű követhetősége. Magas szinten (technológiai megközelítésben) a SOA modell az alábbi szereplők együttműködését definiálja:

- szolgáltatást *nyújtó* entitás (*szolgáltató*, *service provider*): az a szereplő (számítási egység), amely a szolgáltatás által lefedett feladatokat képes elvégezni.
- szolgáltatást *igénybe vevő* entitás (fogyasztó, *service consumer*): az a szereplő, amelynek feladata elvégzéséhez szüksége van arra, hogy egy adott szolgáltatást igénybe vegyen (egy szolgáltatótól). Ez a kliens (lehet vékony, vastag és mobil) az utóbbi időben a smart client fogalom is forog az irodalomban.
- szolgáltatást *közvetítő* entitás (*service broker*): az a szereplő, aki ismeri azoknak a szolgáltatóknak a halmazát, amelyek képesek egy adott szolgáltatást biztosítani. Ez egy köztes szereplő, amely bizonyos esetekben kihagyható a tényleges kommunikációs folyamatból.
- kommunikációs csatorna: ez tipikusan webszolgáltatás, amely a későbbiekben ismertetett SOAP protokoll segítségével támogatja a szinkron és aszinkron kommunikációt. A különböző keresztülívelő problémák kezelésére pedig a megfelelő webszolgáltatás profilok biztosítanak szabványos, mindenki által érthető leírást, megvalósítási útmutatót.

Vegyük észre, hogy ezek a szerepek ebben az esetben is egy adott szolgáltatásra vonatkoztatva jelennek meg, tehát elképzelhető, hogy egy A *szolgáltatást nyújtó entitás* egy másik, B szolgáltatásra vonatkozóan a *szolgáltatást igénybe vevő entitás* szerepében van.

Ezek a szereplők az alábbi módon kapcsolódnak egymáshoz:



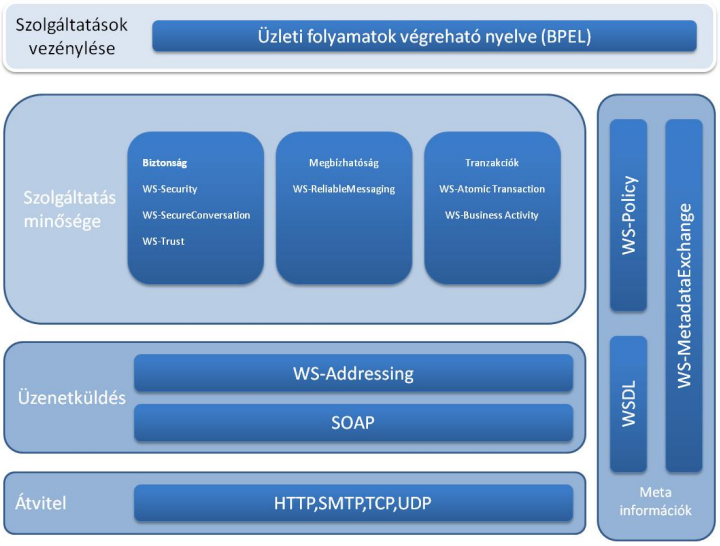
4. ábra: SOA szereplők és kapcsolataik

Az ábrán az alábbi három interakció jelenik meg az egyes szereplők között:

- *közzététel* (*publish*): a szolgáltató az általa nyújtott szolgáltatásokat a megfelelő eszközökkel leírja, és a szolgáltatások leírását közzéteszi a (közismert) közvetítőnél
- *keresés* (*search*, *query*): a szolgáltatást igénybe venni kívánó entitás a közvetítőt felhasználhatja arra, hogy az igényeinek megfelelő szolgáltatást keressen.
- *összekapcsolás (interakció)*: a szolgáltatást igénybe vevő entitás a szolgáltatás és az azt szolgáltató entitás adatait ismerve felveszi a kapcsolatot a szolgáltatóval, és igénybe veszi a szolgáltatást. Vegyük észre, hogy a SOA rendszerben jellemzően üzenet alapú kommunikáció történik, amelyben a szolgáltatást nyújtó a szerver, a szolgáltatást igénybe vevő a kliens.

A SOA alapú rendszerek elterjedésekor nagy hangsúlyt fektettek a közvetítőkre, több nagy cég működtetett kereshető szolgáltatás-adatbázisokat, amelyeket felhasználva böngészhettünk az általuk ismert szolgáltatások és szolgáltatók között. Az utóbbi időben azonban ez kevésbé elterjedt módszer, a közvetítők szerepe csökkent, hiszen a legtöbbször a szolgáltatást igénybe vevő komponensek fejlesztése során pontosan tudjuk, hogy milyen jellemzői vannak az igénybe venni kívánt szolgáltatásnak (ez esetben statikus kötésről beszélünk, míg abban az esetben, amikor fejlesztés során nem ismertek az igénybe venni kívánt szolgáltatás jellemzői, dinamikus kötésről van szó). Eltemetni azonban még korai a háromszög közvetítő elemét, hiszen a szemantikus webszolgáltatások fejlődésével újra jelentős szerephez juthatnak. A szemantikus webszolgálatásokról később lesz szó a jegyzetben.

Ha már felmerültek a szemantikus webszolgáltatások, akkor érdemes néhány szót ejteni a webszolgáltatásokról, hiszen a jegyzet későbbi fejezeteiben nagyon gyakran fogunk ezzel a fogalommal találkozni. A webszolgáltatás egy azok közül a technológiák közül, amik lehetővé teszik a SOA-koncepció gyakorlatba történő átültetését. A webszolgáltatások jellemzően (de nem kizárólag) HTTP protokollba ágyazva közlekedő SOAP üzeneteket takarnak, hogy ez pontosan mit jelent, azt a későbbi fejezetekben tárgyaljuk. Az alábbi ábrán látható néhány szabvány technológia, amely felhasználható webszolgáltatások alkalmazása esetén.



5. ábra: Webszolgáltatás-technológiák

Mivel valamilyen szintű webszolgáltatási eszközkészlet ma már szinte minden fejlesztési platform alatt rendelkezésre áll, ezért a webszolgáltatások kitűnő megoldást jelentenek a platformfüggetlen integrációra. Azaz, ha azt szeretnénk, hogy egy szolgáltatásunk minél szélesebb fogyasztói kör számára elérhető legyen, akkor tegyük elérhetővé a szolgáltatást webszolgáltatáson keresztül! Érdemes azonban figyelembe venni, hogy a haladóbb technológiák (WS-Trust, WS-BusinessActivity, stb.) nem minden platformra elérhetők, ezért ha ezekre szükség van, akkor a megfelelő platformot kell választani. A WS-* profilok biztosítják a kompatibilitást köztesréteg szinten.

Egyelőre elegendő megjegyeznünk, hogy ezek a szolgáltatás-orientált architektúrák alap építőkövei. A későbbi fejezetek alapján egy jóval teljesebb kép fog kialakulni arról, hogy mik is azok a webszolgáltatások, hogyan kapcsolódnak egymáshoz és milyen haladó lehetőségeket biztosítanak.

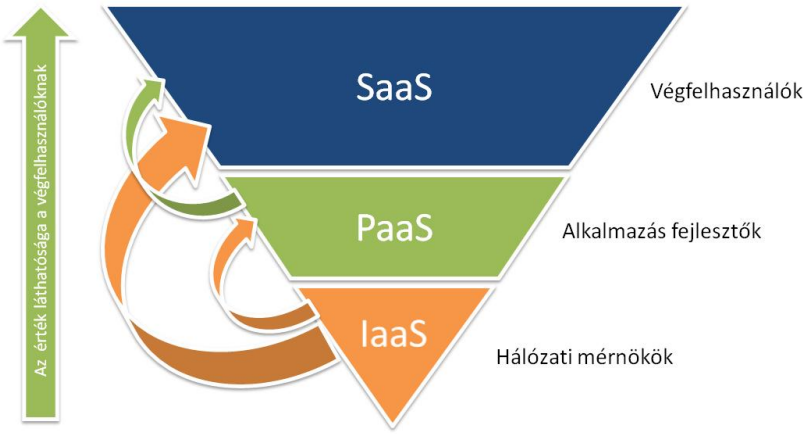
3.4. Virtualizációs szintek

Az elosztott rendszerekkel kapcsolatos kép teljessége érdekében érdemes áttekinteni a mai felhő szolgáltatási szinteket. Ezek a következőek:

- *szoftver mint szolgáltatás* (*Software as a Service*, *SaaS*): ez a legelterjedtebb a hasonló szolgáltatások közül. Gyakorlatilag annyit jelent, hogy a szolgáltató nem adja át az elkészült szoftvert az azt használó ügyfélnek, hanem szolgáltatásként üzemelteti azt. Ilyen módon lehetővé válik, hogy a használónak ne kelljen telepíteni az alkalmazást, ne kelljen az üzemeltetés nehézségeivel bajlódnia. Ezen felül az is lehetővé válik ilyen módon, hogy a szoftvert használó ügyfél annak megfelelően fizessen a szoftverért, hogy mennyit használja azt ténylegesen. Ez természetesen bontható többféleképpen. Például elképzelhető olyan szoftver, hogy minden funkciója elérhető, és az ügyfél az adott hónapban annak megfelelően fizet, hogy az elérhető funkciók közül mennyit használt ténylegesen, de elképzelhető az is, hogy a díjfizetés alapját a forgalmazott adat képezi (az igénybe vett funkciók számától függetlenül), stb.

Ha jobban belegondolunk, ez a fajta szolgáltatás elég régóta jelen van az Interneten, gondoljunk csak a Google Mailre vagy bármely más hasonló cég ilyen jellegű szolgáltatására.

- *platform mint szolgáltatás* (*Platform as a Service*, *PaaS*): így nevezzük azt a szolgáltatást, amikor a szolgáltató nem egy konkrét szoftverterméket biztosít az ügyfelek számára, hanem egy olyan számítási platformot, amelyre az ügyfél készíthet és telepíthet saját szoftvereket. Van olyan PaaS szolgáltatás, aminek a használatához szükséges az, hogy a biztosított platformon futó alkalmazások egy megfelelő API-hoz legyenek igazítva (például adattárolásra csak egy adott, speciális módon van lehetőség), de ez nem szükséges feltétele egy PaaS szolgáltatásnak.
- *infrastruktúra mint szolgáltatás* (*Infrastructure as a Service*, *IaaS*): a platform szintjéhez képest van lehetőségünk még alacsonyabb szintben gondolkodni. Ez pedig nem mást eredményez, mint olyan szolgáltatásokat, amelyekkel a szolgáltató virtuális infrastruktúrát biztosít az ügyfelei számára. Ilyenkor az ügyfelek szempontjából a szolgáltatás felfogható egy adott mennyiségű számítógépként, amelyre bármilyen platformot (és arra a megfelelő szoftvereket) telepíthet vagy fejleszthet. Az így elérhető számítási egységek lehetnek virtuális gépek, de elképzelhető az is, hogy fizikai gépekhez kap hozzáférést az ügyfél.



6. ábra: Felhő-szolgáltatások. Az ábrán az SaaI helyett IaaS

A megfelelő méretű infrastrukturális hátteret felhasználó cégek esetén azonban megvan a fenti szolgáltatásoknak az az előnye, hogy a szolgáltatást igénybe vevő ügyfélnek nem kell részletes és pontos előzetes becsléseket készítenie arról, hogy az egyes rendszerkomponensek milyen terhelésnek lesznek kitéve, mivel az igénybe vett háttér biztosítja azt a rugalmasságot (*elasztikus felhő*), hogy mindig annyi erőforrást biztosít, amennyi szükséges (és ennek megfelelően kerülnek kiszámításra a költségek is). Ilyen módon ezek a szolgáltatások extrém skálázhatóságot biztosítanak.

E szolgáltatásokkal kapcsolatban az előnyök mellett egy fontos hátrányt is meg kell említeni. Ez pedig az adatvédelem. Az ilyen szolgáltatások esetén ugyanis a szolgáltatást igénybe vevő ügyfélnek gyakorlatilag semmilyen rálátása nincs arra, hogy az adatok hol és milyen módon tárolódnak és milyen csatornákon keresztül közvetítik azokat. Azoknál az alkalmazási területeknél, ahol kiemelten fontos az adatvédelem, ezek a megoldások nem vagy csak a megfelelő korlátozásokkal használhatók. A nagyobb IaaS/PaaS szolgáltatóknak (pl. Amazon, Google, Microsoft) azonban legtöbbször érdekük, hogy nagy és tőkeerős ügy-

feleik is legyenek, emiatt a megfelelő szerződések mellett elképzelhető, hogy az ilyen ügyfelek esetén az adatvédelmet is megfelelő szinten lehet biztosítani, illetve léteznek *privát felhő* (*private cloud*) megoldások is, amelyek ezt a hátrányt kiküszöbölik. (Nemrégiben volt például hír, hogy az Egyesült Államok kormányzati szervei egyre inkább kezdik felismerni a felhőben rejlő lehetőségeket.)

3.5. Összefoglaló

Az eddigiek során megismerkedhettünk azokkal az alapvető fogalmakkal melyek ma jellemzik az elosztott programrendszereket. A következő két fejezetben bemutatjuk egyrészt azokat az általános a konkrét alkalmazások funkcionalitásán túlmutató területeket, amelyek egy-egy elosztott rendszerben jó eséllyel megjelennek (keresztülívelő problémák) és azt a szoftverabsztrakciós réteget, amely az elosztottságot és ezeket a keresztülívelő problémákat megvalósítja, összefogja, szervezi, illetve magát az elosztottságot elfedi a szoftverfejlesztő elől.

3.6. Kérdések

1. Mik a szolgáltatás-orientált architektúrák legfontosabb elemei, ezek hogyan kapcsolódnak egymáshoz?
2. Mit nevezünk számítási felhőnek? Milyen típusú szolgáltatásokat tesz lehetővé?
3. Mik az elosztott rendszerekre vonatkozó legfontosabb követelmények?

3.7. Ajánlott irodalom

- Coulouris, G. és mások: *Distributed Systems - Concepts and Design.* 4. kiadás. Addison-Wesley, 2005. ISBN 0-321-26354-5
- Emmerich, W.: *Engineering Distributed Objects.* Wiley, 2000. ISBN 0-471-98657-7
- Estefan, J. A. és mások: *Reference Architecture Foundation for Service Oriented Architecture Version 1.0.* OASIS Committee Draft. Elérhető: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf
- Ganci, J. és mások: *Patterns: SOA Foundation Service Creation Scenario.* IBM Redbook, 2006. Elérhető: http://www.redbooks.ibm.com/redbooks/pdfs/sg247240.pdf
- Graham, S. és mások: *Java alapú webszolgáltatások - XML, SOAP, WSDL, UDDI.* Kiskapu Könyvkiadó, 2002. ISBN 9-639-30104-3.
- Hurwitz, J. és mások: *SOA for Dummies.* Elérhető: ftp://ftp.software.ibm.com/software/solutions/soa/pdfs/SOAforDummies.pdf
- Keen, M. és mások: *Patterns: Extended Enterprise SOA and Web Services.* IBM Redbook, 2006. Elérhető: http://www.redbooks.ibm.com/redbooks/pdfs/sg247135.pdf
- MacKenzie, C. M. és mások: *Reference Model for Service Oriented Architecture 1.0.* OASIS Standard. Elérhető: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

- Papazoglou, M. P. és mások: *Service-Oriented Computing Research Roadmap.* Service Oriented Computing (SOC), Dagstuhl Seminar Proceedings, 2006. Elérhető: http://drops.dagstuhl.de/opus/volltexte/2006/524/pdf/05462.SWM.Paper.524.pdf

4. Átszövődő vonatkozások

A nem funkcionális követelmények kielégítéshez kapcsolódó programrészeket átszövődő vonatkozásoknak nevezik. Ezek bizonyos mértékig megjelennek minden komponensben így érdemes áttekinteni azokat a leggyakoribb átszövődő vonatkozásokat amelyekkel találkozhatunk. E rövid áttekintés után a következő fejezetben már bemutathatjuk azt a réteget (köztesréteg) amely ezen átszvödő problémákat többé kevésbe elfedi a fejlesztő elől.

4.1. Kontextusok, általános problémák

A legtöbb átszövődő vonatkozáshoz valamilyen adathalmaz tartozik, amelynek az élettartama, a rendszeren belüli láthatósága jól definiálható. Ezt az adathalmazt összefoglaló néven kontextusnak nevezzük. Mint majd később láthatjuk ezen kontexusok nem csak a rendszeradatok, hanem a funkcionális követelményeket kiszolgáló adatok szervezésére is igen gyakran alkalmazottak. A Web konténernél például 4 különböző láthatóságú és élettartamú kontextust használhatunk (viszony, kérés, válasz, alkalmazás). Ezen kontextusok átvitele egyik komponensből a másikba, az ezekhez kapcsolódó protokollok esteleges támogatása a web szolgáltatás profilok legfontosabb feladata. Az alábbiakban két fontosabb kontextust tekintünk át.

4.2. Jellemző alkalmazási területek

Következőekben kifejtett kontextus a pedig a biztonságkezelés a másik pedig a tranzakció-kezelés lesz. Ezeket jellemzően támogatják a népszerűbb köztesréteg megvalósítások és számos web szolgáltatás profil is tartozik hozzájuk. Az alfejezetben először bemutatjuk a két említett terület jellemző problémáit, majd mindkét területen bemutatjuk azokat a pontokat, amelyek jellemzően rendszereken keresztül ívelnek.

4.2.1. Biztonságkezelés

Az üzleti információ biztonsága minden cég számára kiemelkedő jelentőségű. Amikor egy-egy rendszert önállóan használunk, a biztonság megvalósítása viszonylag egyszerű feladat. Abban a pillanatban viszont, amint ezek a rendszerek együttesen használva egy nagyobb, elosztott rendszerbe vannak szervezve (kiváltképpen, amikor különböző cégek által üzemeltetett rendszerek vannak összekötve) az információ biztonságos kezelése egészen komplex feladattá válik. Ebben az esetben az egyes rendszereknek esetleg valós időben kell egymással kommunikálniuk úgy, hogy az üzleti tranzakciók egységessége megmaradjon. Az egyes felhasználók, csoportok biztonsági azonosítóit szinkronizálni kell a különböző rendszerek között, továbbá az üzleti adatokat védeni kell mind tárolás mind átvitel során. Általában ezeket a feladatokat együttesen az *AAA* névvel illetik (az *authentication*, *authorization*, *accounting* angol szavak kezdőbetűiből). Biztonság témakörben tehát az alábbi kihívásokra keresünk megfelelő megoldásokat:

- *azonosítás* (*authentikáció*) és *azonosságkezelés*
- *jogosultságkezelés* (*authorizáció*)
- *könyvelés* (*accounting*)
- *általános adatvédelem*

Ebben az alfejezetben e területek jellemző problémáit vázoljuk fel.

4.2.1.1. Azonosítás és azonosságkezelés

Azonosításnak nevezzük azt a folyamatot, amely során egy felhasználó (ez lehet akár konkrét személy akár egy szoftverrendszer) azonossága ellenőrzésre, megállapításra kerül. Ez a folyamat általában feltételez valamiféle előzetes ismeretet a felhasználókról (mint például felhasználónév-jelszó párok minden felhasználóhoz): azonosítót és olyan próbákat, amelyek segítségével a felhasználó azonosságát bizonyítani lehessen. (Ilyen próba például az, hogy *„Ismeri-e a felhasználó az azonosítójához tartozó jelszót?”* vagy hogy *„Rendelkezik-e a felhasználó egy megfelelő titkos kulccsal?”*) Amikor egyetlen rendszert használunk, egy egyszerű felhasználónév-jelszó páros teljes mértékben elegendő lehet a felhasználók azonosításához, azonban ez a fajta felhasználói azonosítás kevésnek bizonyulhat abban az esetben, amikor több rendszernek kell együttműködnie.

Amikor több rendszer működik együtt egy feladat során, akkor egy megoldás lehet az, hogy minden rendszer tárolhatja a saját felhasználóhalmazát és a felhasználókhoz tartozó próbák jellemzőit, ezzel a megoldással viszont megnehezíti az egyes rendszerek között az azonosítási adatok átadását. E nehézség legfőbb oka, hogy ezek az önálló rendszerek különböző adatokat használhatnak azonosításra és különbözőek lehetnek az azonosításhoz szükséges próbák is. Ezáltal előfordulhat, hogy ugyanannak a felhasználónak más az azonosítója két különböző rendszerben ráadásul más az azonosításhoz szükséges próba is. Elképzelhető például, hogy egy számviteli rendszerben Kovács József azonosítója *jkovacs*, és az azonosítóhoz tartozó jelszóval tudja bizonyítani a személyazonosságát. Kovács József azonban hozzáférhet egy ügyviteli rendszerhez is, amelyben *kovacs.jozsef* a felhasználóneve és egy titkos kulcsfájl átadása szükséges a felhasználói azonosításhoz. Ebben az esetben, ha azt akarjuk, hogy a két rendszer együttműködhessen egymással, akkor ezeket az azonosítókat meg kell feleltetnünk egymásnak. Ezt azonosító megfeleltetésnek (angolul *identity mapping*) hívjuk. A felhasználói azonosítók megfeleltetése többféleképpen történhet. Amennyiben egy adott rendszer minden egyes felhasználói azonosítójához meg tudunk adni egy, a másik rendszerben érvényes felhasználói azonosítót, akkor több a többhöz megfeleltetésről beszélünk.

Előfordulhat azonban, hogy elegendő az, ha egy rendszer összes felhasználójához hozzárendelünk egyetlen felhasználói azonosítót a másik rendszerben. Ilyenkor *egy a többhöz* hozzárendelésről van szó. Nyilvánvalóan ebben az esetben elveszik az egyes felhasználók egyedi személyazonossága, azonban bizonyos esetekben ez is elegendő, viszont ilyenkor sokkal egyszerűbb a megfeleltetés.

Azt, hogy különböző alkalmazások esetén érdemes ugyanazt az azonosítási mechanizmust használni (ha lehet), már régen felismerték a cégek, ezért általában úgynevezett *címtárakban* vannak eltárolva a felhasználói adatok, így e címtárak alapján az összes alkalmazott azonosítható az összes alkalmazás számára ahelyett, hogy az egyes alkalmazásokban egyenként lennének nyilvántartva a felhasználók. Emiatt általában az azonosítási tartományok nem egy-egy alkalmazásra vonatkoznak, hanem legtöbbször az adott vállalatra. Emiatt is érdemes leválasztani a felhasználói azonosítókat az alkalmazásokról. Ez általában úgy történik, hogy valamilyen föderatív azonosítási megoldást használunk, ami a felhasználók felé leggyakrabban egyszeri bejelentkezésként (*single sign-on, SSO*) jelenik meg.

Az előzőekből pedig nyilvánvalóan következik, hogy a különböző rendszerek közötti szolgáltatáshívások esetén a kérést kezdeményező (vagy továbító - a személyazonosság-

megfeleltetési mechanizmusnak megfelelően) felek személyazonosságát a szolgáltatáshívásokkal együtt továbbítani szükséges ahhoz, hogy a szolgáltatást biztosító számítási egységek meggyőződhessenek arról, hogy a szolgáltatást kezdeményező fél jogosult-e az adott szolgáltatás igénybevételére.

4.2.1.2. Jogosultságkezelés

Jogosultságellenőrzésnek azt a folyamatot nevezzük, amikor megállapítjuk egy felhasználó jogait egy adott rendszerben. Az ellenőrzési folyamat során a jogosultságellenőrző alrendszer eldönti, hogy egy adott felhasználó jogosult-e hozzáférni a megadott módon a megadott erőforráshoz (az erőforrás itt lehet adat, de lehet művelet is). Általában a jogosultságellenőrzést használjuk arra, hogy az erőforrásokhoz részletes hozzáférés-szabályozást tudjunk megvalósítani.

A jogosultságellenőrzés többféle módszer alapján történhet. A jogosultságellenőrzési séma definiálja azokat a szabályokat, amelyek alapján engedjük vagy tiltjuk a hozzáférést egy adott felhasználó számára az adott erőforráshoz. A fő célja a jogosultságellenőrzésnek az, hogy biztosítsa az adatok *titkosságát* (*confidentiality*) és *integritását* (*integrity*). Egy szolgáltatásorientált rendszerben, ahol többféle különböző alkalmazás együttműködése eredményeképpen történik komplex feladatok végrehajtása, többféle jogosultságellenőrzési séma is szerepet játszhat, hiszen minden egyes alkalmazás meghatározhatja a saját ellenőrzési sémáját. Továbbá ezek az alkalmazások több szinten is ellenőrizhetik a jogosultságokat (művelet szintjén, adatok szintjén, stb.). Ennélfogva ilyen SOA rendszerek esetén a rendszerintegrátorok feladata komplex adathozzáférési szabályok meghatározása. Ezek a komplex szabályok egy átfogó szabálykészletbe foglalhatók, ami biztosítja, hogy a szabályok megfeleljenek a cég biztonsági üzletszabályzatával.

Elég gyakran előfordulhat olyan eset, hogy egy adott szolgáltatáshoz való hozzáférési jogosultság, valamint a hozzáférés módja nem kizárólag a szolgáltatást igénybevevő fél személyazonosságán múlik, hanem azon a környezeten is, amelyben a szolgáltatás igénybevétele történi (a nap milyen szakaszában vagyunk, a kliens rendszeren adottak-e a megfelelő feltételek, más szolgáltatáshívás eredményeképpen adottak-e bizonyos feltételek, stb.) Emiatt ilyen esetekben szükséges lehet a személyazonossági információ mellett a biztonságra vonatkozó egyéb adatok átadásai is, esetleg a biztonsági szabályozás (policy) adott kérésre vonatkozó részei hitelesítve.

4.2.1.3. Könyvelés

Könyvelés alatt azt a folyamatot értjük, amely során a felhasználói tevékenységeket gyűjtjük és tároljuk későbbi feldolgozás céljából. Ennek többféle oka is lehet. A könyvelési adatokat használhatjuk arra, hogy a felhasználói műveletek letagadhatatlanságát (*non-repudiability*) biztosítsuk. Ez azt jelenti, hogy amennyiben egy felhasználó végrehajt egy műveletet a rendszerben, annak nyoma legyen, és amennyiben a későbbiek során probléma merül fel, ne tagadhassa le az adott művelet elvégzését. Ebben az esetben passzív védelemről beszélünk, hiszen ilyen esetekben a könyvelési információ csak akkor kerül felhasználásra, ha valamilyen probléma fellépése azt indokolttá teszi.

Heterogén, elosztott rendszerekben, ahol többféle alkalmazás többféle könyvelést tart nyilván, a könyvelési adat gyakorlatilag a teljes rendszerben szétterülve helyezkedik el. Emellett ezek az alkalmazások általában valamilyen saját formátumot használnak a könyve-

lési információ tárolására. Ezen elosztott, többféle formátumot használó könyvelési adatbázis kezelése nem egyszerű feladat.

A könyvelést ezek miatt tekinthetjük keresztülívelő problémának, azonban a könyvelési adatok hozzácsatolása az a szolgáltatáshívásokhoz és az ezekre adott válaszokhoz általában nem jellemző, hiszen a legtöbb rendszer saját hatáskörében végzi a könyvelést. Elképzelhető azonban olyan eset, hogy a szolgáltatáskérés vagy a rá adott válasz tartalmaz a másik fél számára a könyvelésre vonatkozó javaslatot, információt.

4.2.1.4. Adatvédelem

Az AAA-hoz tartozó megfontolások mellett további szempontokat is érdemes figyelembe venni, amikor az adatok biztonságát kívánjuk biztosítani. Ilyen például az adatok védelme hálózati adatátvitel során (ha jól meggondoljuk, elképzelhető, hogy egy felhasználó megfelelően azonosítja magát, van is jogosultsága a rendszerben a megadott információ megszerzésére, és a hozzáférés tényét a rendszer megfelelően le is könyveli, de az egész biztonsági rendszer haszontalan, ha az adatokhoz – az adatátviteli csatorna védtelensége miatt – átvitel során illetéktelenek is hozzáférhetnek).

Ahhoz, hogy az adatok átvitele biztonságos legyen, az átvitelig közeget is biztosítanunk kell. Egy titkosított átviteli közeg nagyon meg tudja nehezíteni az illetéktelenek dolgát, ha hozzá szeretnének férni az átvitt adathoz. Összességében elmondható, hogy az adatok titkosságát és integritását, valamint az adatküldés (és fogadás) tényének letagadhatatlanságát biztosítani kell az átvitel során is. Ezeket jellemzően titkosított adatátviteli rétegekkel, illetve digitális aláírással és titkosítással szokás megoldani.

4.2.1.5. Keresztülívelő biztonsági problémák

Amint azt ebben az alfejezetben láthattuk, az információbiztonság témakörében számos olyan részterület van, amely elosztott rendszerek esetén válik igazán jelentőssé. A teljesség igénye nélkül ilyen részterületek tehát:

- műveletvégzést felhasználó kilétének (és felhasználói adatainak) továbbítása
- felhasználói jogosultságok kezelésének összehangolása, teljes rendszerre vonatkozó biztonsági előírások kialakítása és betartatása
- könyvelési információ átvitele rendszerek között
- rendszerek közötti információcsere védelme (titkosság, integritás, letagadhatatlanság biztosítása)
- kód alapú biztonság

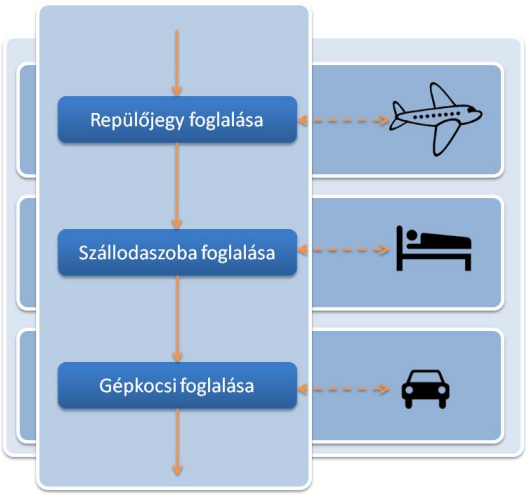
E fejezetnek nem célja bemutatni az e területekre vonatkozó konkrét megoldásokat, azok egy részével a későbbi fejezetekben találkozhatunk (pl. WS-Security, föderatív személyazonosság-kezelés, SSO).

4.2.2. Tranzakció-kezelés

Az adatok biztonságával kapcsolatosan egy olyan fontos probléma is felmerül, ami már önálló területté forrta ki magát, ez pedig az adatintegritás biztosítása. Amikor módosítást végzünk az adatainkon, fontos, hogy minden egyes műveletvégzés után az adathalmaz önmagával konzisztens állapotban legyen. Az adatintegritás megőrzésének érdekében évtize-

dek óta használnak tranzakciókat, ezáltal lehetségessé válik, hogy az összetett adatmódosítási műveletek során minden esetben konzisztens állapotban maradjon az adathalmazunk. Klasszikusan a tranzakció-kezelés problémaköre az adatbázis rendszerekhez kapcsolódik, azonban az évek során magasabb szinten is szükséges volt bevezetni a tranzakciókat és megfelelően módosítani az ide kapcsolódó fogalmakat, módszereket.

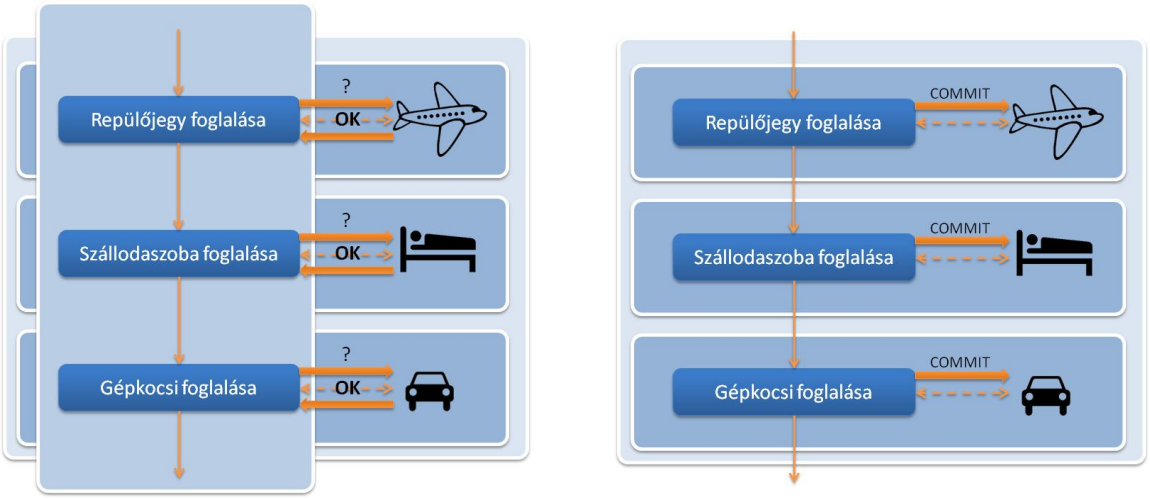
A modern elosztott alkalmazásokban sokszor előfordul ugyanis, hogy egy adathalmaz nem kerül azonnal tárolásra egy megfelelő adatbázisban, továbbá elosztott rendszerek esetén a műveletvégzés sem feltétlenül ugyanazon az eszközön történik. E jelenségek miatt szükség van arra, hogy elosztott környezetben is tudjunk tranzakciókat használni, mégpedig az elosztottságot, mint fontos tényezőt figyelembe véve. Ezt csak úgy tudjuk elérni, ha az egyes szolgáltatások nemcsak azt az adatot kapják meg, amin műveletet kell végezniük, hanem azt az információt is, hogy az adathoz milyen tranzakciós környezet tartozik.



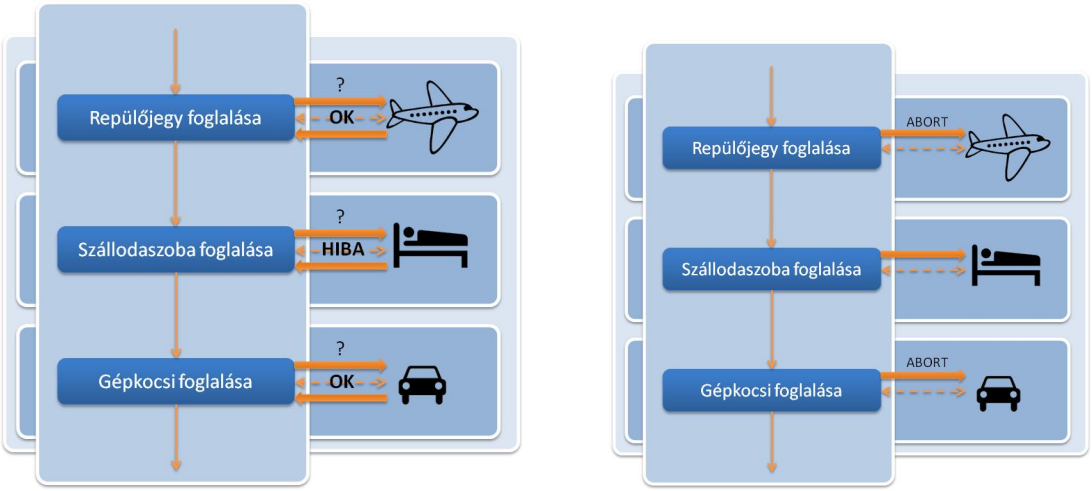
7. ábra: Több rendszeren átívelő tranzakció (példa)

Egy erre vonatkozó példát mutat be a 7. ábra, amely egy elképzelt utazás-előkészítési folyamatot tartalmaz. Egy olyan munkafolyamatot láthatunk az ábrán, amely 3 különböző szolgáltató 3 különböző szolgáltatását érinti. A példa feltételezi azt az előzetesen ismert követelményt, hogy a teljes munkafolyamatot egy tranzakcióként kell kezelni, azaz csak akkor tekinthető sikeresnek a munkafolyamat végrehajtása, ha mind a repülőjegy foglalása, mind a szállodaszoba foglalása, mind pedig a gépkocsi foglalása feladat sikeresen elvégzésre került, mert az illető, akinek a számára végezzük a foglalásokat, kizárólag az összes feltétel együttes teljesülése esetén képes a munkáját megfelelően elvégezni. Amennyiben valamelyik feladat sikertelenül hajtódik végre (pl. nincs szabad szállodaszoba az adott időszakra), akkor a teljes tranzakciót vissza kell vonni (repülőjegy foglalással és gépkocsi foglalással együtt). A teljes problémát bonyolítja, hogy az egyes foglalások is lehetnek összetett műveletek, amiket önálló tranzakcióként kell végrehajtani. Ebben az esetben tehát van 3 olyan tranzakciónk, ami egy-egy szolgáltatót érint, valamint egy negyedik, amely az összes szolgáltatót érinti.

Az ilyen jellegű feladatok megoldására találták ki az úgynevezett kétfázisú commit módszert, ami lehetővé teszi, hogy az egyes tranzakcióknak a „jó” tulajdonságai megmaradjanak. A 8. ábra és a 9. ábra mutatja be probléma megoldását kétfázisú commit felhasználásával.



8. ábra: Kétfázisú commit („minden rendben” eset)



9. ábra: Kétfázisú commit („hiba a szállodaszoba foglalása közben” eset)

Az elosztott tranzakció-kezelés esetén tehát a különböző rendszerek között a szolgáltatás kéréshez és a rá adott válaszokhoz csatolni kell azt az információt, hogy milyen tranzakciós környezetben történik a műveletvégzés. Az ilyen tranzakciós információ is jellemzően olyan, amit kontextus adatként kiválóan lehet kezelni, és természetesen adódnak a megfelelő (tranzakciós) hatókörök is.

4.3. Összefoglaló

A fejezetben kifejtett két példa (biztonság- és tranzakció-kezelés) jól mutatja az igényt a különböző szolgáltatások esetén a kérésekhez és rájuk adott válaszokhoz csatolható információ lehetőségére. Ez azonban csak két olyan keresztülívelő problémakör, ahol a kontextusok (és a hozzájuk tartozó hatókörök) alkalmazása hozzásegít egyes részfeladatok hatékony és elegáns megoldásához. Kis jártassággal és megfelelően rugalmas eszközkészlettel további keresztülívelő (és egyéb) problémákra vonatkozóan saját magunk is kialakíthatunk az előzőekhez hasonló kontextusokat és akár hatóköröket is, amelyeket a szükségeknek megfelelően az adott alkalmazási területhez igazíthatunk. A kontextusok hatékony segítsé-

gében a köztesréteg megoldások hathatós segítséget nyújtanak. Néhány lehetőséggel gyakorlati szempontból is fogunk találkozni a jegyzet hátralévő fejezeteiben.

4.4. Kérdések

1. Milyen problémák jelentkeznek az elosztott felhasználó-azonosítás területén?
2. Milyen problémák jelentkeznek az elosztott jogosultságkezelés területén?
3. Milyen problémák jelentkeznek az elosztott könyvelés területén?
4. Milyen problémák jelentkeznek az adatok védelmének területén elosztott rendszerekben?
5. Milyen problémák lépnek fel, ha több rendszeren keresztül ívelő tranzakció-kezelést akarunk megvalósítani?
6. Mire jó a kétfázisú commit (2PC) módszer, és hogyan ad megoldást a problémára?

4.5. Ajánlott irodalom

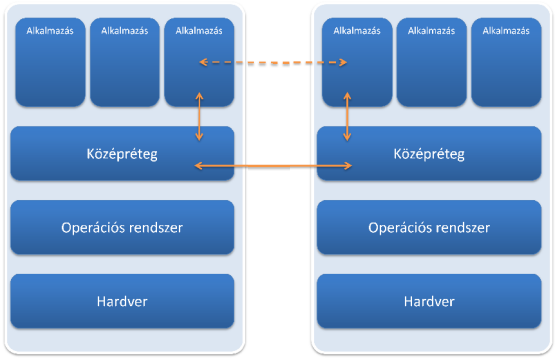
- Bernstein, P. A. és mások: *Concurrency Control and Recovery in Database Systems.* Microsoft Research, 1987. Elérhető: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx
- Buecker, A. és mások: *Understanding SOA Security Design and Implementation.* IBM Redbook, 2007. Elérhető: http://www.redbooks.ibm.com/redbooks/pdfs/sg247310.pdf
- Feingold, M. és mások: *Web Services Business Activity Framework (WS-BusinessActivity).* 2005. Elérhető: http://public.dhe.ibm.com/software/dw/specs/ws-tx/WS-BusinessActivity.pdf
- Feingold, M. és mások: *Web Services Atomic Transaction (WS-AtomicTransaction).* 2005. Elérhető: http://public.dhe.ibm.com/software/dw/specs/ws-tx/WS-AtomicTransaction.pdf
- Little, M. C. és mások: *Java Transaction Processing: Design and Implementation.* Prentice Hall, 2004. ISBN 0-130-35290-X
- Little, M. C.: *A History of Extended Transactions.* InfoQ, 2006. Elérhető: http://www.infoq.com/articles/History-of-Extended-Transactions
- Nadalin, A. és mások: Web *Services Security: SOAP Message Security 1.0.* OASIS Standard, 2004. Elérhető: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

5. Köztesréteg

Az előző fejezeteken láthattuk, hogy a fejlesztőknek számos olyan nem funkcionális – rendszer szintű követelménnyel is foglalkoznia kell, amelyek speciális tudást igényelnek. Nem igazán várhatjuk el egy egyszerű webes szoftver fejlesztőjétől, hogy mestere legyen a hibatűrő rendszerek tervezésének az egyszerű üzenetvesztések kezelésétől (az üzenetek elveszhetnek, késhetnek, mi a rendszer akutális állapota, ha tetszőleges késleltetések, üzenetvesztések lehetnek egy csatornán?) egészen a kauzalitás, vagy a bizánci típusú (amikor egy kommunikáló csoportban ahol szavazásos konszenzust akarunk elérni valakik tetszőlegesen szabálytalanul működhetnek) hibák kezeléséig. A klasszikus operációs rendszerek nem nyújtanak megoldásokat e problémák kezelésére. A képességeik kimerülnek a nyers hálózati kapcsolatok (UDP/TCP) és a processzek megfelelő kezelésében. Szükség van tehát egy rétegre az operációs rendszer és az alkalmazás között, ami ezeket a problémákat megfelelő absztrakciós szintre emeli annak érdekében, hogy a programozó átlássa, és tudjon valamit kezdeni vele. Mint majd látjuk ez nem feltétlenul a **futtató környezet (runtime)**, hanem egy attól esetenként független réteg, amit **köztesrétegnek (middleware)** nevezünk.

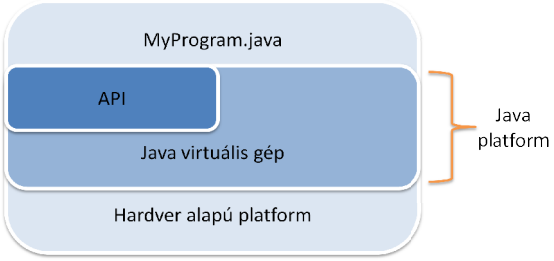
5.1. A köztesréteg fogalma, eredete

A **köztesréteg (middleware)** mint olyan először egy 1968-as konferencián jelent meg. A réteg bevezetésének célja a különböző fájlrendszerek szemantikájának és szintaktikájának elrejtése volt az alkalmazás elől. A köztesréteg helye napjainkban az operációs rendszer és az alkalmazás között található, de funkcionalitása/fókusza jelentősen módosult.



10. ábra: A köztesréteg helye egy elosztott rendszerben

A köztesréteg feladatát ma sokan a különböző alkalmazások együttműködésének elősegítésében látják, más tartományban a köztesréteg a skálázhatóság, elosztottság problémáit fedi el a fejlesztő elől. Abban talán egyetértenek a különböző irányok képviselői, hogy a köztesréteg egy magasabb absztrakciós szint segítségével segít az elosztottság megfelelő kezelésében (legyen ezt a heterogenitás, mint dimmenzió, vagy a lokáció, mint dimmenzió). Ezzel a definícióval el tudjuk választani attól a **futtató környezettől (runtime environment)** amely egy héjat ad a program köré, amin keresztül különböző nem csak elosztottsággal kapcsolatos problémákra magasszintű eszköztárat nyújt. A JVM mint olyan tehát egy **futtató környezet,** de nem köztesréteg mivel maga a JVM csak lehetőséget ad az elosztottság kezelő keretrendszer beillesztésére de megoldást nem. Az EJB konténer viszont már köztesrétegnek nevezhető mivel lehetőséget ad számos köztesréteg szolgáltatás használatára.



11. ábra: A JVM mint futtató környezet

A köztesréteg, mint láttuk több dimmenzió mentén több paradigm mentén ad egy magasabb absztrakciós szintet az elosztottság kezelésére. A következő fejezetben áttekintjük azokat az alapvető paradigmákat, amelyek mentén egy köztesréteg szolgáltatásokat nyújthat.

5.2. A köztesréteg alap típusai

A köztesréteg egyes típusai a konkrét megvalósításoknál nincsenek ilyen atomi szinten elkülönítve mivel ezeket együtt érdemes alkalmazni. Itt az egyes paradigmákat írjuk le:

- **Üzenetorientált Köztesréteg (Message Oriented Middleware)**: Segítségével üzenet alapú kommunikáció valósulhat meg. Az alapvető absztrakciói a csatornák, források, előfizetők és az üzenetek. Pont-Pont, Pont-Több Pont kommunikációs paradigmákat egyaránt megvalósíthatunk vele. Alapvetően aszinkron kommunikációt valósít meg, azaz egy üzenet elmegy, akkor a program futhat, tovább nem várja meg a választ. Az üzenetek viszik át a különböző kontextusokhoz tartozó szolgáltatásokat, a legtöbb megvalósításban ez valamennyire automatizált (pl.: az üzenet küldője megfelelő módon átkerül a fogadó oldalra is és ott az üzenet kezelése megfelelő jogosultsággal zajlik le.)
- **Csoportkommunikáció Köztesréteg (Group Communication Middleware)**: Bár ez is üzenet kommunikáción alapuló kommunikációt valósít meg a cél gyakran jóval összetettebb mint a sima üzenetküldésnél. Olyan primitíveket támogathat, amelyek segítségével például megvalósítható az elosztott konzisztenica (pl.: szavazás alapú többségi döntés). Ezen köztesréteg tipikusan nincs kiajánlva a fejlesztőnek, hanem valahol mélyebb rétegekben jelenik meg (egy példa erre a JBoss JGroups ami a klaszter alapja, vagy a Google elosztott zárolást megvalósító rétege ami a BigTable alapja).
- **Objektum Köztesréteg (Object Oriented Middleware)**: Segítségével az objektum orientált szintaktika és szemantika megtartásával virtualizálható a konkrét futtatás helye. Megoldja tehát az objektumok fellelését, az esetleges replikálásból eredő konzisztencia problémákat és a távoli metódushívás problematikáját. A Corba és a klasszikus JEE idején élte fénykorát, amikor az elképzelés az volt, hogy a köztesréteg a terhelés vagy egyéb metrika figyelembevételével tetszőlegesen helyezgetheti, az objektumokat ez nem zavarja a rendszert. Ma is megtalálható ez a szolgáltatás az elosztott gyorsító tárakban de a teljesítménybeli igénye miatt megfontoltan kell használni.
- **Távoli Eljárás Köztesréteg (Remote Procedura Call Based Middleware):** Az előző paradigma alap építőköve. A távoli eljárás hívás transzparens megvalósítására koncentrál. Ez azonban a gyakorlatban nem igazán valósul meg mivel a távoli eljá-

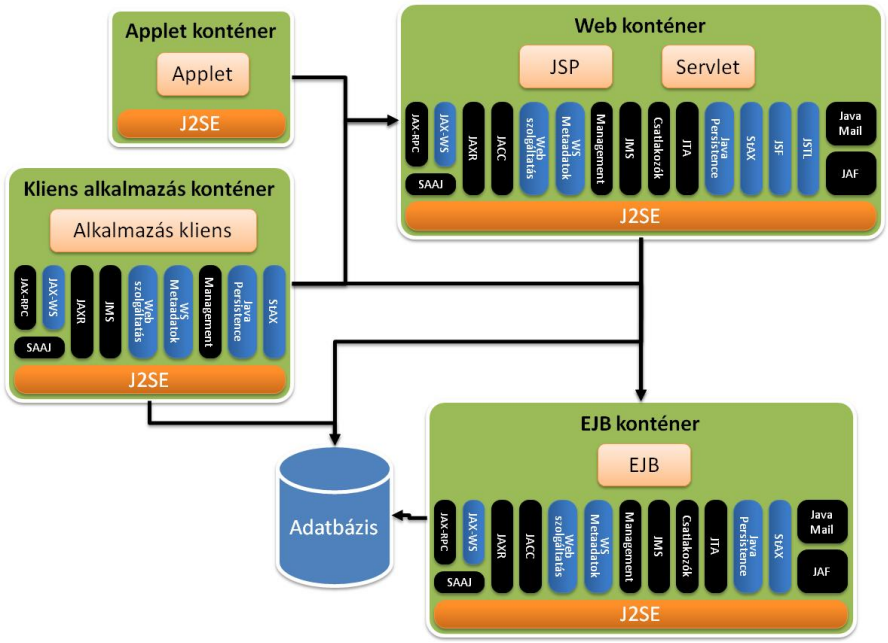
ráshívásban küldött és kapott objektumok tipikusan érték és nem referencia szerint adódnak át. Erre a programozónak oda kell figyelnie. Alapvető absztrakciói a **csonk**, amely a hívó oldalon elfedi a távoli objektumot és a **váz** amely a távoli objektumnál elfedi a hívó objektumot.

- **Adatbázis Köztesréteg (Database Based Middleware)**: Feladata a különböző adatbázis kezelők sajátosságainak elfedése az alkalmazás elől. Sokan ezt nem tartják köztesrétegnek. A JDBC például ezt valósítja meg.
- **Tranzakció-Kezelő Köztesréteg (Transaction Processing Middleware)**: A feladata az erőforrások konzisztens állapotba tartása tranzakció-kezelés támogatásával. Az erőforrások tipikusan adatbázisok, de lehetnek más adattárak is (pl.: fájlok). Az ACID tulajdonságok biztosítására az **erőforrás kezelők** (pl.: adatbázis) a **tranzakció kezelők** segítségével kétfázisú (vagy egyéb típusú) tranzakciókat tudnak végrehajtani. Ezen rendszerek az üzenet orientált köztesrétegekkel együtt régóta a robosztus infrastruktúra alapjait képviselik.
- **Portál Köztesréteg (Portal Middleware)**: Ez egy új kategória feladata szemben az előzőekkel ahol tipikusan a **háttér rendszerek (back-end)** voltak valamilyen módon integrálva az **előtét rendszer (front-end)** integrálása. A portálok lehetőséget adnak az egyes más-más helyről származó GUI elemek integrált egy felületben történő integrálására, úgy, hogy közben külön kontextust biztosítanak a portál szintjén lehetővé téve a portálon megjelenő elemek interakcióját. Alapvető absztrakció a portlet amely vagy távol, vagy a portálon fut és ez fér hozzá a portál kontextushoz (pl.: felhasználó azonossága, …).
- **Felhő köztesréteg (Cloud Middleware)**: E réteg feladata az előző fejezetekben említett CAP kritériumok valamilyen párosításának megvalósítása. Erre a köztesrétegre a nagyon nagyfokú skálázhatóság a jellemző nagyon nagy adatmennyiségek mellett. Egy példa erre az Amazon Dynamo köztesrétege amely egy lapos név érték tárat biztosít.

5.3. A köztesréteg megvalósítása

Maga a köztesréteg lehet **implicit** vagy **explicit**. Az **explicit köztesréteg** egy olyan absztrakt programozói interfészt ad, amit a programozónak kell explicit módon hívogatnia amikor a szolgáltatásait igénybe szeretné venni. Az **implicit köztesréteg** hasonló szolgáltatásokat nyújthat mint az explicit csak ez esetben a programozónak csak a saját üzleti logikájának megvalósításával foglalkozik és nem vesz tudomást a köztesrétegről mert ez számára láthatatlan. A szolgáltatásokat nem API hívásokkal ú.n.: kézi vezérléssel veszi igénybe, hanem automatikusan a szükséges ponton a keretrendszer ezt az alkalmazás logika részéve varázsolja. Ezen „varázslás” majd később látni fogjuk, megtörténhet interceptorokkal, IoC konténerekkel, de Aspektus Orientált megoldásokkal is.

Egy köztesréteg-szabvány a Java EE, ami nem egy konkrét köztesréteget definiál, hanem pontosan azt a modellt és felületet, amibe bele tudjuk illeszteni az alkalmazásainkat, szolgáltatásainkat. A szabvány továbbá ajánlásokat fogalmaz meg a konkrét köztesréteg-implementációkra vonatkozóan, de egy sereg dolgot nem szabályoz. Az ilyen nem szabályozott dolgokat a köztesréteg gyártói úgy valósítják meg, ahogyan jónak látják. A Java EE szabvány által definiált magas szintű modellt láthatjuk az alábbi ábrán:



12. ábra

A JavaEE modell egyes elemeivel gyakran fogunk találkozni a jegyzet hátralévő fejezeteiben.

5.4. A köztesréteg feladatai, lehetőségei

A fentiekből következik, hogy egy megfelelő köztesréteg rengeteg terhet levesz a fejlesztők válláról, és az elosztott alkalmazásaink kevesebb hibával rendelkező és szabványos megoldásokat használhatnak ilyen módon. A köztesréteg jóságát az általa biztosított transzparencia képességekkel és ezek mértékével mérhetjük. Az ANSA 1989, ISO/IEC 1996 International Standard on Open Distributed Processing testület foglalta össze ezeket a transzparencia tulajdonságokat az alábbi csoportokba:

- **helyszín áttetszőség**: nem lehet megállapítani azt, hogy akivel kommunikálunk hol van. Transzparens a másik entitás (pl.: objektum) elhelyezkedése. Ilyen képességet ad részben pl. az RMI (a referenciák kezelésében nem tudja teljes mértékben kielégíteni ezt a követelményt).
- **hozzáférés áttetszőség**: ugyanolyan szintaktikával, szemantikával érjük el a helyi és a távoli erőforrásokat is. (ez gyengébb, mint az előző mert pl. az RMI esetén ugyanaz az interfész, de attól függően, hogy a távoli objektum exportált-e vagy sem máshogy működnek a referenciák)
- **replikáció áttetszőség**: adott erőforrás több helyen is megtalálható, nem lehet megállapítani, hogy mikor melyik replikával kommunikálunk. (Ezt nyújtja a klaszterezés, amikor az objektumok memóriabéli állapotát is replikálja)
- **hiba áttetszőség**: a bekövetkezett hibák nincsenek kihatással a rendszer működésére (ez alacsony szinten lehet pl.: TCP, magasabb szinten megfelelő kivételkezelés)

- **párhuzamosság áttetszőség**: az erőforrásokat többen is használják egy időben egymástól függetlenül. Ezt a klasszikus java stack esetén szál kezeléssel lehet megoldani. J2EE esetén ez pl.: a konténer dolga (hogyan kezeli a viszony babokat, ...)
- **migráció áttetszőség**: adott komponens tetszőleges mozgatható a különböző futtató környeztek között. Ez volt a J2EE egyik célkitűzése amit nem igazán tudott megvalósítani, a felhő vagy rács infrastruktúra valamilyen mértékben megvalósítja ezt (megfelelő granuralitás esetén).
- **teljesítmény áttetszőség**: a köztesréteg automatikusan skálázódik. Amennyiben több erőforrásra van szükség akkor többet allokál. Ezt ma leginkább a felhő megoldások tudják produkálni. Ott is leginkább az IaaS esetén (pl.: a Google PaaS rengeteg olyan nem dokumentált limit értékkel rendelkezik ami nem teszi lehetővé a skálázást)
- **programozási nyelv áttetszőség**: a köztesréteg ugyanazt nyújtja a különböző programozási nyelveknek és közöttük hidat képez. Erre egy jó példa a .NET platform és a hozzá kapcsolódó keretrendszerek.

A következő fejezetben néhány ismertebb köztesréteget tekintünk át és helyezzük azt el a szolgáltatástérképen.

5.5. Köztesréteg példák

Az előző fejezetekben felsorolt paradigmákohoz illeszkedő azokat megvalósító, legismertebb köztesrétegeket tekintjük át ebben a fejezetben. A JEE-vel kapcsolatos köztesrétegekkel és az ESB-vel később jóval részletesebben is megismerkedünk.

5.5.1. JGroups

A JGroups egy klasszikus csoportkommunikációs köztesréteg megvalósítás, amely kihasználva a hálózat képességeit is (pl.: multicast) különböző képességű csoportkommunikációs megoldásokat biztosít. A JGroups képezi a JBoss EJB és JBoss Web tároló klaszterezésének alapját is. Alapvetően a csoportok kezelését biztosítja, de a csoportkommunikációt alkotó üzenetkezelésnél számos megbízhatósági szintet tud megvalósítani (Atomi, FIFO, Kauzális, Teljes Sorrend). Ezt TCP/UDP mellett JMS igénybevételével is biztosítani tudja. Alapvető elemei a csatorna és a csatorna alatt lévő protokoll halmaz. Egy-egy csatorna jelenti azt a virtuális médiumot amelyen keresztül a sima kommunikáción túl olyan absztrakt elosztott az alatta lévő protokollok által biztosított szolgáltatások is elérhetőek mint az elosztott zárolás, többségi szavazás… Ezekkel mint már írtuk a programozó ritkán találkozik mert elfedi ezt például az elosztott objektum gyorstár (pl.: Infinity) vagy az elosztott objektum címtár (pl.: JNDI).

5.5.2. JEE Web Tároló

A Web Tároló (Web Container) a JEE által specifikált legnépszerűbb tároló, nem véletlen, hogy a GWT is ezt ajánlja mint futtató környezetet. Alapvetően egy futtatási környezetet biztosít a http kérések lekezelésére. Ezen felül bizonyos mértékig lekezeli a párhuzamossággal kapcsolatos problémákat az adatbázis rétegre támaszkodva. Amiért köztesrétegként itt szerepel az az, hogy van klaszter szolgáltatása is amelyben például a viszony kon-

textusban tárolt adatokat konzisztens módon egy vagy több helyre replikálja lehetővé téve ezzel a robosztus rendszerek kialakítását. Mint már írtuk e szolgáltatás mögött a JGroups áll (a JBoss megvalósítás esetén, más megvalósítás pl.: IBM WebSphere más csoportkommunikációs megvalósítást használ)

5.5.3. JEE EJB Tároló

Az EJB tároló értelmét gyakran megkérdőjelezik megemlítve azt, hogy elegendő a Web tároló is a szolgáltatás réteg (vagy üzleti logika) megvalósítására. Ez egyszerű alkalmazás esetén lehet, hogy így van de olyan komplexebb alkalmazásnál ahol a tranzakciók több adatbázis kapcsolaton is túlnyúlnak és nem adatbázis hanem Java kód szinten kellene ezeket végrehajtani az EJB konténer kikerülhetetlen. Hasonló a biztonság is: amikor részletes esetleg metódus szintű jogosultságkezelést szeretnénk akkor szintén az EJB tárolót célszerű használnunk. A különböző babok által biztosított absztrakciók, amelyekkel szinkron vagy aszinkron kommunikációt tudunk megvalósítani ismét az EJB-t igényelnek. Amennyiben komponens orientált fejlesztést szeretnénk ismét az EJB konténer adja meg hozzá a megfelelő kereteket (komponensek definiálása, monitorozása, menedzselése, …). A web konténerhez hasonlóan itt is tudunk klaszterezni, de ezen túl támogatva van az üzenet orientált, objektum és távoli eljárás köztesréteg paradigmák megvalósítása is.

5.5.4. Google AppEngine

A **share nothing (csak adatbázis szinten van szinkronizácó)** elvét követve autonóm elemekből építkezik (amit már láttunk a tér alapú architektúrában a **PU – Processing Unit** absztrakciót), ahol a konzisztencia, állapotok szinkronizációja a Bigtable feladata amely a CAP kritériumok BASE permutációját valósítja meg, azaz partíciótűrő és magas rendelkezésre állású a konziszetnica rövid távú kárára. Maga a már említett feldolgozó egység (processing unit) a web konténer által definiált futtató környezetre építhet, ez alatt pedig a Java virtuális gépre. Egy-egy ilyen web alkalmazás képezi a terheléselosztás alapját. A Google köztesréteg ezen web alkalmazásokat helyezi el annyi virtuális gépre amennyi szükséges. Ezen web alkalmazások között a konzisztenciát a felhő adattár Google implementációja a BigTable valósítja meg.

5.5.5. ESB

Az előzőekben látott köztesrétegek alapvetően az alkalmazások kiszolgálásra lettek megtervezve. Az ESB és az SCA alkalmazás rendszerek kiszolgálását valósítják meg. Az **ESB – Vállalati Szolgáltatás Busz (Enterprise Service Bus -ESB)** mint ahogy a nevében is benne van egy busz absztrakciós segítségével valósítja meg a különböző rendszerek közötti kommunikációt. Az ESB tekinthető a SOA egyfajta megvalósításának is. A busz üzenet alapú kommunikációt támogat és képes a különböző helyszíneket virtualizálni (klaszterezés segítségével). Számos absztrakciót támogat melyek az üzenetek irányításával, manipulációjával foglalkoznak. Bővebben a 11-12 fejezetekben beszélünk az ESB-ről.

5.5.6. SCA

Ellentétben az ESB-vel mely az üzenetekre azok manipulálásra koncentrál, az **SCA (Szolgáltatás Komponens Architektúra – Service Component Architecture - SCA)** egy nagyobb szkópot céloz meg. Alkalmazásokat lehet benne definiálni a SOA koncepció men-

tén. Míg az ESB java specifikus (gyakorlatilag a JBI JSR 208 megvalósítása) addig az SCA nyelv független. Az SCA-ban szerelvényeket lehet definiálni melyeket a futtató környezet megfelelő módon virtualizálhat és kezelhet. Ezen szerelvények más szerelvényekre vagy szolgáltatásokra (komponensekre) építhetnek (a SOA paradigma mentén).

5.6. Összefoglaló

E fejezet célja a köztesréteg szerepének, lehetséges típusainak és képességeinek bemutatása volt. A köztesrétegek használata napjaink alkalmazás vagy alkalmazásrendszer fejlesztésekor elkerülhetetlen. A trendek azt mutatják, hogy egyre fontosabbá válik a BASE és az ACID megközelítést is vegyítő rendszerek használata, ezek megfelelő integrációja még nyitott kérdés. A jegyzet következő részében áttekintjük a napjainkban használatos programozási nyelvi paradigmákat mivel ezek megfelelő kiválasztása nagyban hozzájárulhat a fejlesztők produktivitásához.

5.7. Kérdések

1. Mit nevezünk köztesrétegnek?
2. Mik egy köztesréteg legfontosabb feladatai?
3. Hogyan támogatja a köztesréteg az egymással kommunikáló számítási egységeket?
4. Hogyan támogatja a köztesréteg az alkalmazások és szolgáltatások fejlesztőit?
5. Hogyan viszonyulnak a klasszikus értelemben vett köztesréteg-megoldások a mai, modern felhőalapú szolgáltatásokhoz?

5.8. Ajánlott irodalom

- Coulouris, G. és mások: *Distributed Systems - Concepts and Design.* 4. kiadás. Addison-Wesley, 2005. ISBN 0-321-26354-5
- EBU project group on Middleware in Distributed programme Production: *The Middleware Report.* 2005. Elérhető: http://tech.ebu.ch/docs/tech/tech3300.pdf
- Emmerich, W. és mások: *The impact of research on middleware technology.* ACM SIGOPS Operating Systems Review. Volume 41 Issue 1, 2007.
- Jendrock, E. és mások: *The Java EE 6 Tutorial.* Oracle, 2010. Elérhető: http://download.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf

II. rész

Nyelvi paradigmák,

trendek

6. Nyelvi paradigmák, trendek - logika megvalósítása

Az előző fejezetben láthattuk az elosztott rendszerek irányába mutató trendeket és azt a réteget mely az elosztottságot igyekszik elfedni a fejlesztő elől. Ezzel egyfajta magas szintű áttekintést nyertünk a területről, lehetséges stratégiai irányokról. Egy következő absztrakciós szint lehetne, ha a köztesrétegre koncentrálva bemutatnánk annak részletesebb környezetét az alkalmazáson belül és alkalmazás rendszerek szintjén is. Ehhez azonban ismernünk kellene a köztesréteg alatt lévő nyelvi környezetek képességeit, az általuk nyújtott lehetőségeket. Ezen ismeretek nélkül a programozók gyakran esnek abba a hibába, hogy csak egy nyelvi környezetre megoldásmódra koncentrálnak és figyelmen kívül hagyják a programozási nyelvek fejlődésének eredményeit. Jelen fejezet célja tehát mielőtt a következő részekben belemélyedünk a köztesrétegek és a hozzájuk kapcsolódó technológia megoldások, képességek bemutatásába, hogy áttekintsük azokat a megközelítésmódokat, paradigmákat melyeket érdemes ismernünk amikor az üzleti logika vagy az adatábrázolás megvalósításához használható paradigmákat értékeljük.

Annak érdekében, hogy az előbb ismertetett adatábrázolás és üzleti logika mélyebb tartalmát is megismerjük a következő fejezetben a rendszerfejlesztés folyamatának vázlatos bemutatásának segítségével bemutatjuk ezen fogalmak valós az egyes rendszerfejlesztési lépésekhez köthető tartalmát. Ezek után egy gyors áttekintést adunk a programozási nyelvek fejlődéséről, generációiról, majd a leggyakrabban alkalmazott logika leírási és adatkezelési megközelítéseket mutatjuk be. A későbbi fejezetekben látni fogjuk, hogy egy-egy rendszer ezen nyelvek, leírásmódok kombinációja segítségével épül fel.

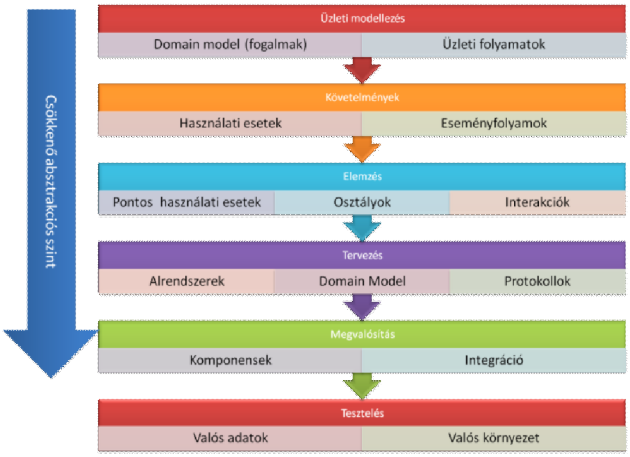
6.1. Rendszertervezés alapok

A **Rational Unified Process (RUP)** egy igen elterjedt és népszerű rendszerfejlesztési metodológia mellyel jelen tárgy hallgatója már megismerkedett a Rendszerfejlesztés című tantárgy keretében. Így itt most csak azon aspektusokat tekintjük át melyek szükségesek a jegyzet megértéséhez.

A RUP a rendszerfejlesztés folyamatát két dimenzió mentén taglalja: az egyik az időbeliséget veszi alapul és négy fázison át valósítja meg a fejlesztést a másik dimenzió ezzel párhuzamos az egyes fázisokban kisebb vagy nagyobb súllyal megjelenő eljárásokra bontja a munkát. Az egyes eljárások során a probléma leírásának absztrakciós szintje egyre alacsonyabb lesz egyre közelebb kerül a konkrét megvalósításhoz alkalmazott környezethez. A fejlesztő produktivitását nagyban befolyásolja, hogy ezen folyamat során mekkora az általa használt eszköztár által biztosított absztrakciós szint. Az adott keretrendszer által biztosított absztrakciós szintet első körben maga a keretrendszer által használt programozási nyelv vagy nyelvek határozzák meg, második körben pedig maga a keretrendszer által megvalósított funkcionalitás.

Az iterációk számának növekedésével a RUP féle folyamat során két egymástól többékevésbé jól elkülönülő területet dolgoz fel. Az egyik terület arról szól, hogy a rendszer milyen adatokkal fog foglalkozni ezeket hogyan ábrázolja, itt milyen kényszerek vannak magából az adatból fakadóan. A másik terület a rendszer működésével az általa megvalósí-

tott funkcionalitással foglalkozik. Eme területek felderítésére megfelelő absztrakciós szinten történő leírására számos módszertan, modellező eszköz, megközelítés létezik.

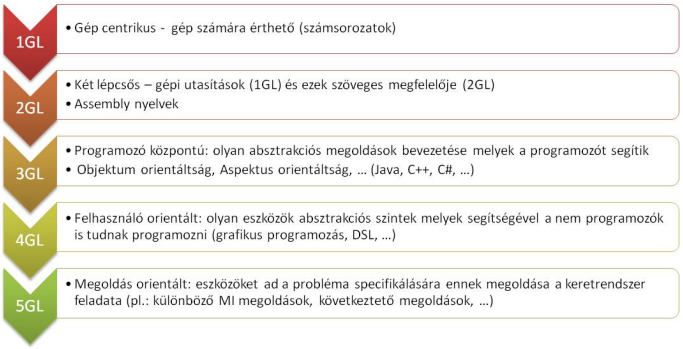


13. ábra

A RUP eljárás lépéseit a 13. ábra szemlélteti. Az egyes lépések alatt a legtöbb esetben jól elkülöníthető az adat és a logika ábrázolásához használt absztrakciós szint. Az üzleti modellezés esetében a feladat a konkrét céltartomány megértése, leírása úgy a folyamatok mind a kezelt adatok, fogalmak szempontjából. A követelmények kidolgozásánál már egy absztrakciós szinttel alacsonyabban (a szoftverhez viszonyítva) kiválasztott használati esetekkel és ezen használati esetek közötti eseményfolyamokkal foglalkoznak. Az elemzés fázisában a használati eseteket precízebb, pontosabb leírása segítségével absztrakt osztályokat képeznek és ezek interakcióját pontosítják. A tervezés folyamán az absztrakt osztályokból egyrészt konkrét **domain model** másrészt alrendszerek és ezen alrendszerek közötti protokollok lesznek. Ezen elemek kerülnek megvalósításra és integrálásra. Mint láthatjuk nem mindegy, hogy a kiválasztott nyelv, felhasznált fejlesztő eszköz milyen absztrakciós szint nyújtására képes. A továbbiakban ismertetett tartomány specifikus nyelvek (**Domain Specific Language - DSL**) vagy folyamat nyelvek (**Process Language**) képesek arra, hogy a felhasználó saját maga alakíthassa az üzleti logikát, vagy az üzleti logika igen közel legyen a felhasználó által adott specifikációhoz. Ezen eszközök megközelítések igen jelentős fejlesztői produktivitás növekedést képesek nyújtani. A következőekben röviden áttekintjük a programozási nyelvek fejlődését és az egyes generációk képességeit.

6.2. A nyelvi környezet melyre építhetünk

Egy-egy konkrét programozási nyelvet a szintaxisával (a különböző utasítások és azok megengedett kombinációi) és szemantikájával (a nyelvre jellemző utasítások jelentése) írható le. A programozási nyelvek fejlődésénél a fejlesztők produktivitásának növelése az egyik fő mozgatórugó. Mivel a számítási, tárolási kapacitás folyamatos növekszik ezért a teljesítmény és hatékonyság nem igazán jelent meg mint fontos szempont.

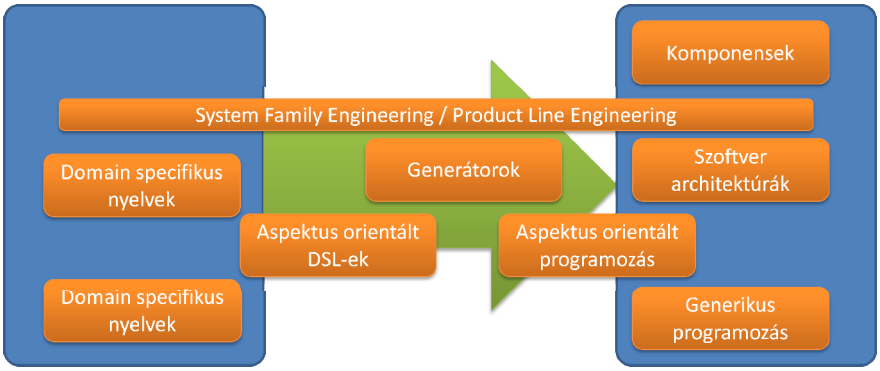


14. ábra

Az 14. ábra látható a különböző programozási nyelv generációk egymásra épülése és a köztük lévő szintlépések rövid leírása.

- Az **első generációs (1GL)** programozási nyelvek minden különösebb absztrakció nélkül a gépi kódot biztosítottak a programozóknak. Végső soron ez fut a különböző processzorokon, de ma már sokkal magasabb absztrakciós szinteken folyik az, algoritmusok adatstruktúrák specifikálása.
- A **második generációs (2GL)** programozási nyelvek a számkódok helyett emberi szemnek jobban áttekinthető és jobban megérthető parancsszavakat használnak. Ilyen megoldásokat ma csak azokon a helyeken alkalmaznak ahol a sebesség kritikus. A második generációs nyelveket tulajdonképpen az assembly nyelvek jelentik.
- **A harmadik generációs (3GL)** programozási nyelveket használják ma leginkább a szoftverfejlesztésben. Ezen nyelvek célja olyan absztrakciós megoldások, eszközök létrehozása melyek segítségével a szoftverfejlesztők magasabb szinten tudják modellezni úgy az algoritmusokat mint az adatokat. Ilyen nyelvek pl.: a Java, C, C++, C#, ... Itt jelentek meg azok a képességek is, hogy a különböző modulokba, vagy más struktúrákba lehessen szervezni egy-egy megoldás kódbázisát. Napjaink legelterjedtebb absztrakciós paradigmája az objektum orientáltság. mely bár hatékonyabb mint az elődei, azonban a tapasztalat azt mutatja, hogy az újrafelhasználhatóság, karbantarthatóság de még a kifejező képesség területén sem túlzottan hatékony. Az osztályok melyek az újrafelhasználható elemkészlet alapjait képezik túlzottan kicsinek bizonyultak. A különböző keretrendszereket nehéz együttműködésre bírni, így a produktivitás messze elmarad a kívánatostól. A komponensek használata a megfelelő kontraktusok mentén specifikált interfészekkel ugyan segítené az újrahasznosítást de minél nagyobb egy komponens annál kevésbé újrahasznosítható. A tervezési minták ugyan kellően általánosak ahhoz, hogy újrahasználhatóak legyenek, de nem implementációk, csak absztrakt vázak amelyek így nehezen beépíthetők. A kutatás és a gyakorlat azt mutatja, hogy jelentős produktivitás növekedés csak paradigma váltással érhető el.
- A **negyedik generációs (4GL)** programozási nyelvek célja az absztrakciós szint további növelése és a hangsúlyt a nem programozók irányába mozdul el. Több párhuzamos ág is kialakult. Egy ilyen a szoftver termék családokban gondolkodó (**System Family Engineering, Product Line Engineering**) megközelítés mellyel az adott probléma tartományra szánt megoldások fejlesztésénél kihasználják a

szinergiákat. Az új rendszer variánsok gyorsan elkészíthetőek mivel újrafelhasználható elemekből építkeznek (közös architektúra, komponensek, stb.), a különbségeket különböző szisztematikus megoldásokkal kezelik. A Generatív szoftver fejlesztés (**Generative Software Development**) a rendszer család fejlesztés megközelítésen alapul és a rendszer családtagok automatikus létrehozására fókuszál. A generálás alapja a tartomány specifikus nyelven megadott leírás. A 15. ábra egy áttekintést ad a különböző technológiákról, módszerekről a szoftver generálás területén. A terület specifikus nyelvek (**Domain Specific Language**) olyan testreszabott nyelvek melyek az adott alkalmazás tartomány igényei szerint lettek kialakítva. Tartomány specifikus jelöléseket és absztrakciókat használnak az adott tartomány tudásának reprezentálására.



15. ábra: Program Család Generálás áttekintése

- Az **ötödik generációs (5GL)** programozási nyelvek, környezetek célja, hogy egy újabb absztrakciós szinttel ne a probléma megoldására, hanem annak specifikálására fókuszáljanak a fejlesztők, felhasználók. A megoldás keresése a keretrendszer feladata. A különböző MI megoldások tartoznak ebbe a kategóriába. De ebbe a családba tartoznak a különböző következtetéseket adó megoldások is.

Összefoglalva tehát láthatjuk, hogy a fejlődés a produktivitást szolgálja. Ma leginkább a 3GL nyelvek vannak használatban és néhány helyen megjelennek a 4GL megoldások is. Az 5GL megoldások ma még szinte csak az akadémiai szférában léteznek, de egyes elemei már megjelennek a gyakorlatban is. A továbbiakban áttekintjük a logika leírására használt három legelterjedtebb megközelítésmódot majd hasonló módon az adatleírásra használt három legelterjedtebb megközelítésmódot.

6.3. A logika modellezése, megvalósítása

Az üzleti logika szorosan összefügg ugyan az adatok kezelésének módjával de gyakran túlmutat azon. Amikor például nem egy, hanem nagyszámú elem, objektum viselkedését írjuk le, vagy olyan elemekét melyek nincsenek konkrétan megvalósítva, hanem csak absztrakt szinten jelennek meg a rendszerben (felhasználói interakció, rendszerek közötti interakció, ...) akkor már nem csak az adatmodell és annak helyességével foglalkozunk hanem az üzleti vagy felhasználói logikával. Az **üzleti logika** esetünkben hosszabb folyamatokat és több együttműködő entitást jelent míg a felhasználói logika tulajdonképpen az üzleti logika egy lehetséges felbontása mely úgy időtartamát mind a résztvevő entitások és hatókörét tekintve is kisebb tartományt fed le.

Az előző áttekintésből látható volt, hogy ma a 3. generációs programozási nyelvek vannak döntő többségben alkalmazva, míg egyes esetekben már megjelennek a 4. esetleg részben 5. generációs képességeket is felvillantó nyelvek. Maga az adat kezelés/modellezés nincs minden esetben szigorúan elkülönítve az logikától, esetenként ez csak tervezési minták szintjén jelenik meg. Mi a kiválasztott cél tartományunkban (SOA + Klaszter) azokat a paradigmákat mutatjuk be melyek napjainkban népszerűek illetve várhatóan népszerűek lesznek a közeljövőben. Az **Objektum Orientált** paradigmával kezdjük majd ezen paradigma hiányosságait bemutatva kitérünk az ú.n.: **POP - Post Object oriented Paradigm** azaz az Objektum Orientált Paradigmákon túllépő azokat kiegészítő paradigmákra, megközelítésekre (AOP, dinamikus nyelvek). Majd két teljesen más csak a logikára koncentráló 4. illetve részben 5. generációs paradigmát tekintünk át (Processz nyelvek, Szabály nyelvek)

6.3.1. Objektum orientált és POP megközelítések a logika leírására

Az objektum orientált szoftver tervezési fejlesztési paradigma ma általánosan elfogadott és szinte minden modern programozási nyelv valamilyen mértékben épít erre a paradigmára. A paradigma lényege már ismert az olvasó előtt, itt csak röviden ismertetnénk a lényegét. A logikát és az adatkezelést is objektumok formájában ábrázoljuk. Egy-egy objektum egyszerre foglalkozik az adatok ábrázolásával és a viselkedéssel is. Az objektumok között származási hierarchiákat lehet kialakítani az általánostól a specifikusabb objektumok felé. Mint azt már említettük nincs elkülönítve az adatmodellezés a logikától, így a származási hierarchia is együtt kezeli a viselkedést (metódusok formájában) és az adat modellezést (adattagok formájában). Annak ellenére, hogy a sima procedurális megközelítéshez képest nyújtott magasabb absztrakciós szintje nagy előrelépést jelentett számos olyan terület van ahol a mai OO paradigmát megvalósító nyelvek nem nyújtanak megfelelő produktivitást nyújtó megoldást. Ilyen a már az első részben említett ú.n.: általános vagy keresztülívelő problémák kezelése, vagy a dinamikusabb, időben változó objektum szerkezetek kialakítása. Egy naplózást, vagy biztonság kezelést gyakran rendszer szinten szeretnénk megadni, de előfordul, hogy ez rendszer modulonként változik, ennek a kezelésére ugyan használható az öröklődés is, de ez nem kellően dinamikus. Ezekre a kihívásokra az OO paradigmát továbbfejlesztve egyrészt a **dinamikus nyelvek** másrészt az **Aspektus Orientált** paradigma próbál megoldást nyújtani.

Az Aspektus Orientált paradigma segítségével az objektumok képességei (tárolt adatok, viselkedés) paraméterezhetőek, dinamikusan változtathatóak. Az ú.n.: keresztülívelő problémák kiszervezhetőek akár rendszer akár modul szinten aspektusokba és az objektumok viselkedése ezen aspektusokkal egészül ki akár helyszíntől függően is.

A dinamikus nyelvek ezt a viselkedést viszik egy lépéssel tovább úgy, hogy az objektum viselkedés illetve adattagjai bárhol megváltoztatható.

Mint láthattuk az Aspektus Orientált és a Dinamikus nyelvek az Objektum Orientált paradigmát egészítik ki, de magával a logika és az adatábrázolás elkülönítésével nem foglalkoznak, ez a programozókra hárul akik ezt tervezés minták illetve keretrendszerek segítségével tehetik meg. Szintén látható az, hogy ezen paradigmákat megvalósító nyelvek többsége a 3. generációs nyelvek családjába tartozik, mivel nem lép túl az objektum orientáltságon mint absztrakciós szinten és ezzel nem igazán alkalmas arra, hogy a programozókon kívül más is alkalmazza. A következőekben két olyan paradigmát, megközelítést tekin-

tünk át amelyek a logika megadására koncentrálnak és ezt 4. illetve részben 5. generációs megközelítésmóddal teszik meg.

6.3.2. Folyamatok, folyamat nyelvek

A munkafolyamat leíró vagy folyamat leíró nyelvek nem számítanak forradalmian újnak, már a múlt század 80-as éveinek elején is voltak ilyen kezdeményezések. Ekkor azonban a szoftver fejlesztést az adatközpontú világnézet uralta ahol az adatok voltak a fejlesztők fókuszában és nem a folyamatok. Napjainkban értünk egy olyan trendfordulóhoz ahol már kellő intelligencia található a különböző keretrendszerekben ahhoz, hogy a folyamatok álljanak a fejlesztők, sőt az adott tartomány szakértői azaz a nem szoftver fejlesztők figyelmének központjában. Napjaink munkafolyamat leíró nyelvei általában grafikus megjelenítéssel rendelkeznek és magát a folyamatot megfelelő fejlesztő eszközök segítségével nem programozó is meg tudja tenni. Ilyen típusú folyamat leíró nyelvekkel találkozhatunk úgy a hagyományos üzleti folyamatok szintjén (pl.: BPEL, JBPM, ...) de találkozhatunk velük a felhasználói folyamatok szintjén is (pl.: page-flow). Ezen megoldások segítségével a logika dinamikusan változtatható akár futásidőben is.

A folyamat leíró nyelvek leginkább munkafolyamat kezelő rendszerekben jelennek meg. Ezen rendszerek feladata az üzleti vagy felhasználói folyamatok modellezése, futtatása, monitorozása munkafolyamat modellek segítségével. Munkafolyamat definíciók (**munkafolyamat séma - workflow scheme**) segítségével definiálhatóak az aktivitások és a közöttük logikai relációk. Az aktivitás a munka egy atomi eleme. A munkafolyamat definíciókat hasonlóan az OO osztályokhoz példányosítani kell a futtatáshoz. Az egyes aktivitások a munkafolyamaton belül átmenetekkel vannak összekötve. A konkrét aktivitás sorozatok egymással párhuzamosan is futtathatóak. Az aktivitásokat különböző szerepkörű humán vagy gépi entitásokhoz rendelhetjük és esetenként magát az aktivitást is ezek az entitások végzik el. Adatkezelés szempontjából két adattípust szoktak megkülönböztetni: az egyik a vezérlő információ amelyet a munkafolyamat egyes csomópontjaiban szükséges döntésekhez használjuk és tipikusan a munkafolyamat futása alatt léteznek míg a másik adattípus a produkciós adat melyben olyan objektumok vannak melyek léte nem függ a munkafolyamattól.

A munkafolyamat nyelvek képességeinek megértéséhez talán a munkafolyamat minták áttekintése a legmegfelelőbb módszer. Wil van der Aalst összefoglalta a különböző folyamatleíró nyelvekben megvalósított mintákat. Ezeket tekintjük át röviden:

- Egyszerű vezérlő minták: azon minták melyek a folyamat vezérlés elemi műveleteit írják le:
    - Szekvencia: egy munkafolyamaton belül egy aktivitás akkor kerül aktív állapotba, ha az előtte lévő aktivitás befejeződött.
    - Párhuzamos elágazás: a munkafolyamat olyan pontja ahol az egy szálú végrehajtás több párhuzamos szálra ágazik.
    - Szinkronizálás: A munkafolyamat olyan pontja ahol több alfolyamat/ /végrehajtási szál egyesül egy folyamattá/végrehajtási szállá egyesülnek így szinkronizálva a végrehajtásukat.
    - Kizárólagos választás: egy olyan döntési pont ahol valamilyen döntési elv alapján egy végrehajtási ágat választanak ki a több lehetséges közül.

  - Egyszerű egyesítés: több folyamat/végrehajtási ág egyesül egy végrehajtási ágba szinkronizálás nélkül, azaz bármelyik szál teljesül az indikálja az új közös szál indítását.

- Összetett elágazás és szinkronizálás minták: a kötegelt kezelés és a szinkronizáció témakörében fellelhető bonyolultabb minták:
  - Többes választás: a munkafolyamat egy olyan pontja ahol egy vagy több végrehajtási száll indul adott döntés meghozatala után.
  - Szinkronizációs egyesítés: a munkafolyamat egy olyan pontja ahol több végrehajtási szál is egyesül egy végrehajtási szállá. A párhuzamos útvonalaknál szükséges a szinkronizáció míg az alternatív útvonalaknál nem.
  - Többes egyesítés: egy olyan pont a munkafolyamatban ahol több ág egyesül szinkronizáció nélkül. Az új ág annyiszor indul el ahányszor a beérkező ágak befejeződnek.
  - Megkülönböztető: egy olyan pont a munkafolyamatban amely egy beérkező ág teljesülése után indítja tovább a szálat. Ezután megvárja amíg minden beérkező szál teljesül és ezeket figyelmen kívül hagyja. Miután mindegyik teljesül újra a számláló állapotba kerül és kezdődik a tevékenysége előről.
  - N-ből M csatlakozás: M párhuzamos ág egyesül egy ágba, hasonlóan az előzőhöz N ág teljesülése után továbbmegy és minden ág teljesülése után újra N ág teljesülése után továbbmegy.
- Strukturális minták: olyan minták amelyek a munkafolyamat modellek struktúráját határozzák meg
  - Tetszőleges ciklus: a munkafolyamat egy olyan pontja ahol egy vagy több aktivitást többször is végre kell hajtani.
  - Implicit befejezés: egy adott végrehajtási ágat akkor kell befejezni amennyiben már nincs végrehajtandó aktivitás, a munkafolyamatot pedig akkor amikor olyan aktivitás amely aktív vagy aktívvá tehető (és a munkafolyamat nincs versenyhelyzetben)
- Több példányt is érintő minták: egy adott aktivitásnak több egyszerre futó példánya is lehet
  - Több példány amelyekről már a tervezéskor tudtunk: az adott aktivitás párhuzamosan futó példányairól már tervezéskor tudunk.
  - Több példány amelyekről futtatáskor tudtunk: az adott aktivitás párhuzamosan futó példányainak száma valamilyen futásidőben keletkező adattól függ (még mielőtt az adott aktivitást aktiválnák).
  - Több példány amelyekről nem tudtunk: az adott aktivitás párhuzamosan futó példányainak száma valamilyen futásidőben keletkező adattól függ (csak az aktiválás után derül ki).
  - Több példány érintő szinkronizáció: Egy aktivitás több példányban is létezik és ezek mindegyikének teljesülése után kell új aktivitást indítani.
- Időbeli relációk: két vagy több aktivitás közötti viszonyt kifejező minta család

    - Átlapolt szekvencia: a sima szekvenciális végrehajtástól különböző időbeli viszony. (akkor kezdje, ha a másikat elkezdte, nem fejeződhet be előbb mint a másik ,...)

- Állapot alapú minták: a rendszer állapotkezelő képességét leíró minták
    - Elhalasztott XOR elágazás: a lehetséges végrehajtási ágak közül egyet választ de csak akkor amikor azt elkezdi végrehajtani (többet is elindít, de csak a leggyorsabban induló futhat, a többit visszavonja)
    - Átlapolt párhuzamos forgalom irányítás: a végrehajtási ágak egy csoportja kerül végrehajtásra, de a sorrend futásidőben dől el, és egyszerre csak egy fut (egy példányon belül).
    - Mérföldkő: adott aktivitás csak akkor indul el, ha a rendszer adott állapotban van. Ezt tipikusan egy mérföldkő elérését jelenti és azt, hogy ennek az ideje még nem járt le.
- Visszavonási minták: az aktivitások futási engedélyének visszavonását leíró minták
    - Aktivitás visszavonás: egy futásra váró szál futási engedélyét visszavonjuk.
    - Teljes munkafolyamat példány visszavonás
- Munkafolyamatok közötti szinkronizáció: a csoportba a különböző munkafolyamat példányok vagy a különböző munkafolyamat példányai közötti szinkronizációt leíró minták tartoznak
    - Üzenet alapú kommunikáció: egy üzenet továbbítsa a az egyik munka-folyamat egyik aktivitásától a másik munkafolyamat aktivitásához.
    - Üzenet koordinálás: a küldő egy választ vár az első fogadótól.
    - Tömeges üzenetküldés: egy adott üzenet küldői egyszerre aktívak: ez többes küldésnek nevezzük
    - Tömeges üzenet kézbesítés: adott üzenet fogadói párhuzamosan futnak.

A fentiekben láthattuk azt, hogy a folyamat leíró nyelvek családját megalapozó minta gyűjtemény a valós üzleti folyamatokból veszi a megvalósítandó mintákat. A különböző munkafolyamat motoroknak vagy nyelveknek egy jó mérőrúdja az, hogy az itt felsorolt mintákból mennyit és milyen szinten támogat. Összefoglalva tehát a munkafolyamat alapú megközelítést ismerhettük meg ebben az alfejezetben. A következő alfejezet egy teljesen más megközelítést mutat amelyet azonban igen gyakran együtt szoktak alkalmazni a folyamat nyelvekkel.

6.3.3. Szabály nyelvek, szabály motorok

Mint láthattuk a tudás ábrázolásának számos módja van. Egy ilyen eddig még nem tárgyalt de egyre gyakrabban alkalmazott mód a szabályokkal történő ábrázolás. Az erre épülő rendszereket Produkciós Szabály Alapú rendszereknek nevezik (**Production Rule Engine**). Ezen rendszerek a bemeneti adatokból (tényeknek nevezik ebben a körben - **Facts**) és a szabályokból (**Rule**) a szabályokban meghatározott eseményeket kimenetet generálnak.

A szabályok leírására számos szabálynyelv, módszer született. A terület egy gyakorlati felosztása következő:

- Szabály modellezés: Az URML nyelv segítségével a szabályok hasonlóan különböző UML nyelvekhez vizuálisan modellezhető.
- Objektum orientált szabály rendszerek: Olyan szabály nyelv implementációk melyek az objektum orientált szemantika, szintaktika segítségével oldják meg az adatok kezelését. A szabály nyelvek is gyakran valamilyen OO nyelvhez hasonló szemantikával vannak megvalósítva.
- Szemantikus web szabály nyelvek: Itt az erőforrások (adatok) hozzáférése az URI-n alapuló eszköztár segítségével tipikusan ontológiák bevonásával lehetséges. A gyakorlatban ez azt jelenti, hogy az adatokat gráf formájában ábrázolják melynek szabályrendszerét az ontológia leírás adja meg.
- MI szabály rendszerek: Azon megoldásokat szokták ide sorolni melyek erős többnyire erős formalizmussal támogatva valamilyen fokú tanulási képességekkel rendelkeznek.

Nehéz átfogó képet adni a különböző megoldások képességeiről mivel nincs egy olyan közös mérce amellyel mérhető lenne egy-egy szabálynyelv tudása, képessége ezért a továbbiakban egy a népszerű platform a **Drools** képességei segítségével mutatjuk be a szabály nyelvek lehetőségeit (ez egyébként nagyrészt követi a JSR-94-es ajánlást).

Mielőtt kitérnénk magára a nyelvre fontos megemlíteni a szabály motor két működési módját mert ezek kihatással lesznek például a deklarált változókra:

- Állapotmentes működés: ekkor az adatállomány melyre a szabályok ki lesznek értékelve megváltoztathatatlan.
- Állapottartó: Az adatok változhatnak, adott szabályok hatására új adatok jöhetnek létre melyek a szabályok újbóli kiértékelését vonhatják maguk után.

A szabály nyelv az alábbi váz segítségével adja meg a szabályokat:

```
package  - A csomag nevét kell megadni
imports – az importált csomagok
globals – a definiált változók
functions – a definiált függvények
queries - lekérdezések
rules - szabályok
```

```
rule "name"
        attributes
when LHS

then RHS

end
```

16. ábra

A szabályok mint ahogyan az ábrán is láthatjuk alapvetően két részből állnak. A bal-oldalinak (**left hand side** LHS) nevezett szakasz a feltételeket írja le míg a jobboldalinak (**right hand side** - RHS) nevezett szakasz a következményeket írja le.

A szabály nyelv lehetőségei tömören:

- **Globális források használata**: segítségével definiált globális változókon keresztül a szabály elérheti az alkalmazás objektumait.
- **Függvények deklarálása**: segéd kód részek megfelelő strukturálását segíti. Leggyakrabban a következmények megvalósítását segíti.

- **Típusok deklarálása**: Új típusokat hozhatunk létre futásidőben illetve meta adatokkal egészíthetjük ki a meglévő adatmodellünket.
- **Lekérdezések**: a LHS-ben leírt kifejezések segítségével lehet meghatározni a szabályt. Itt halmazműveletekkel (pl.: contains) reguláris kifejezéseket használhatunk melyek kiegészíthetőek időbeliségen (pl.: during) alapuló logikával (amennyiben használjuk a CEP modult - **Complex Event Processing**)
- **Szabályok**: A RHS segítségével a következményeket írják le.

Egyáltalán nem mindegy, hogy hogyan történik meg a tények és a szabályok összepárosítása. A naiv megoldásnak mely esetén minden tényt minden szabállyal összepárosítunk igen komoly skálázhatósági problémái vannak. Ennek a problémának a kiküszöbölésére számos különböző módszert fejlesztettek ki. Ezen módszerek alapvetően a szabályok ütemezésének módjában különböznek egymástól. A legtöbb megoldás az ú.n.: előre láncolásos (**forward chaining**) megközelítést alkalmazza amely az alábbi csoportokra bontható:

- Következtető (**Inference**): ezek tipikusan a HA AKKOR (IF - THEN) logika megvalósítására szolgálnak. Pl.: szabad-e az adott betegnek adott típusú gyógyszert szednie? (HA magas a vérnyomása AKKOR adott gyógyszert nem szabad szednie) Ezt tipikusan az alkalmazás hívja meg adott adat halmazra.
- Esemény Feltétel Akció (**Event Condition Action**): A reaktív szabály motor detektálja a beérkező eseményeket és ezek mintázatára reagál. (pl.: riasztja a gondozót ha a beteg a fürdőszobába ment és már több mint 20 perce ott van). Ellentétben az előzővel ezt az események beérkezése aktiválja.

A szabály motorok egy másik osztálya a hátrafelé láncoló (**backward chaining**). Ekkor a motor azokat az esemény/tény kombinációkat keresi ki amikor adott feltételek igazak.

6.4. Összefoglaló

A fejezetben három különböző eljárásmódot mutattunk be a logika leírására. Ezen módszerek azonban nem helyettesítik, hanem inkább kiegészítik egymást. A legtöbb alacsony szintű algoritmus továbbra is OO vagy POP nyelvek valamelyikében lesz megvalósítva. A magasabb szintű szakértelmet leíró logika azonban már egyre gyakrabban folyamat nyelvekkel kerül megvalósításra. Az olyan bonyolult szabályrendszerek ahol sok egymástól többnyire független szabály van és ezeket a szabályokat menedzselni is szeretnénk egyre gyakrabban lesznek szabály motorokkal megvalósítva. Egy jó példa ezen három megközelítés integráló keretrendszerekre a **Red-Hat Drools** keretrendszere illetve az **IBM Websphere** termékcsaládjába tartozó **ILOG** és **BPEL** platformok. Ezek lehetővé teszik ezen három megközelítésmód integrált használatát.

6.5. Kérdések

1. Melyek a különböző generációs programozási nyelvek fontosabb jellemzői?
2. Mi a POP?
3. Milyen lehetőségeket nyújthatnak a processz nyelvek?
4. Mire jók a szabály nyelvek?

6.6. Ajánlott irodalom

- W. van Der Aalst, A. Ter Hofstede, és M. Weske, “Business process management: A survey,” Business Process Management, 2003, o. 1019–1019.
- Drools - JBoss Community. http://www.jboss.org/drools
- K. Czarnecki, “Overview of Generative Software Development,” Unconventional Programming Paradigms, 2005.
- R. Lämmel, J. Visser, és J. Saraiva, Szerk., Generative and Transformational Techniques in Software Engineering II, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

7. Nyelvi paradigmák, trendek - adatkezelés

Egy információs rendszerre gyakorlatban úgy gondolunk, mint adatbázisok és programok összességére, ahol a programok feladata az adatbázisokból származó adatok manipulálása, tárolása és lekérdezése. A programok legtöbbször egy jól megtervezett folyamat modell köré lettek tervezve, implementálva, mely az üzleti célokat elégíti ki. Ennek megfelelően az adatbázis is egy meghatározott **adatábrázolási modell** köré épített. Ez határozza meg, hogy milyen típusú adatokat tárol és kezel a rendszer. Az adatábrázolás kezdeti szinten az adatok **koncepcionális megfogalmazásával** történik, melyet a gyakorlatban **logikai adat modellnek** hívunk.

Az adatmodellezést a gyakorlatban nem csak az adat elemek definiálására használják, hanem az azokból felépíthető struktúra és köztük lévő kapcsolatok definiálásra is. Így az adatok egy standard, konzisztens, kiszámítható struktúráját kapjuk meg, mely lehetővé teszi az erőforrások megfelelő menedzselését. Egy alkalmazás adatmodellje a teljes rendszerre kihatással van, mivel minden felsőbb szintű alkalmazás az alsó rétegekre épített. Így például egy felhasználói felület is csak annyi információval tud szolgálni, mint amennyit eltárolunk. Ugyanakkor a felső szintű logika kiépítése is mindig az alsó rétegekre van alapozva. A való életben úgy gondolhatunk erre, hogy megfelelő alapozás nélkül nem lehet rendesen építkezni. Nem véletlen tehát, hogy minden projekt kritikus része az adatmodell körültekintő megtervezése, tehát megéri további erőfeszítéseket tenni annak tervezése során.

Jelen fejezet az adatmodellezés folyamatáról ad egy rövid áttekintést, illetve a három legelterjedtebb adatábrázolási, adattárolási módszert írja le, azok előnyeit, hátrányit és kombinációit. Mielőtt még részletesen tárgyalnánk a fejezet lényegét, fontos megkülönböztetnünk néhány alapfogalmat, melyek elengedhetetlenek az anyag megértéséhez.

Egy rendszer **adatmodellje** szerkezeti felépítést próbál szolgáltatni az általa feldolgozott információk logikai felépítéséről és a rajtuk végezhető műveletekről. Fontos megemlíteni, hogy itt mindig csak logikai kifejezésekkel foglalkozunk. Egy adatmodell általában a következő elemeket tartalmazza: egyed, rajtuk értelmezhető tulajdonságok, az egyedek közötti kapcsolatok. Ezen ábrázolások tényleges, fizikai megfogalmazása távol esik az adatmodellezéstől.

Az **adatmodellezés** során az adatmodellben tárolt információkat próbáljuk összesíteni, összegezni, kiegészíteni, illetve az adatmodellben felvett adatok közötti kapcsolatokat ábrázoljuk, elősegítve a számítógépes információ-feldolgozást.

Az adatmodellezés során megfogalmazott logikai információkon értelmezett műveleteket **adatkezelésnek** nevezzük. Az adatkezelés tényleges feladata az adatokon értelmezett lekérdezhetőségek, módosítások, beszúrások és egyéb műveletek megfogalmazása logikai szempontból. Ezen műveletek később fontos szerepet játszanak a fejlesztés életciklusa során, hiszen csupán ezeken az interfészeken keresztül tudunk kapcsolatot teremteni a fizikai adatokkal.

7.1. Az adatmodellezés fontossága

Az adatmodell egy hatékony eszköz annak kifejtésében, hogy a rendszer mire képes és milyen követelményei vannak ehhez. Az adatmodell ereje a **tömörségben** rejlik és ez a

tömörség mellett is képes implicit meghatározni képernyőterveket, riportokat, folyamatokat, amik az adat feltöltését, lekérdezését és törlését végzik. Többnyire a tényleges adatmennyiség áttekintése sokkal kevesebb idővel jár, mint a rajta értelmezett definíciók, szabályok átnézése.

Egy program hosszú időn keresztül gyűjtött adatai értékes eszközök a cég számára. Így egy pontatlan adat megléte csökkenti az információk értékét és gyakran költséges vagy javíthatatlan következményekkel jár. Vegyünk példának egy egyszerű dátumok tárolására alkalmas mezőt. Egy nem megfelelő specifikáció esetén felmerülhet a kérdés, hogy európai vagy amerikai stílusú dátumok tárolására használatos a mező. Ezen feltétel hiánya az információk megbízhatatlanságát eredményezi. Így az adatábrázolás kulcsszerepet játszik abban, hogy jó adat minőséget érjünk az által, hogy egy **közösen értelmezhető leírást** fogalmazunk meg az adatokról.

7.2. Egy jó adatmodell ismérvei

A tervezés során egy jó adatmodell elérése érdekében a következőben tárgyalt kritériumokat kell szem előtt tartani. Ezek a szempontok mindegyike hozzájárul rendszerünk sikerességéhez.

- **Teljesség**: Fontos, hogy az adatmodell minden lehetséges attribútummal már rendelkezzen a tervezés során. Egy új tulajdonság, adattag felvétele nagy költségekkel jár a későbbi karbantartás során, ezért fontos, hogy egy teljes **adatmodellt** hozzunk létre.
- **Nem-redundáns felépítés**: Megeshet, hogy **redundáns** modellt dolgozunk ki, melyben ugyanazon információ többször is előfordul. Egy ilyen eset, ha például tárolunk életkort és születési évszámot. A redundáns adatok, adattagok figyelmen kívül hagyása túl nagy tárhely kihasználtságot vagy konzisztencia problémákat okoz. Van azonban olyan eset ahol a redundáns adat nagyban hozzájárul a kellő teljesítmény eléréséhez.
- **Üzleti szabályok érvényre juttatása**: Minden cég saját **üzleti szabállyal** rendelkezik. Ennek szem előtt tartása első pillantásra nem nyilvánvaló, de a folyamatos fejlesztés során erre is nagy figyelmet kell fordítani. Ezen szempont teljes figyelmen kívül hagyása komoly kockázatot rejt magában és korrekciója is jelentősen nehézkes.
- **Újrahasználhatóság**: Egy jó adat modell segít abban, hogy az információk újra felhasználhatóak legyenek. Például egy biztosító rendszer által tárolt információ később újra felhasználhatóak demográfiai adatok statisztikájára.
- **Stabilitás és flexibilitás**: Egy jól megtervezett rendszer számára az előre látható változtatások lehetősége is támogatott. A piaci életben az egyik legkritikusabb faktor az, hogy a rendszernek képesnek kell lennie az idő folyamán a változásokhoz igazodnia. Így egy új termék vagy szabály (policy), egyes esetekben törvény bevezetése esetén is képesnek kell lennie a rendszernek alkalmazkodni. Egy modell **stabil**, ha előre láthatólag semmilyen szintű változtatásra sem szorul. A stabilitás kérdésköre felbontható, hogy kevésbé vagy jobban stabil a rendszer aszerint, hogy mennyi változtatásra van szükség a fejlesztési ciklus során. **Flexibilisnek** tekintjük, ha minimális változtatásokkal érhető(ek) el az új követelmény(ek).

További fontos tényezők még az érthetőség, relevancia, költség, pontosság, hozzáférhetőség. Ezek alapján elmondható, hogy bár az adatábrázolás csak egy kis részét teszi ki a teljes rendszer tervezésének, mégis rendkívüli hassal bír az egészre nézve. Plusz idő és erőforrás fektetése az adatmodell tervezésébe sok esetben kifizetődik a jövőben.

7.2.1. Az adatmodellezés menete

Minden adatmodell létrehozása a követelmények összegyűjtésével kezdődik. A követelmények egy **fogalmi/koncepcionális** modellben kerülnek rögzítésre. Feladata a domain szemantikus megfogalmazása gyakran adatszótárak segítségével. A domain-ben minden elem egy entitásként van reprezentálva. Minden entitás bizonyos értelemben részt vesz a rendszer egészének megértésében, tehát szükséges a rajtuk értelmezett kapcsolatok kifejtése/ábrázolása, de csak olyan szinten, melyre maga a modell képes. A koncepcionális modell kivitelezését (implementáció) hívjuk **logikai adatmodellnek**. Az utolsó lépésként a logikai modellből fel lehet építeni a már tényleges fizikai tartalommal rendelkező sémát, amit **fizikai adatmodellnek** nevezünk. Fizikai modellhez sorolható információ például az adatbázis domain leírásához szükséges minden definíció: indexek, feltételek, kulcsok, kapcsolók. Ezek és más fizikai információk alapján lehetőség nyílik a megfelelő tárhely allokációjára.

Az adatmodellezés során kritikus lépés a megfelelő adatbázis implementáció kiválasztása. Gyakori hiba, hogy egy jól megtervezett adatmodellhez egy rosszul megválasztott adatbázis társul, mely nem támogatja a megfogalmazott követelményeket. Két általános absztrakt ábrázolási módszer terjedt el:

- Általános/szintaktikai adat modellezés
- Szemantikus adat modellezés

Az első a hagyományos adatmodellek általánosítását jelenti. Ezzel együtt standard relációs típusok kifejezésre alkalmas, amik egy modell esetén előfordulhatnak. Ilyen kapcsolatok például osztályozási relációk, ami meghatározza, hogy az adat milyen típusba sorolható, vagy a rész-egész reláció. Ezen és más relációk kifejezésével korlátos tények fejezhetőek ki az osztályok között a természetes nyelvi kifejezőkészség mellett, ami viszont maga után vonja azt a következtetést, miszerint különböző domain-ek használhatósága limitált, mert mindig csak olyan kifejezés típusok alkalmazhatóak, melyek már korábban definiálására kerültek, illetve melyet a saját szemantikája megenged.

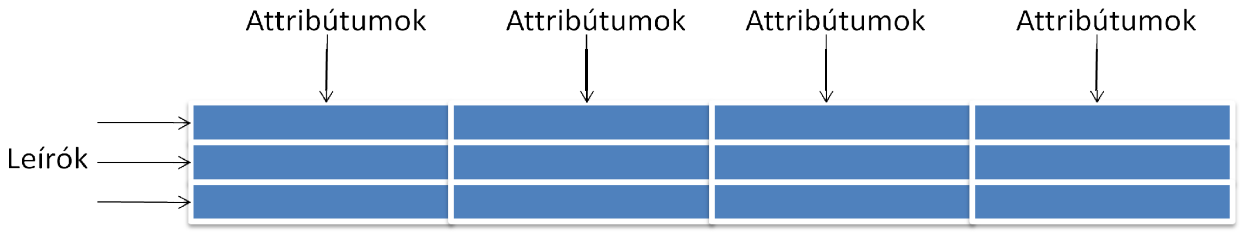
A szemantikus adatábrázolás egy teljesen más megközelítést alkalmaz - az előző típus korlátoltságának köszönhetően -, mely szerint maga az adat jelentése is definiálása kerül és a kapcsolata a többivel a környezetén belül. A szemantikus modell tehát egy absztrakció, mely kapcsolatot próbál teremteni a valós kifejezések és a fizikailag tárolt adatok között. A szemantikus adatmodellezés általános célja tehát több jelentés/információ kinyerése az adatokból, felhasználva a mesterséges intelligenciában használt absztrakt kifejezéseket. Ezzel gyakorlatilag gépi értelmezés valósul meg.

A következőkben röviden áttekintjük a három legelterjedtebb szintaktikus és szemantikus adatmodelleket.

7.2.1.1. Relációs adatmodell

Az első és legegyszerűbb szintaktikus módszer a relációs modellezés, ami sok más módszer ősének tekinthető. Az alap koncepció, hogy az adatbázis állítmányok halmaza a véges változók felett, kifejezve ezzel korlátokat a lehetséges változókon vagy azok kombinációin. Célja, hogy egy deklaratív módszert adjon az adatok és a rajtuk értelmezhető lekérdezések specifikálására: közvetlenül meghatározza, hogy mit tartalmaz az adatbázis és mit kívánunk kinyerni belőle. Ráhagyja az adatbázist menedzselő rendszerre az adatok strukturálásának leírását és a lekérdezés eredményeinek folyamatát.

A modell magvát a relációk adják, ami még kiegészül további szabályokkal: kulcsok, kapcsolatok, kötések, funkcionális függőségek, tranzitív függőségek, többértékű függőségek. A reláció a modell alap eleme.



17. ábra: A reláció felépítése

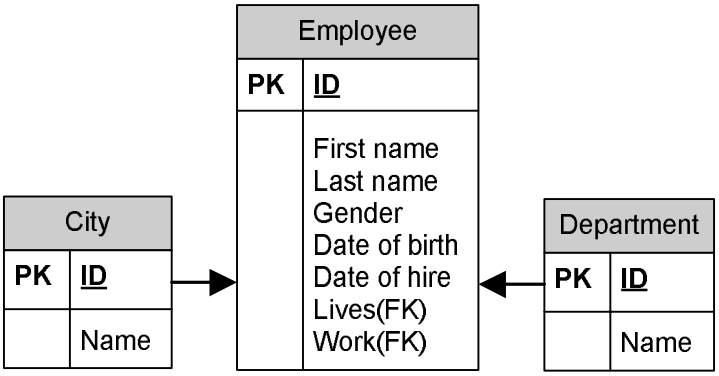
A modellel kapcsolatos általános szabályok a következők:

- Egy reláció (fájl, tábla) egy kétdimenziós táblának felel meg.
- Az attribútum (mező, adat elem) a tábla egy oszlopa.
- Minden oszlop egyedi névvel rendelkezik a táblán belül.
- Minden oszlop homogén.
- Minden oszlop saját domain-nel rendelkezik (tehát a felvehető értékek halmaza specifikált).
- Egy bejegyzés (tuple) a tábla egyetlen sorának felel meg.
- A sorok és oszlopok sorrendje nem meghatározott.
- Egy sor minden értéke kapcsolódik egy másik fogalomhoz vagy annak egy részéhez.
- Ismétlődő csoportok nem megengedettek.
- Egy bejegyzés csak egyszer szerepelhet (kulcsok használata).
- A cellák egyetlen értéket tartalmaznak.

Minden tábla rendelkezik legalább egy speciális oszloppal, melyet kulcsnak hívunk. Ezek célja a tábla indexelése, ezzel felgyorsítva a keresést. A kulcsok használatával két különböző táblában, de hasonló halmazból származó értékek között kötések alakíthatóak ki. Ezzel különböző táblák értéke egyszerre kérdezhető le, megfeleltetve benne bizonyos értékeket egymásnak. Például egy rendelés (id, termék kód) és termék (termék kód, termék ár) táblánál szükséges a két tábla egyszeri lekérdezése annak érdekében, hogy vásárláskor egy összesített árat adjunk a vevő felé, mindezt a „termék kód” kapcsolásával. Természetesen

több különböző tábla között is adható kapcsolás. Köszönhetően, hogy ezek a kapcsolások lekérdezés időben definiáltak, a relációs adatmodell egy rendkívül dinamikus sajátosságát adják.

A relációs modell is rendelkezik hátrányokkal. Komplex adatok tárolásának hiánya jellemző tulajdonsága a modellnek (képek, videók). További hátrány a mezőkre vonatkozó határértékek, megszorítások, amelyeket már a tervezési fázisban definiálnunk kell. Például egy szövegmezőre vonatkozó megszorítás figyelmen kívül hagyása adatvesztéssel járhat. Komplex relációs adatbázisok közötti kommunikáció nagy és költséges munka. Teljesítmény terén is alulmarad a relációs modell másokkal szemben, tekintve, hogy táblák sokaságát kell felvennünk egy kész rendszer működtetéséhez és a táblák gyakran robosztus információmennyiséggel rendelkeznek.



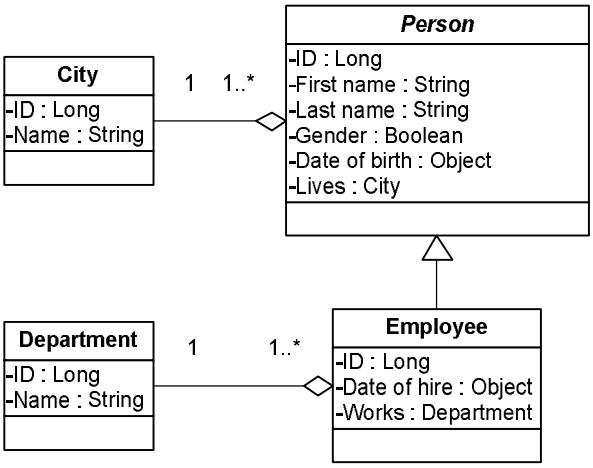
18. ábra: Példa a relációs adatmodellre

7.2.1.2. Objektumorientált adatmodell

A szintaktikus módszerek másik elterjedt modellje az objektumorientált adatmodellezés, mely már gyakorlatilag egy kiterjesztettje a relációs modellnek, ahol is az adatbázis képességei az objektumorientált (OO) nyelv képességeivel vannak kombinálva.

Elterjedtségét és népszerűségét leginkább annak köszönheti, hogy a programozók által implementált entitások objektum szinten közvetlenül tárolhatóak, módosíthatóak vagy nagyobb objektumcsoportban hozhatóak létre. A modellben felhasználhatóak az objektumorientált paradigma sajátosságai, így például az öröklődés által történő általánosítás és specializálás is.

A megoldás hátránya, hogy nem igazán létezik jó megoldás a tartós, kereshető, módosítható perzisztenciára. Emiatt csak a memóriában szoktak OO modelleket használni a tartós perzisztenciát pedig a klasszikus relációs adatbázisok valósítják meg.



19. ábra: Példa az objektumorientált adatmodellre

7.2.1.3. Szemantikus adatmodell

Az előző két paradigma nem nyújt elég gazdag kifejező fogalmi modellt bizonyos problémákra, amelyekre nem illeszkedik a jól megszokott sablon. Az utóbbi évtizedek alatt számos olyan modell emelkedett ki a többi közül, amelyek mindinkább próbáltak a nagyobb kifejezőkészségre szert tenni és gazdagabb szemantikát adni az adatbázisoknak. Ezen modellek halmaza adja meg a szemantikus adatmodelleket, amiket szélesebb körben ontológiáknak neveznek.

A szemantikus adatábrázolás célja, hogy egy olyan struktúrát szolgáltasson, mely a leginkább tükrözi a való világ kifejezőkészségét. Általában ez a modell kiegészül egy absztrakt világgal. Három fontos absztrakció szükséges egy adatmodell számára:

- Osztályozás
- Tartalmazás
- Általánosítás

A szemantikus világban mind a három absztrakció egy típusleíráshoz vezet. Összevetve más modellekkel egy nagyon fontos eszköz a szemantikus rendszerek esetében a „koncepció”. Gyakorlatilag minden koncepciók köré csoportosul, mert mindent a való világban is fogalmakkal azonosítunk. Ezen koncepciók között bármilyen, az emberi nyelvezet által kifejezhető kapcsolat létrehozható.

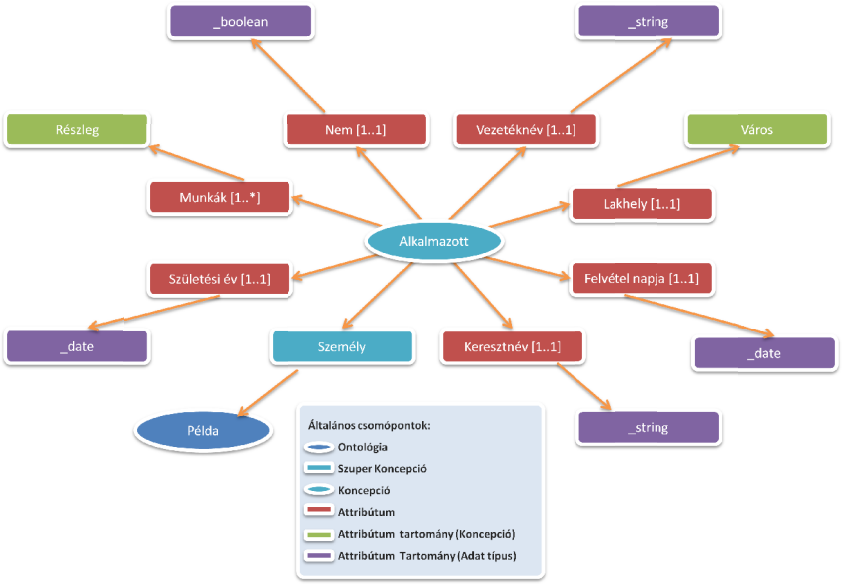
Az adatmodellek leírása maga után von bizonyos integritási szabályok meglétét. Két lényeges integritási szabályt említhetünk meg a szemantikus adatmodellekkel kapcsolatban:

- Kapcsolatképesség
- Konvertálhatóság

Az első jelentése, hogy minden koncepció attribútuma kapcsolódik egy másik koncepcióhoz. Persze minden egyes fogalom több másikkal is kapcsolatban állhat tulajdonságainak köszönhetően. Ebből származik azaz észrevétel, miszerint gyakorlatilag végtelen domain-t tudunk alkotni. A második fogalom célja kifejteni, hogy minden koncepció egyedi, tehát nincs módszer arra, hogy kettő ugyan azt a jelentést hordozza, ugyanígy attribútumok között sem. További megszorítások fordulhatnak még elő az adatok

modelljében, továbbá gyakran komplex feltételek megfogalmazására is szükség van, amellyel a domain konzisztenciája teljessé válik.

Mint már ismert, a szemantikus adatmodell legfőbb eleme a koncepció, mellyel minden leírható egy bizonyos domain-en belül. Ez csak egy entitás fogalmi jelölését jelenti. Egy szemantikus adatmodellben példányokat is tudunk létrehozni. Vegyünk példának egy entitást, mely egy személy fogalmi leírását adja meg. Ezen entitás példányait emberi nevek alkotnák, ahol minden példányhoz járulnak további attribútumok, amivel önmagukat írják le. Például a felvett személy entitáshoz tartozó példányok: Bob, Jim, Sue, Betty, stb. Ezen nevek mindegyike egy bizonyos személy típusú objektumra mutat, tehát Bob egy olyan sorra mutat, ami megadja Bob sajátosságait, mint például név, nem, családi állapot, továbbá minden olyan attribútum mely szerepel a személy koncepciónál.



20. ábra: Példa szemantikus adatmodellre

7.3. Modellek közötti megfeleltetések, átjárások

Látható, hogy minden modellnél léteznek előnyök és hiányosságok, tehát a fejlesztő és tervező csapat feladata, hogy kiválassza a saját rendszerükhöz leginkább alkalmas megoldást. Gyakran a megoldás nem vagy-vagy jellegű hanem és-és jellegű. Azaz mindhárom paradigmának megvan a maga szerepe/réteg a rendszerben és szükség van olyan ragasztó rétegekre melyek hidat képeznek a különböző absztrakciós szintek/megközelítések között.

Egy ilyen híd/technológia az Object-Relational Mapping (ORM), mely a relációs és az objektumorientált világ között képez átjárást. Ez a megoldás megtartja a relációs sémából kapott perzisztencia képességet, az objektumorientáltságból vett előnyökkel (lásd fentebb). Mindezzel lehetségessé válik az objektum szintű manipuláció és hozzáférés, anélkül, hogy felfednénk ezen objektum(ok) adatbázis szintű állapotát. Legfőbb előnye, hogy a programozó számára konzisztens képet ad az objektumokról.

Az ORM eszközök fontos feladata, hogy minden, az összekapcsolt technológiákban található kifejezéseket meg tudjon egymásnak feleltetni. Az egyik ilyen legegyszerűbb példa, ha egyetlen perzisztens osztályt és tábla között akarunk megfeleltetni. Ekkor az

osztály minden attribútuma megfeleltethető a tábla oszlopainak. Egy másik gyakori eset, ha az öröklődést szeretnénk szimulálni. Erre három lehetőséget kínál az ORM:

- Horizontális megfeleltetés
- Vertikális megfeleltetés
- Szűrő megfeleltetés

Horizontális esetben, minden osztály és a hozzá tartozó absztraktak is külön táblákba kerülnek. Vertikális esetben az összetartozó absztrakt és leszármazott osztályok egy táblába kerülnek, viszont továbbra is minden osztály külön szerepel. Végül szűrő esetben minden összetartozó osztály egy táblába kerül.

Egy másik kritikus eset, ha objektumok közötti relációt kell megfogalmazni. Ilyen relációk az egy-egy, egy-több, több-több asszociációk és az aggregációk. Minden esetben meg kell fogalmaznunk az osztályok közötti ilyen szintű kapcsolatot annak érdekében, hogy a megfelelő táblatípusok generálódjanak, például külső kulcsok elhelyezésénél. Aggregáció esetében képesnek kell lenni minden belső osztály módosítására a külsőn keresztül is. A belső osztály elhelyezése is minden esetben az ORM feladata.

Természetesen egy ORM technológiának képesnek kell lenni minden egyszerűbb adattípus lekezelésére. Komplex objektumok (kollekciók) esetében viszont más a helyzet. Legtöbbször a fejlesztő feladata megtervezni a kollekciók definiálását tárolási szinten, ám beépített módszerekre is támaszkodhat. Így például egy lista kollekció megfeleltetése a lista indexek a tábla indexeivel történő megfeleltetést jelenti. Saját osztályt tartalmazó kulcs-érték pár (Map) kollekció esetében viszont nem egyértelmű, hogy mi alapján szeretnénk indexelni. Hasonló módszerek léteznek minden közismertebb komplex típusra: enumeráció, halmaz, vektor. Számos ORM megoldás ismert, ezek közül eggyel (**Hibernate**) részletesebben is foglalkozunk a 8. fejezetben.

A szemantikus adatmodellek számára is létezik megfeleltető eljárás, az adatbázisban tárolt objektumok szemantikus leírókká alakítására. Egy ilyen technológia a DBOM1, mely OWL ontológiák használatával képes leírni egy relációs adatbázis tartalmát.

7.3.1. Összefoglalás

A fejezetben három különböző absztrakciós szintet biztosító adatmodellezési paradigmát tekintettünk át. Ezeket gyakran együtt alkalmazzuk és ekkor szükséges az átjárás a különböző absztrakciós szintek között. Ezekből egy ilyen területet az ORM-et ismertettünk. A 8. fejezet részletesen foglalkozik az ORM területen fellépő problémákkal és lehetséges megoldásokkal.

7.4. Kérdések

1. Miért fontos az adatmodellezés?
2. Mik egy jó adatmodell ismérvei?
3. Mi az adatmodellezés folyamata?

[1] DataBase Ontology Mapping

4. Mi a három legelterjedtebben használt adatmodell típus?
5. Mi az ORM eszközök feladata, előnyeik?

7.5. Ajánlott irodalom

- Graeme C. Simsion, Graham C. Witt, “Data modeling essentials”, Morgan Kaufmann, 3rd edition, 2004.
- Michael C. Reingruber, William W. Gregory, “The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models”, Wiley, annotated edition, 1994.
- David C. Hay, “A comparison of data modeling techniques”, Essential Strategies, Inc, 1999.

III. rész

Alkalmazás fejlesztés

Az előző fejezetekben bemutatásra kerültek azok az alapvető problémák, amelyek alkalmazásrendszer fejlesztésekor felmerülnek, továbbá felvázoltuk azokat a lehetséges elméleti megoldásokat, amelyek választ adnak a fenti problémákra. Jelen rész célja, hogy bemutassa azokat a konkrét technológiákat, amelyek segítségével lehetséges egy elosztott, robusztus, skálázható és hibatűrő rendszer megvalósítása.

A jelen fejezetben bemutatott alkalmazás-fejlesztési megoldásokat összefoglaló néven vállalti (enterprise) alkalmazásfejlesztésnek nevezzük. Az vállalti alkalmazásfejlesztés lényegesen különbözik olyan korábban megismert megoldásoktól, amiket jellemzően asztali vagy mobil környezetben alkalmaznak. Az vállalti alkalmazás keretrendszerek köztesrétegek segítségével biztosítják azokat a magas szintű fejlesztői kereteket, amik elfedik a szálkezelést, a tranzakció kezelést, absztrahálják a hálózati és adatbázis-kapcsolatokat. A fejezet célja bemutatni azt a futtató környezetek képességeit, rálátást adni azokra a technológiákra, amikkel a felület-tervezéstől az adatok pertisztálásáig lehetőség nyílik egy teljes független rendszer fejlesztésére.

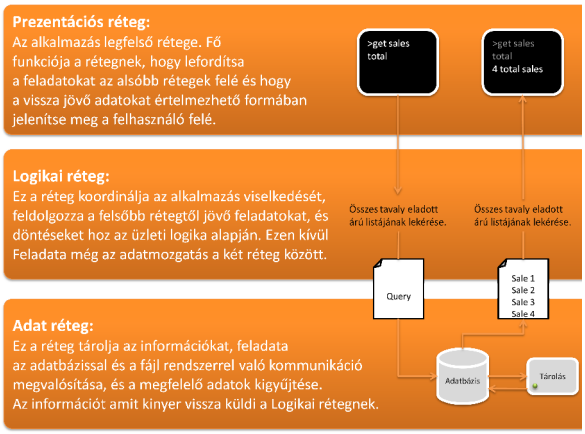
A klasszikus háromrétegű alkalmazás architektúra az adat, logika és megjelenítési rétegekből áll. A tervezési mintákat figyelembe véve az olvasó azt gondolhatná, hogy a háromrétegű architektúra megegyezik az MVC (*Model-View-Controller*) mintával, azonban ez nem teljesen igaz. A háromrétegű architektúrában lineáris az információáramlás, mivel a felső rétegek mindig a közbenső rétegeken keresztül érik el az alsó rétegeket (21. ábra). Ezzel szemben az MVC szerkezeti felépítése trianguláris. Ez annyit jelent, hogy az MVC esetén a megjelenítési réteg frissíti a vezérlő réteget, a vezérlő réteg frissíti a modellt, a modell réteg pedig ezután közvetlenül visszahat a megjelenítési rétegre (22. ábra).

Az optimális, jól karbantartható és effektív logika megteremtéséhez – objektum-orientált paradigma esetén – több, esetleg már ismert kritériumnak kell megfelelni úgy, mint például:

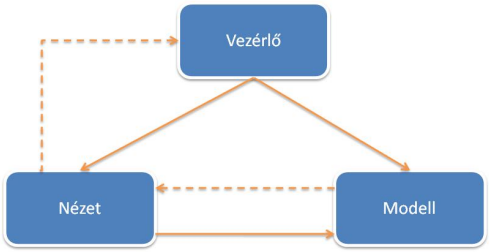
- karbantarthatóság (*maintainability*)
- újrafelhasználhatóság (*reusability*)
- skálázhatóság (*scalability*)

A klasszikus tervezés esetén a fenti kritériumok megteremtésének feladata mind a fejlesztőre hárul. A fejlesztő felelőssége annak megértése, hogy amennyiben a szoftver karbantartható, úgy minimalizálni tudja a változtatáskor, javításkor, illetve új funkciók bevezetésekor igényelt erőforrásokat. A skálázhatóság pedig a növekvő igények kiszolgálá-

sának problémamentes kezelését jelenti (pusztán erőforrás növeléssel, szoftver újratervezés nélkül). Mindezek mellett, ha a fejlesztő bármilyen segéd-keretrendszer nélkül próbálna meg a középső szint (middle tier) képességeinek megfelelő rendszert létrehozni, szembe kellene néznie olyan klasszikus, ám korántsem triviális kihívásokkal, mint például a konkurens szálak, folyamatok hatékony és megfelelő módon való kezelése, az adatok és információ integritásának megőrzése érdekében. Ahogyan ezt már a 2., 3. fejezetben is bemutattuk, e problémák orvoslására való a köztesréteg.



21. ábra: Háromrétegű architektúra

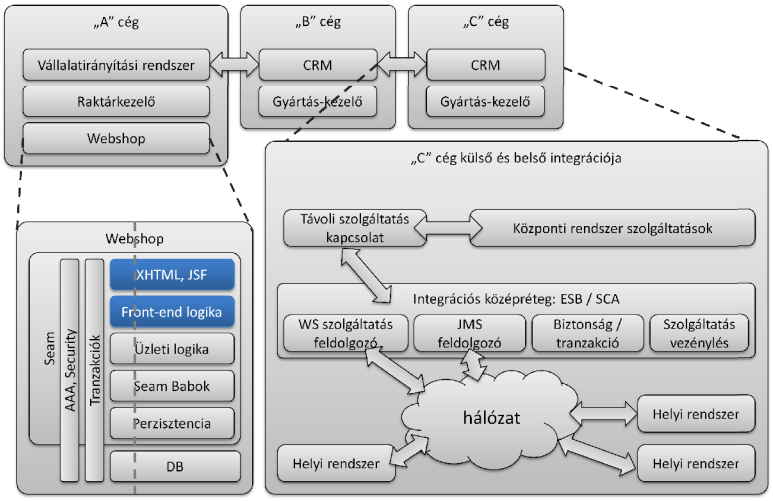


22. ábra: MVC modell

8. Felhasználói interakció – Megejelenítési réteg

8.1. Bevezetés

Ma a számítógépes rendszerek jelentős részének egyik fő célja **CRUD** (**Create-Read-Update-Delete**) műveletek elvégzése. Jellemzően a CRUD műveletek forrását emberi felhasználók jelentik. Egy jellemző irodai környezetben a számítógépes rendszerek használatának jelentős részét űrlapok kitöltése és adatböngészés teszi ki. A fejezet célja bemutatni azokat a technológiákat, melyek lehetőséget adnak a felhasználóval történő kommunikációra, továbbá a fejezet felvázolja a felmerülő kérdéseket, majd megfelelő technológiai választ ad rájuk. A 23. ábra kiemelt részei jelzik azt, hogy jelen fejezet az átfogó rendszer mely részeit taglalja.



23. ábra

8.2. Elvárások a felületekkel szemben

Az informatika fejlődésével a felhasználói felületek is fejlődtek, szabványosodtak. A felületekkel szembeni felhasználói elvárások az egyszerű beviteli-mezőtől a speciális komponensek felé mozdulnak. Minden esetben cél az intuitív felület-kialakítás, a produktív használat, azonban a rendelkezésre álló szabványok lassan követik ezt az irányt (HTML5), valamint egyes böngészők is lassan követik, követték a szabványokhoz való alkalmazkodást. Ennek következménye, hogy a speciális komponensek magasabb absztrakciós szintek jelennek meg. Ma felhasználói elvárás az intuitív, interaktív felület, a cég arculatába illő, mégis szabványos design, a böngésző-függetlenség, a jó vizualizáció, valamint az accessibility (képességeikben korlátozott felhasználók támogatása). Fejlesztői elvárások között fő szempont a jó karbantarthatóság, értve ezalatt a redundancia-mentes fejlesztést és a felhasználók felé történő kihelyezést is. A jó karbantarthatóság kérdéskörére a következőkben bemutatott fejezetek adnak választ, a felhasználók felé történő kihelyezés pedig a web felépítéséből adódik.

8.3. RIA, Okos kliens

A böngészők elterjedésével létre jött egy olyan platform, mely az operációs rendszerek felé egy újabb absztrakciót adva lehetőséget ad az alkalmazásfejlesztés egy új módjára: a böngészőkkel egy olyan operációs rendszer platform-transzparens réteg jött létre, mely az alkalmazás-fejlesztési szempontból lehetőséget nyitott könnyen karbantartható központból menedzselt rendszerek fejlesztésére. A böngészők fejlődésével, valamint különböző beépülők, úgynevezett beépülő modulok segítségével a böngészők képessé váltak arra, hogy felváltsák a hagyományos asztali alkalmazásokat, mind grafikai élmény, mind funkciógazdagság vonatkozásában. Így született meg a RIA, teljes nevén Rich Internet Application fogalma.

A RIA mint fogalom alapvetően két részre osztható: egyrészt a böngészők természetes fejlődésével a böngészők egyre intelligensebb és hatékonyabb megoldásokkal rendelkeznek, másrészt a beépülőm modulok segítségével tetszőleges kiegészítések alkalmazhatók.

A böngészők területén a HTML fejlődésével (HTML5), a Javascript és a CSS egyre szabványosabb támogatásával megerősödött platform ad átütő lehetőségeket.

8.3.1. A böngésző mint platform

Mára a böngészők váltak elsődlegessé a hálózati feladatok ellátásában, ellentétben a korábbi trenddel, amikor az asztali alkalmazások kommunikáltak zárt protokollokon valamilyen szerverrel vagy központtal.

Számos böngésző létezik, azonban csak relatív kevés megjelenítő, úgynevezett böngészőmotort (browser engine) tekintünk alkalmasnak napi használatra, képesnek arra, hogy ellássa a felhasználók és alkalmazásfejlesztők igényeit. A teljesség igénye nélkül a következő lista tartalmazza a legsűrűbben alkalmazott grafikus böngészőmotorokat és a mai piacot uraló böngészőket:

- Trident: Internet Explorer 4-8
- Chakra: Internet Explorer 9+
- Gecko: Firefox, SeaMonky
- WebKit: Safari, GoogleCrome,
- Presto: Opera 7+

8.4. Rendelkezésre álló böngésző alapú alaptechnológiák

A web megjelenése és a web szabványosodása között hosszú idő telt el. Az ezredforduló környékén vált széles körben az üzleti szférában is elterjedté a web mint kereskedelmi és munkaplatform. A kezdeti megoldások (Java Applet) helyett ma a web mint pehelysúlyú vékony-kliens platform vált uralkodóvá. Ennek a folyamatnak egyik indikátora volt a szabványt mind jobban támogató böngészők megjelenése.

Technológiai oldalról a ma rendelkezésre álló tárház jellemzően a következő: HTML+ +CSS a megjelenésért, speciális beépülő modulok a speciális feladatokért, JavaScript a kliens oldali viselkedésért felel, szerver oldalon pedig jellemzően valamilyen dinamikus nyelvet támogató kiszolgáló vagy Java alkalmazásszerver található.

8.4.1. HTML

Számos írás említi a HTML (HyperText Markup Language) nyelvet mint programozási nyelvet, habár a neve is mutatja nem programozási, hanem jelölő nyelv. A mai web alapját jelentő technológia a ’90-es évek elején jelent meg, a W3C szervezet több szabványosítási folyamata után jelenleg több variáns és verzió támogatott (XHTML 1., 1.1, HTML 4.01). Jelenleg a HTML 5 előkészítése folyik. A HTML szállítási protokollja a HTTP 1.1.

A HTML-t eredetileg weboldalak megjelenítésére, annak mind strukturális, mind formázási megjelenítésére használták. Felépítését tekintve támogatta az oldalak egymásba ágyazását, táblázatokat, betűátméeretezést, keretezést, valamint interakciót tekintve az egyszerű beviteli mezőket. A szabvány mai változatát tekintve a HTML-ből kikerültek a formázásra vonatkozó nyelvi elemek. A szabvány legújabb változata támogatja a hang és videó lejátszást, rajzoló felületet, valamint az eddig általános célú div mellé számos az oldal struktúráját leíró új nyelvi elem is bekerül, beleértve a navigációra, menüre, cikkre, bekezdésre vonatkozó elemeket. További újdonság a HTML legfrissebb változatában a változatos input mezők használata, reguláris kifejezések támogatása az egyszerű input mezőkben, alapvető validációs funkciók megvalósítása. Mára a HTML[1] elsődleges célja mára az adatmegjelenítés lett. A megjelenítésért, kliens oldali logikáért mára más technológiák felelősek. Maga a HTML nyelv az SGLM(Standard Generalized Markup Language, szabványos általános jelölőnyelv) nyelven alapul. Az SGML-t arra tervezték, hogy a számítógép által olvasható formátumban lehessen dokumentumokat tárolni. Tervezéskor figyelembe kellett venni a tervezőknek, hogy olyan leírónyelvet készítsenek amelyek évtizedekig megőrzik a dokumentumok olvashatóságát. Így a HTML is megőrizte ezeket a tulajdonságokat.

A HTML-t teljesen felváltotta volna a XHTML ami már XML nyelven is alapul, de ez nem teljesen történt meg, helyette mindkét jelölőnyelv szabvány párhuzamosan van jelen a weben. Az XHTML abban tér el jelentősen a HTML-től, hogy szigorúbb formai követelményeket követel meg. Jelentős változásokat a HTML 5 szabvány hordoz, ami azonban még szerkesztői fázisban van.

Felépítését tekintve a HTML erősen épít az XML-re, azonban nem minden HTML szabvány felépítése ad XML struktúrát, a HTML önmagában megengedőbb. Nyelvi szempontból azonban nagyban hasonlít az XML-hez, úgynevezett tag-ek és attribútumok alkotják a nyelvet. Egy HTML oldal struktúráját tekintve szigorúan fejlécre <head>…</head> és az oldal törzsére <body>…</body> részre bomlik. a fejléc felelős az oldal címéért, a formázási (CSS) és viselkedési (szkript nyelvek, jellemzően JavaScript) betöltésének jelzéséért. A szabvány lehetőséget ad beágyazott CSS és szkript deklarációra is.

A HTML törzsét tekintve egy oldal elrendezését, és a benne megjelenő adatokat írja le. A HTML felelős azért, hogy az oldalon **mi** legyen, a CSS azért, hogy **hogyan nézzen ki**, míg a szkript nyelv azt írja le, hogy **hogyan viselkedjen**. A HTML önmagában nem támogat semmilyen sablonozó megoldást, a fejlesztőknek maguknak, vagy az általuk használt keretrendszernek kell gondoskodnia a sablonozásról. Az oldal felépítését tekintve jellemzően két taktika kínálkozik az oldal elrendezésének megvalósítására: a táblák használata, és az úgynevezett tableless design. Míg az előbbi erősen ellenjavalt, a tableless design jellemzően tag-ek segítségével és CSS használatával valósítja meg az elrendezést. Ma a weboldalak többsége ezt a megoldást alkalmazza.

8.4.2. CSS

A CSS, teljes nevén Cascading Style Sheets egy szabály alapú stílus leíró nyelv. Feladata mára megjelenés leírása, valamint néhány animációs esemény kezelése. A HTML-hez hasonlóan a W3C tartja karban az ajánlásait. Jelenleg a CSS2 és a CSS3 bizonyos részei jelennek meg a ma használt fővonalbeli böngészőkön.

A CSS maga egy stílusleíró nyelv ami a webes dokumentumok(HTML, XHTML, SVG, XUL, stb.) megjelenését írja le. A nyelv legnagyobb értéke az, hogy különválasztja a megjelenést a tartalomtól. Egyes dokumentumokhoz több különálló, akár eszközspecifikus stíluslap is tartozhat (kijelző, mobil, nyomtatás), így még rugalmasabb lehet a dokumentumok megjelenése. Az egyes stílusok külön fájlokban vagy akár a dokumentumon belül is definiálhatóak.

Felépítését tekintve a css minta{ szabályok } alakú kifejezésekből épül fel. Minta esetén számos megoldás létezik, beleértve a direkt hivatkozást (*#idName*), tag hivatkozásokat (*div*), a stílusosztály alkalmazását (*.styleClass*), valamint a pszeudó osztályokat (*:hover*). A CSS2 és CSS3 a minták közötti kapcsolatok definiálását átfogóbban támogatja, ez azt jelenti, hogy lehetőség van a különböző vivatkozások egymás után írásával a DOM fából komplett részfák leveleire szűrni (pl.: #mainContent div.subcontent>p, mely a mainContent id attribútummal rendelkező tag-en belül található összes subcontent class-szal renelkező div tag közvetlen p-t tartalmazó tag-jeit jelenti). A pszeudo osztályok utószűrést jelentenek: lehetőséget ad arra, hogy adott tag-re bizonyos tulajdonsága esetén szűrjünk (pl.: :hover jelenti azt, hogy ha az egér az adott tag fölött van, akkor érvényes a formázás).

A formázási szabályok kulcs:érték párokként vannak definiálva. Számos kulcs lehet egyszerű, vagy összetett (összetett szabályra példa: background: black url(image.png) no-repeat; egyszerű szabályra példa: background-color: black;). Alapvető CSS formázási szabályok a

- betűtípusra vonatkozók: font, font-weight, font-size, text-decoration, stb.
- a dobozolásra vonatkozók: width, height, margin, padding, border, stb
- színre vonatkozók: color, background, background-*, stb
- layoutra vonatkozók: float, overflow, display, stb
- osztályozási beállítások: whitespace, list-style, list-style-*

A css formázási szabályok a CSS2-ről a CSS-3-ra tovább bővültek, megjelent a lekerekítés, árnyékolás, a 2dimenziós transzformációs műveletek: eltolás, forgatás, valamint az átlátszósági színcsatorna és újabb szín modellek. Az alábbi példa mutatja a CSS néhány lehetőségét:

A piacon megtalálható böngészők sajnos nem egyformán támogatják a CSS szabványt, ami megjelenésbeli különbségeket okozhat az egyes böngészőkben.

8.4.3. JavaScript

A JavaScript-et a Netscape fejlesztette ki, kezdetben még Mocha néven. Maga a JavaScript egy objektumalapú, interpretált szkript nyelv, aminek a szintaxisa nagyban hasonlít a Java nyelvhez, azonban dinamikus és nem típusos nyelv. A nyelvet először '97 és '99 között

szabványosították ECMAScript néven, a jelenlegi JavaScript 1.5 az ECMA-262 Edition 3 szabvány alatt fut.

A nyelv felépítését tekintve mind a nem típusosság, mind a dinamizmus magyarázatra szorul. A nem típusos jelen esetben azt jelenti, hogy az objektumoknak látszólag nincs típusa, azt maga a nyelvi értelmező találja ki. A megoldás számos előnyt és hátrányt hordoz magában: a klasszikus OO paradigmák alkalmazása nem egyértelmű, az egységbezárás és a származtatás nem triviális, ugyanakkor a nyelv típustalansága kompaktabb kódot eredményez. A dinamizmust tekintve a JavaScript szabad utat ad az objektum-hierarchia bővítésére, futás közbeni módosítására. Új attribútumok, tagfüggvények vehetők fel létező objektumokhoz. Egy speciális mező a nyelvben, mellyel minden objektum rendelkezik az úgynevezett prototype. Mivel a nyelv nem típusos, ezért a klasszikus példányosítás nem létezik, az objektumok létrehozása klónozással történik. A prototype adja meg a klón objektumok struktúráját és viselkedését leíró metódusait.

A JavaScript fő célja egy adott oldal viselkedésének leírása. Ennek megfelelően hozzáfér a weboldal úgynevezett DOM fájához (Document Object Model). A DOM fa többek között az aktuálisan megjelenő weboldal HTML reprezentációjának objektumhierarchiába felépített változata. Minden HTML tag-hez külön objektum tartozik, minden tag attribútuma pedig egy elemi vagy összetett mezőnek felel meg a DOM fában. A JavaScript implementációk jellemzően képesek arra, hogyha valami módosul a DOM fában, akkor az azonnal a HTML struktúrában is módosul, illetve a képernyőn a felhasználó észlelheti a módosulást.

A JavaScript futtatása jellemzően böngészőben zajlik. Ennek megfelelően jelentős különbségek lehetnek az egyes böngészők között, mivel az egyes gyártók máshogy implementálták a JavaScript feldolgozásáért és futtatásáért felelős részeket. Habár maga a JavaScript nyelv szabványos, számos böngésző-specifikus függvénykönyvtárat nem definiál. A JavaScript nyelvi specifikáció nem szól sem a DOM fa felépítéséről, sem semmilyen böngésző-specifikus komponensről, beleértve akár az eseménykezelést. Ennek következtében a különböző böngésző-gyártók másképpen implementáltak számos komponenst, egyes területeken de facto szabványok léteznek („Netscape like event handling”).

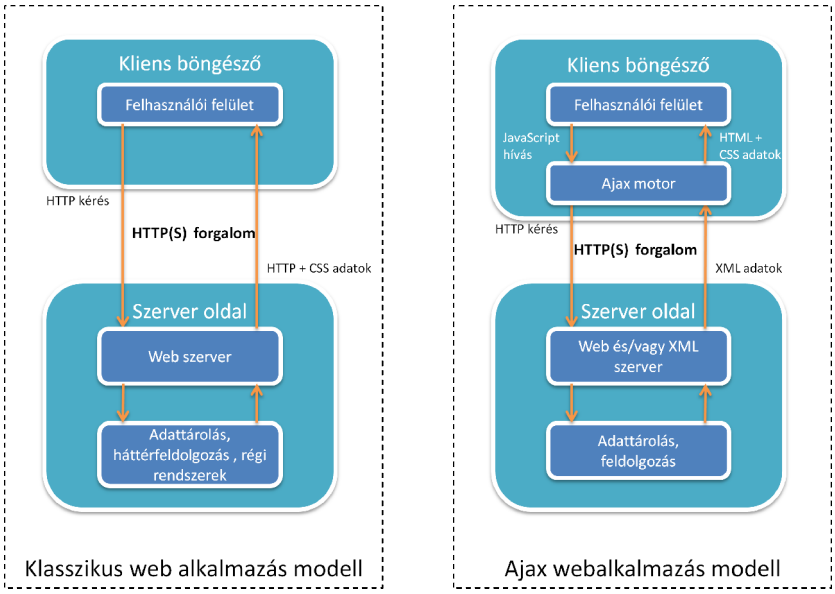
Az előbb említett problémák leküzdésére számos alacsony szintű (értve ezalatt JavaScript szintű) nyelvi könyvtár kínálkozik, ide értve a Prototype, a JQuery, MooTools, stb. A keretrendszerek lehetővé teszik azt, hogy böngésző-implementáció függetlenül kezeljük a DOM fát, eseményeket, ilyen függvénykönvtárak alkalmazásával úgymond transzparenssé tehetők a böngészők. Az említett függénykönyvtárak számos esetben kikerülő megoldást (workaroundot) adnak olyan esetekben, amikor egy böngésző nem képes az adott funkció közvetlen elvégezésére.

Napjainkra elég nagy szerep hárul a JavaScript-re, mert az egyes oldalak interaktív viselkedéséhez elengedhetetlen. A JavaScript az alapja a mára kulcsfontosságúvá vált AJAX megoldásoknak is.

8.4.3.1. XMLHttpRequest, AJAX és további hasonló jelentésű szavak

A böngészők fejlődésével alapvető igény mutatkozott arra, hogy a böngésző az oldal újratöltése nélkül tudjon kommunikálni a szerverrel. A megoldás egyik fő motivációs tényezője, hogy ha csak pár elem módosul az oldalon, akkor ne kelljen az egész oldalt újratölteni, a böngészőnek újrarajzolni, elég legyen csak az adott kis területnek a változását követni. A megoldást kezdetben iframe-mel, beágyazott weboldal kerettel oldották meg, az AJAX

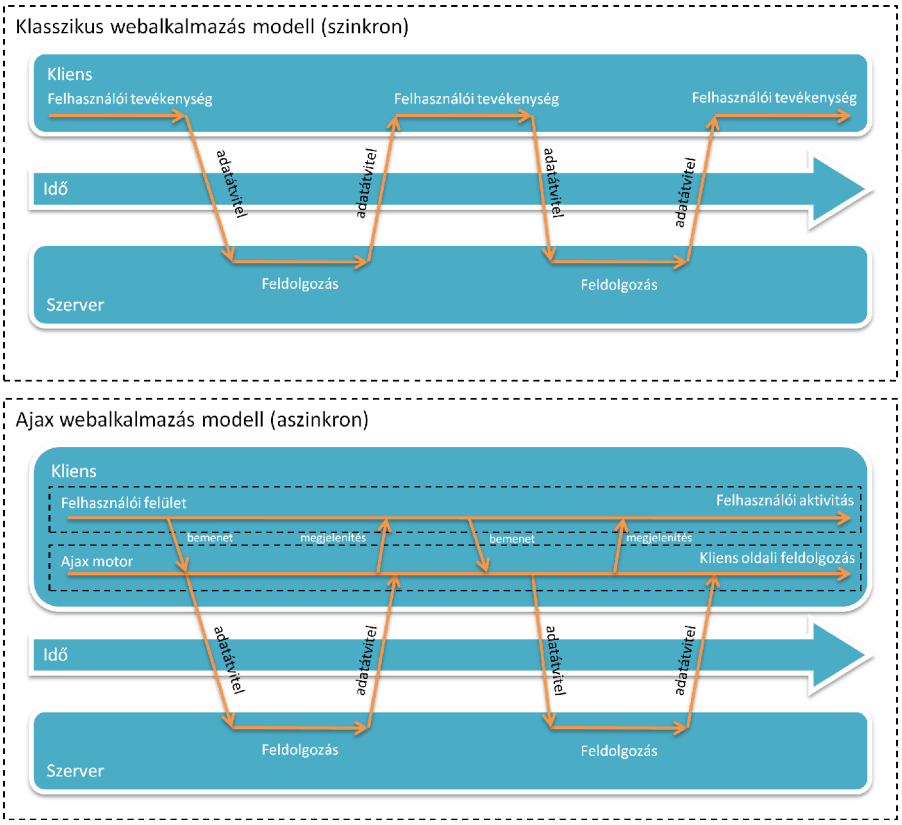
fogalma és az XMLHttpRequest objektum gyakorlati alkalmazása később alakult ki. Az AJAX motivációját a 24. ábra mutatja.



24. ábra

Az interaktív webalkalmazások fejlesztésékor egyik alapvető megoldás az AJAX (Asynchronous JavaScript and XML) technika használata. Az interaktivitást a gyors és kismennyiségű adatcsere adja. A legtöbb felhasználói esemény esetében nem szükséges a teljes oldal újratöltése, hanem ehelyett a kliens csak kismennyiségű adatokat cserél a szerverrel, ami jóval gyorsabb mint a teljes oldal újratöltése.

A kommunikációt tekintve többféle megoldást alkalmaztak (Applet, frame, iframe tag), mára az XMLHttpRequest objektum terjedt el, melyet jellemzően valamilyen köztes rétegen keresztül szokás használni (JQuery, Prototype, MooTools). A hívást tekintve megkülönböztethetünk szinkron és aszinkron hívásokat. Mivel a JavaScript önmagában aszinkron ütemezőkkel rendelkezik, jellemzően az aszinkron hívási megoldás és az onComplete callback (visszahívó) függvény használata a gyakoribb. A két modellt a 25. ábra szemlélteti.



25. ábra

Az AJAX alkalmazásával a következő interaktív megoldások váltak lehetővé (a teljesség igénye nélkül):

- szerver oldali keresőlista böngészőben megjelenítése (pl.: suggestion box, az egyik első AJAX alapú demo)
- interakcióra cserélődő, módosuló felületek
- nagy mennyiségű adat hatékony megjelenítése a kliens oldalon (on demand letöltődés, példa erre a Richfaces ScrollTable-je, vagy a Facebook wall)
- szociális alkalmazás API-k (pl.: Facebook API, Twitter API)
- …

8.4.3.2. A JavaScript mint futtató környezet

A JavaScript motor nem más, mint a JavaScript nyelv egyfajta interpretere. Napjainkra nagy hangsúly helyeződik a böngésző eme komponensére, ez felelős leginkább a felhasználói interakciós élményért. A nyelv értelmezett, ez teszi lehetővé a dinamikus objektumbővítést. Minden böngészőmotor más JavaScript értelmezővel rendelkezik, éppen ezért teljesítményük eltér, egyes böngészők más-más funkcióban adnak jó teljesítményt. A JavaScript motorra szoktak úgy is tekinteni, mint egyfajta virtuális gépre, azonban ez önmagában nem állja meg a helyét, mivel nem különálló alkalmazás egyes kivételektől eltekintve a JavaScript motor szerves része a böngészőnek, éppen ezért nem leválasztható. Példa erre a Mozilla esete, mely a böngésző felületének viselkedését a XUL leírónyelvvel és JavaScripttel valósítja meg.

Korábban a JavaScriptet tartották az informatika egyik sötét foltjának, mára azonban a kép árnyaltabbá vált: egyrészt már évek óta elérhetők egyes böngészőkhöz jó, de legalábbis használható fejlesztő eszközök: (Firefox: Firebug, Chrome: beépített, Internet Explorer 6-8: Web Developer Toolbar, IE9: beépített), másrészt mára lehetővé vált a JavaScript hatékony hibakeresésre.

8.4.4. Virtuális gépek a böngészőn belül

Nem a JavaScript az egyetlen virtuális gép a böngészőn belül. Vannak olyan problémák amelyeket egyáltalán nem, vagy csak nagyon körülményesen lehet megoldani az előzőekben említett technológiákkal, technikákkal. Ezek a problémák az erős 2 dimenziós és 3D-s számításokat igénylő alkalmazásoknál és a helyi erőforrások elérésénél jelentkeznek. A böngészők napjainkra kezdik felvenni a versenyt a beépülő modulokkal ezen a téren (pl.: 3D kártya támogatásának kihasználása a böngészőn belül).

A beépülő modulok jellemzően olyan problémákat fednek le, melyek máshogy nem megoldhatók, és az új szabványok sem (HTML5, CSS3) adnak rá hatékony megoldást. Példa erre a videólejátszás: habár a HTML5 támogatja, számos videoportál nem tér át Adobe Flash Player-ről, mert a HTML5 nem rendelkezik DRM támogatással (Digital Rights Management). További példa a Google Earth, Microsoft Live illetve OVI Maps esete, ahol a Google, Microsoft, Nokia egy-egy a böngészőbe beágyazódó pluginnel oldja meg a háromdimenziós térképek megjelenését.

A jelenlegi trend szerint a legelterjedtebbek a következők:

- Adobe Flash: összetett vizualizációra (pl.: grafikonok, hang és videostream lejátszás, játékok)
- Java Applet: csak bizonyos speciális feladatokra (pl.: aláírt applet helyi fájlokhoz hozzáférése),
- Silverlight: hasonlóan az Adobe Flash-hez ez is többek között összetett vizualizációra képes.

A legnagyobb probléma ezekkel, hogy nehézsúlyúak és a böngészőkben általában külön telepítést igényelnek. A pluginek azonban számos korlátozást jelentenek: egyrészt nem mindenütt állnak rendelkezésre, másrészt nehezen karbantarthatók, zárt kódokat alkalmaznak. A rendelkezésre állás hiányára jó példa a mobil környezet, ahol számos gyártó egyáltalán nem támogat Adobe Flash lejátszót (pl.: Apple), más gyártók limitált Adobe Flash lejátszási képességekkel rendelkező beépülőket biztosítanak (pl.: korábban az Android Flash player). Másik tipikus korlátozó tényező szokott lenni a nagyvállalati rendszer, ahol központilag szabályozott/tiltott a bővítmények telepítése. A zárt, byte kód pedig az ellenőrzést, szűrést nehezíti, például a biztonsági kockázatot jelentő kódok letöltésének észlelését.

8.5. Jelenleg rendelkezésre álló absztrakt technológiák

Az előző fejezetben említett megoldások alacsonyszintű kérdésekre adnak alacsonyszintű problémákat, azonban nem nyújtanak átfogó támogatást az alkalmazásfejlesztésre. Ahhoz, hogy hatékonyan lehessen alkalmazást fejleszteni a fenti technológiák felhasználásával, mindenképpen ajánlott egy összetettebb fejlesztői platform alkalmazása.

A webre történő fejlesztést alapvetően két irányból közelítik meg a magasszintű fejlesztői keretrendszerek: az asztali alkalmazásfejlesztés irányából és a web működés logikájának irányából.

Az asztali fejlesztési irány fő célként tűzi ki, hogy úgy lehessen webre fejleszteni, hogy az alkalmazásfejlesztőnek ne kelljen törődnie a webes felület az asztali felületekétől eltérő működésével, a webes felületek leíró és programozási nyelveivel. Az ilyen jellegű keretrendszerek (RAP, GWT) teljes mértékben elfedik a web működését, a webes felületek fölé saját ablakozó, valamint widget rendszert adva.

A webes működési irányt támogató technológiák célja, hogy a web logikájába illeszkedő megoldást adjon a fejlesztésre, a kliens-szerver viselkedésmintát kihasználva biztosítson magasszintű fejlesztői támogatást (JSP, JSF). Ennek megfelelően jellemzően a HTML eszközkészletét kiegészítő tag library-k tartalmazzák az abasztrakt komponenseket, melyből HTML generálódik.

A továbbiakban sorra vesszük az előbb említett technológiákat mindegyiknél rámutatva a jelenlegi fejlesztésekre, bemutatva azok legújabb kiegészítéseit.

8.5.1. JEE Web tároló

A web konténer a háromrétegű modell felső rétegét (front-end) valósítja meg. A web konténernek felépítését tekintve a feladata a vékony kliens (böngésző) kiszolgálása. Ez azt jelenti, hogy a web konténernek kell tartalmaznia minden erőforrást (statikus és dinamikus weboldal leírókat, kliens oldali szkripteket, képeket), valamint a vékony kliens viselkedését leíró erőforrásokat (oldal navigációs leírókat). Az interakcióra vonatkozó fejlesztésről az Interakció fejezet tartalmaz részletes leírást.

Kezdetben a dinamikus weboldalak generálását tisztán Java Servlet-ek végezték. A servlet egy HTTP kérést dolgoz fel és a kérésnek megfelelően egy HTTP választ generál le, lehetővé téve a dinamikus válasz adást. Különböző URL-ekhez különböző servletek-et lehet hozzárendelni, így egy-egy címen más-más servlet figyelhet. A servlet-ek kezelését a servlet tároló (**servlet container**) végzi. Ez a komponens végzi az egyes servlet-ek életciklusának kezelését, valamint itt rendelődnek hozzá az egyes servlet-ek az egyes URL-ekhez. Természetesen a kérések párhuzamosan érkeznek a servlethez, a fejlesztőknek nem kell foglalkoznia a servletek párhuzamosításának kérdéskörével (az servlet objektumban nem szabad osztály szintű adattagokat felvenni). A servlet szigorúan véve a HttpServlet objektumból származik, a servlet konténer felelős a futtatásáért. A servlet felépítését tekintve eseménykezelőként működik. Az alábbi eseményeket kezeli: doGet, doPost, doPut, doDelete, doHead, doOptions, doTrace, valamint az init, destroy, getServletConfig és getServletInfo metógusai írhatók felül. Habár a REST kivételével a weben a GET és POST típusú HTTP üzenetküldés az elterjedt, a HTTP szabvány további metódusait is támogatja a hagyományos HTTPServlet is.

A servlet konténer, más néven servlet engine az alkalmazás szerver vagy webszerver azon része, mely szigorúan véve az alkalmazáshoz érkező kéréseket kezeli, a kérésekre választ kap, biztosítja a a MIME kódolást és dekódolást. A servlet konténer jellemzően a webszerver része, képes kezelni a HTTP és HTTPS kéréseket. A szervlet konténer tartalmazhat szűrőket, melyek képesek módosítani a kéréseket vagy válaszokat, valamint tartalmazhat biztonsági szabályokat.

A Servlet konténer nagy előnye, hogy lényegesen gyorsabb, mint a hagyományos CGI vagy Apache modulok, standard Java API-t használ, éppen emiatt a Java API-khoz hozzáfér.

A legfrissebb Servlet konténer a v3-as. A Servlet 2.5-höz képest újragondolták a servlet feladatait. A HTTP 1.0-n túl a HTTP1.1 is támogatott, mely támogatja a perzisztens kapcsolatokat (keep alive), ennek köszönhetően támogatottak a kapcsolatonkénti szálak, a thread pool. Továbbá az AJAX nem page-by-page modelt igényel, sokkal sűrűbb és kisebb mennyiségű adatmozgatás történik a szerver között. Ezt a Servlet v3 már támogatja. Érdekes újdonsága a Servlet v3-nak továbbá a Server push, mely az interakció-intenzív alkalmazásoknál hasznos (pl.: GTalk, Facebook chat), a service streaming, long polling, aszinkron polling. A technikát reverse AJAX-nak is nevezik.

A Servlet v3 előrelépést jelent a konfigurációk tekintetében is: a hagyományos web.xml konfiguráción túl lehetőség nyílik annotációkkal is servletek, servlet filterek létrehozására. Ez a megoldás lényegesen csökkenti a rendszer tagoltságát, növeli a karbantarthatóságot, produktivitást.

Maga a Servlet konténer API definiál három szkóp típust: Request, Session és Application scope-ot. Ezek a szkópok, másnéven hatókörök definiálják azt, hogy adott objektum mennyi ideig él, és ki számára érhető el:

- a Request (kérés) szkópba helyezett objektumok a kérés-válasz folyamatig élnek, utána nem elérhetőek
- a Session (munkafolyamat) szkópba helyezett objektumok a HTTP viszony ideje alatt élnek. A session lejárta után szabadulnak fel, felhasználónként egyediek
- Az Application scope-ba helyezett objektumok az alkalmazás futása ideje alatt élnek, minden felhasználó számára elérhetők.

A Java servlet-ek kezdetben csak Java nyelven íródtak. Azonban HTML tartalmak generálásánál ez elég körülményes és nehezen használható. A nehézség abból fakad, hogy gyakorlatilag kézzel kell Java kódból előállítani a HTML tartalmat, de ugyanez igaz más tartalomtípus esetében is. A gyakorlatban ez a megoldás karakterláncokkal vagy bájt tömbökkel való munkát jelent, ami nehézkes lehet a HTML, XML, stb felépítése miatt, valamint a rétegek elkülönítését nem támogatja.

Az előbb vázolt nehézségek miatt több megoldást is megalkottak, a JSP-t, JSF-et, ugyanakkor meg kell említeni a facelet-eket, mint megoldást.

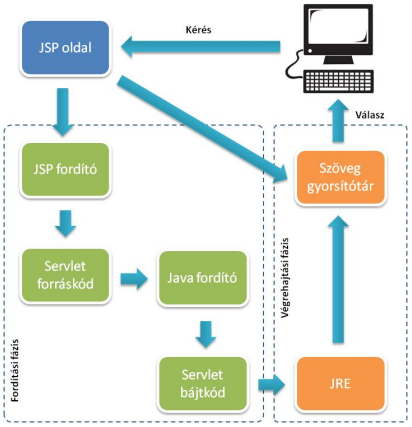
A bevezetett a szintnek a feladata, hogy egy egyszerűbb módon generálja le az egyes weboldalakat.

A JSP, JSF és a facelets kapcsolata a következő:

- A JSP egy korábbi szabványos, mely weboldalak generálására szolgál sablonok segítségével
- A JSF egy standardizált Java keretrendszer, mely MVC modellre épül
- A facelet-ek az MVC modell egy alternatív megjelenítési (View) formáját jelentik, ahol a leíró teljes egészében XML. Csak JSF-re van értelmezve.

8.5.2. Java Servlet Pages

A **Java Server Pages (JSP)** saját nyelvet használ, ami egyszerűbbé teszi a dinamikus weboldalak kialakítását. Egy JSP oldal szerkezetileg megfordítja a servletek által megismert állapotot: nem Java servletekből kell HTML kódot előállítani, hanem a rendszer lehetőséget ad arra, hogy HTML kódba ágyazzunk Java kódot. A megoldás szerkezetileg hasonló a klasszikus PHP-s megoldáshoz, ahol a HTML kódba kerülnek PHP scriptek. Működését tekintve azonban a JSP nem értelmezett, hanem fordított.



26. ábra: JSP életciklusa.

A JSP a következő képességekkel rendelkezik:

- kód blokkok injektálása ”<% Java kód %>”
- importok kezelése: hagyományos java osztályszintű importok
- bean-ek kezelése: A JSP a page, request, session és application szkópban lévő bean-eket támogatja
- további oldalak beágyazása
- tag library-k támogatása

Az előbbi lista néhány pontja magyarázatra szorul. Bevezetésre került a bean scope-ok közé egy page scope, melyben az adott oldalon elérhető objektumok vannak. A további oldalak inculde-olása vagy beillesztése lehetővé teszi azt, hogy más oldalak kerüljenek adott helyre, nagyjából a c-s include-hoz hasnoló megoldást jelent fejlesztői szempontból.

8.5.3. Java Servlet Faces

A web rohamos fejlődésével egyre komplexebb tartalomra és tartalom kiszolgálása vált szükségessé. A JSP önmagában csak egy sablonozó keretrendszer, nem választja szét az MVC modell rétegeit. A JSP lehetőséget ad arra, hogy a View rétegbe (JSP) a kontrolba tartozó betétek kerüljenek (JSP-ben Java kódbetétek).

A fejlesztés megkönnyítését újrahasznosítható komponensekkel lehet például jelentősen felgyorsítani. A JSF (Java Server Faces) egy olyan keretrendszer ami a JSP tapasztalatain alapul, komponens alapú fejlesztést tesz lehetővé. A JSF keretrendszer a következő részeket tartalmazza:

- grafikus felhasználói komponenseket megvalósító API-kat, amik többek között képesek a felhasználói események kezelésére és képesek az egyes komponensek módosítására,
- Java babokat, amelyek a különböző komponensek meghajtásáért felelősek,
- JSP könyvtárakat, amelyek megvalósítják azokat az interfészeket, melyekkel JSP oldalakról lehet JSF komponenseket vezérelni és hivatkozni,
- a kiszolgáló oldali eseménymodellt.

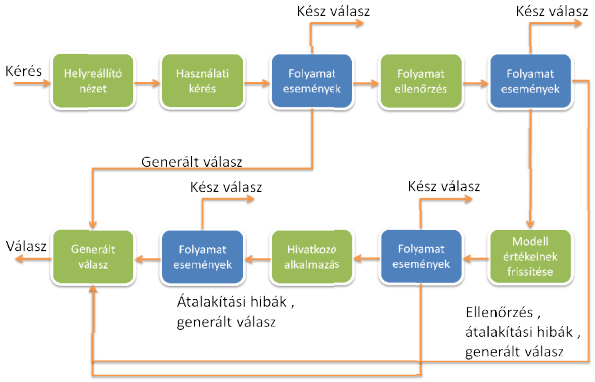
A JSF fejlesztésekor újragondolták a front-end fejlesztési lehetőségeit ötvözve azzal a céllal, hogy az MVC szétválasztható legyen. Ennek megfelelően az MVC-ben a modelnek felelnek meg az entitás babok, a View-nak a JSF leírói, a kontrolnak pedig a háttér üzleti logika babok. A JSP nem valósít meg komponens alapú eseménykezelést, a UI elemek a szerveren nincsenek reprezentálva, nem úgy, mint a JSF-ben. A JSF esetén ez technikailag azt jelenti, hogy minden egyes elemhez, ami a felhasználói felületen megjelenik, tartozik egy Java reprezentáció, egy olyan komponens, mely a UIComponent-ből származik. Ennek megfelelően minden egyes elemről lekérhető az őt reprezentáló objektum (binding attribútum), amely segítségével Java kódból szerkeszthető, módosítható.

A felhasználhatóságot a továbbiakban segíti az is, hogy a JSF képes sablonok (esetünkben **facelets**) kezelésére is, így tetszőleges oldalrészek létrehozhatóak és újra felhasználhatóak, több helyen alkalmazhatóak. A facelets egy nyílt forráskódú web sablon rendszer, amely a JSF-hez készült. A sablonok szabványos XML alapú dokumentumok, amelyek HTML kódon kívül a JSF komponensekhez tartozó XML elemeket is tartalmazhatnak.

A JSF további előnye, hogy a teljes kommunikációs ciklust vezérli: nem csak a HTML űrlapok megjelenítést írja le, de egyúttal képes a HTML űrlapok ellenőrzésére, validálására, a modellbe történő visszaírására. A validáció a webes alkalmazások működésének kritikus része: nem mindegy, hogy a felhasználók milyen formátumú és típusú adatokat adnak meg az egyes beviteli mezőknek. Validációs megoldások a JSF előtt is léteztek, csak jóval körülményesebb volt a használatuk, minden egyes mezőre külön meg kellett adni egy olyan eljárást, ami ellenőrzi a mező tartalmát. A hibák felhasználó felé történő visszajelzése is egyszerűsödött a JSF-ben. A JSF esetén a validáció már automatikus, ha szükséges és lehetséges, a JSF a konverzióról is gondoskodik. Természetesen saját konverterek és validátorok is készíthetőek a különleges formátumokhoz. A JSF a következő validációs képességekkel rendelkezik:

- beépített validációs komponensekkel
- alkalmazás szintű ellenőrzéssel
- egyéni ellenőrző komponensek készítésének lehetőségével
- belső validációs metódusok készítésének lehetőségével a háttér babokban

A konverterek és validátorok az egyes mezőkhöz egyszerűen definiálhatóak, mert a faceletek-ben egyszerűen XML-el elemekkel lehet őket hozzárendelni az ellenőrizendő mezőkhöz. A felhasználó által hibásan megadott értékek esetén is képes a hibának megfelelő hibaüzenetet a fejlesztő által megadott helyre automatikusan kiírni. A fejlesztés e részét nagyban meggyorsítja, főleg olyan esetekben, ahol rendkívül sok ellenőrizendő beviteli mező van.



27. ábra

A JSF további előnye, hogy teljesen elfedi a háttérben használt technológiákat, technikákat. Így a fejlesztőnek nem szükséges széles körben elsajátítania ezen ismereteket. Elég csak a JSF-et és a facelet-eket ismernie. Így például nem kell belemélyednie a JavaScript eseménykezelés mélységeibe, ha AJAX-ot is használó web alkalmazás készítése esetén.

Ugyanakkor a JSF előzőekben említett előnye, könnyen a hátrányává is válik azokban az esetekben, amikor egy olyan web alkalmazás készítése a cél, amelyben minden teljesen egyedi. A JSF további előnye, hogy az egyes oldalakhoz navigációs szabályokat (**navigation rules**) lehet megadni. Ezen szabályok írják le azt, hogy melyik oldalról melyik oldalra lehet eljutni. Akár olyan szabályok is definiálhatóak, amelyek egy oldalt csak akkor engednek megjeleníteni, ha a felhasználónak joga van az adott oldalnak a megtekintésére. Az egyes fejlesztői eszközökben ezeket a szabályokat grafikus felületen keresztül lehet létrehozni, így még egyszerűbbé és gyorsabbá ezek megvalósítása.

A JSF nemrégiben megjelent továbbgondolt változata a JSF2, mely számos területen szakít a korábbi JSF1.2-ben található megoldásokkal és sok területen újításokat hordoz: ideértve az a nézetek deklarációját, kompozit komponenseket, AJAX életciklus támogatást, oldal állapotának mentését, rendszereseményeket, implicit és feltételes navigációt, preemtív navigációt, HTTP/GET módú adattovábbítást, új szkópokat, továbbá egyszerűsíti a konfigurációt, hibakezelést és erőforrás-betöltést. Az új JSF bevezeti a project-stage fogalmát. Ezek közül kiemelünk most néhányat:

Nézet deklarációk: gyakorlatilag a korábban ismert Facelets 1.x API és implementáció részévé vált a JSF2-nek, amely segíti a sablonozás megvalósítását, melyet eredetileg a JSF1.x nem tartalmazott.

Kompozit komponensek: A JSF1.2 nem teszi lehetővé saját tag könyvtárak egyszerű kezelését. a JSF2-ben ez egyszerű a kompozit komponensek lehetővé teszik saját komponensek interfészeinek deklarációját, valamint hatékony interfész-írást. A megoldást a http://java.sun.com/jsf/composite taglib composite:interface és composite:implementation tagek jelentik.

AJAX életciklus támogatás: Egy új eseménykezelő, az f:ajax jelenik meg, melynek render attribútuma képes megmondani, hogy mely elemeknek kell frissülnie az AJAX hívás után. A megoldás hasonló az a4j/richfaces hasonló megoldásaihoz (pl.: a4j:support), mely ezt JSF1.2 környezetben kerülőmegoldással oldja meg.

Állapotmentés és visszatöltés: Lehetőséget ad a komponensfa mentésére és visszatöltésére, olyan segédfüggvényeket definiál, melyek lehetővé teszi ezeknek a műveleteknek az

elvégzést. Hasonló képességekkel rendelkezik, mint amit korábban az Apache Trinidad valósított meg.

Rendszeresemények: a komponensek számára lehetőséget ad arra, hogy feliratkozzanak rendszer-eseményekre, mint pre-validáció, pre-rendering.

Navigáció: egyrészt lehetővé válik a faces-config.xml nélküli navigáció, másrészt bevezetésre kerül a navigation case, mely case-if szabályok segítségével navigációt definiál. A preemptív navigáció pedig átláthatóvá teszi a navigációs döntéseket, konfigurálhatóvá teszi a NavigationHandler-t.

HTTP/GET alapú adattovábbítás: A JSF önmagában korábban rendelkezett http/ /GET alapú adatküldéssel. Néhány új tag ezt egészíti ki: egyrészt az f:metadata tag segítségével, másrészt a h:link és h:button alkalmazásával.

Új szkópok: Egyrészt bevezetésre kerül a view szkóp, mely a request-tel ellentétben nem egy kérés-válasz időtartamáig él, hanem egészen addíg, míg azt a view-t el nem hagyjuk. Továbbá bevezetésre kerül egy flash szkóp, mely a rövid ideig tartó kérés-válasz sorozatok (conversation) kezelésére szolgál. A JSF2 továbbá meghagyja a lehetőséget a saját szkóp állapotok definiálására is. Technikailag ez azt jelenti, hogy programkódból lehetséges annak megadása, hogy adott bab milyen szkópba kerüljön.

Konfiguráció egyszerűsítései: A nehézkes és nehezen karbantartható faces-config.xml-ben elhelyezett managed-bean xml leírók helyett az új @ManagedBean Java annotációk alkalmazhatók. Ezzel megszűnik a konfiguráció és kód szétválasztása, lehetővé válik az egyszerűbb kód-karbantartás és nő a rendszer átláthatósága.

Hibekezelés: az új kivételkezelő API lehetővé teszi azt, hogy hiba esetén részletesebb információk legyenek kinyerhetők a hiba okáról és környezetéről, pl kódsorok száma, mely lehetővé teszi a hatékonyabb hibakeresést.

Erőforrás betöltés: Korábban nem volt szabványos megoldás a weboldalhoz hozzátartozó további leírók betöltésére, mint például CSS vagy JavaScript fájlok. Erre a JSF2 standardizált módot biztosít. Tovább a @ResurceDependency annotáció segítségével függőségek is definiálhatók.

Project stage: A JSF2 bevezeti a projekt állapot fogalmát melyek a következők lehetnek: Development, Production, SystemTest, és UnitTest. A JSF megjelenítő motor ennek az állapotnak megfelelően tudja optimalizálni a működését, hogy éppen hibakeresés a fő szempont, vagy éppen a performancia.

8.5.4. Felületi integráció: portletek, portál

A webes felületek hatékony fejlesztésének következő lépése, hogy az alkalmazás-integráció a felhasználói felületek szintjén is megjelenjen. A Java világában erre nyújt megoldást a portleteket befoglaló portál megvalósítása (JSR 168, JSR 286).

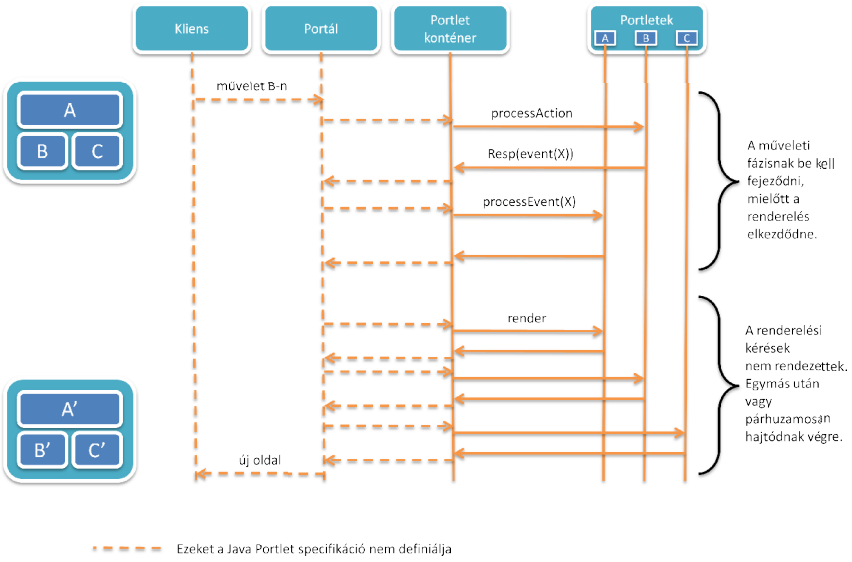
A portál egy rendkívül sokféleképpen értelmezett fogalom. Az első és talán érthetőbb definíciója az, hogy a portál egy olyan technológia-független web alapú gateway, amely lehetővé teszi különböző helyről származó releváns tartalmak együttes megjelenítését (iGoogle, netwibes). A másik, talán pontosabb definició a következő: egy olyan szoftver architektúra, melynek a szerver oldali komponense felelős a tartalmak összegyűjtéséért, az egyes tartalomdarabokat az úgynevezett portletek szolgáltatják. Mindezt a portál úgy

valósítja meg, hogy a servletekkel, JSF/JSF-fel ellentétben a portálban elhelyezkető úgynevezett portletek nem komplett HTML kódot generálnak (rendereének), hanem csak HTML kód darabokat, amit a portál motor rak össze egy teljes oldallá.

A portál nagy előnye, hogy különböző alkalmazások egyszerűen integrálhatóak egy alkalmazásba. A portál biztosítja az egyszeri azonosítást (SSO) a különböző komponensekhez. Technikailag a portál portletekből épül fel, a portál kezeli a portleteket. Minden egyes portlet külön alkalmazásnak tekinthető, a referencia és a létező megvalósítások is lehetőséget adnak a portletek közötti úgynevezett inter-portlet kommunikációra.

Technológiáját tekintve a frissített portál szabvány is elmarad a jelen szabványoktól. a JSR 262-ben definiált újragondolt portál szabvány is csak a 2.4-es servlet konténert létét írja elő.

A portletek felelősek az alkalmazások futtatásáért. A portletek egy portlet-konténerben futnak, mely nem más mint a szervelt konténer egy kiterjesztése. A kérések kezelését a következő ábra mutatja:



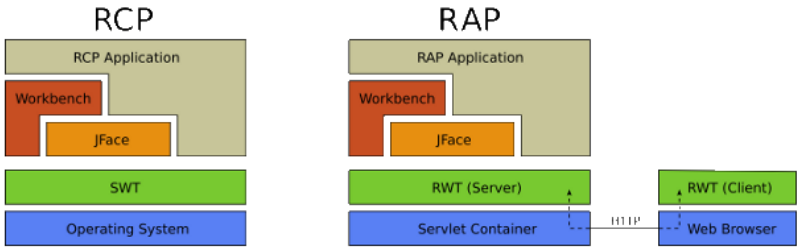
28. ábra

Súlyát tekintve a portáltól magától sokat vártak, azonban elég kevés az a terület, ahol jól alkalmazható. Leginkább a céges intranetes megoldások esetén menedzsment felületeként elterjedt. Fejlesztését tekintve számos nehézséget lehet felhozni: a hagyományos JSF alapú fejlesztést újra kell gondolni, az alkalmazások portolása portletekké nehézkes, a portletek nem támogatják a hagyományos javascript-es, AJAX-os megoldásokat, erre egyedi megoldásokat kínálnak.

8.5.5. RAP

A **RAP (Rich Application Platform)** egy nyílt forráskódú keretrendszer web alkalmazások fejlesztésére. A keretrendszer fő motivációja az úgynevezett single sourcing. Ez azt jelenti, hogy a keretrendszer fejlesztői célul tűzték ki, hogy az **RCP (Rich Client Platfrom)** alkalmazások webes felületen megjelenjenek anélkül, hogy a fejlesztőknek törődniük kelljen azzal, hogy a webre fejlesztenek vagy desktop alkalmazást írnak. A keretrendszer olyan megoldást ad, mely segítségével egyszerűen készíthetők asztali alkalmazásokhoz hasonló

alkalmazások böngészős környezetre. A keretrendszer hasonlóan az RCP-hez illeszkedik az **Eclipse** beépülő modul (**plugin**) szemléletébe, az elkészített alkalmazások maguk is Eclipse beépülő modulok. A RAP segítségével tisztán Java nyelven, JavaScript és HTML kód írása nélkül fejleszthetők webes alkalmazások. A keretrendszer teljes egészében elfedi a kliens oldali feladatokat, a web esetén a HTML és JavaScript kódok írását, illetve a CSS fájlok létrehozását. Az alkalmazás fejlesztése során a felhasználói felület az asztali fejlesztés során megszokott komponensekből épül fel. A 29. ábra összeveti az RCP és RAP platformokat:

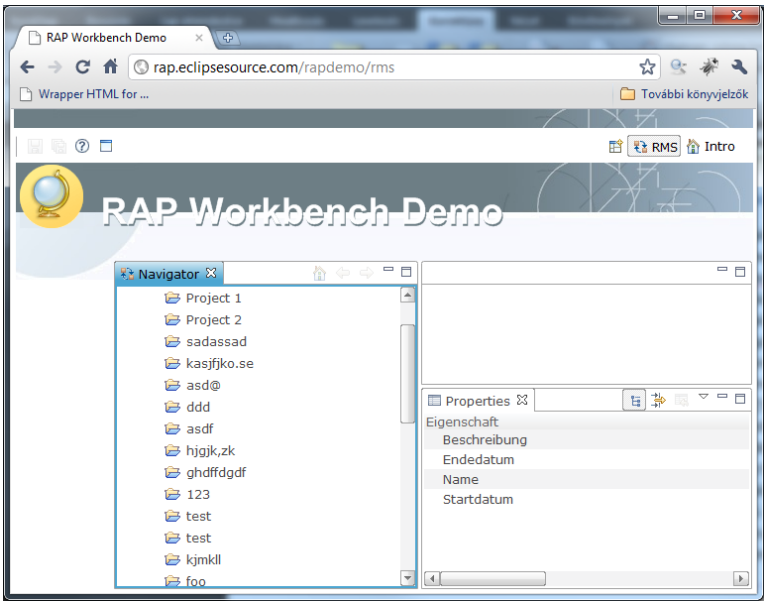


29. ábra: RCP és RAP felépítése

A alkalmazás háttérlogikáját az RCP/RAP alkalmazás írja le. Ez platformfüggetlen réteg mellyel a fejlesztőknek leginkább foglalkoznia kell. A Workbench felelős a munkaterületek kezeléséért, az ablakozásért, míg a JFace rétege egy általános komponenskezelőt valósít meg. Az RCP alkalmazás esetén az SWT widgetkezelő rendszer felelős az elemek kirajzolásáért, az eseményhurkok kezeléséért. A megjelenítés a gazda operációs rendszer feladata. Ezzel szemben a RAP alkalmazás servlet konténerre épít, az RWT a servlet konténeren futó HTML és CSS alapú komponenskezelést és JavaScript alapú eseménykezelést valósít meg.

A RAP technikailag az **Equinox** keretrendszerre támaszkodik. Az Equinox az OSGi (erről majd bővebben beszélünk a 11. fejezetben) az Eclipse közösség által készített implementációja, egy modul és szolgáltatás kezelő keretrendszer Java nyelven, mely menedzseli az OSGi komponenseket és modulokat, valamint kezeli a köztük lévő függőségeket is. Az Equinox RAP esetén Servlet tárolóban fut (**Servlet Container** - alapvetően egy olyan burok amely a HTTP parancsokhoz ad megfelelően absztrakt kezelő eszközöket).

A RAP keretrendszer jelentős részét a három nagy RCP alrendszer (SWT, JFace, Workbench) újraimplementálása teszi ki. Az SWT (Standard Widget Toolkit) tartalmazza az UI komponenseket, melyekből az alkalmazások felépíthetőek. Az SWT RAP implementációja az *RWT* (*RAP Widget Toolkit*). Az RWT nagy alrendszerek közül ez az egyetlen, ahol jelentősebb eltérések találhatók az eredeti alrendszerhez képest, ennek oka a böngészős környezetre való váltás. A JFace az SWT (RWT) felett elhelyezkedő Java segédosztály gyűjtemény, egyszerűsíti az UI fejlesztést, de nem rejti el az SWT (RWT) rétegeket. Támogatja, hogy a fejlesztőket az MVC (Model-View-Controller) elv alkalmazásában, ezáltal szétválaszthatóak az alkalmazás adatokért és megjelenítésért felelős részei. A Workbench egy Eclipse szemléletű felhasználói felület modell (mutatja be), mely kommunikációs infrastruktúrát szolgáltat, és az alkalmazás funkcionalitását különböző területekre osztja, így a felhasználói felület elemei között hierarchikus és logikai csoportosítás képezhető.



30. ábra: RAP Workbench Demo

A RAP alkalmazás a szerveroldalon fut, a kliens oldali böngészőben csak az alkalmazás megjelenítése történik, és az UI felületen végzett interakciók továbbítása a szerveroldali rész számára. A RAP kliensoldali szolgáltatásait a qooxdoo JavaScript keretrendszer segítségével valósították meg. A qooxdoo segítségével böngésző független RIA-k (Rich Internet Application) készíthetők. Egy RAP alkalmazás indulásakor a böngészőbe letöltődnek a szükséges JavaScript függvénykönyvtárak és az alkalmazás kezdő képernyőjét előállító JavaScript kód. Az előzetesen regisztrált felhasználói tevékenység hatására (például: bizonyos egér műveletek, billentyűzet használat, UI elemek eseményei) a kliens oldal AJAX hívásokkal felveszi a kapcsolatot a szerveroldallal, ahol az eseményekre kötött hurkok meghívásra kerülnek, majd az esetlegesen eközben az UI felületen történő változásokat az AJAX kérésre küldött válasz JavaScript fogja a böngészőben frissíteni. Értelemszerűen bizonyos megszorítások szükségesek a kezelt eseményekre vonatkoztatva a webes felület és a szerver közötti kapcsolat miatt.

8.5.5.1. Előnyök

A RAP egyik legfőbb előnye az úgynevezett **single sourcing**, ami azt jelenti, hogy egy RCP alkalmazást elegendő egyszer megírni, utána esetlegesen minimális átalakítással, RAP-ban is futtathatóvá tehető. A single sourcing megoldással készíthető webes alkalmazás is, ezáltal támogatja az újrafelhasználhatóságot. A keretrendszer további előnye, hogy meglévő technológiákra, eszközökre épül: ha a fejlesztő számára az adott technológia ismert, nem szükséges új elsajátítása a webes alkalmazásfejlesztéshez. Az alkalmazott fejlesztési modell így csökkenti a betanulási költségeket (**learning curve**). További előny az MVC szemlélet támogatása, mely az RCP alkalmazásokból következik.

A single sourcing megoldásból következik, hogy a webes alkalmazások elkészítéséhez nincs szükségünk HTML és JavaScript ismeretekre. A RAP Java alapú, a fejlesztés során használható a teljes Java API, vagy bármilyen külső Java függvénykönyvtár. A Java fejlesztés során megszokott fejlesztő eszközök szintén használhatók. A keretrendszer továbbá megoldást nyújt arra a problémára, hogy különböző böngészők különbözőképpen értelmezik a JavaScriptet, valamint a CSS megvalósítások sem egyeznek. A fejlesztőnek így

nem szükséges böngésző-specifikus fejlesztéssel foglalkoznia. A keretrendszer támogatja különböző erőforrások (például: képek) gyorstárazását, így a böngésző csak egyszer tölti le azokat, ez fejlesztői oldalról nem igényel többletmunkát, automatikusan támogatott. A gyorstárazás segítségével csökken a szerver és a hálózat terhelése. RAP további sávszélesség használat csökkenést elősegítő szolgáltatása a tömörített adatforgalom támogatása, valamint az, hogy a felhasználói tevékenységek során csak a változások kerülnek átküldése, így egy-egy egyszerű esemény kezelése kevesebb, mint 1 KB adatforgalommal megoldható.

A keretrendszert folyamatosan fejlesztik, ezáltal egyre több SWT-ben elérhető funkció megtalálható benne, valamint egyre könnyebb ezek használata. A kényelmesebb fejlesztést támogatja a RAP Tooling elnevezésű Eclipse plugin is, melynek segítségével egyszerűsödik a RAP projektek létrehozása, testre szabása, valamint maga a fejlesztés is.

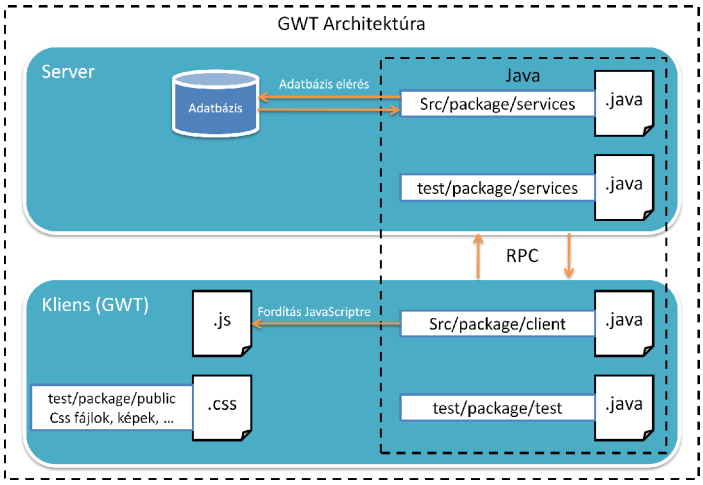
8.5.5.2. Hátrányok

A keretrendszert használva a web alkalmazásoknál megszokott testreszabhatóság stílus és design terén jóval kötöttebb, mintha a HTML és JavaScript eszközökre közvetlenül fejlesztenénk. Az RWT-ben meglévő UI komponensek köre ugyan bővíthető, viszont a bővítés támogatása elég szegényes, az ehhez elérhető dokumentáció szintén. A fejlesztés során mindenképp figyelembe kell venni, hogy az RCP alkalmazásokkal ellentétben itt egy több felhasználós környezetre történik a fejlesztés, ami sok esetben többletmunkát és alaposabb körültekintést, átgondolást igényel. A kliens oldalra kihelyezett logika csak a megjelenítésre és eseménykezelésre korlátozódik, így a szerver oldal terhelése nem csökkenthető a feladatok egy részének kliens oldalra történő átszervezésével. Az elkészített alkalmazásoknál minden egyes esemény szerveroldali kommunikációval jár, nem lehet az egyszerűbbeket sem kliens oldalon lekezelni. Sok eseménykezelő használata esetén az alkalmazás lelassulhat, több sávszélességet igényelhet.

Az alkalmazások Java EE környezetbe történő integrációja nehézkes. A keretrendszer az RCP-hez képest korlátozottabb, mivel hiányzik belőle az ott meglévő események egy része, valamint több SWT komponens tulajdonság.

8.5.6. GWT

A GWT (**Google Widget Toolkit**) egy nyílt kódú fejlesztői eszköztár összetett webes alkalmazások készítéséhez. A projekt célja, hogy a fejlesztők számára lehetővé tegye nagy teljesítményű web alkalmazások gyors elkészítését anélkül, hogy a fejlesztőknek foglalkozni kellene a különböző böngészők sajátosságaival (különböző módon megjelenített HTML oldalak, különböző JavaScript API-k, CSS). A GWT alkalmazások felhasználói felülete komponensekből épül fel, úgynevezett widget-ekből. A widget-ek segítségével a felhasználó számára információk jeleníthetők meg, valamint azokon keresztül valósul meg az adatbevitel. A GWT nem pusztán egy egyszerű JavaScript API GUI készítéshez, hanem a teljes fejlesztői folyamatot támogató eszközök összessége. A GWT alkalmazások Java-ban íródnak, majd a GWT fordító a megírt alkalmazást JavaScriptre fordítja, amely böngészőben futtatható. Nem csak a felhasználói felület és a felhasználói interakció kezelése történik a kliens oldalon, hanem a teljes alkalmazás a böngészőben fut. A fejlesztéshez a GWT SDK biztosítja a szükséges osztálykönyvtárakat, tartalmazza a fordítót, Eclipse plugin-t, hibakeresést.



31. ábra

A GWT a RAP által ismertett modellel ellentétben nem egy létező technológiára épít (SWT), hanem újat hoz létre. A megvalósítási modell is különbözik a RAP-étól, cél a teljes fejlesztési, tesztelési folyamat Java oldalon legyen tartható, és szükség esetén lefordítható legyen HTML+CSS+JavaScriptté. Ennek megfelelően a GWT ToolKit tartalmaz emulátort, mely lehetővé teszi a kód böngészőfüggetlen tesztelését, valamint egy fordítót, mely cél-környezetre fordít. A Google a cél-környezetre fordítást teljesítményre optimalizálta: ez alapvetően azt jelenti, hogy kicsi, „tömörített" fájlokat jelent, melyek ráadásul platform-függőek: a transzparanssé tevő JavaScriptes függvénykönyvtárak helyett adott böngészőre optimalizáltak jelennek meg. Ezáltal jobb teljesítmény biztosítható böngésző-oldalon.Technikailag a GWT a következőket tartalmazza:

- Java→JavaScript fordító
- Fejlesztőkörnyezeti támogatás (Debugger)
- Tesztelő keret
- Saját JRE

A GWT továbbá rendelkezik távoli eljáráshívás támogatással, lokalizációval, kódgenerációval. Támogatja a Java5 nyelvi elemeket, oldal optimizálót, képes a képek hatékony kezelésére. A GWT-ext pedig további komponenseket tartalmaz, mely ugyan szigorúan véve nem része a GWT-nek, azt egészíti ki újabb komponensekkel.

A kommunikációt tekintve a GWT számos csatornát támogat:

- GWT-RPC
- JSON
- XML

Ezek a megoldások gyárilag támogatottak, azonban a GWT sem SOAP, sem XML-RPC támogatással nem rendelkezik.

8.5.6.1. Előnyök

Előnyeit tekintve a GWT hasonló okok miatt célozza a Java nyelvi fejlesztést, mint a RAP. A RAP-pal ellentétben viszont nem igényel más jellegű előképzettséget, újonnan definiált, a JavaScripthez hasonló eseménykezelési logikát követ.

Egy GWT web alkalmazás működéséhez nem szükséges komolyabb szerver oldali erőforrás, az egész alkalmazás futhat csak kliens oldalon. Természetesen ez függ az alkalmazás által használt szolgáltatásoktól, hogy mennyi logika helyezhető kliens oldalra. A keretrendszer támogatja a legtöbb ismert böngészőt, az elkészített alkalmazások böngésző függetlenek. A keretrendszer böngészőnként generál optimalizált és tömörített kódot, a fejlesztőknek nem szükséges böngésző specifikus HTML, CSS valamint JavaScript kódokat készíteniük, így a webes alkalmazások alacsonyabb költségekkel, hatékonyabban készíthetők el amit a JSNI (JavaScript Native Interface) biztosítja számunkra.

```
public class BookJs extends JavaScriptObject
{
   protected BookJs () { }

   public final native String getName () /*-{
      return this.name;
   }-*/;

   public final native String getAuthor () /*-{
      return this.author;
   }-*/;
}
```

32. ábra

A GWT fejlett JRE emulációval rendelkezik, emiatt a Java-ban megszokott nyelvi elemek, típusok használhatók GWT-ben is, annak ellenére, hogy a JavaScript esetleg nem támogatja az adott típust (például: Long). A szokásos asztali böngészők mellett, támogatja a mai modern telefonokon, tábla pc-ken megtalálható mobil böngészőket is. Az egyes böngészőkre csak a rájuk optimalizált kód kerül letöltésre, így az alkalmazások teljesítménye és hálózati sávszélesség használata jobb lesz.

A keretrendszer számos, alapesetben is elérhető UI komponens használatát támogatja, amennyiben ezek nem lennének elegendőek saját komponensekkel; egyszerűen bővíthető, a bővítés menete jól dokumentált. A komponensek megjelenése, stílusa teljesen testre szabható, a teljes CSS eszközkészlet használható. Figyelembe kell venni, hogy ebben az esetben elveszíthetjük a böngészőfüggetlenséget. A böngészőben az adott weboldalhoz tartozó DOM (**Document Object Model**) objektumok is elérhetők a programozó számára. A GWT az UI felületek egyszerűbb elkészítését a GWT Designer Eclipse plugin segítségével. Az eszköz segítségével vizuális módon szerkeszthető az alkalmazás a fejlesztő számára rögtön látható, hogyan fog kinézni az működés közben. A GWT a gyorsabb alkalmazásindulást, valamint a kevesebb erőforrás felhasználást fejlett gyorstárazási technikákkal támogatja, mindezt úgy, hogy az alkalmazás fejlesztése során ez ne okozzon problémát.

A GWT elfedi a programozó elől az alacsony szintű JavaScript programozást, a fejlesztőnek nem szükséges azzal foglalkoznia, hogy az alkalmazása végül böngészőben fog futni, nagy előny más rendszerekkel szemben, hogy annak ellenére, hogy ez a JavaScript el van fedve, szükség esetén egyszerűen használhatók a JavaScript képességek.

A keretrendszert folyamatosan fejlesztik, egyre több területen, ahol régebben nem volt megoldása a napjainkra már van (például: log, MVP tervezési minta). A GWT képes együttműködni más keretrendszerekkel: egyrészt támogatja a JUnit teszteseteket és teszt keretrendszert, valamint képes mád alkalmazás keretrendszerekkel, például a JBoss Seam keretrendszerrel is együttműködni, az ottani alkalmazások számára UI felületet biztosítani. Szabványos WAR fájl készítésével elősegíti, hogy a GWT alkalmazások a legtöbb Java alkalmazásszerverre telepíthetők legyenek. A kliens és szerver oldal együttműködésének elősegítését távoli eljáráshívás támogatásával biztosítja, ezáltal böngésző oldalon nem támogatott vagy JavaScript technológiával nem megoldható funkciók használatához könnyen meghívható a szerveroldal.

8.5.6.2. Hátrányok

A keretrendszer hátrányai egyrészt az alkalmazott böngészőoldali JavaScript technológiából, másrészt a programozási nyelvként használt Java és a megszokott Java környezet közötti eltérésekből adódnak. A GWT bár biztosít JRE emulációt és számos előnye van annak, hogy Java nyelven fejleszt a GWT alkalmazásfejlesztő, viszont folyamatosan szem előtt kell tartani a GWT Java implementáció és a hagyományos Java közötti különbségeket. Ilyenek például a Long típus túlzott használatából eredő esetleges teljesítmény problémák, valamint, hogy a reguláris kifejezések JavaScriptben és Java-ban eltérő jelentéssel bírnak. Fejlesztés során ezen kívül nem használhatjuk a teljes Java eszközkészletet, sem külső JAR-okat, viszont meg kell említeni, hogy számos felmerülő problémára a keretrendszer biztosít saját megoldást (például: XML kezelés, log, tesztelés). Fontos tudni, hogy a lebegőpontos számítások a JavaScript interpreterek sajátosságai miatt, nem feltétlenül adják ugyanazt az eredményt, mint egy Java alkalmazás esetén. A keretrendszer nem támogatja továbbá a reflexiót és a finalizációt. Kivételkezelés során a stack trace információk, csak hibakereső módban érhetők el. A szálkezelés szintén hiányzik. A fejlesztőknek fontos tudniuk, hogy napjaink böngészői biztonsági okokból korlátozzák a távoli JavaScript hívásokat, emiatt egy AJAX hívás célja csak ugyanaz a domain lehet, mint ahonnan maga az alkalmazás is származik (**SOP, Same Origin Policy**).

8.6. Technológiai kitekintés

Az új hardver-eszközök megjelenésével egyre nagyobb teret kapnak az érintőképernyős eszközök, a multitouch megoldások, melyek további lehetőségeket adnak a webes felületek. Emiatt azonban olyan komponensek szükségesek és a különböző felületek felépítését is úgy kell megvalósítani, hogy az könnyedén használható legyen az ilyen eszközökön. Általában itt a legfőbb problémát az jelenti, hogy kellően nagy méretben kell megvalósítani a komponenseket, hogy a felhasználók ezt használni tudják. Szerencsére már találhatóak olyan keretrendszerek a fejlesztéshez, amelyek ezeket a lehetőségeket figyelembe veszik.

A jövőben valószínűleg ez a tendencia továbbra növekedni fog és fontos lesz szemmel tartani az új hardver eszközök igényeit. Az érintőképernyős megoldások nem csak nagy méretben, hanem kisebb mobiltelefonok esetében is egyre jobban fognak terjedni a csökkenő áruk miatt. E miatt viszont egyre többen használnak majd mobil böngészőket. Szerencsére ezek a böngészők követik az új technológiákat, szabványokat. A nagyobb rendszerek böngészői már fel vannak készítve a HTML 5 által nyújtott számos lehetőségre.

A HTML 5, és a vele együtt beköszönő új megoldások jelentősen átalakíthatják a web alkalmazásokat. Az egyik főbb tervezési célja a HTML 5-nek, hogy kiváltsa azokat a

megoldásokat amiket eddig csak a böngészőbe telepített különböző beépülő modulokkal lehetett elérni. Megjelenő elemek a HTML 5-ben az újfajta beviteli mezők (email, dátum), grafikai képességek (canvas), valamint streamelt adatfolyamok támogatása (audio, video tag). A nyelv emellett szakít a korábbi HTML verziók alapját képező SGML-el, de visszafelé kompatibilitását megőrzi, így a böngészőkben használt elemzők az alapvető elemeket fel tudják majd használni. A megjelenítés is teljesen különválik, így már csak kizárólag CSS-el lehet formázni az elemeket. A HTML 5 kiegészítéseként használható lesz a WebGL, amely egy olyan OpenGL alapú réteg ami képes lesz 3D-s tartalmak megjelenítésére teljes hardveres támogatás mellett. Ez újabb lehetőségeket ad a web alkalmazásoknak, mely új lehetőségeket biztosít a webes megjelenésben. Az előzőekben bemutatott új megoldások még bevezetés alatt állnak és csak részben implementáltak. A legtöbb alapjául szolgáló specifikációt azonban már elfogadták, így az alapjaik biztosítva vannak és a böngésző gyártók is felkészülhetnek az újításokra.

8.7. Összefoglaló

A fejezetben felvázolásra került néhány olyan technológia, amely segítségével hatékonyan lehetőség nyílik webes felületek készítésére, mely az alkalmazás-szerveren futó fejlesztett alkalmazások felhasználói interfészeit alkotja. A továbbiakban bemutatásra kerülnek azok a technológiák, melyek háttér-logikát képesek biztosítani a fent vázolt technológiáknak, lehetővé téve a teljes alkalmazás-rendszerek funkcionalitásának lefedését.

8.8. Ellenőrző kérdések

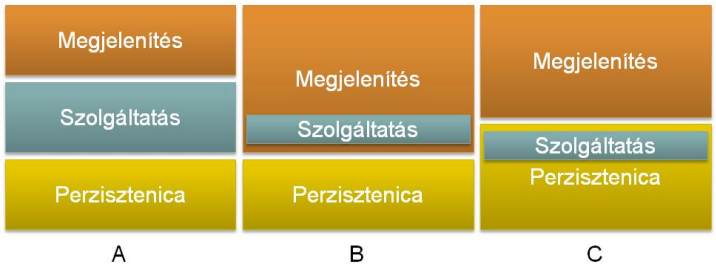
1. Milyen megoldásokkal lehetséges elfedni a webes front-end sajátosságait?
2. Milyen technológiai kihívások jelennek meg a webes felületek fejlesztésekor, hogy lehet ezt feloldani?
3. Mik a GWT és a RAP előnyei, hátrányai?
4. Milyen elképzelés mentén épül fel a JSF? Mi a működési mechanizmusa?
5. Mire való a portál, mire jók a portletek?

8.9. Referenciák

- http://w3.org – http, HTML, CSS, JavaScript W3C szabványok, ajánlások
- “Rich Client Platform - Eclipsepedia.” http://wiki.eclipse.org/index.php/Rich_Client_Platform .
- “Google Web Toolkit - Google Code.” [Online]. Available: http://code.google.com/intl/hu-HU/webtoolkit/
- JSR314: JSF specifikáció http://www.jcp.org/en/jsr/detail?id=314

9. Háttérlogika – Üzleti logika réteg

A jegyzetben több helyen is bemutattuk a rétegezett architektúrát és lehetséges rétegeket. Ezen rétegek közül a megjelenítés és a perzisztencia szükségessége, feladata magától értetődő: szükség van a felhasználói események minél gyorsabb lekezelésére és szükség van az adatok tartós tárolására is. Az üzleti logika, vagy más néven a szolgáltatás réteg feladata azonban nem ilyen világos. Ha megnézzük a ma használt alkalmazás architektúrákat akkor gyakran látjuk, hogy ez a réteg mintha hiányozna.



33. ábra

A 33. ábra A oszlopa mutatja JEE által javasolt szolgáltatás réteg kiemelt helyzetét. A B ábra azt az esetet ábrázolja amikor a szolgáltatás réteg nincs külön kiszervezve, hanem az eseménykezelés részeként van lekódolva. Ekkor a megjelenítés közvetlenül a perzisztencia biztosító réteggel kommunikál. Ez gyakran előfordul amikor a Web konténerben futó servlet-ből JDBC-n keresztül közvetlenül az adatbázist szólítjuk meg. Bár e tervezési minta amikor a szolgáltatás és a megjelenítés réteg összeolvad egyszerű komoly problémákat okoz amikor például nem csak GUI hanem más módon is hozzáférhetővé kell tennünk az alkalmazás logikánkat (pl.: van egy okos kliensünk), a gyorsítótárazást is nehézkesen használjuk ebben a modellben. A viszony réteg szintű klaszterezés szintén megvalósíthatatlan. A nagyon egyszerű alkalmazások kivételével tehát nem érdemes ezt a tervezési mintát követnünk. A C ábra azt az esete mutatja be amikor az üzleti logika a perzisztencia rétegben valósul meg. Ennek a tipikus módja tárolt eljárások használata. A PHP ökorendszerben írt szoftverek nagy része ezt a mintát követi. Vegyük észre, hogy ez esetben a programunk jó részében nem használjuk az objektum orientált paradigmát mivel a tárolt eljárások rekordok és nem objektumok szintjén működik. A biztonság egy másik olyan kérdés amit ez esetben magunknak kell megoldani gyakran a tárolt eljárásokhoz csapott „where” feltételekkel. Ez a módszer tehát nem igazán produktív. E két példából jól látható annak az előnye, ha van egy szolgáltatás rétegem. Ez a réteg különös jelentőséggel bír az AJAX-ot aktívan használó alkalmazásoknál mivel a itt tudunk hatékony gyorsítótárazás megoldásokat biztosítani. Nem mindegy, hogy a felhasználónként beérkező másodpercenkénti kéréseket memóriából vagy adatbázisból szolgáljuk ki. Az sem mindegy, hogy a szolgáltatás logikában egy objektumorientált nyelven (pl.: JAVA) vagy egy csak adatmanipulációra szakosodott nyelven valósítom meg (SQL). Azokat a keresztülívelő problémákat/ /szolgáltatásokat mint a biztonság, naplózás, tranzakciók, ... nem mindegy, hogy hogyan veszem igénybe, valósítom meg. Egy jó szolgáltatás réteget kiszolgáló köztesréteg/tároló ezeket a szolgáltatásokat magas szinten biztosítja a programozó számára.

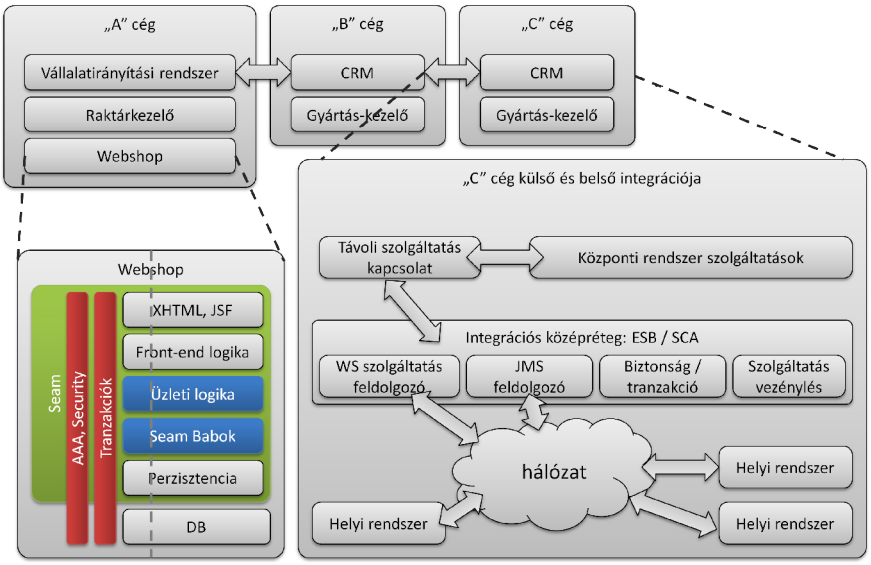
Egy jó kérdés az, hogy mit érdemes a szolgáltatás rétegbe helyezni és mit a többi rétegbe. Egy ökölszabályként azt mondhatjuk, hogy ami a felhasználói eseménykezeléshez

szükséges logika azt a megjelenítés rétegbe kell elhelyezni, ami ettől nagyobb szkópú azt viszont a szolgáltatás rétegbe. Ami alkalmazásokon átnyúló azt pedig a később ismertetett SOA koncepciót megvalósító ESB vagy SCA konténerbe.

A háttérlogika, vagy üzleti logika (*business logic*) definíciója szerint tehát arra hivatott, hogy a program lényegi algoritmusait tárolja, végrehajtsa (erről már említést tettünk a 4. fejezetben). Emellett felelős az információ közvetítéséért az adatbázis réteg és a megjelenítési réteg, valamint a további kapcsolódó rendszerek között. Ebben a fejezetben a vállalati (*enterprise*) Java megoldások által nyújtott előnyök és funkciók kerülnek kihangsúlyozásra. A következőkben felvázolunk egy klasszikus háromrétegű alkalmazás architektúrát, valamint bemutatjuk az üzleti technológiában használatos rétegeket, illetve azt, hogy ezek a megoldások milyen módon könnyítik funkcionális eszköztárukkal a fejlesztést.

A szintek elkülönítése megkönnyíti és egységessé teszi a szoftverfejlesztést. Kezdetben a sok funkcionális rész és a konvenciók betartásának kényszere azt a hatást keltheti, hogy igen bonyolult e konténerek megfelelő módon való használata. Azonban amint a fejlesztő számára letisztul, melyik egység milyen előre definiált funkciókat biztosít számára, melyeket nem kell implementálni, azonnal rájön az üzleti világ nyújtotta funkciók hasznosságára. A fejezetben bemutatjuk továbbá a tervezési mintákban fontos szerepet betöltő újrafelhasználhatóság, karbantarthatóság, skálázhatóság szempontjait is.

A 34. ábra kiemeli azokat a részeket, amellyel jelen fejezet foglalkozik.

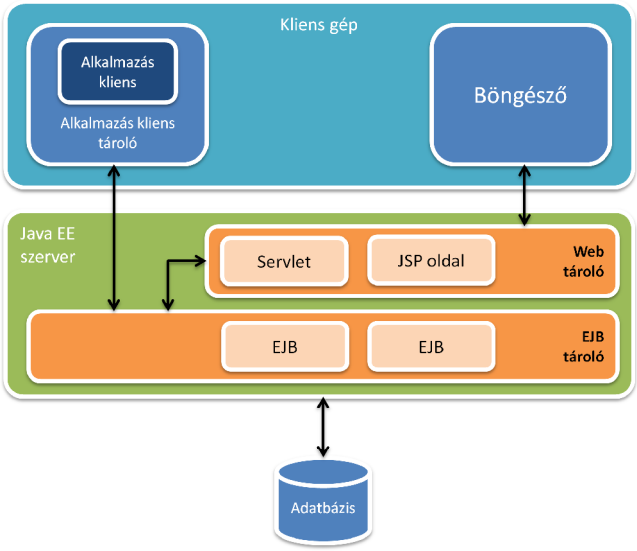


34. ábra

9.1. A Java EE keretrendszer üzelti réteg támogatása

A konténer, mint absztrakciós szint feladata, hogy interfészeket adjon a komponensek és az alacsony szintű platformfüggő funkcionalitások között, amiket a komponensek használnak. A konténer beállítások a Java EE szerver által nyújtott szolgáltatásokat szabják testre. Ilyenek például a többszálúság, biztonság, tranzakció-kezelés, névkezelés (Java Naming and Directory Interface, JNDI) és a távoli kapcsolatkezelés. A konténer magasszintű szolgáltatásai:

- *Biztonság*: a Java EE biztonsági modell biztosítja a menedzselt babok vagy web komponensek olyan módon való konfigurálását, hogy a rendszer erőforrások csak felhatalmazott felhasználók által legyenek elérhetők.
- *Tranzakció-kezelés*: a Java EE tranzakciós modell biztosítja az egy tranzakcióban szereplő összes művelet közötti összefüggések definiálását ezzel megoldva azt, hogy a tranzakció egyetlen egységként legyen kezelhető.
- *JNDI* (Java Naming Directory Interface): a JNDI-kérés szolgáltatás egységes interfészt biztosít több névkezelési és könyvtárszolgáltatás számára, ezzel lehetővé téve az alkalmazás komponenseknek, hogy elérjék ezeket a szolgáltatásokat.
- *Távoli kapcsolódás*: a Java EE távoli kapcsolódási modell (remote connectivity model) a kliensek és vállalati babok közötti alacsony szintű kommunikációt kezeli. Miután a vállalati bab létrejött, a kliens meghívja a műveleteit oly módon, mintha azonos virtuális gépben foglalnának helyet.



35. ábra: Java EE szerver és konténerek (oracle.com)

A 35.ábra foglalja össze a konténer típusokat. Az alkalmazásfejlesztésben a telepítési (*deployment*) folyamat során a Java EE alkalmazás komponensek gyakorlatilag az ábrán feltüntetett konténerek egyikébe kerülnek elhelyezésre. A Java EE szerver ebben a kontextusban a futtató környezetet reprezentálja és rendelkezésre bocsájtja az EJB illetve web konténereket. A következőkben a vállalati konténerek és a háttérlogika megvalósításai, szabványai és konkrét megvalósítások kerülnek kifejtésre.

9.2. EJB konténer

Az EJB (**Enerprise JavaBean**) a háromrétegű modell középső rétegét alkotja. A web konténerhez hasonlóan szintén az alkalmazásszerver általa menedzselt modulról van szó, mely egységbe zárja az adott működési logikát leíró komponenseket. Segítségével gyorsan és egyszerűen fejleszthetők elosztott, tranzakciós, biztonsági rendszert alkalmazó és újrafelhasználható Java technológián alapuló alkalmazások. Az EJB a Java EE specifikáció

része. Olyan szerver oldali modell, amely az alkalmazás üzleti logikáját tartalmazza. A következőkben bemutatjuk az EJB 3.0 képességeit.

Az EJB 3.0-ás kiadása az Enterprise JavaBean architektúra fejlesztés szempontjából való egyszerűsítését tűzte ki célul. Ezen egyszerűsítések több aspektust érintenek:

- Egyszerűsíti a vállalati babok interfész definíciós követelményeit: nem szükségesek a home és komponens interfészek.
- Egyszerűsíti a babok és a konténerek közötti együttműködést: a vállalati babok nem szükséges implementálniuk a javax.ejb.EnterpriseBean interfészt.
- Egyszerűsíti az API-k számára a babok környezetéhez való hozzáférést: bevezeti a függőség injektálás (**dependency injection)** lehetőségét és az egyszerűbb look-up API-kat (implicit köztesrétegként a tároló feltölti a szükséges referenciákat).
- A deployment (telepítési) leírók alternatívájaként bevezeti a Java meta adat annotációkat.
- Egyszerűsíti az objektum perzisztenciát azzal, hogy perzisztencia komponensek helyett egyszerű objektum relációs leképezési lehetőséget biztosít közvetlenül Java osztályok segítségével.

9.2.1.1. Vállalati bab osztály (Enterprise Bean)

Az EJB 3.0 API használata során a fejlesztő elsődleges programozási egysége a vállalati bab osztály. Az vállalati bab osztályhoz annotációkon keresztül rendelhetők meta-adatok, amivel az EJB konténer számára meghatározható bizonyos szemantika és követelményeket (konténer-szolgáltatási kérelem, strukturális és konfigurációs információk a konténer futtatókörnyezet számára).

A vállalati bab osztály típusát mindenképp meg kell adni. Erre a célra használható annotáció, de deployment leíróval is megadható.

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
   private float total;
   private Vector productCodes;
   public int someShoppingMethod(){...};
   ...
}
```

9.2.1.2. Vállalati interfészek

A vállalati babok megadásához szükség van egy interfész definícióra, amely segítségével az adott bab bejegyzésre kerül a regisztrációs adatbázisba (JNDI).

A **viszony (session)** és **üzenetvezérelt** (**message-driven**) babok üzleti interfészt igényelnek. Az üzenetvezérelt babok interfészét általában az üzenetküldés típusa határozza meg.

A vállalati bab osztályokra a következő szabályok vonatkoznak: a bab osztálynak implementálnia kell a megadott business interfészét. Egy bab osztálynak több interfésze is lehet a következő szabályok szerint:

- Ha a bab osztály csak egy interfészt implementál, akkor ez az interfész az osztály business interfészének tekinthető. Ez local interfész lesz, hacsak a @Remote annotációval nincs remote business interfészként definiálva.
- Ha egy bab osztály több – a lentebb felsoroltaktól eltérő – interfésszel is rendelkezik, akkor a business interfészeket explicit módon meg kell jelölni @Local vagy @Remote annotációval. A következő interfészek nem lesznek figyelembe véve annak eldöntésekor, hogy egy osztály több business interfésszel rendelkezik vagy sem: java.io.Serializable, java.io.Externalizable, az összes javax.ejb csomagban található interfész.
- Egy üzleti interfész nem lehet egyszerre helyi és távoli interfésze ugyanannak, babnak.
- Az üzleti interfész nem terjesztheti ki a javax.ejb.EJBObject vagy javax.ejb.EJBLocalObject interfészeket.

További szabályok vonatkoznak a kivételek kezelésére: a üzleti interfészek metódusain definiálhatók különböző kivételek. Azonban egy üzleti interfész metódus nem dobhat java.rmi.RemoteExceptiont, hacsak az interfész nem egy távoli üzlet interfész vagy az osztály nincs @WebService vagy a metódus @WebMethod annotációval ellátva. Amennyiben valamilyen probléma lép fel a protokoll szinten, akkor egy EJBException dobódik, ami becsomagolja a konténer által dobott RemoteExceptiont. Továbbiakban bemutatásra kerül néhány speciális technika, amely segítségével lehetővé válik néhány fejlesztési technika hatékony alkalmazása.

9.2.1.3. Állapotmentes viszony babok (Stateless session beans)

Az állapotmentes viszony bab (stateless) nem perzisztens bab. Állapota ugyan lehet, de miután a hívott metódus végrehajtódott rajta, az állapot többet nem determinisztikus, és nem is releváns. Mivel az állapotmentes viszony bab példányainak nincs állapota, így a metódusainak végrehajtásához szükséges összes adat a metódusok paraméteréből származik.

Amennyiben az állapotmentes viszony bab web szolgáltatást implementál, úgy nem szükséges web szolgáltatás interfész definiálása. Azokat a metódusokat, amelyek a web szolgáltatás műveleteiként definiáltak, @WebMethod annotációval kell ellátni. Azt a viszony bab osztályt, amit web szolgáltatás végpontként szolgál (**endpoint**), @WebService osztály szintű annotációval kell ellátni.

Az állapotmentes viszony bab osztályokat @Stateless annotációval kell ellátni és nem szükséges implementálnia a javax.ejb.SessionBean interfészt (EJB2).

A PostConstruct és PreDestroy életciklus események vannak rajtuk értelmezve. A PostConstruct callback függvények a konténer által indított dependency injection után hívódnak meg, még mielőtt bármilyen üzleti metódus meghívásra kerülne. A PreDestroy visszahívók (**callback**) a bab példány megszűnése előtt kerülnek végrehajtásra. Az interceptor létrejöttekor a hozzárendelt vállalati bab kontextusát használva végrehajtódik a dependency injection.

9.2.1.4. Állapottartó viszony babok (statefull session beans)

Egy objektum állapota magába foglalja a példány tagváltozóinak érétkét. Egy állapottartó viszony bab (statefull) példánya egy egyedi kliens munkamenetet reprezentál. Az állapotát a kliens viszony (session) ideje alatt őrzi meg. Ha a kliens eltávolítja a babokat vagy terminál, a munkamenet véget ér és az állapot elvész. Az állapot effajta tranziens viselkedése nem jelent problémát, mivel a viszony végeztével nincs szükség az állapot további megőrzésére.

Az állapottartó viszony babokat a @Stateful annotációval kell ellátni és nincs szükség a javax.ejb.SessionBean vagy java.io.Serializable interfészek implementálására (EJB2). Az állapottartó session babok a következő életciklus eseményekhez kapcsolódó visszahívási (callback) metódusokat támogatják: construction, destruction, activation, és passivation. A dependency injection az életciklus és üzleti metódusok meghívása előtt megtörténik.

Az állapottartó viszony babok egy menedzselt metódusán használt @Remove annotáció hatására a konténer eltávolítja a bab példányt az annotált metódus meghívása esetén a lefutása (normális vagy abnormális) után.

9.2.1.5. Üzenetvezérelt babok (message-driven beans, MDB)

A message-driven bean (MDB) olyan vállalati babok, ami üzenetek aszinkron feldolgozását teszi lehetővé Java EE alkalmazások számára. Gyakorlatilag eseményfigyelőkhöz hasonló JMS üzenetfigyelőként viselkedik, azt leszámítva, hogy JMS üzeneteket kap események helyett. Az üzeneteket küldhetik Java EE komponensek (alkalmazás kliens, másik vállalati babok vagy web komponens), JMS alkalmazások vagy akár Java EE technológiát nem használó rendszerek.

A session és a üzenetvezérelt babok között az alapvető különbség az, hogy a klienseknek nincs az interfészeken keresztül hozzáférésük az üzenetvezérelt babokhoz. A viszony babokkal ellentétben az üzenetvezérelt babok csak bean osztály deklarációval rendelkeznek, interfésszel nem.

Az üzenetvezérelt babok több szempontból is hasonlítanak az állapotmentes viszony babokra:

- a MDB példányok nem tárolnak adatokat vagy állapotokat a kliensek számára
- a MDB példányok azonosak, ezzel lehetővé téve a konténer számára, hogy az üzeneteket bármelyik példányhoz hozzárendeljék
- egy MDB képes több különböző klienstől érkező üzenetek feldolgozására

Az üzenetvezérelt babok tagváltozói képesek bizonyos állapotok tárolására a kliens üzenetek feldolgozása során. Például JMS API kapcsolatok, nyitott adatbázis kapcsolatok vagy vállalati bab objektum referenciák. A kliens komponensek nem tartalmaznak MDB-ket és nem hívhatják közvetlenül a metódusaikat. Egy kliens például egy olyan JMS üzeneten keresztül érheti el az MDB-t, ahol a MDB osztály a MessageListener az üzenet célállomásán.

Az üzenetvezérelt babok a további tulajdonságokkal rendelkeznek:

- kliens üzenet érkezésekor végrehajtódnak
- aszinkron módon viselkednek

- relatív rövid élettartamúak
- nem reprezentálnak közvetlen adatbázis adatokat, de hozzáférhetnek és frissíthetik azokat
- állapotmentesek
- kezelik a tranzakciókat

A fenti három komponensfajta tartalmazza a szabványos JAVA EE 5 architektúrán futó alkalmazások üzleti-logikai viselkedés-leíróját.

9.2.1.6. Meta adat annotációk és telepítési (deployment) leírók

Meta adatnak nevezzük azokat a leírókat, melyek a tényleges feladatokat végőz kód kiegészítéseként a telepítési vagy a viselkedési információkat tartalmaznak. A J2SE 5.0 egyik fontos újítása a programkód annotációjának lehetősége (JSR-175). Az annotációkat felhasználva a programozók a Java forrásfájlok segítségével összetett xml konfigurációs fájlok írása nélkül befolyásolhatják az alkalmazások viselkedését.

Az annotációk használata az egyik kulcsfontosságú egyszerűsítése az EJB 3.0 szabványnak, mivel a meta adatok annotációval specifikálhatóak. Az annotációk feladata a konténerek viselkedésével kapcsolatos elvárt követelmények, szolgáltatások vagy erőforrások injektálása és objektum relációs hozzárendelések megadása. Az annotációk az Enterprise JavaBean specifikáció korábbi verzióiban használt deployment leírók alternatívájaként is használhatók. A specifikáció szerint a deployment leíró a meta adat annotáció alternatívája, vagy az annotációk felülírásának egy módja.

9.2.1.7. Eseményvezérelt programozás

A vállalati babok esetén lehetőség van hurkok beépítésére, ezek az interceptorok. Az interceptor egy olyan metódus, amely elfogja a menedzselt babok metódushívásait vagy életciklus hurok eseményeket (**callback**). Az interceptor metódus megadható a babban vagy a babhoz rendelt interceptor osztályban. Az interceptor osztály egy olyan (a babtól különböző) osztály, melynek metódusai business metódushívásokra válaszolva vagy életciklus eseményekre hívódnak meg. Interceptorok session és üzenetvezérelt babokhoz definiálhatók.

Az interceptor osztályokat a @Interceptors annotációval lehet megadni a bab osztályokon. Az alapértelmezett interceptorok (amelyek minden viszony és üzenet vezérelt babra alkalmazva vannak) telepítés leírók definiálják. Egy bab osztályon több interceptor is szerepelhet, azonban ekkor a meghívás sorrendje a definiálás sorrendjétől függ. Egy interceptor osztálynak rendelkeznie kell egy paraméter nélküli konstruktorral. Az interceptorok állapotmentesek (**stateless**). Az interceptor példány életciklusa megegyezik azzal a bab példányéval, amelyhez hozzá van rendelve.

Az életciklus hurok (callback) interceptorok arra szolgálnak, hogy bizonyos értesítéseket szerezzünk a menedzselt babok életciklus eseményeiről. Ezeket a metódusok a következő annotációkkal lehet megadni: @PostConstruct, @PreDestroy, @PostActivate, @PrePassivate.

Az alkalmazáslogika építőkövei a munkamenet babok. A viszony babok életciklusát tekintve felhasználónként új, egyedi példányt jelent. Alapvetően két fajta menedzselt viszony babok különböztetünk meg: állapotmentes (**stateless**) és állapottartó (**stateful**) viszony babokat (**session bean**).

9.2.1.8. Perzisztencia

Ahogyan az korábban is kiemelt szinten taglaltuk, az EJB 3.0 fő célja a fejlesztési munka megkönnyítése, a megírandó kód egyszerűsítése, karbantarthatóságának könnyítése. Nincs ez másként a perzisztencia területén sem. Az EJB technológia egyik nagy újítása a Java Persistance API (JPA) bevezetése, ami egyszerűsíti az entitás perzisztencia modellt és olyan lehetőségeket biztosít, amelyek az EJB 2.1-ben nem voltak jelen. Foglalkozik relációs adatok Java objektumokká („perzisztens entitások") való leképezésével, ezek relációs adatbázisban való tárolásával, amely segítségével az adatok később is hozzáférhetők lesznek. A perzisztencia egyszerűsítése érdekében a JPA szabványosítja az objektum relációs leképezést. A perzisztenciával kapcsolatos problémákkal, lehetőségekkel a Perzisztencia fejezet foglalkozik.

9.2.2. Összefoglaló

A fejezet ezen részében bemutatásra kerültek azok a szabványok, amelyek leírják a vállalati alkalmazásokban használt alkalmazáslogika megoldások alapjait nyújtó alrendszerek képességeit. A következőkben bemutatásra kerül egy nyílt kódú implementáció, mely a korábban említett szabványokat teljes mértékben támogatja, ugyanakkor az előző fejezetben bemutatott felhasználói interakcióval kapcsolatos leírt technológiákat is megfelelően integrálja.

9.3. Web Babok

A JBoss Seam egy olyan alkalmazás keretrendszer, ami a Web 2.0-ás alkalmazások jelenlegi és következő generációjának kifejlesztését szolgálja olyan technológiákat egyesítve és integrálva, mint az AJAX, Java Server Faces (JSF), Enterprise Java Beans (EJB3), Java portletek és üzleti folyamatmenedzsment (Business Process Management – BPM).

A Seam kifejlesztésének célja a komplexitás csökkentése volt, mind architekturális szinten, mind fejlesztői szemponból. A Seam segítségével a fejlesztők komplex web alkalmazásokat készíthetnek POJO osztályok, komponensekre bontott felhasználói felületelemek és XML leírók felhasználásával.

A Seam előtt egyedül a HTTP munkamenet (session) segítségével volt lehetséges a webes alkalmazások állapotának kezelése. A Seam többféle, állapotfüggő, különböző összetettségű kontextust nyújt a felhasználói felület párbeszéd alapú folyamatainak területétől az üzleti folyamatok területéig, amivel a fejlesztők felszabadulnak a HTTP munkafolyamatok kötöttségei alól.

9.3.1. Keretrendszer az EJB 3.0 használatához

Az EJB 3.0 megváltoztatta az EJB komponensekről alkotott hagyományos elképzelést: kezdetben az EJB durva szemcsézettségű, „nehéz" objektumokat jelentett. A Seam segítségével viszont „könnyű", finom szemcsézettségű egyszerű objektumokként is tekinthetünk a komponensekre. A Seamben minden osztály lehet EJB-komponens – a Seam megszünteti a megjelenítő komponensek és az üzleti logikát megvalósító komponensek közötti megkülönböztetést, és egységes komponens-modellt hoz létre a nagyvállalati Java platform számára.

A Seam-et az EJB 3.0-val való használatra tervezték: ez lehetővé teszi az új komponens modell használatát. Mivel a Seam-ben bármilyen osztály lehet EJB komponens, nincs szükség további (különben szükségtelen) rétegek bevezetésére a keretrendszer működéséhez. Természetesen nincs szükség kódírásra az EJB 3.0 webes keretrendszerhez történő integrálásához, mivel a Seam ezt már magában foglalja.

Fontos megjegyezni, hogy nincs feltétlen szükség az EJB 3.0-ra a Seam használatához. Amennyiben olyan környezetben történik a fejlesztés, amely nem támogatja az EJB 3.0-át, a Seam erre is nyújt alternatívát. Jó példa erre a JBoss beágyazott (embedded) konténere, amely lehetővé teszi nem EAR-alapú komponensek futtatását könnyűsúlyú („lightweight") alkalmazásszervereken, mint például a Tomcat.

A Seam a következő területeken mutat újat, többet a többi nyílt kódú alkalmazásfejlesztő keretrendszertől: JSF és EJB3 integráció, AJAX integráció, munkafolyamatok átfogó kezelése, perzisztencia és biztonság. A továbbiakban a Seam ezen képességei kerülnek bemutatásra.

9.3.2. JSF integrálása EJB 3.0-val

A Java EE 5-nek talán a két legnagyobb újítása a JSF és az EJB 3.0. Míg az EJB 3.0 a szerver-oldali üzleti- és perzisztencia logika új komponense, addig a JSF a megjelenítési réteg megvalósításáért felel. Ezen összetevők közül azonban egyik sem képes önmagában minden feladat megoldására. A két komponenst legjobb együtt használni. A Java EE 5 specifikáció nem kínál szabványos módot a két modell integrálására. Szerencsére mindkét modell megalkotói előre látták ezt és olyan kiterjesztési pontokat biztosítottak, melyek segítségével kiterjeszthetők és integrálhatók más keretrendszerekkel.

A Seam egyesíti a JSF és az EJB 3.0 komponenseket, elkerülve ezzel az úgynevezett „ragasztó" kódot (olyan forráskód, mely az alkalmazás logikai működésre nincs hatással, csak a különböző komponenseket, API-kódokat köti össze), így a fejlesztők az üzleti logikára koncentrálhatnak. Lehetséges olyan Seam alkalmazásokat írni, melyekben minden menedzselt osztály EJB-komponens.

A Seam a JSF-et további funkciókkal terjeszti ki a többablakos működéshez és a munkaterület kezeléséhez, a modell-alapú ellenőrzéshez, a munkafolyamat-kezeléshez (ld. jBPM - a következő fejezetben kifejtésre kerül), a lokalizációhoz (különböző nyelvterületekhez való igazítás), illetve az oldaltöredékek gyorsítótárazásához.

9.3.3. AJAX integráció

A Seam támogatja a legjobb nyílt forrású JSF-alapú AJAX megoldásokat (JBoss RichFaces és ICEfaces). E technológiák felhasználásával anélkül lehet AJAX támogatással felruházni a felhasználó felületet, hogy JavaScript kódot írnánk.

A Seam egy beépített JavaScript távelérési réteget kínál, melynek segítségével aszinkron módon lehet komponenseket hívni kliens-oldali JavaScriptből köztes műveleti réteg beiktatása nélkül. Emellett lehetőség nyílik JMS (Java Message Service) üzenetek küldésére és fogadására AJAX hívások segítségével. Az AJAX technológiáról és a felhasználói felület megoldásairól bővebben a felhasználói interakció fejezetben olvashatunk.

9.3.4. Munkafolyamatok kezelése

A Seam átlátszó üzleti folyamat-kezelést biztosít JBoss jBPM (a Munkafolyamatok fejezetben kifejtésre kerül) segítségével. A Seam folyamatleíró nyelvek támogatásával megkönnyíti a komplex munkafolyamatok és feladatkezelés megvalósítását. A Seam lehetőséget nyújt olyan jPDL nyelvű munkafolyamatok készítésére a megjelenítési réteg szintjén, melyet a jBPM használ az üzleti folyamatok leírása során. A JSF rendkívül gazdag eseménymodellt biztosít a megjelenítési réteg számára. A Seam ezt a modellt a jBPM üzleti folyamataival kapcsolatos eseményekkel terjeszti ki, ezáltal egy egységes eseménykezelési modellt biztosítva a Seam egységes komponens modelljéhez.

9.3.5. Perzisztencia

Üzleti alkalmazások egyik kulcskérdése az adattárolás, perzisztencia. A Seam állapotkezelő architektúrája eredetileg perzisztenciával (különösen az „optimista" tranzakció kezeléssel) kapcsolatos problémák megoldására lett kifejlesztve. A skálázható on-line alkalmazások esetén szinte minden esetben „optimista" tranzakció kezelést használnak. Egy atomi szintű (például adatbázis vagy a Java Transaction API) által definiált tranzakciónak nem szabad áthidalni egy teljes felhasználói interakciót, mert a zárolások miatt a párhuzamos adatbázis műveletek szinte lehetetlenné válnak.. A legtöbb munka során azonban először adatokat kell megjeleníteni a felhasználó számára, majd nem sokkal később frissíteni is kell azokat. A Hibernate-et úgy tervezték meg, hogy támogassa az „optimista" tranzakciót magába foglaló perzisztencia-kontextust.

A Seamet és az EJB 3.0-át megelőző úgynevezett állapotmentes architektúrák nem rendelkeznek az „optimista" tranzakciót megvalósító konstrukciókkal. Ehelyett ezek az architektúrák atomi tranzakciókat megcélzó perzisztencia-kontextust szolgáltatnak. Az EJB 3.0 felismerte ezt a problémát és bevezette az állapottartó komponenst (egy állapottartó viszony bab) kibővített perzisztencia-kontextussal megcélozva a komponens teljes élettartamárát. Ez a probléma részleges megoldása, mivel újabb két probléma merült fel:

- az állapottartó viszony bab életciklusát manuálisan kell kódból kezelni a web rétegben
- a perzisztencia-kontextus állapottartó komponensek között való terjesztése az azonos „optimista" tranzakciók között lehetséges, bár nehézkes

A Seam az első problémát úgy oldja meg, hogy a megfelelő szkópú babokat (conversation scope) biztosít a műveletekhez. (A legtöbb művelet „optimista" tranzakciót reprezentál az adatrétegben.) Ez a legtöbb egyszerű alkalmazásban elegendő, ahol a perzisztencia réteg tartalmának terjesztése nem szükséges. Problémát okozhat azonban olyan összetettebb alkalmazásokban, ahol fontos az azonos műveletben szereplő, egymással interakcióba lépő komponensek közötti perzisztencia-kontextus terjesztés. Így a Seam műveletorientált perzisztencia-kontextussal bővíti ki az EJB 3.0 perzisztencia-kontextuskezelési modelljét.

A Seam párbeszéd-modellje (conversation model) sok perzisztenciával kapcsolatos programozási problémára ad megoldást, amelyeket a hagyományos állapotmentes webes alkalmazás architektúrák okoznak. Akár Hibernate-et, akár JPA-t használunk, a Seam leegyszerűsíti a kiterjesztett perzisztencia-kontextusok használatát. Segít a szükségtelen

állapotreplikáció elkerülésében, amikor kiterjesztett perzisztencia-kontextus van használatban, klaszterezett környezetben.

9.3.6. Annotációk

Az annotáció, mint java nyelvi elem a JLS 3-as verziójában (Java Language Specification 3, a Sun Java 5 implementálja) jelent meg. Lehetőséget ad különböző nyelvi elemek jelölésére „annotálására”.

Hagyományosan a Java közösség azzal a zavaró problémával küzdött, hogy pontosan milyen meta-információk számítanak valójában konfigurációnak. A J2EE és más népszerű könnyűsúlyú konténerek XML-alapú telepítési leírást biztosítanak mind olyan dolgokra, amelyek valóban konfigurálhatók (a rendszer különböző telepítései között), mind olyan beállításokhoz, amelyek Java nyelven nehezen kifejezhetőek. A Java 5 annotációi gyökeresen megváltoztatták ezt a helyzetet.

Az EJB 3.0 magába foglalja az annotációkat és a „kivételek megadásával történő konfigurációt” mint a konténerekkel való információ-megosztás legkönnyebb deklaratív formáját. A JSF azonban még mindig erősen függ a bőbeszédű XML konfigurációs fájloktól. A Seam kiterjeszti az EJB 3.0 által kínált annotációkat a deklaratív állapotkezelés és a kontextusok elhatároláshoz. Ezzel elkerülhetővé válik a JSF által kezelt babok deklarációja, továbbá olyan szintre csökkenti a szükséges XML leírók számát, hogy csak a szükségszerűen XML fájlban megadandó információk esetében kell azokat használni (például a JSF navigációs szabályok).

A Seam az első programozási modell, ami lehetővé teszi a Java 5 annotációk használatát végponttól végpontig, a perzisztencia rétegtől egészen a felhasználói felületig. Sosem kell bőbeszédű XML leírásokkal foglalkozni. Ez nem azt jelenti, hogy a Seam nem használ XML-t (a Seam egy kifinomult XML-alapú komponens konfigurációs képességgel is rendelkezik), hanem hogy az általános programozási feladatok során a fejlesztő nem vész el az XML-ek rengetegében.

9.3.7. Automatizált integrációs tesztelés

Az automatizált egység-alapú (unit) tesztekre minden fejlesztési folyamat során szükség lehet. Több mint veszélyes azonban kizárólag az egység-alapú tesztelésre támaszkodni. A legtöbb hiba a komponensek közötti együttműködésre és a komponensek illetve a konténer-környezet közötti együttműködésre vezethető vissza. Az egység-alapú tesztek nem képesek arra, hogy megfelelően modellezzék a konténer viselkedését, és rendszerint nem írják le az összetett, komponensek közötti együttműködést sem. A Seam az automatizált együttműködés-tesztelés innovatív megközelítését vezeti be, ahol egy kérés vagy párbeszéd teljes folyamata szimulálható, és tesztelhető a Java kód minden rétege az alkalmazásban, a megjelenítéstől az adattárolásig.

Mivel a Seam komponensek egyszerű java objektumok, ezért természetüknél fogva tesztelhetők egység-alapú teszteléssel. A komplex alkalmazások esetén azonban az egységek tesztelése önmagában nem elegendő. Ezért a Seam az alkalmazások könnyű tesztelhetőségét alapfunkcióként nyújtja.A JUnit kiegészítésével TestNG tesztek készíthetők, melyek a felhasználóval történő teljes interakciót létre tudják hozni, a rendszer minden komponensét kipróbálva. A tesztek közvetlenül a fejlesztői környezetből (IDE)

futtathatóak, mely alatt a Seam automatikusan közzéteszi az EJB komponenseket az alkalmazásszerveren a beágyazott JBoss segítségével.

9.3.8. Munkaterület-kezelés és többablakos böngészés

A Seam alkalmazások lehetővé teszik a felhasználók számára a böngészőlapok közötti szabad navigálást. A böngészés során minden lapon különálló, egymástól biztonságosan elválasztott folyamat (párbeszéd) zajlik. Ezen felül az alkalmazások kihasználhatják a munkaterület-kezelés nyújtotta lehetőségeket azáltal, hogy a felhasználók váltogathatnak a folyamatok (párbeszéd, munkaterület) között egyazon böngészőlapon belül. A Seam segítségével lehetőség nyílik nem csak a többablakos, hanem a többablakos-szerű működésre is egy ablakon belül.

9.3.9. Biztonság

A Seam a biztonsági modell (Security API) segítségével különféle biztonsági képességekkel ruházza fel a Seam-alapú alkalmazásokat. Többek között az alábbi területeket öleli tel:

- Azonosítás: egy bővíthető, JAAS-alapú azonosítási réteg, amelynek segítségével a felhasználók különféle biztonsági szolgáltatások felhasználásával azonosíthatók.
- Személyazonosság-kezelés: futásidőben történő, a Seam alkalmazás felhasználóinak és szerepköreiknek kezelésére szolgáló API
- Jogosultságkezelés: egy meglehetősen összetett engedélykezelő keretrendszer, mely támogatja a felhasználói szerepköröket, perzisztencia- és szerepköralapú engedélyeket és egy cserélhető engedélyfeldolgozót, melynek segítségével könnyedén megvalósítható saját biztonsági logika
- Engedélykezelés: Seam komponensek beépített csomagja, mely lehetővé teszi az alkalmazások biztonsági stratégiájának egyszerű kezelését
- CAPTCHA támogatás: segítségével elkerülhető a Seam alapú weboldalak esetén az automatizált szoftverekkel vagy szkriptekkel történő visszaélés.

9.3.10. Összefoglaló

Az előzőekben bemutatásra került az EJB 3.0 egy konkrét implementációja. A következőkben egy szinttel elvontabb alkalmazás-fejlesztési megoldás kerül bemutatásra, amely lehetővé teszi, hogy az alkalmazás-logikát akár olyanok számára is lehetővé tegyük, akik nem szoftverfejlesztők. A megoldás segítségével lehetővé válik egy olyan interdiszciplináris tudás alkalmazása, melyre az üzleti modell nyelvek előtt nem volt lehetőség.

9.3.11. Üzleti folyamatkezelés (BPM)

A Seam az EJB 3.0 támogatása mellett másik nagy újdonságként vezeti be az üzleti alkalmazások számára kiemelkedően fontos üzleti folyamatmodellező nyelvek támogatását. Minden nagy rendszer számára fontos üzleti folyamatainak rendszerezése, modellezése annak érdekében, hogy egy jól átlátható képet adjon a rendszer egészéről vagy komponenseinek működéséről. Az erre használatos gyakori kifejezés az üzleti folyamatmodellezés (Business Process Modeling, BPM), amely az üzleti folyamatok modellezésének eljárása,

így az elemezhető és továbbfejleszthető. A BPM tipikusan mellőzi az informatikai részleget a fejlesztésből, szabad utat biztosítva így az üzleti folyamatok fejlesztőinek a szabad tervezésre. Minden BPM technológia saját eszközöket szolgáltat a tervezésre, így például modellező és szimuláló eszközzel, programozói és végfelhasználói felülettel.

Több előny is felsorakoztatható a BPM mellett, így például:

- Formalizálja a meglévő folyamatokat és rávilágít a fejlesztésekre.
- Automatikus, hatékony folyamatok készítésének megkönnyítése.
- Növekvő termelékenység és csökkenő emberigény.
- Nehéz feladatok számára egyszerű megoldás.
- Szabályokon, megkötéseken alapuló problémák egyszerűsítése.

Számos BPM-tervezési eljárást dolgoztak már ki, ezek közül mégis a leggyakoribb módszer a gráf-struktúra alapú fejlesztés. A gráf-orientált BPM nyelv egy jól meghatározott és könnyen értelmezhető irányítási modellt ad különböző csomópontok és köztük lévő logikai függősekre. E csomóponttípusok nyelvenként más és más értelmezést nyernek. Annak ellenére, hogy az ilyen szintű folyamatmodellezést leggyakrabban rendszerek egészére szokás definiálni, vannak esetek, amikor kisebb feladatok megfogalmazására is tökéletes. Ilyen például a weboldalak közötti átmenetek, tehát logika kialakítása, melyet oldalafolyamatnak (pageflow) nevezünk. Ekkor minden csomópont egy oldalnak felel meg, a köztük lévő kapcsolatok pedig a weboldalak közötti logikának felelnek meg. Egy célszerű példa: a banki átutalás; ekkor ellenőrizendő tényező a maximális, megerősítés nélküli összegek kezelésére vonatkozó szabály. E szabály figyelembe vételével egy „visszaigazoló” vagy „figyelmeztető” oldalra küldhetjük tovább a felhasználót. Jellemzően ezen oldalfolyamat felett további folyamatmodellek helyezhetők el, melyek a banki átutalás részletes lépéseit takarják: igényfeldolgozás, bankszámlán lévő összeg ellenőrzése, fedezettől függően pedig átutalás megkezdése vagy elutasítása, végül visszaigazolás.

Egy másik gráf alapú modellező a BPEL, mely webszolgáltatások között próbál egy komplex folyamatot létrehozni. E gráf alapú eszközök közös tulajdonsága, hogy az objektum-orientált programozás képességeit használják ki, miközben szolgáltatnak a modellhez további funkciókat, természetesen nem felfedve a háttérben használt objektum-orientált programozás mivoltát.

9.3.11.1. JBPM

További elterjedt technológia az üzleti folyamatok modellezésére a jBPM. A jBPM a JBoss fejlesztői által kifejlesztett nyílt forrású eszközök összessége. Gyakorlatilag a BPEL-hez hasonló folyamatmodellt ábrázoló technológia, mely saját nyelvvel rendelkezik, ez a jPDL. A jPDL mind az üzleti folyamatok tervezői, mind a fejlesztők számára közös felületet és folyamatmenedzselő lehetőségeket nyújt. Használatával számos előny érhető el: feldolgozó motorja rugalmas és skálázható megoldást nyújt alkalmazások és szolgáltatások között. Lévén, hogy tervezésekor a szolgáltatás-orientált architektúrát (SOA) vették alapul, számos más technológiával képes együttműködni (webszolgáltatások, Java üzenetkezelés, stb.). Jellemzője még az automatikus állapotkezelés, változók és feladatok kezelése, időzített folyamatok, verziókezelés. Használatával átláthatóbbak a végfelhasználók közötti folyamatok és az alkalmazások együttműködése. A jPDL egyesíti a folyamatok leírását, végrehajtá-

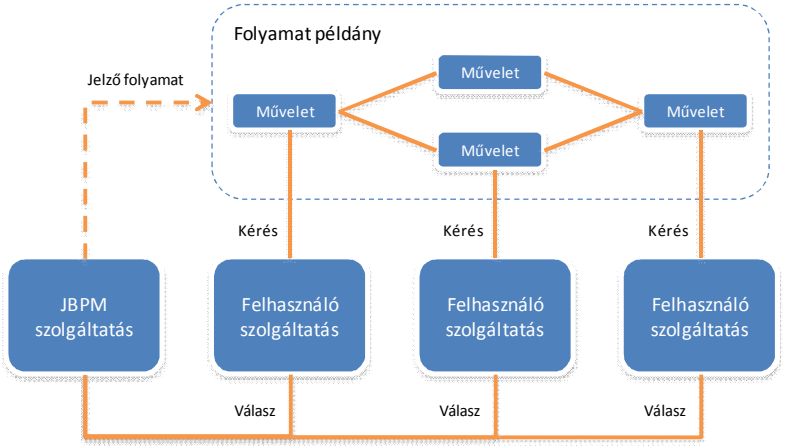
sát és menedzselését, hogy egy központosított platformot adjon a rendszer és a felhasználók közötti kommunikáció kezelésére.

Összehasonlítva a 4. fejezetben megfogalmazott követelményeket, a jBPM-re elmondható, hogy a felsorolt vezérlési minták mindegyike sikeresen alkalmazható ebben a környezetben, ehhez mindössze a nyelvben már beépített elemeket kell felhasználni. Ezek rendre:

- Egyszerű csomópontok: gyakorlatilag bármilyen interakcióval azonosíthatók.
- Feladatok: ezek az emberi interakcióknak feleltethetők meg. E pontok végrehajtásakor a modell várakozó állapotba kerül addig, amíg ez végrehajtásra nem kerül.
- Állapotok: a feladatokkal ellentétben gyakorlatilag közvetlenül képesek befolyásolni, szüneteltetni a végrehajtást, amíg ennek befejezésére egy külső jel nem érkezik.
- Elágazások és csatlakozás: a segítségükkel válik igazán teljessé a párhuzamosított végrehajtás. Jellemzően a rajtuk kívül eső végrehajtás addig nem folytatódik, amíg minden, a köztük lévő átmenet be nem fejeződött.
- Döntési pontok: két használati eset lehetséges: a folyamat leírójának kell a döntést meghozni, az általa használt adatok alapján, vagy esetleg ember vagy külső rendszer is részt vesz a döntési adat elkészítésében.
- Átmenetek: az elemek közötti átjárhatóságot biztosítják.
- Kezdő és végpontok
- „Szuper csomópont”: több elem egy felsőbb szintű elembe történő csoportosítása.

Egyedüli kivétel a több példányt is érintő minta lehet, mivel leíróhoz tartozó példányok külön léteznek. Ekkor a példányok közötti szinkronizáció szükséges a lehetségesen felvehető csomópontok segítségével. Visszavonási minták tekintetében a jBPM korlátozott, mivel irányított gráfot használ a futtatásra az egyedüli lehetőség a példány törlése. Működését tekintve egy folyamat leírást vár inputként. Ez a folyamat magába foglalja a azon tevékenységek sorozatát, amelyek a modellt leírják. Ezen tevékenység legtöbbje egy tranzakciós lépésnek felel meg. Az így előállított folyamat leírások mindegyike futtatás során egy példányként jön létre. Ezen példányok kezeléséért felelős a jBPM. Minden, a példánnyal kapcsolatos tevékenység automatikusan rögzítésre kerül. Ezekből az adatokból riportok generálhatóak aztán, hogy üzleti információkat szolgáltassanak a menedzsment számára.

A SOA-alapú rendszerek esetén gyakran használják a buszt mint integrációs platformot-, mely köré az elosztott szolgáltatások csoportosulnak. Felmerülhet az igény e szolgáltatások csoportosítására, a köztük értelmezendő koreográfiára bonyolult folyamatok végrehajtása közben, vagy az üzleti folyamat szolgáltatásként való meghajtására. A jBPM biztosítja ezt a fajta integrációt is egy szolgáltatásbusz (Enterprise Service Bus, ESB) használatával. Kétfajta lehetőség adott az ESB szolgáltatások meghívására. Az első a kérés-válasz (szinkron) típusú akciót képviseli, ami egy üzenetet helyez el a szolgáltatás buszon és egészen addig várakozó állapotban marad, amíg a válasz meg nem érkezik a meghívott szolgáltatástól. Ezt a fajta végrehajtást szemlélteti a 36. ábra. A második típusú szolgáltatáshívás egyszerűen egy üzenetcsomagot helyez el a buszon (aszinkron módon), és folytatja az előírt feldolgozást.



36. ábra: Kérés-válasz típusú akció (forrás: infoq.com)

Összegezve elmondható, hogy egy folyamatmodell tervezése fontos aspektusa egy rendszer tervezésének, erre remek lehetőséget ad a jBPM. A technológia nagy előnye másokkal szemben, hogy saját buszra csatolva egy nagyon erős és rugalmas platformot kapunk, szolgáltatás-orientált architektúra megvalósítására.

9.4. Összegzés

A fejezetben megtudhattuk, hogy hogyan lehetséges az alkalmazások logikájának fejlesztése úgy, hogy abból jól karbantartható, elosztott, skálázható és biztonságos rendszer keletkezzen. Bemutattuk a Java EE 5 specifikáció fontosabb pontjait, egy konkrét megvalósítást, majd annak kiegészítő képességeit (jBPM), mely lehetővé teszi a kód egységbe zárását, karbantarthatóságát. A következő fejezetben bemutatásra kerülnek azok a megoldások, amivel az adatok lementésre illetve visszaállításra kerülhetnek.

9.5. Ellenőrző kérdések

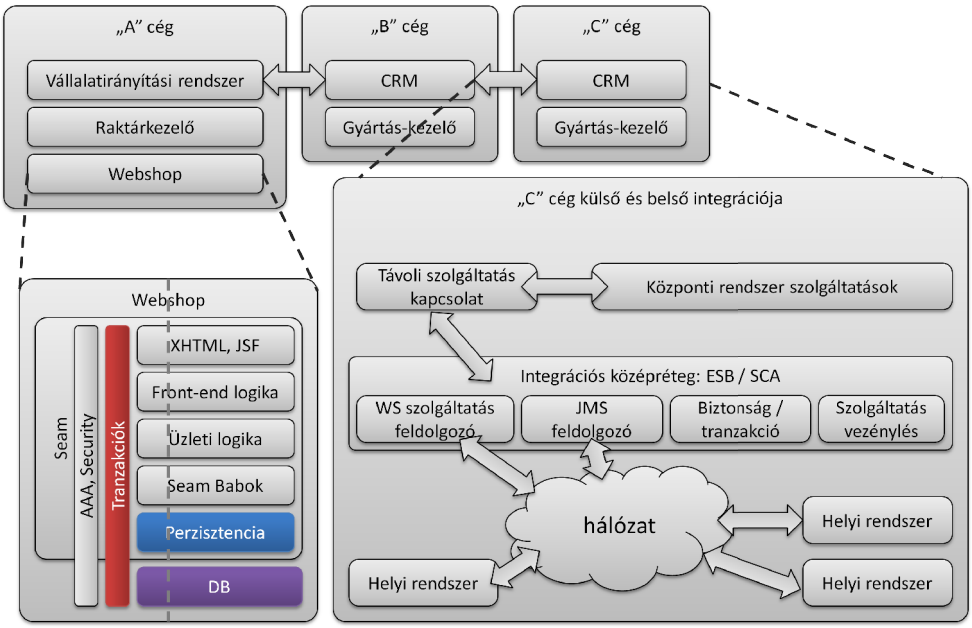
1. Mi a háromrétegű architektúra három rétege? Mely réteg mire szolgál?
2. Milyen előnyei vannak a Java EE5 implementációnak? Nevezzen meg egy Java EE 5 implementációt?
3. Mire szolgálnak az EJB és Web konténerek?
4. Milyen képességei vannak a Seam keretrendszernek?
5. Mi a jBPM, mire szolgál?

9.6. Referenciák

- Michael Havey, Essential business process modeling.
- Boris Lublinsky, “Using JBoss ESB and JBPM for Implementing VMS Solutions,” http://www.infoq.com/articles/jboss-esb-jbpm.
- EJB specifikációk http://www.oracle.com/technetwork/java/javaee/ejb/index.html
- JBoss Seam: http://seamframework.org/

10. Adatkezelés – Perzisztencia réteg

Az előző fejezet ismertette az alkalmazás fejlesztéséhez szükséges megoldásokat, viszont csak említés szintjén ismertette azokat a módszereket, amelyekkel lehetőség nyílik az adatok perzisztens (hosszú távú) tárolására. Az adatkezelés az alkalmazások, így a vállalati alkalmazásoknak is egyik sarkalatos pontja, különös tekintettel arra, hogy a konkurens hívások hatékony kezelése még bonyolultabbá tehetik a problémát. A 37. ábra kiemelt részei jelzik azt, hogy jelen fejezet az átfogó rendszer mely részeit taglalja.



37. ábra

10.1. Bevezetés

Ahogyan azt az 5. fejezetben leírtuk, az alkalmazásfejlesztés egyik sarkalatos pontja az adatok megfelelő modellezése.

Ezen fejezet az objektum orientált paradigma segítségével megvalósított adatmodellek relációs adatbázisba történő leképezését és kezelését támogató keretrendszert mutatja be. Bemutatásra kerül, hogy az objektum-relációs leképezések során felmerülő problémákra a konkrét megvalósítások milyen megoldást kínálnak.

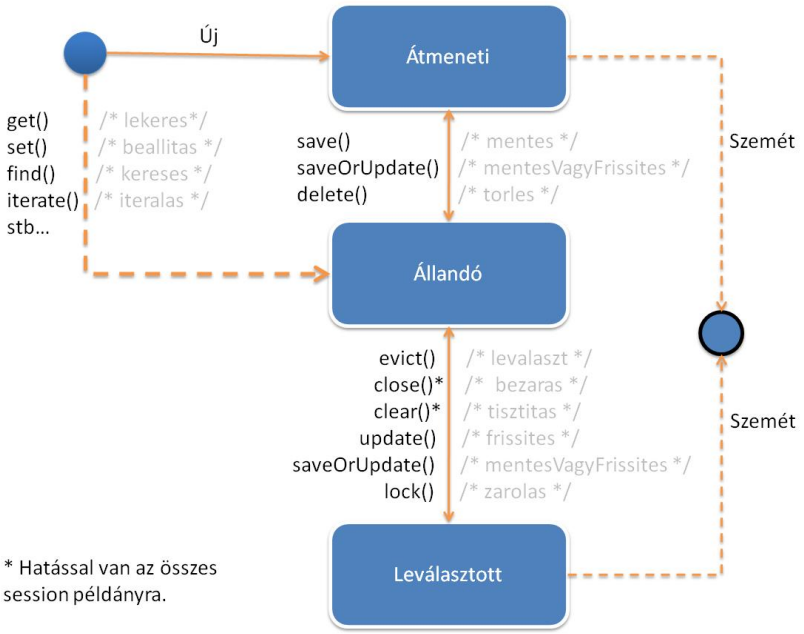
10.1.1. Entitás és életciklus

Az adatkezelés kontextusában entitás alatt olyan objektumot értünk, amelyek adatbázisban mentődnek. Az entitások a perzisztencia folyamatában különböző állapotokon mennek keresztül. A következőkben áttekintjük ezen állapotokat.

- Tranziens*:* Az újonnan példányosított (a new operátorral) objektumok még nem perzisztensek. Először az úgynevezett tranziens állapotba kerülnek. Ebben az állapotban nem kapcsolódnak semmilyen adatbázis táblához. Ezek az objektumok nem lehetnek tranzakcionálisak, azaz a rajtuk végrehajtott bármely módosítási művelet

nem lehet része tranzakciónak. Ha egy tranziens objektumot perzisztálni akarunk, vagy más szóval a tranziens állapotból perzisztensbe akarjuk helyezni, a perzisztencia menedzser save függvényét kell meg hívni az objektumra, vagy egy perzisztens objektumból hivatkozni kell a tranziens objektumra.

- Perzisztens: A perzisztens állapotban lévő objektumpéldány rendelkezik egy elsődleges kulccsal azonosított érvényes adatbázis bejegyzéssel. Egy entitás perzisztens állapotba lép a perzisztencia menedzser save függvénye segítségével. Ekkor a perzisztencia menedzserrel is összerendelésre kerül. Abban az esetben, ha a tranzakció menedzser elfogad egy tranzakciót, a benne résztvevő perzisztens objektumok az adatbázissal szinkronizálódnak. A tranzakció végeztével csak azok a perzisztens entitások kerülnek szinkronizálásra, amelyek módosultak. Ezt a folyamatot dirty chechking-nek nevezik. Piszkos objektumnak (dirty) nevezzük azt a perzisztens entitást, amely valamilyen tranzakció során módosult, de nincs szinkronizálva az adatbázissal. Ha egy perzisztens objektumon meghívásra kerül a perzisztencia menedzser delete metódusa, az adatbázisból a bejegyzés eltávolításra kerül valamint a perzisztens állapotban lévő objektum átkerül tranziens állapotba.
- Leválasztott (detached): A perzisztens objektumok a tranzakció végeztével is léteznek és tartalmaznak adatokat, azonban leválasztódnak a viszony bezáródásával. A leválasztott állapotban lévő objektumoknál nem garantált az adatbázissal való szinkronitás.



10.1.2. Állapotátmenetek

Miután megismertük az entitás egyes állapotait, megvizsgáljuk, milyen műveletek vagy eseményes következtében történnek meg az állapotváltozások.

A new operátor segítségével példányosított entitás még tranziens (nem menedzselt és nem is perzisztens) állapotban van. Amennyiben ezek után semmilyen perzisztencia művelet nem kerül rajta végrehajtásra, úgy a szemétgyűjtés során az általa lefoglalt memória

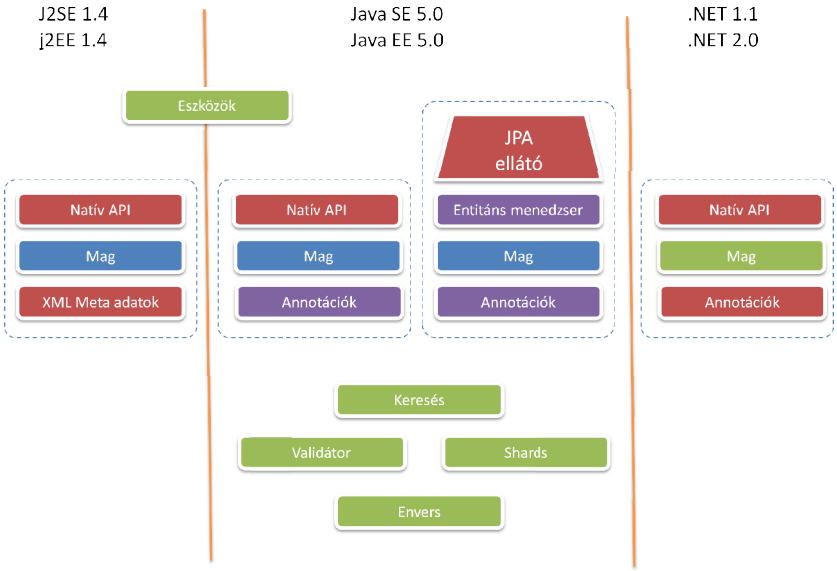
felszabadul. A save művelet hatására az entitás menedzselt (perzisztens) állapotba kerül és a tranzakció lezáródásával (commit) bekerül az adatbázisba.

A perzisztens állapú entitáson végrehajtott delete művelet hatására törlődik a perzisztens entitást, tranziens állapotba kerül. Tranzakció commit esetén az adatbázisból is törlődik. Az evict, close vagy clear műveletek végrehajtása után az entitás kikerül a perzisztencia kontextusból, leválasztott (detached) állapotba kerül. Ekkor a nem használt tranziens objektumokhoz hasonlóan a szemétgyűjtő eltávolíthatja őket a memóriából. Egy leválasztott entitás állapotának menedzselt entitásba való másolására szolgálnak az update parancs. Ekkor a leválasztott entitás ismét perzisztens állapotba kerül.

Amennyiben az adatbázisban tárolt információkkal szeretnénk példányosítani egy entitás objektumot, akkor használhatóak a load, find és get metódusok. a betöltött adatokkal egy perzisztens objektum jön létre.

10.2. Hibernate

A következőben a Hibernate nevű objektum-relációs leképező (ORM) és perzisztencia keretrendszerrel fogunk megismerkedni. A Hibernate-et Gavin King indította útjára 2001-ben, amivel az EJB2 stílusú entitásbabok használatára kívánt egy alternatívát megalkotni. Elsődleges célja volt, hogy jobb perzisztálási képességeket biztosítson, mint amit az EJB2, azáltal hogy egyszerűsítette a komplexitást és új képességeket vezetett be. 2003-ban a Hibernate fejlesztőcsapata belekezdett a Hibernate2 kifejlesztésébe, amely számottevő fejlődést mutatott az első kiadásokkal szemben és ezzel a Hibernate-et tette a Java perzisztencia „de facto” standardjává.



38. ábra

10.2.1. A Hibernate alapvető képességei

A Hibernate segítségével olyan perzisztens osztályokat (enitásokat) hozhatunk létre, melyek követik a hagyományos objektum orientált paradigmát, mint a leszármaztatás, polimorfizmus, összerendelés, kompozíció és a Java kollekciós rendszerét. Nem szüksége-

sek interfészek vagy absztrakt osztályok definiálása a perzisztens osztályokhoz, és bármely egyszerű Java osztályt vagy adatstruktúrát képes perzisztálni. Ezen felül mivel nincs build-time forrás- vagy bájtkód generálás és feldolgozás ezért gyorsabb build folyamatot biztosít.

A Hibernate továbbá támogatja a laza inicializálást, különböző kapcsolódó mező kitöltési stratégiákat (fetching) és az optimista zárolást automatikus verziózással és időbélyegezéssel. Nincs szükség speciális adattáblákra vagy mezőkre, a legtöbb SQL lekérdezést futásidő helyett már a rendszerinicializáláskor legenerálja a Hibernate keretrendszere.

A Hibernate támogatja a Hibernate Query Language-et (HQL), Java Persistance Query Language-et (JPQL), az úgynevezett Criteria Query-ket és natív SQL lekérdezéseket.

10.2.2. Leképezés XML használatával

A Hibernate-nek, ahogyan minden más ORM eszköznek szüksége van olyan meta információkra, amelyek az adatok egyik reprezentációjából a másikba (és fordítva) való leképezését írják le. A Hibernate 3.2-es verzióját megelőzően ezeket az információkat XML fájlokban lehetett megadni. Ezek az úgynevezett mapping fájlok. A mapping fájlok leírják, hogy az adott entitás az adatbázis mely táblájának felel meg, az entitás mely mezői mely adatbázis oszlopokba kerülnek leképzésre. A mapping fájl struktúráját a következőképp néz ki:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.company.project.domain">
    <class name="Event" table="EVENTS">
     <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

A hibernate-mapping tegek között egy class határozza meg az osztály-tálba összerendelést. Az ORM leképezés alapján az adatbázis egy táblája egy osztályt, a tábla egy sora egy objektumot reprezentál.

10.2.3. Annotációk

Az objektum relációs leképezéshez a Hibernate 3.2-es verziójától kezdve lehetséges a JDK 5.0 által kínált annotációk használata. Az annotációk használhatók az XML leképezések mellett és azok helyett is.

A Hibernate tartalmazza a standardizált Java perzisztencia és az EJB 3.0 (JSR 220) objektum relációs leképezés annotációit, továbbá olyan Hibernate specifikus annotációkat, melyek a teljesítmény optimalizálás és bizonyos speciális leképezések során használhatóak. A Java perzisztencia annotációi mellett a Hibernate annotációinak alkalmazásával hasznosíthatók a Hibernate natív képességei.

A JPA (Java Persistance Api) entitásai egyszerű Java objektumok (POJO), melyek egyben Hibernate perzisztens entitások is. Az annotációk két jól elkülöníthető csoportba sorolhatók: logikai (leírják az objektum modellt, két objektum közötti kapcsolatot, stb.) és fizikai leképző annotációk (leírják a fizikai sémát, táblákat, oszlopokat, indexelést, stb.).

Minden perzisztálandó POJO osztály egy entitás és a @Entity annotációt használva osztály szinten definiáljuk.

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

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

A @Entity az osztályt, mint entitást definiálja, a @Id az osztály egyedi azonosító adattagját határozza meg. A példában látható Flight osztály a Flight táblában kerül tárolásra az id oszlopot használva egyedi kulcsként.

Attól függően, hogy az annotáció mezőre vagy metódusra van alkalmazva, a Hibernate által használt hozzáférési típus lehet mező vagy tulajdonság. Az EJB3 specifikáció megköveteli, hogy az annotáció a hozzáfért elemtípuson legyen, ez lehet metódus tulajdonság hozzáférés esetén vagy osztály adattag mező hozzáférés esetén. Azonban a két alkalmazásmód együttes alkalmazása kerülendő. A Hibernate a hozzáférés típusát a @Id vagy @EmbeddedId annotációk helyéből következteti ki.

```
@Entity
@Table(name="tbl_sky")
public class Sky implements Serializable {
   ...
}
```

Amennyiben meg akarjuk határozni az adattábla nevét, használhatjuk a @Table osztály szintű annotációt. Segítségével megadhatjuk az entitás számára a tábla, katalógus vagy séma nevét. Amennyiben nincs @Table annotáció, úgy az alapértelmezett tábla név az osztály egyszerű (csomag nélküli) neve lesz.

Az entitás minden nem statikus vagy tranziens adattagja (mező vagy metódus, a hozzáférési típustól függően) perzisztálandónak tekintett, hacsak nem rendelkezik @Transient annotációval. Az annotációval nem rendelkező adattagok úgy viselkednek, mintha a megfelelő @Basic annotációval lennének ellátva. A @Basic annotáció lehetőséget nyújt a fetch-elési stratégia megadására.

```
public transient int counter; //transient property

private String firstname; //persistent property
```

```
@Transient
String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

A Hibernate annotációinak itt csak egy nagyon kis szeletét láthattuk. Azonban ezekből a példákból is jól kitűnik, hogy az XML leképezéshez viszonyítva jóval kényelmesebben lehet az objektum-relációs leképezést a mapping fájlok használata helyett annotációk segítségével megvalósítani.

10.2.4. Azonosító generálás

A Hibernate lehetőséget biztosít az entitások számára kötelezően megadandó egyed azonosítók automatikus generálására. Több különböző generálási stratégiát ismer. A következőkben láthatjuk ezen stratégiák közötti különbségeket:

- Identity*:* a DB2, MySQL, MS SQL Server, Sybase és HypersonicSQL által használt identity mezőt támogatja. Értéke long, short vagy int típusú lehet.
- Sequence (seqhilo): egy hi/lo algoritmust felhasználva hatékonyan generál long, short vagy int típusú azonosítókat egy nevesített adatbázis sorozat alapján.
- Table (MultipleHiLoPerTableGenerator): a sequence-hez hasonló stratégia azzal a különbséggel, hogy itt csak tábla szintű egyedi kulcsok jönnek létre az adatbázis szintű helyett.
- Auto*:* az előző három stratégia valamelyikét használja az adatbázis képességei alapján.

10.2.5. Asszociációk leképzése

Az asszociációk helyes leképzése az egyik legnehezebb feladat. A Hibernate segítségével lehetséges mind egyirányú (unidirectional), mind kétirányú (bidirectional) kapcsolat megvalósítása. Továbbá a Hibernate támogatja az egy-egy, egy-több és a több-több kapcsolatok leképezését. Bizonyos esetekben az asszociációk megfeleltetéséhez kapcsolótáblákra is szükség van. A nullértékű külső (idegen) kulcsok használata mellőzött gyakorlat a hagyományos adatmodellezési technikákban, azonban a Hibernate-nek nem okoznak gondot, de alkalmazásuk kerülendő.

10.2.6. Perzisztens kollekciók

A Hibernate lehetőséget biztosít teljes kollekciók perzisztálására. Ezek a kollekciók majdnem bármely Hibernate típust tartalmazhatnak beleértve az alap adattípusokat, egyedi típusokat, komponenseket és más entitásokra való hivatkozásokat. Az érték és referencia értelmezése közötti különbség ebben a környezetben különösen fontos. Egy kollekcióban szereplő objektum kezelhető értékként (az életciklusa teljes mértékben a kollekció tulajdonosától függ) vagy egy önálló életciklussal rendelkező entitásra való hivatkozásként. Az utóbbi esetben a kollekció csak a két objektum közötti kapcsolatot tartalmazza.

A kollekciókat tartalmazó mezőket interfész típusként kell deklarálni. Ezen interfészek a következők lehetnek: java.util.Set, java.util.Collection, java.util.List, java.util.Map, java.util.SortedSet, java.util.SortedMap. Lehetséges olyan egyedi típus használata is, ami implementálja az org.hibernate.usertype.UserCollectionType interfészt. A kollekciók példányosítása során használhatóak a Java standard kollekció implementációi (java.util.ArrayList, java.util.HashSet, stb.) Amennyiben az entitás perzisztálásra kerül, úgy a Hibernate automatikusan lecserélni a példányokat a saját implementációira. Így visszatöltés esetén a kollekciókat csak interfészek típusaira kényszeríthetők.

A kollekció példányok általában a hagyományos érték típusokkal azonos módon működnek. Automatikusan lementésre kerülnek, ha egy perzisztált entitás hivatkozik rájuk és törlődnek, ha a hivatkozás megszűnik. Ha egy kollekció egyik perzisztált objektumból átkerül egy másikba, akkor annak elemei is átkerülnek egyik táblából a másikba. Két entitás nem hivatkozhat ugyanarra a kollekcióra. A mögöttes relációs modell miatt egy kollekció típusú adattag értéke nem lehet null. A Hibernate nem tesz különbséget a null és az üres kollekció között.

10.2.7. Öröklődés

A Hibernate következő három alapvető öröklődés leképzési stratégiát támogatja:

- osztály hierarchia szerinti táblák
- alosztályonkénti táblák
- konkrét osztályonkénti táblák

Ezeken felül támogat még egy negyedik, az előzőektől eltérő típusú polimorfizmust, az implicit polimorfizmust.

Az öröklődési hierarchiák egyes változataihoz lehetséges eltérő leképzési stratégiákat használni. Így használható az implicit polimorfizmus a teljes hierarchiában való polimorfizmus megvalósítására. Habár a Hibernate nem támogatja az azonos <class> gyökérelem alatti <subclass>, <joined-subclass> és az <union-subclass> leképezéseket, a hierarchia szerinti és az alosztályonkénti táblák stratégiái kombinálhatók.

Lehetséges különböző mapping fájlokban a <subclass>, <union-subclass> és <joined-sebclass> definiálása a <hibernate-mapping> elem alatt. Így új mapping fájlok hozzáadásával az osztály hierarchia kiterjeszthető. Meg kell adni egy extends attribútumot az alosztály leképezésnél, ahol a paraméter értéke az ősosztály neve. Korábban ez a technika figyelembe vette a mapping fájlok sorrendjét. A Hibernate3 óta azonban az extends kulcsszó használatakor a fájlok sorrendje nem számít, csak az egyes fájlokon belül az ősosztályok definíciójának kell megelőznie a leszármazott osztályokat.

10.2.8. Fetch-elési stratégiák

A kapcsolódó objektumok betöltése kritikus fontosságú feladat. Egy körkörös hivatkozású domain-modell esetén (pl.: kompozit minta) egy rosszul megválasztott betöltési stratégia képes a teljes tábla betöltésére, objektum-reprezentációinak létrehozására.

A Hibernate egy kifinomult fetch-elési eljárást használ annak érdekében, hogy az asszociált objektumok csak akkor kerüljenek betöltésre (bizonyos megkötésekkel), amikor azokra az alkalmazásnak szüksége van. A különböző fetch-elési módszerek a relációs leképezések megadásakor definiálhatók, de a HQL vagy kritéria lekérdezések során felülírhatók.

A Hibernate3-ban a következő fetch-elési stratégiákat használhatók:

- Join fetching: a kapcsolódott példányt vagy kollekciót egy OUTER JOIN segítségével azonos SELECT parancsban tölti be.
- Select fetching: a kapcsolódott példányt vagy kollekciót egy második SELECT kérdezi le. Abban az esetben, ha expliciten nincs letiltva a lazy fetching, a második select csak akkor kerül végrehajtásra, ha a kapcsolódott objektumhoz az alkalmazás valamely komponense hozzá akar férni.
- Subselect fetching: az előzőleg betöltött összes entitáshoz kapcsolódott kollekciót egy második SELECT kérdezi le. Abban az esetben, ha expliciten nincs letiltva a lazy fetching, a második select csak akkor kerül végrehajtásra, ha a kapcsolódott objektumhoz az alkalmazás valamely komponense hozzá akar férni.
- Batch fetching: egy optimalizált select fetching. A Hibernate elsődleges vagy külső kulcsok listájának megadásával a kapcsolódott entitások vagy kollekciók egy csoportját kérdezi le egy önálló SELECT-ben.

A Hibernate szintén különbséget tesz a következők fetching stratégiák között:

- Immediate fetching: egy asszociáció, kollekció vagy attribútum azonnal fetch-elésre kerül, amint a gazda objektum betöltődött.
- Lazy collection fetching: a kollekció akkor töltődik be, ha az alkalmazás valamilyen műveletet végez rajta. Ez az alapértelmezett.
- „Extra-lazy” collection fetching: a kollekcióból csak a szükséges elemek töltődnek be. Hacsak feltétlenül nem szükséges, a Hibernate megpróbálja elkerülni a teljes kollekció memóriába való betöltését. Ez nagyméretű kollekciók esetén hasznos.
- Proxy fetching: egy egyszerű érték akkor kerül betöltésre, amikor az objektumon az azonosító getter-től különböző metódus meghívásra kerül.
- „No-proxy” fetching: egy egyszerű érték akkor kerül betöltésre, amikor a példányváltozóhoz hozzáférés történik. A proxy fetching-hez képest ez a megközelítés kevésbé „lazy”, már akkor is megtörténik a betöltés, ha az azonosítóhoz hozzáfernek. A transzparenciát az is növeli, hogy az alkalmazásban nem látszódik a proxy. Ez a megközelítés buildtime bájtkód szerkesztést igényel és kevéssé használt.

- Lazy attribute fetching: egy attribútum vagy egyszerű érték hozzárendelés akkor kerül betöltésre, amikor a példányváltozóhoz hozzáférés történik. Ez a megközelítés buildtime bájtkód szerkesztést igényel és kevéssé használt.

A fetch-elés módszer kiválasztásakor alapvetően két kérdésre kell válaszolni: a mikor és hogyan kérdésére. A fetch-elés alapvetően a teljesítményt hivatott javítani, azonban körültekintően kell választani a lehetséges technikák közül. Logikusnak tűnik, hogy csak olyan objektumokat töltsünk be, amelyeket valóban használni fogunk, azonban előfordulhat olyan eset, amikor a proxy-k nem oldódnak fel vagy az entitás leválasztott (detached) állapotba kerül és ez kivételes működéshez vezet. Ez előfordulhat a session lejártával vagy a tranzakció végeztével.

10.2.9. Lekérdezések

Az eddigiekben láthatott technikák és lehetőségek mind az objektumok relációs leképezéséről és az azokkal kapcsolatos egyes problémák lehetséges megoldásairól szóltak. Azonban nem láthattuk, hogy az adatbázisban tárolt objektumokhoz hogyan férhetünk hozzá. Az SQL világában az adatbázisból való információ kinyerésére a lekérdezések használhatók. Nincs ez másként a Hibernate esetében sem. Azonban a Hibernate a hatékony objektum hozzáférés elősegítéséhez a hagyományos SQL mellett a következő lekérdező nyelveket is támogatja:

- Hibernate Query Language*:* A Hibernate Query Language (HQL) a Hibernate saját, az SQL-hez nagyon hasonló lekérdező nyelve. Az SQL-el szemben azonban teljesen objektum-orientált, olyan lekérdezéseket írhatunk benne, melyekben a táblák és oszlopok helyett osztályneveket és adattagokat használhatunk. Kezeli az öröklődést, a polimorfizmust és az asszociációt.
- Natív SQL: A Hibernate lehetőséget biztosít a kapcsolódott adatbázis SQL dialektusában írt natív lekérdezések használatára. A natív lekérdezések segítségével kihasználhatók az adatbázis speciális szolgáltatásai, mint például Oracle Connect kulcsszava. Natív lekérdezéseket használva egyszerűen migrálhatók SQL/JDBC alapú alkalmazások Hibernate-re. A Hibernate3 támogatja a tárolt eljárások használatát is.
- Named Query: Annotációk vagy Hibernate mapping fájlok segítségével osztály szinten definiálhatók úgynevezett HQL named query-k. Natív SQL lekérdezésekhez is definiálhatók named query-k mapping fájlok segítségével és a HQL named query-khez hasonló módon használhatók.
- Criteria query: A Hibernate Criteria query API segítségével objektum-orientált környezetben lehet dinamikus lekérdezéseket létrehozni olyan módon, hogy az entitásokhoz bizonyos feltételeket, kritériumokat definiálunk.

10.2.10. További képességei

Az eddigiekben a Hibernate olyan lehetőségeit láttuk, amelyek biztosítják az objektum-relációs lekérdezés megvalósítását. A következőkben röviden áttekintünk még néhányat a Hibernate által nyújtott szolgáltatások közül.

10.2.10.1. Bab érvényesítés (Bean validation)

A Bean Validation egységesíti a domain modell szintű megszorítások definiálását és deklarálását. Például meghatározhatjuk, hogy egy adattag értéke sosem lehet null vagy egy számlaegyenleg csak pozitív szám legyen, stb. Ezek a domain modell megszorítások közvetlenül az entitás babok adattagjainak annotációjával vannak meghatározva. A validációs eljárások az alkalmazás különböző rétegeiben (megjelenítési, adathozzáférési) kerülhetnek végrehajtásra anélkül, hogy megismétlődnének. A Bean Validation és annak referencia implementációja a Hibernate Validationt ezen célok elérésére hozták létre.

A Hibernate és a Bean Validation integrációja két szinten történik. Egyrészről ellenőrizheti, hogy a memóriában lévő osztálypéldányok megsértenek-e valamilyen megszorítást. Másrészről alkalmazza a megszorításokat a Hibernate metamodellen és beilleszti azokat a létrehozott adatbázissémába.

10.2.10.2. Intercepotorok

Az interceptor interfészek lehetőséget biztosítanak az alkalmazás számára olyan callback függvények használatára, amelyek megfigyelik és/vagy módosítják a perzisztens entitások adattagjait mentés, frissítés, törlés vagy betöltés előtt. A technika egy lehetséges felhasználási területe az audit információk követése.

10.2.10.3. Szűrők

A Hibernate3-ban megadhatók osztály illetve kollekció szintű előre definiált szűrőfeltételek (filterek). Egy szűrőfeltétel segítségével a lekérdezések where részéhez hasonló megszorításokat adhatók meg az osztályokon és kollekciókon. Ezek a feltételek tetszés szerint paraméterezhetőek. Az alkalmazás futásidőben engedélyezhetik a szűrőket és meghatározhatják a paramétereiket. A szűrők egyfajta, az alkalmazásból paraméterezhető adatbázis view-ként is tekinthetők. Definiálhatók mind annotációk, mind konfigurációs fájlok segítségével.

10.2.10.4. Hibernate Search

Az olyan teljes szöveges kereső motorok, mint az Apache Lucene, hatékony szöveges lekérdezési technológiával bővítik az alkalmazásokat. A Hibernate Search olyan problémákra nyújt megoldást, mint indexek naprakészen tartása, index struktúra és domain modell közötti eltérések, lekérdezésekkel kapcsolatos ütközések. A Search néhány annotáció segítségével indexeli a domain modellt, gondoskodik az adatbázis – index szinkronizációról és szabad szöveges lekérdezések segítségével érhetők el hagyományos menedzselt objektumok. A Hibernate Search az Apache Lucene-t használja.

10.2.10.5. Gyorstárazás

A Hibernate-et használó webalkalmazások teljesítményét nagyban növelik a gyorsítótárak (cache) alkalmazása. A gyorsítótár tárolja az adatbázisból betöltött adatokat, ezzel lecsökkentve az alkalmazás és az adatbázis közötti kommunikáció mennyiségét abban az esetben, ha az alkalmazás ismételten hozzá akar férni a már betöltött adatokhoz. Ekkor az alkalmazás a gyorsítótárban lévő adatokat fogja használni. Abban az esetben, ha a gyorsítótárban nem szereplő adatokra van szükség, úgy ismét az adatbázishoz fog hozzáférni. Mivel az adatbázis felé irányuló kérések hosszabb lefutásúak, mint a gyorsítótár felé indítottak, így a gyorsítótárak használatával a hozzáférési idő és az adatforgalom csökkenni fog az

alkalmazás és az adatbázis között. Mivel a gyorsítótár mérete limitált és csak az alkalmazás aktuális állapotában lévő adatokat tárolja, így időről időre üríteni kell.

A Hibernate kétszintű gyorsítótárat alkalmaz:

- ***Elsődleges gyorsítótár (first-level cache):*** Az elsődleges gyorsítótár mindig a Session objektumhoz van rendelve. Alapértelmezetten a Hibernate ezt a tárat használja. Ezen a szinten a gyorsítótárazás tranzakciónként történik, tehát egy tranzakción belül nincs ismételt adatbázis hozzáférés, tehát csak a tranzakció végeztével hajtódnak végre az adatbázis frissítések. Alapvetően csökkenti a tranzakcióban generálandó SQL lekérdezések számát.
- ***Másodlagos gyorsítótár (second-level cache):*** Ez a tár a Session Factory objektumhoz van rendelve. Tranzakciók futtatása közben az objektumokat Session Factory szinten tölti be. Ezek az objektumok az egész alkalmazás számára elérhetőek lesznek, nem csak egy tranzakcióban. Mivel az objektumok már bekerültek a gyorsítótárba, így ha egy lekérdezés eredményében olyan objektum van, amit a gyorsítótár tartalmaz, nincs szükség adatbázis tranzakcióra. A Hibernate több különböző gyorsítótár implementáció használatát támogatja. Minden implementációnak más teljesítménye, memória felhasználása és beállítási lehetőségei vannak.

10.2.10.6. Replikáció

Replikáción entitás vagy entitások halmazának egyik adatforrásból másikra való átmásolását értjük. A replikáció célja a megbízhatóság, hibatűrés vagy az elérhetőség növelése. A Hibernate által támogatott replikáció során megadhatók bizonyos viselkedési módok annak definiálása érdekében, hogy a rendszer hogyan kezelje a cél adatbázisban már létező adatokat:

- Ignore: figyelmen kívül hagyja az azonos azonítóval rendelkező adatbázis bejegyzéseket.
- Overwrite: felülírja az azonosakat.
- Exception: kivételt dob, ha azonos azonosítóval rendelkező sort talál.
- Last version: felülírja az adatot, ha korábbi verziószámmal rendelkezik, egyébként figyelmen kívül hagyja.

10.2.10.7. Hibernate Shards

Időnként előfordul, hogy több adatbázist kell egyszerre kezelni mertpl.: elosztott architektúrát használunk (pl.: a Kalifornia és India közötti hálózati késleltetés túl nagy egy egyszerű adatbázisnak) vagy egyéb nem technikai okokból az adatokat több különböző adatbázisban kell tárolni. Ez megnehezíti az alkalmazás fejlesztését. A Hibernate Shards egy olyan keretrendszer, ami a Hibernate fölé ad horizontális partícionálási lehetőséget.

Horizontális partícionálás során különböző adatsorokat különböző táblákban helyezünk el. Például az 50000 alatti irányítószámú vásárlókat a CustomerEast táblában, míg az a felettieket a CustomerWest táblában helyezzük el. Így a két partíciós tábla a CustomerEast és a CustomerWest, de készíthetünk egy olyan nézetet melyben a két tábla uniójával az összes vásárló egy helyen látható.

10.3. Összefoglaló

Ebben a fejezetben bemutatásra kerültek azok a módszerek, amelyek lehetővé teszik azt, hogy az adatokat hatékonyan tároljuk adatbázisban, valamint jól karbantartható és hatékony technológiát mutattunk be arra, hogyan lehet relációs adatbázisba objektumokat tárolni.

10.4. Ellenőrző kérdések

1. Mi az entitás életciklusa, milyen elemei vannak? Vázolja fel az életciklus-folyamatot.
2. Mik a Hibernate fő képességei?
3. Hogyan alkalmazható a Hibernate, milyen asszociációs és öröklődési leképezéseket ismer a keretrendszer?
4. Milyen fetch-elési stratégiákat ismer a Hibernate?
5. Mik a Hibernate további, kiegészítő képességei? Milyen előnyökkel járnak ezek?

10.5. Referenciák

- Hibernate annotations: http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/
- Why Hibernate? - JBoss Community. http://www.hibernate.org/

IV. rész

Alkalmazásrendszerek fejlesztése

Az előző fejezetekben olvashattunk alkalmazásfejlesztésről, szolgáltatások fejlesztéséről. Alkalmazásrendszerek fejlesztéséhez elengedhetetlen ezek a komponensek integrálása. Jelen fejezet ezt a témát tárgyalja, hogyan lehet független szolgáltatások és alkalmazások egységesítését megvalósítani, közös kommunikációs, interakciós réteget alkalmazni azok együttműködésére, így kialakítva egy szolgáltatásorientált rendszert. Mielőtt belemennénk a részletekbe, érdemes pár körülményt és fogalmat megismerni az integráció kapcsán.

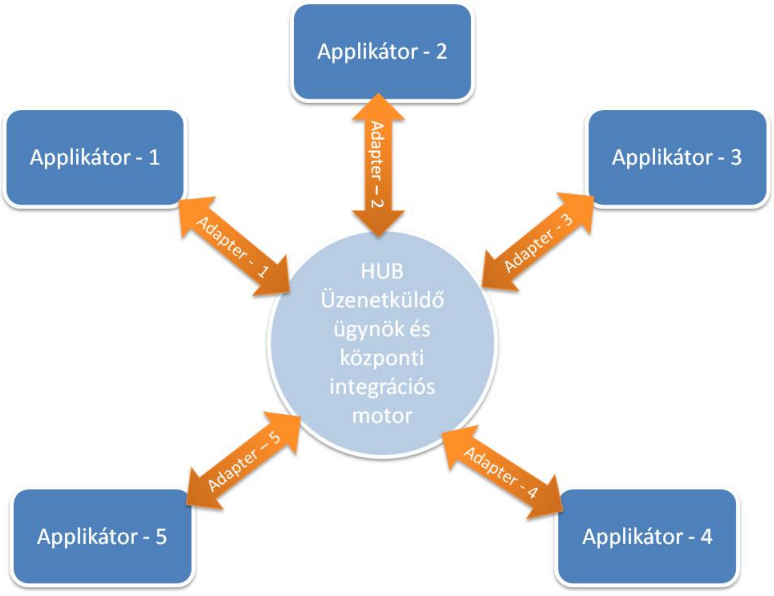
Az alkalmazásintegráció az IT világ egyre növekvő érdekeltségű területe a hatékonyabb információcsere és áramlás biztosítására egy cégeken belül a részlegek között, sőt cégek, vállalatok között. Ahhoz, hogy lássuk az integrációs technológiák fejlődésének okát, vizsgáljuk meg egy nagyvállalat infrastrukturális felépítését és annak követelményeit, következményeit.

Egy vállalat működését tipikusan több száz alkalmazás felhasználásával irányítják, amelyek egy adott részleg üzleti folyamatait fedik le, mint például számlakezelő rendszerek, számviteli rendszerek, raktárkészlet kezelő alkalmazások, kommunikációs szolgáltatások, stb. Emellett számtalan, a külvilág felé irányuló weboldalak és más információmegosztó szolgáltatások futnak. A felhasznált alkalmazásokkal szemben reális elvárás az, hogy azok, mint egy integrált egész operáljon az adott vállalat sikeréért, amelynek az első problémája, hogy az alkalmazások lehetnek saját fejlesztések, szabad felhasználású szolgáltatások, vagy pénzért megvásárolt piaci termékek, így ha az adott alkalmazás nem rendelkezik szabványos integrációs interfészekkel, akkor a forráskód hiányában nagyságrendekkel megnehezíti az ilyen irányú törekvéseket. A vállalatok esetében az alkalmazások számossága önmagában nem befolyásoló tényező, az informatika mai hardver helyzetét és annak képességeit tekintve.

Felvetődhet a kérdés, hogy miért nem használnak ezek a vállalatok egy nagy rendszert, amely minden számukra fontos képességgel és üzleti folyamat támogatásával fel van vértezve? Sokszor a fejlesztők és a fejlesztői eszközök képességeit meghaladó bonyolult rendszerek létrehozása lenne a feladat, hogy a számtalan elvárásnak, komplex üzleti folyamatoknak maradéktalanul megfeleljen egy ilyen rendszer, amely kifejlesztése rengeteg pénzt

és időt emésztene fel, és közel sem biztos, hogy sikeresen zárulna a projekt. A szoftverfejlesztő cégek számára sem megfelelő irány, így nem is céljuk ilyen „mindentudó” rendszerek fejlesztése, hiszen egy ilyen nagy alkalmazás teljesen cég-specifikus felhasználói eseteket és igényeket elégíteni ki, és nem, mint általános megoldás jelenne meg a piacon, vagyis a termékük nem lenne újrafelhasználható, eladható más cégek számára. Az általánosítás és konfigurálhatóság lehetősége egy ekkora méretű rendszerben szintén közel lehetetlen. Ez indukálja a másik fontos tényezőt, hogy mivel nem létezik olyan alkalmazás, amely a teljes cég üzleti folyamatait lekezelné pontosan olyan módon, ahogy az a cég számára a legmegfelelőbb, a vállalatok több, terület-specifikus rendszert, szolgáltatást és alkalmazást telepítenek, amelyek egyenként csak egy részhalmazát fedik le a teljes cég igényeinek, de ami követelményeket teljesítenek, azokat pontosan kielégítik. Mindezek alapján kijelenthető, hogy az integráció nem ideiglenes probléma, amit jelenleg meg kell oldani, hanem egy alapvető elvárás, amely a vállalatok életében mindig is meg fog maradni a jövőben is. Ezt a feladatot az IT világban Üzleti Alkalmazásintegrációnak nevezik, angol nevén Enterprise Application Integration (EAI).

Az EAI magába foglalja a szolgáltatások között forgalmazott üzenetek fogadásának módszereit, transzformációját, kézbesítését, irányítását, valamint az üzleti folyamatok vezérlését. Tipikusan aszinkron üzenetkezelést alkalmaznak a kommunikációra, de ha a követelmények igénylik, lehet szinkron üzenetváltást is alkalmazni. Két alapvető architektúrát különböztetünk meg, ezek a Bus, és a Hub-and-Spoke felépítés. Mindkettő felhasználható szolgáltatások fejlesztésére, és így már szolgáltatásorientált rendszerről beszélhetünk.



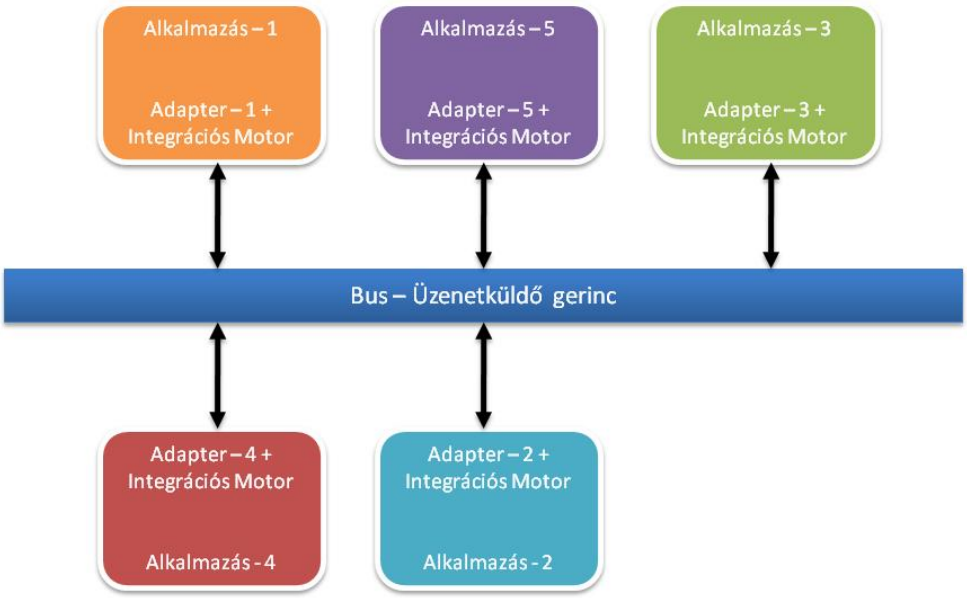
HUB/SPOKE Architektúra: Központi üzenetküldő ügynök integrációs motorral és adapterekkel

39. ábra: Hub-and-Spoke architektúra

A Hub-and-Spoke architektúra központosított üzenetbrókerből (Hub) és a szolgáltatásadapterekből (Spoke) áll, amely becsomagolja az alkalmazásokat, hogy képesek legyenek a Hub felől üzeneteket fogadni, a Hub felé üzeneteket küldeni, valamint a Spoke feladata az üzenetek transzformálása, amennyiben szükséges. Ez a központosított architektúra egyszerűen menedzselhető és karbantartható, mivel mind az üzenetek vezérlése és kézbesítése egy helyen történik, így eléggé költséghatékony megoldást jelent. Másfelől pont ez a

sajátossága teljesítménybeli problémákat okozhat egy bizonyos számú szolgáltatás becsatlakoztatása esetén, amelyet az úgynevezett Federated Bus kiterjesztéssel lehet leküzdeni. A Federated Bus architektúrában több, elosztott Hub dolgozik együtt, amelyek mindegyike tartalmaz lokális és globális információkat is a rendszerről, így ha valamely szolgáltatás az adott Hub-on nem érhető el, a rendszer a többi Hub felé képes közvetíteni a kéréseket.

A Bus architektúra, hasonlóan a Hub-and-Spoke megvalósításhoz egy központi üzenetbróker gerinchálózatot jelent, amelyen keresztül képesek a rendszerben regisztrált alkalmazások egymással kommunikálni, valamint rendelkezik adapter modulokkal, amelyek a regisztrált szolgáltatások interfészeként szolgálnak, és feladatuk az üzenetek transzformálása. A legfontosabb különbség a két felépítés között, hogy ezek az alkalmazás-adapterek a Bus-tól függetlenül futhatnak, de az adott alkalmazással mindenképpen közös platformon. Ez a megvalósítás teljesítmény szempontjából kifinomultabb megoldást jelent az integráció terén, bár az adminisztráció nagyobb kihívást jelent.



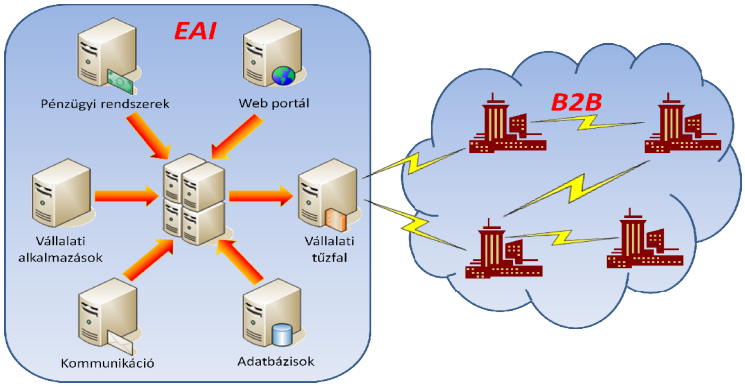
40. ábra: Bus architektúra

Az Üzleti Alkalmazásintegráció (EAI) tipikusan vállalaton belüli (Intra-Enterprise Application Integration) integrációt jelent, míg a vállalatok közötti integrációval (Inter-Enterprise Application Integration) nem foglalkozik. Az EAI integráció lokális, vállalatorientált problémákra koncentrál, így hiányt szenved pár, a vállalatok határain átívelő integrációt jellemző tulajdonságokban. Ilyen például a kifinomult biztonságkezelés, mivel vállalaton belül ez nem támaszt olyan szintű követelményeket, mint vállalatok között.

A vállalatok, cégek közötti együttműködés, üzleti határokon átívelő információcsere legalább annyira fontos kritériumokat támaszt, mint a belső üzleti folyamatok ilyen irányú támogatása. Az integráció ezen megközelítését Üzlet-2-Üzlet (Business-2-Business, B2B) integrációnak nevezik. A B2B integráció legfontosabb feladata egy adott cég kereskedelmi partnereivel történő kapcsolat megvalósítása, így értékláncokat kialakítva vállalatokon átívelő módon, illetve a közös termékfejlesztések és értéknövelt szolgáltatások létrehozásának támogatása. Bár a B2B integráció informatikai szempontból alapvetően hasonló megoldásokat ad az együttműködés, üzenetváltás lehetőségének kialakítására, mint az EAI,

mivel mindkét integrációs megoldás üzenetbróker köztesréteget alkalmaz a legtöbb esetben, fontos a két területet megkülönböztetni.

Fejlesztési szempontból a legtöbb B2B integrációs megvalósítások alkalmazása előtt az EAI megoldásokat kell érvényre juttatni, hiszen logikusan végiggondolva a belső információs rendszerek integrációja szükséges és elengedhetetlen feltétele annak, hogy létrehozhassunk, valamint hatékonyan üzemeltethessünk hosszú távon külső rendszerek, az adott vállalat kereskedelmi és üzleti partnerei felé kommunikációs interfészeket.



41. ábra: EAI és B2B

Mindkét integrációs szempont nagyon fontos, és a szolgáltatás-orientált világban is léteznek megfelelő eszközök, standardok ezek megvalósítására. Jelen fejezet ismerteti ezeket, mint például az SCA, vagy az ESB. A SOA alapelv adja az integrált rendszerben a szolgáltatások keresését, regisztrációját, és együttműködését.

Az üzenetkezelés egy integrált rendszerben a SOA alapelvet követve, elsősorban webszolgáltatásokra épült, és azt a HTTP rétegen átküldve szinkron kommunikációs módot jelentett. Az integrációnál mégis az aszinkron megoldás ad több lehetőséget, és normalizáltabb, még lazábban csatolt rendszerkomponenseket, ahogy azt a B2B és EAI is ajánlja, így egyre nagyobb szerepet kapott az Esemény-vezérelt Architektúra (Event-Driven Architecture, EDA) is egy integrációra tervezett köztesrétegben.

Az esemény-vezérelt architektúra (EDA) a SOA egy kiegészítése. Az EDA támogatást ad események létrehozásához, felderítéséhez, lekezeléséhez. Egy eseményt az állapotokban bekövetkezett fontos változásnak tekintünk. Ebben az architektúrában a tervezés és megvalósítás folyamatainak azt az elvet kell szem előtt tartani, hogy a rendszer eseményeket továbbít lazán csatolt programkomponensek, szolgáltatások között. Két fő eleme van egy ilyen rendszernek, az esemény kibocsátói (agent) és az esemény feldolgozói (sink). Az esemény feldolgozóinak a feladata, hogy egy esemény bekövetkezését követően azt minél előbb lekezelje. Ezt nem feltétlenül kizárólag ő teszi meg, tovább tudja adni a feldolgozást, illetve akár új eseményeket indíthat el. Az esemény-vezérelt architektúrára épülő rendszerek tervezésükből adódóan normalizáltabbak, mint más aszinkron környezetek. Az SOA és EDA kombinációjából egy olyan egymással kommunikáló szolgáltatáscsomagot hozhatunk létre, ahol az események is indukálhatják a szolgáltatások aktiválását triggerek segítségével.

Az alkalmazásrendszerek fejlesztéséhez a következőkben felsorolt problémákat kell megoldani, amelyek így az integrációs eszközünktől, a köztesrétegtől való elvárások. A

köztesréteg fejlettségének és alkalmazhatóságának egy jó mérőrúdja, hogy a felvázolt problémákra milyen megoldásokat nyújt.

- ***Szolgáltatás becsomagolás, üzenet transzformáció***

  Normál esetben egy integrált rendszerben valamilyen üzenetbróker köztesréteg áll az üzleti alkalmazások között, amely képes kiváltani az összes Point-to-Point kapcsolatot, ezzel elősegítve az integráció, változások, fejlesztések, bővítések elvégzésének egyszerűsödését. Ehhez a köztesrétegnek egy jól definiált módon kell becsomagolnia a kapcsolódó alkalmazásokat és azok szolgáltatásait. Ez a közös modell meghatároz egy alap üzenethalmazt, amelyeket a köztesréteg majd kapni fog, és továbbítania kell. Természetesen a csatlakoztatott IT megoldások nem feltétlenül voltak/vannak felkészítve erre a kommunikációra, tehát a köztesrétegnek biztosítania kell azt a megfelelő átalakítást egy üzenethez, amely elvégzése után az alkalmazás képes lesz fogadni és feldolgozni azt. A rendszer egyik legkritikusabb része a ***szolgáltatás becsomagolása***, illetve az ***üzenetek megfelelő transzformálása***, mivel ha a köztesréteg nem képes a teljes funkcionalitás leképezésére, egyes kapcsolatokat a köztesréteget megkerülve kell kiépíteni, ami költséges, valamint nagymértékben csökkenti a hatékonyságot és növeli a komplexitást. Egy jó integrációs köztesréteg sok, különféle kommunikációs interfésszel rendelkező szolgáltatás becsomagolását támogatja, valamint több transzport lehetőséget és üzenet transzformálási lehetőséget biztosít.

- ***Szolgáltatásvezénylés (orchestration) és koreográfia (choreography)***

  Szolgáltatásoknak – akár egy alkalmazásban szereplő modulokat, akár egy rendszerben a résztvevő alkalmazások tekintjük – az integrációja és kommunikációs lehetőségének biztosítása egy alkalmazott köztesrétegben még csak az első szintje az alkalmazásrendszerek fejlesztésének. Szükség van magasabb szintű, rendszereken átívelő üzleti folyamatok köré szervezésére ezeknek a szolgáltatásoknak. Az ilyen üzleti folyamatok, mint összetett szolgáltatások jelennek meg a rendszerben, amelyek ***létrehozását és vezénylését*** is támogatnia kell a köztesrétegnek. A rendszerben résztvevő szolgáltatások rendelkezhetnek olyan leíró meta-információkkal, amelyek definiálják, hogy az adott szolgáltatás milyen módon vehet részt egy érvényesnek tekintett üzenetváltási szekvenciában, vagyis az ***együttműködés koreográfiáját***. Az összetett szolgáltatások által magas szinten definiált üzleti folyamatokból a szolgáltatásorientált rendszerekre jellemző generikus logikával rendelkező alkalmazások építhetőek fel.

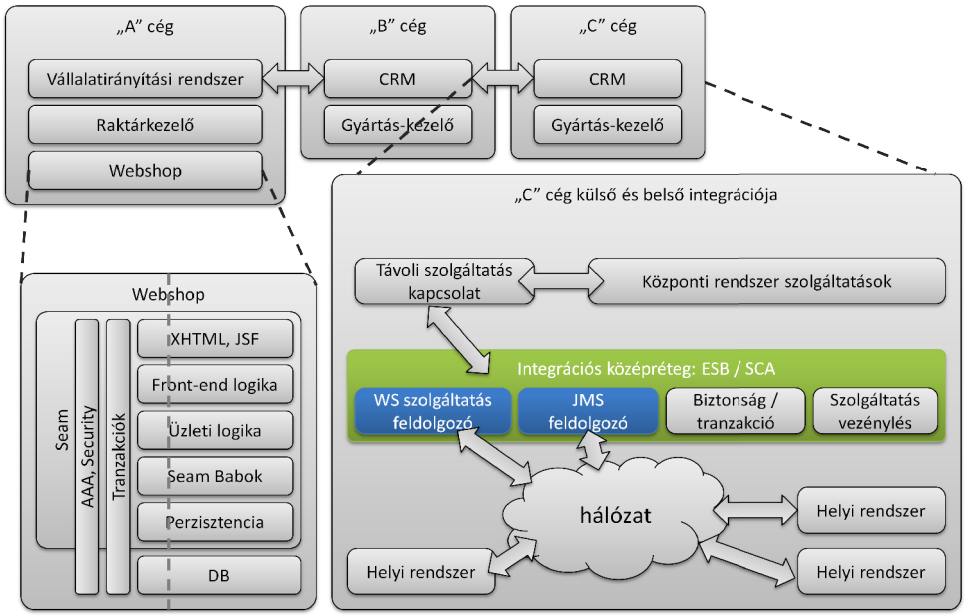
- ***Keresztülívelő problémák***

  Az üzleti alkalmazások, szolgáltatások egyik sarokpontja a megfelelő biztonsági feltételek és követelmények betartása, betartatása, valamint az adat konzisztencia megőrzése. Ez különösen fontos egy integrált szolgáltatásrendszerben, ahol a teljes rendszer működését befolyásolják ezek a tényezők. Így a megfelelő kontextusok beállításainak és a transzport réteg feletti továbbításának lehetősége az integrációs köztesréteg feladata. A legtöbb esetben a ***biztonsági és tranzakciós kontextusok továbbítása*** a szolgáltatások, alkalmazások közötti transzformált üzenetek kiterjesztését jelenti.

Jelen fejezet ez a három fő problémának és megoldási lehetőségeinek tárgyalásával foglalkozik, és felépítését tekintve az alfejezetek követik a problémák megemlítésének sorrendjét.

11. Szolgáltatás-integráció megvalósítása

Jelen fejezet a szolgáltatások rendszerbe illesztését, valamint az integrált szolgáltatások közötti kommunikációs lehetőségeket és szabványokat tárgyalja. Az így létrehozható alkalmazásrendszerek alapjául szolgáló köztesrétegek és azok legfontosabb tulajdonságait mutatjuk be, amely csak felületes áttekintést ad, de próbál átfogó képet adni a jelenleg elterjedt és alkalmazott megoldásokról. A 42. ábra kiemelt részei jelzik azt, hogy jelen fejezet az átfogó rendszer mely részeit taglalja.



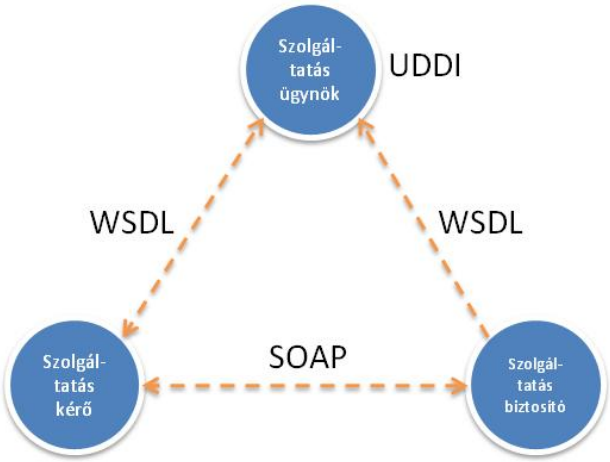
42. ábra

11.1. Webszolgáltatások és a REST specifikáció

A szolgáltatás-integráció alapköve olyan szinkron vagy aszinkron alapú kommunikációs interfészek megléte, amely lehetővé teszi, hogy a két integrálandó komponens kommunikálni tudjon. Az üzleti szoftverfejlesztés során hamar felismerték a problémát (CORBA, 1991), teljesen átfogó, generikus megoldás azonban csak az XML elterjedésével született.

A webszolgáltatás a W3C definíciója szerint „olyan szoftverrendszer, ami az interoperábilis gép-gép (M2M) integrációt támogatja hálózat felett. Az interfésze gép által feldolgozható formátumban van leírva, pontosabban a WSDL (Web Service Definition Language) által. A rendszerek SOAP (Simple Object Access Protocol) üzenetek által leírt módon lépnek kapcsolatba egymással, jellemzően HTTP protokollon keresztül, XML szerializációt és egyéb webes szabványokat felhasználva".

A webszolgáltatások két részre oszthatók: nagy vagy hagyományos webszolgáltatások, és RESTful webszolgáltatások. A következőkben ezek a szolgáltatástípusok kerülnek bemutatásra.



43. ábra: Webszolgáltatás alapmodell

A webszolgáltatások a W3C által definiált alapmodellt alkalmazva a **43**. ábra által mutatott elemekből épül fel. A webszolgáltatás definíciós nyelvet a W3C specifikálta [8][9] (a WSDL2 ajánlásként szerepel). A WSDL célja leírni azt a modellt, amellyel a webszolgáltatások kommunikálnak. A WSDL leírás definiálja a következőket: port/végpont (definiálja a kapcsolat végpontokat), kötés (binding – összerendeli a műveleteket a csatornákkal), porttípus/ /interfész (befoglalja a műveleteket), művelet (definiálja az egyes műveletek be- és kimenetét), üzenet (definiálja a művelet által fogadható és küldött üzenetformátumokat), típusok (definiálja az egyedi összetett típusokat). A webszolgáltatásokkal történő együttműködés, azok meghívása SOAP alapú üzenetek formájában lehetséges. A SOAP eredeti célja az objektum-modell szerializálásának ábrázolása XML segítségével. A protokollt szintén a W3C specifikálta.

Az UDDI egy platform-független XML-alapú regisztrációs adatbázis, mely lehetővé teszi webszolgáltatások bejegyzését, keresését egy adott rendszerben. Az UDDI szabványt az OASIS definiálta. Az OASIS definíciója szerint a „az UDDI definiál egy regisztrációs szolgáltatást webszolgáltatások és egyéb elektronikus és nem elektronikus szolgáltatások számára.” Az UDDI specifikációi meghatározzák többek között azt az API-t, amivel elérhető az UDDI regisztrációs adatbázis, azt az XML sémát, ami a regisztrációs adatbázis adatmodelljét és SOAP üzenetformátumait tartalmazza. Emellett a SOAP API WSDL definícióit és az UDDI regisztrációs adatbázis definíciókat írja le.

A webszolgáltatások köré számos szabvány készült. Ezek a specifikációk kiegészítik, átfedik egymást. Közösen WS-* jelzővel illetik az ilyen jellegű szabványokat. A WS-* szabványok a következő csoportokba sorolhatók:

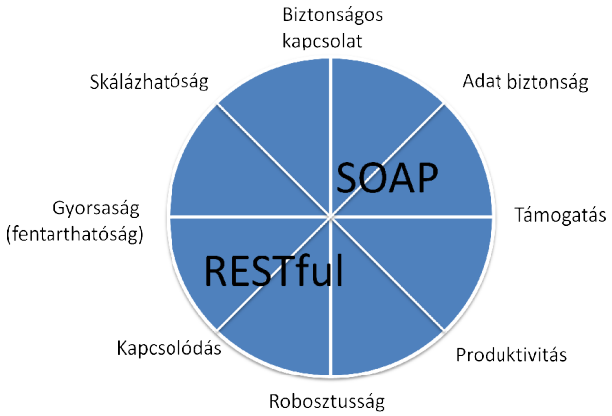
- Együttműködési profilok (Interoperability Issues)
- Üzleti folyamat specifikáció (Business Process Specifications)
- Meta-adat specifikáció (Metadata Specifications)
- Menedzsment specifikáció (Management Specifications)
- Megbízhatósági specifikáció (Reliability Specifications)
- Tranzakciós specifikáció (Transaction Specifications)
- Erőforrás specifikáció (Resource Specifications)
- Üzenetküldési specifikáció (Message Specifications)
- XML specifikáció (XML Specifications)

A fent említett profilok és WS-* specifikációk olyan fejléc és tartalom kiegészítéseket biztosítanak a standard nehéz webszolgáltatásokhoz, melyek kiegészítik a különböző szinkron hívások, az átviteli csatorna és alapprotokoll okozta hiányokat, gyengeségeket, érintve a biztonság, elosztottság, tranzakció-kezelés, kontextuskezelés, menedzsment, jogosultságkezelés, disszemináció okozta problémákat. Megvalósítását tekintve jellemzően nem minden WS-* specifikáció megvalósítása elérhető az egyes webszolgáltatás-megvalósításokon.

A REST (REpresentational State Transfer) architektúra célja a HTTP (vagy más) szabványban rejlő lehetőségeket kihasználva meghatározni egy interfészt a hagyományos HTTP GET, POST, PUT, DELETE kérések (vagy ezek megfelelőjének) használatával. A REST architektúra az üzenetek küldése vagy metódushívások helyett az állapottartó erőforrások manipulálására koncentrál. A REST architektúra épülhet a SOAP fölé, vagy közvetlenül a HTTP közegre. A REST architektúra néhány megszorítást követel meg, ami az állapottartó (stateful) tulajdonsága valamint a közvetítő közeg (HTTP) tesz szükségessé:

- kliens-szerver alapú
- állapotmentes protokoll: a kommunikáció állapotmentes kérés-válasz alapú, a szerver nem őrzi a kliens állapotát
- gyorsítótárazható (a HTTP-hez hasonlóan): a válasznak jeleznie kell, hogy az adott válasz gyorsítótárazható-e vagy nem
- rétegezett rendszerek támogatása: a kliensnek nem kell tudnia, hogyan kapcsolódott a szerverhez, köztes rétegek növelhetik a közeg biztonságát, megbízhatóságát, egyéb jellemzőit
- kód kérésre (opcionális): a szerver képes ideiglenesen kiegészíteni a választ
- egységesített interfész

A RESTful architektúra jellemző adatformátuma a JSON, XML vagy YAML vagy tetszőleges érvényes MIME típus.



44. ábra: A SOAP és RESTful kapcsolata

11.2. Üzenetorientált támogatású integrációs megvalósítások

A szolgáltatásintegráció és alkalmazásrendszerek kifinomultabb köztesrétegei több lehetőséget biztosítanak különböző megvalósítású szolgáltatások együttműködésére és hatékonyabb kommunikációs réteget is biztosítanak számukra. A független szolgáltatások laza

csatolása szükségessé teszi ezekben a rendszerekben az eseményvezérelt, üzenetorientált paradigma felhasználását. Azt kombinálva a SOA alapelvekkel, rendkívül eredményes keretrendszereket nyújtanak az alkalmazásrendszerek számára. A manapság leginkább elterjedt EAI és B2B keretrendszerek ezért rendelkeznek egy úgynevezett Üzenetorientált Köztesréteggel (Message-Oriented Middleware, MOM).

A MOM köztesréteg feladata az aszinkron üzenetkezelés megvalósításával a rendszerben résztvevő alkalmazások és szolgáltatások szétválasztása (decoupling). Tipikusan egy központi üzenetsor (message queue) képezi a MOM köztesréteg magját, amelyet szokás üzenetbrókernek is nevezni. Ehhez csatlakoznak a szolgáltatások, amelyek így nem közvetlenül kommunikálnak, az üzenetek küldőinek nem kell pontosan ismerniük a fogadókat, mert minden kommunikáció az üzenetsoron keresztül történik. Ez adja a lazán csatolt kommunikáció alapját.

Az üzenetbróker az üzeneteket képes tárolni is, így nem követelménye a rendszernek, hogy a küldő és fogadó egyszerre elérhető legyen a rendszerben. A köztesréteg feladata az üzenetek kézbesítése és a kézbesítés vezérlése, amely segítségével a többesküldés is lehetővé válik, valamint a köztesréteg alkalmazásával az üzenetek közös formátumra történő átalakítása sem jelent problémát.

A legelterjedtebb MOM megvalósítás a Java Message Service (JMS). A JMS az első üzleti integráció támogatására született üzenetorientált API, amely jelentős ipari támogatást kapott. Tervezését, kialakítását a Sun Microsystems végezte más partnerekkel együttműködve a Java Community Process által gondozott JSR-914-es (Java Specification Request) specifikációban. A JMS-t az integrációra és különböző üzleti alkalmazások laza csatolására tervezték, lehetőséget adva azoknak az információcserére, mégpedig szabványos, aszinkron módon.

A JMS specifikáció kétféle üzenetkezelési modellt fektet le, a közzététel/feliratkozás (Publish/Subscribe) és a kérés/válasz (Request/Response) módszerét. Előbbi esetén az üzenetküldést az adatszolgáltató kezdeményezi, és a köztesréteg feladata az üzenetre feliratkozott szolgáltatások felé történő továbbítása. Utóbbi módszert tekintve, az üzenetküldést az adatot, információt igénylő szolgáltatás kezdeményezi, és a köztesréteg az adatszolgáltató felé továbbítva azt az adott szolgáltatás válasz üzenetével tér vissza az eredeti kezdeményezőhöz.

A JMS interfészek és jelentések csoportját határozza meg. Egy JMS keretrendszer a következő részekből, résztvevőkből tevődik össze:

1. JMS szolgálatató (JMS Provider)
2. Üzenetorientált köztesréteg, amely megvalósítja a JMS specifikációt.
3. JMS kliensek (JMS Clients)
4. Szolgáltatások, alkalmazások, amelyek üzeneteket fogadnak és küldenek.
5. Üzenetek (Messages)
6. A JMS kliensek közötti információcserét lehetővé tevő objektumok. Az üzenetek három részből állnak, ezek a fejléc (header), a tulajdonságok (properties) és a törzs (body). A fejléc azonosítja az üzenetet és definiálja a kézbesítés vezérlését (routing). A tulajdonságok rész opcionális része az üzenetnek, amely információkt tárolhat az üzenet tartalmáról például szűrések (filters) számára. A törzs rész, amely szintén

opcionális része az üzenetnek, tartalmazza az aktuálisan megosztásra kerülő adatot, információt.

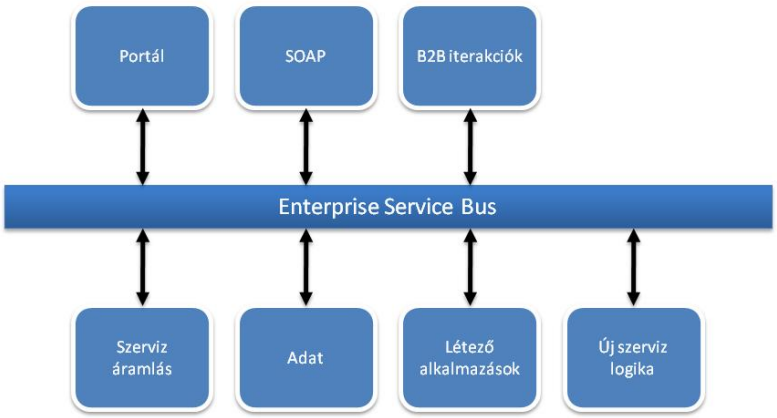
7. Adminisztratív objektumok (Administered Objects)
8. Adminisztrátorok által előre konfigurált JMS objektumok a JMS kliensek számára.

11.2.1. Az Enterprise Service Bus

Az egyik legfontosabb üzenetorientált köztesrétegre építő, szolgáltatásintegrációt megvalósító keretrendszer specifikációja a Java Business Integration (JBI). A JBI egy olyan specifikáció, amely szolgáltatás-orientált architektúra (SOA) egy megvalósítási lehetőségét adja. Alapjában véve egy konténert valósít meg, amely biztosítja szolgáltatások könnyű integrációját, hogy ezzel lazán csatolt kompozit alkalmazásokat hozhassunk létre. A szolgáltatások ehhez a konténerhez tudnak kapcsolódni a csatoló komponensek (Binding Component, BC) segítségével, vagy a konténer maga tartalmazhatja őket, mint a feldolgozó (Service Engine, SE) része. A konténerek egységes modellé alakítják a kapcsolódó szolgáltatásokat. A kommunikációt ezek a szolgáltatások között pedig a Normalizált Üzenet-irányító (Normailzed Message Router, NMR) busz bonyolítja le, amely a négy ismert, a WSDL 2.0 specifikációból vett normalizált XML alapú üzeneteket (Message Exchange Pattern, MEP) továbbítja.

Az Enterprise Service Bus (ESB) gyakorlatilag a JBI magját képező NMR-t valósítja meg, vagyis az üzenetbróker szerepét tölti be. Az ESB egy üzenetorientált, elosztott rendszerintegrációs köztesréteg, amely feladata az üzenetek kézbesítése és a kézbesítés vezérlése (routing), valamint egy köztes réteg biztosítása a szolgáltatások számára a biztonságos és megbízható interakciók támogatására. Az ESB a gyakorlatban hálózaton elosztott szolgáltatáskonténerekből áll. A konténerek olyan szolgáltatásokat nyújtanak a rendszer számára, mint például az alkalmazás adapterek, üzenetvezérlés és transzformáció, valamint a kommunikációs protokollok és transzfer rétegek széles skáláját. Az ESB üzenetorientált köztesrétege tipikusan JMS alapon van megvalósítva, ami garantálja az üzenetek kézbesítését. A szolgáltatások adaptereken keresztül kapcsolódnak a buszhoz vagy valamely támogatott üzenetküldési lehetőséget felhasználva. Az ESB beépített szolgáltatáskonténerei között megtalálhatóak a webszolgáltatások kezelését biztosító modulok is.

Az ESB köztesréteg számos erőforrás és szolgáltatások interakcióját vezérli, amit tranzakciós környezet biztosításával is támogat. Általános célja, hogy az együttműködést az eredeti szolgáltatások módosítása nélkül legyünk képesek azokat rendszerbe szervezni. Az ESB egy ideális megvalósítása a SOA rendszereknek, mivel egy általános keretrendszert biztosít minden szolgáltatás integrálásához egy összetett üzleti megvalósítás eléréséhez biztonságos, megbízható, nagy teljesítményt adva skálázható módon.



45. ábra: Enterprise Service Bus

Az ESB megvalósítások számos általános jellemzővel rendelkeznek, amelyeket röviden áttekintünk.

Híváskezelés (Invocation)

Az ESB különböző kommunikációs lehetőségeket biztosít az integrált erőforrások és szolgáltatások elérésére, amelyek közé tartoznak a webszolgáltatások, beleértve az UDDI, SOAP, WSDL specifikációk, valamint a WS-* szabványok MOM köztesrétegre való átültetését. A JMS, J2EE Connector Architecture (JCA) szintén kötelező érvényű kommunikációs lehetőségnek tekinthető az ESB szempontjából, valamint a TCP, UDP, HTTP és SSL protokollok támogatása. Számos ESB implementáció rendelkezik JBI, RMI, JDBC, POP3, FTP vagy XMPP támogatással is.

Kézbesítésvezérlés (Routing)

A kézbesítésvezérlés feladata az üzenetek címzettjeinek meghatározása és az üzenet továbbítása számára, ezzel köztes réteget képezve a küldő és fogadó között. Az ESB rendszerben az üzenetek címzettjeinek azonosítását tipikusan általános erőforrás-azonosítók (Uniform Resource Identifier, URI) valósítják meg. Az üzenetek pontos célállomásának meghatározására számos módszer áll rendelkezésre, amelyek különböző Router szolgáltatások megvalósítását eredményezték. A tartalomalapú útválasztó (Content Based Router, CBR) az üzenet tartalmát megvizsgálva hozza meg a döntést, hogy ki a címzett, így a küldőnek nem kell pontos célállomást megjelölni az üzenet busz felé történő küldésekor. A CBR szolgáltatások felhasználhatóak üzenetek szűrésére is. A többesküldést (amikor egy üzenetet több címzett számára is továbbítani kell) szintén kézbesítésvezérlő segítségével valósíthatjuk meg, pontosabban Recipient List Router használatával. A Splitter Router segítségével egy üzenetet adott szabályok mentén több üzenetre bonthatunk, és az így előállt üzenetrészeket más-más célállomás felé kézbesíthetünk. Ennek ellentéte az Aggregator Router, amely több üzenetet vár össze, és egyesítve azokat egyetlen üzenetet továbbít a címzett szolgáltatás felé. A Resequencer Router több, nem sorrendben érkezett üzenetet képes adott indexelés mentén ismét sorrendbe állítani, és így a helyes sorrendben továbbítani. A kézbesítésvezérlő módszereket egymással kombinálva még komplexebb, bonyolultabb kézbesítési eljárást képezhetünk, valamint az ESB köztesrétegben a kézbesítésvezérlők dinamikusan, megfelelő menedzsment üzenetekkel futásidőben módosíthatóak.

Közvetítés (Mediation)

A közvetítő szolgáltatások a felelősek az összes felhasználható kommunikációs protokoll és üzenetformátum transzformálásának és becsomagolásának elvégzéséért a különböző implementációt követő integrált szolgáltatások között, hogy azok képesek legyen a különbségek ellenére is együttműködni.

Csatolók (Adapters)

Az adapterek feladata a szolgáltatások, alkalmazások rendszerbe illesztése. Olyan beépített adaptereket implementáltak az ESB környezetben, amelyek képesek üzleti erőforrás-tervezés (Enterprise Resource Planning, ERP), beszállítói lánc-kezelés (Supply Chain Management, SCM) és vásárlói kapcsolatok kezelése (Customer Relationship Management, CRM) alkalmazások rendszerbe foglalására. A beépített adapterek felhasználásával az alkalmazásintegráció egyszerű és gyors módon végezhető el komplex rendszerek esetén is, ha azokat az adott alkalmazáscsaládra jellemző szabványok mentén implementálták, legalább az kommunikációs interfészek terén.

Biztonság (Security)

Az ESB keretrendszerek támogatást adnak a biztonságos üzenetküldéshez, így képesek az üzenet tartalmát titkosítani, majd a titkosítást visszafejteni. Konfigurációs beállításokkal az integrált szolgáltatások elérését és meghívását is szabályozhatjuk (szintén biztonsági megfontolásokból).

Karbantartás (Management)

A legtöbb ESB keretrendszerbe kifinomult nyomkövetési és naplózási eljárást építettek bele a rendszer működésének, hatékonyságának monitorozása érdekében. Ezek közé tartozik például az üzenettár (MessageStore), amely az ESB-n keresztülhaladó üzeneteket tárolja későbbi vizsgálatok és statisztikák számára. Monitorozni lehet a rendszer egy adott szolgáltatásának teljesítményét, a küldött, fogadott és sikeresen feldolgozott üzenetek számosságát komponensenként. Az ESB megvalósítások rendelkeznek valamilyen felhasználói felülettel a busz és szolgáltatásainak adminisztratív vezérlése céljából.

Összetett esemény-feldolgozás (Complex Event Processing, CEP)

Az aszinkron üzeneteket használva, azokat eseményekként felfogva a rendszerben lehetőség nyílik az események értelmezésére, eseménykorreláció és esemény alapú mintaillesztés alkalmazására, ezzel az eseményvezérelt architektúra képességeit is használni lehet használni.

Folyamatvezénylés (Process orchestration)

Az ESB keretrendszerek tartalmazhatnak a munkafolyamatok üzleti folyamatok köré szervezését támogató motorokat az integrált szolgáltatások számára. Az így definiált folyamatok a résztvevő szolgáltatásokat adott sorrendben meghívva egy komplex szolgáltatást hoznak létre. Ilyen eszköz a webszolgáltatásokhoz készített üzleti folyamatvégrehajtási nyelv (Web Services Business Process Execution Language, WS-BPEL), amelyet később még részletesebben ismertetünk.

Fejlesztői eszközök (Integration Tooling)

Az ESB-alapú szolgáltatásrendszerek fejlesztésének támogatására egyre több fejlesztői eszköz és környezet jelenik meg, amely ESB megvalósítás specifikus, és így annak az összes szolgáltatását és egyedi sajátosságát egyszerűen, sokszor kód írása nélkül valósíthatjuk meg és használhatjuk fel grafikus felületeken.

Az ESB, saját elosztott felépítéséből fakadóan remekül klaszterezhető, és ezért az elosztott rendszerek megvalósítására előszeretettel alkalmazták. A széles körben elterjedt ESB implementációk közé tartoznak az OpenESB, a JBossESB, valamint a FuseESB és MuleESB.

11.2.2. A Service Component Architecture

A Service Component Architecture (SCA) egy olyan programozási modell, amely lehetővé teszi az üzleti funkciók és komponensek általánosítását, valamint ezek felhasználását üzleti megoldások összeállításakor. Az SCA segítségével deklaratív módon lehet az egyes szolgáltatások közötti interakciókat és szolgáltatásminőségi jellemzőket (pl. biztonság- és tranzakció-kezelés) leírni. Mivel a szolgáltatások közötti interakció és annak minősége deklaratív módon definiálható, a fejlesztőknek lehetőségük van arra, hogy az üzleti logikára koncentráljanak, ezáltal a fejlesztési ciklus egyszerűsödik és rövidül. Ez elősegíti olyan újrafelhasználható komponensek fejlesztését, amelyeket különböző környezetekben is fel lehet használni.

A szolgáltatások szinkron és aszinkron módon is kommunikálhatnak egymással, és bármilyen technológiával készülhetnek. Az SCA rugalmasságot is biztosít a telepítések számára. Egy SCA segítségével összeállított rendszer egy egységként kerül telepítésre úgy, hogy a hálózaton keresztül akár további eszközökre is ki lehet terjeszteni. Az ilyen megoldás egyszerűen újrakonfigurálható anélkül, hogy a programkódon változtatni kellene. Az SCA programozási modellt követő alkalmazások képesek nem SCA komponensekkel együttműködni. Ez igaz mind a két irányba, vagyis a nem SCA komponensek képesek SCA alkalmazásokat hívni, és SCA alkalmazások is képesek nem SCA alkalmazásokat hívni.

Az SCA gyakorlatilag specifikációk egy csoportja, amik üzleti SOA rendszermodellt definiálnak, és a következő területeket célozzák meg:

- Programozási és implementáció-függetlenség
- A futtatókörnyezettől, köztesrétegtől való függetlenség
- Laza csatolás a komponensek között
- Biztonságkezelés, tranzakció-kezelés és megbízhatóság-kezelés szabály (Policy) alapon
- Rekurzívan előálló komponenscsomagok támogatása

A specifikáció első, kezdetleges verziója 2005-ben jelent meg az OSOA gondozásában, amely olyan informatikai vállalatok és termékfejlesztők egy konzorciuma, akik érdekeltek SOA paradigmára épülő alkalmazásrendszer fejlesztésében. 2007-ben jelent meg a specifikáció 1.0-ás verziója, és azóta az OASIS vette át a specifikáció további kidolgozását.

Az SCA specifikációcsaládot négy fő terület köré lehet csoportosítani, a komponens-szervezés modelljére (Assembly Model Specification), a komponens implementációkra (Component Implementetion Specifications), a csatolási specifikációra (Binding Specifications), valamint a szabály-keretrendszer specifikációra (Policy Framework Specification). A következőkben ezeket tekintjük át röviden.

11.2.2.1. Komponensszervezés

A modell definiálja, hogy hogyan lehet összetett szolgáltatásokat definiálni, vagyis mely szolgáltatások összetételével áll össze, és ezek a szolgáltatásokat mely komponensek nyújtják. Minden SCA komponens lehet összetett komponens, és annak szolgáltatásai más komponensek szolgáltatásaiból épülhetnek fel (recursive composition).

11.2.2.2. Komponens implementációk

Az SCA komponensek implementációját írja adott nyelvekhez, mint például a Java, Spring, BPEL, C++, COBOL. A komponens az implementáció egy futásidőben létrejött példánya. Egy komponens gyakorlatilag bármilyen nyelven implementálható, amíg azok az SCA specifikációkban tárgyalt absztrakciókat megtartják.

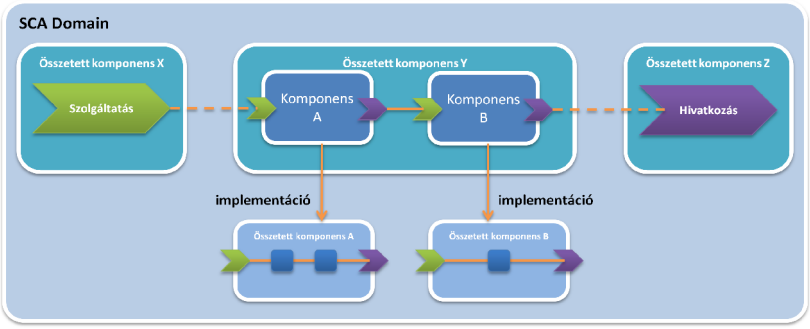
11.2.2.3. Csatolási specifikációk

A csatolási specifikációk definiálják a komponensek által biztosított szolgáltatások elérésének módját, amely a jelenlegi specifikáció alapján lehet JMS, SOAP, EJB és JCA alapon megvalósítva (binding types). A komponensek közötti kommunikációt megvalósító csatolási típusok a futtató köztesréteg által nyújtott lehetőségek mentén, természetes módon kiegészíthetők más lehetőségekkel, mint például az RMI vagy a JSON.

11.2.2.4. Szabály-keretrendszer specifikáció

A szabály keretrendszer definiálja, hogyan lehet nem-funkcionális követelményeket a szolgáltatásokhoz hozzárendelni. Két típust különböztetünk meg, az együttműködési szabályokat (interaction policies), amelyek a szolgáltatás hívója és a szolgáltató közötti együttműködést befolyásolják (titkosítás, azonosítás), valamint az implementációs szabályokat (implementation policies), amelyek a futtató környezet és a komponens közötti együttműködést befolyásolják (tranzakció-kezelés, jogosultságkezelés).

Az SCA alapelve, hogy a rendszer minden alap építőköve egy komponens, amely szolgáltatásokat biztosít a rendszer számára. A komponensek összetett komponensekké szervezhetőek (composite), amely a résztvevő komponensek szolgáltatásait is, illetve az általa definiált összetett szolgáltatást is kiajánlhatja. Az összetett komponensek felhasználásával alkalmazások, üzleti folyamatok képezhetőek, modellezhetőek. Minden komponenst a neve azonosít és különböztet meg a rendszerben. Az így létrehozott komponenseket egy úgynevezett SCA tartományon belül lehet telepíteni, amely tipikusan egy cég vagy szervezet saját üzleti folyamatait tartalmazó konténer. A tartományon belül a szolgáltatások csatolása az SCA csatolási típusokkal valósul meg, valamint a szabály keretrendszerben (Policy Framework) definiált megszorítások is csak a tartományon belül érvényesek. Az SCA tartományon kívül eső szolgáltatások elérésére a szabvány protokollok, úgymint a webszolgáltatások használatosak.



46. ábra: Service Component Architecture

Az SCA elosztott rendszerként is működhet, ahol különböző csomópontok együttesen alkotják magát a tartományt több különböző fizikai gépen futva. Az így létrehozott elosztott SCA tartományon belüli komponensek továbbra is dinamikusan kapcsolódhatnak egymáshoz, mivel a címzés ebben az esetben sem helyi címzés, hanem a komponens neve alapján érhető el a rendszerben az általa nyújtott szolgáltatás.

Számos implementáció készült már az SCA specifikációhoz, bár még egyik sem mondható kiforrottnak, tekintettel arra, hogy a specifikáció nem túl régi. Ezek egy összefoglalója megtalálható az OSOA oldalán.

Az OSOA a különböző szolgáltatások különböző típusú adatforrásokból származó adatainak egy egyszerűsített specifikáció mentén való kezelését is definiálja, amelyet gyakran használnak együtt az SCA alapon szervezett rendszerekkel. Az Service Data Object (SDO) egy adat programozási modell specifikációja. Lehetővé teszi az alkalmazások, és keretrendszerek számára az adatok egyszerű lekérdezését, megtekintését, kötését, frissítését és elemzését. A programozónak nem kell a mögöttes technológiával foglalkoznia, csak az üzleti logikára kell koncentrálnia.

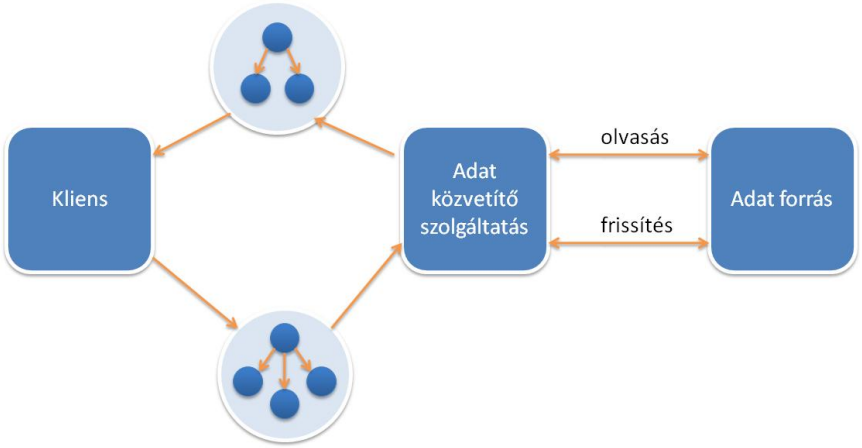
Az SDO célja az egységes adatkezelés különböző adatforrások mentén, segítséget nyújt mind a statikus, mind a dinamikus adatokkal dolgozó API-k számára, támogatja a keretrendszereket, az adatelérési tervezési mintákat, valamint lehetővé teszi a programkód és az adateléréssel kapcsolatos kód különválasztását és kezelését. A legfontosabb előnyei:

- Egyesített, központosított adatábrázolási megoldás a különböző adatforrásokból érkező adatok számára, amelyek lehetnek XML, RDB, POJO, SOAP alapúak
- Támogatja a statikus (erősen típusos) és dinamikus (gyengén típusos) programozási modellt
- Meta-adatokat biztosít az általa tárolt és továbbított adatstruktúra leírásához
- Támogatja a leválasztott (detached) adatok kezelését, valamint az adaton végzett módosítások nyomon követhetőségét
- Nyelvfüggetlenség

Az SDO egy jól definiált, egyesített API-t ad a különböző típusú adatok kezeléséhez. Ezt kiegészítve a adatközvetítő szolgáltatás (Data Mediator Service, DMS) leegyszerűsíti az adatkezelést, amikor a különböző adatforrásokhoz akarunk hozzáférni, elvégzi az adatok transzformációját a szolgáltatások által kezelhető és értelmezhető SDO formátumra.

Az SDO architektúra a nem összefüggő adatgráf (Disconnected Data Graph) koncepciójára épül. A kliens egy adatgráf formájában kapja meg az adatokat, majd a változásokat is ezen a gráfon keresztül közli az adatforrás felé. Az adatforráshoz való kapcsolódásért a

DMS komponens a felelős. A DMS komponens kérdezi le az adatokat, gyártja le a gráfot (ami az adat objektumokat tartalmazza), majd közvetíti visszafelé is a változásokat. A kliens nincs közvetlen kapcsolatban vele, csak a gráffal (olvassa azt és frissíti).



47. ábra: Az SDO működése

Az SDO két legfontosabb eleme az adatobjektum (Data Object) és az adatgráf (Data Graph). Az Data Object az adatokat tulajdonságok (properties) egy halmazaként reprezentálja. A primitív típusokon kívül referenciákat is tárol más objektumokra. A meta-adat API segítségével tud információkat szerezni a program a típusokról, a kapcsolatokról és a megszorításokról. A Data Graph tárolja az adatok összességét, az adat objektumok egy halmazát. A rendszerben a Data Graph számít egy egységnek, amivel műveleteket végzünk, így a változások követéséért is felelős.

Az SDO specifikációnak napjainkban már több implementációja is létezik.

11.3. Az OSGi és kapcsolata a SOA világával

Az alkalmazásfejlesztés és alkalmazásrendszer fejlesztés legelterjedtebb implementációs nyelve a Java. Ennek ellenére több hiányosságot is fel lehet fedezni a nyelvben, amelyek a nyelv egyedi sajátosságai, és kihatással vannak a köztesrétegek, alkalmazásszerverek működésére is.

Egy nagyobb alkalmazás, így az alkalmazásrendszerek is több modulból, szolgáltatásokból épülnek fel. Az egyes modulok különböző részfeladatokat végeznek el, lehetnek saját illetve mások által megírtak. Java nyelv esetén ilyen modulok például a JAR fájlok. A Java nyelv nem az extrém-moduláris programfejlesztés támogatása céljából jött létre, ezáltal a modulok kezelése terén vannak hiányosságai.

Egy program soha nincs készen, így az egyes modulok, szolgáltatások sem, azok fejlesztése folyamatos, ezáltal adott modulból több verzió is lehet. Amennyiben a verziók funkcionalitásban is különböznek, futási időben ez problémákat fog okozni. A Java nem tudja kezelni az egyes modulok különböző verzióit önmagában.

A classpath egyszintű, minden osztály elérhető mindenki számára, ami egyrészt jó, nem kell semmi plusz beállítás, másrészt egy hatalmas úgynevezett „classpath hell” keletkezik, ahol egy helyen van az összes elérhető osztály. Amennyiben a JAR fájlok közül többen is előfordul egy adott osztály (pl.: javax.servlet.http.HttpServletRequest), a közös classpathba

ezek közül csak egy kerül be. Lehetséges, hogy a különböző JAR fájlokban az adott osztálynak különböző verziói érhetőek el, különböző metódusokkal, akkor ez könnyedén NoSuchMethodException kivételhez vezethet. Az egyes modulok a classpathban elérhető osztályoknak csak kis részét használják, így teljesen feleslegeges mindenkinek minden szolgáltatáshoz hozzáférnie. Ez a probléma megjelenik az alkalmazásrendszerek esetén is, amikor egy köztesrétegen futtatunk több szolgáltatást.

Ezekre a problémákra próbál megoldást adni az Open Services Gateway initiative (OSGi) specifikáció. Az OSGi modulokból épül fel, a modulokat batyuknak, angol nevén bundle-öknek nevezik. A bundle egy olyan JAR fájl, ami annyival több a Java környezetben használt JAR fájloktól, hogy tartalmaz egy kiegészített MANIFEST.MF fájlt is, ami egy leírást ad az adott modulról. Ez a fájl tartalmazza a modul nevét, a modul verziószámát, a modul által nyújtott publikus csomagokat, illetve hogy milyen külső, más modulok által hirdetett csomagokat használ fel, vagyis az adott modul függőségeit.

```
Bundle-ManifestVersion: 2                                   (1)
Bundle-Name: Hello World API                                (2)
Bundle-SymbolicName: hu.uszeged.hello                       (3)
Bundle-Version: 1.0                                         (4)
Export-Package: hu.uszeged.hello;version="1.0"              (5)
Import-Package: org.apache.hello;version="2.0"              (6)
```

(1) A meta adat formátum verziója

(2) A Bundle felhasználó számára olvasható neve

(3) A bundle neve, ezt használja az OSGi keretrendszer

(4) A bundle verziója

(5) A csomag, amit megoszt más bundle-ökkel

(6) Az általa használt csomag(ok).

48. ábra: Az OSGi metaadatok

Az Import-Package és Export-Package kezdetű sorok a MANIFEST.MF fájlban megoldják a classpath, és a láthatósági problémákat. A Export-Package megadja, hogy az adott modulból milyen csomagok érhetőek el más modulok által, ezzel a többit elrejti. Az Import-Package megadja, hogy milyen külső erőforrásokat használunk fel, illetve ezeknek milyen verziójú változatát.

Az OSGi szolgáltatás platform specifikáció két részből áll: az OSGi keretrendszerből és a szabványos szolgáltatásokból. A keretrendszer a futtató környezet, ami az OSGi funkcionalitását biztosítja. Az OSGi szövetség sok szabványos szolgáltatást specifikált. Ezek a szolgáltatások olyan Java interfészek, amiket a batyuk implementálni tudnak. Ilyen szolgáltatások például a naplózás, I/O konnektorok, különböző adminisztrációs eszközök, stb.

Az OSGi szövetség csak egy specifikációt ad, hogy a keretrendszernek hogyan kell működnie. E specifikáció alapján létezik több konkrét megvalósítás, mint például Apache Felix vagy Eclipse Equinox, bármelyiket használhatjuk, mivel a működésük azonos. Az OSGi

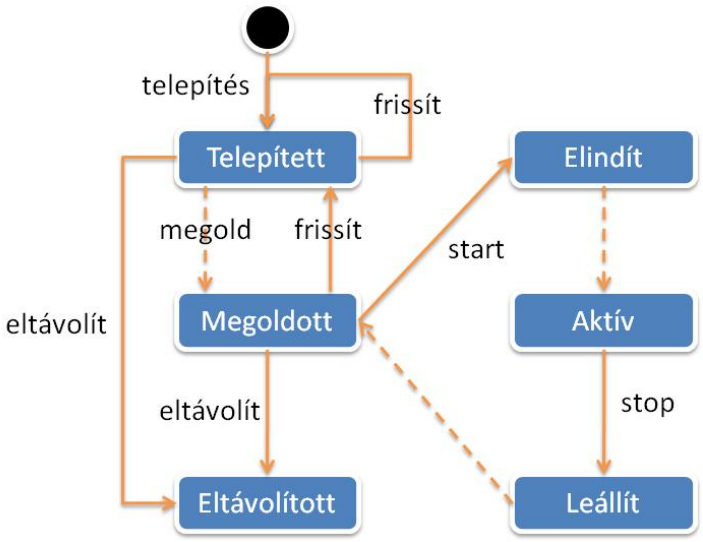
specifikáció a keretrendszert három különböző rétegre osztja. Ezek a rétegek a szolgáltatás réteg, az életciklus réteg és a modul réteg. Hasonlóan a többi rétegezett architektúrához, minden réteg az alatta levő rétegtől függ. Ezáltal lehetőség van az alsó rétegek használatára úgy, hogy a felső rétegeket nem érintjük, fordítva viszont nem lehetséges.

A modul réteg definiálja a bundle-t. A bundle általában nem egy teljes alkalmazás, hanem az alkalmazás egy logikai modulja. A bundle egy JAR fájl, ami tartalmaz még meta-adatokat is magáról a modulról, illetve arról hogy milyen igényei és szolgáltatásai vannak a többi modul, a keretrendszer illetve a rendszer felé. Minden egyes bundle rendelkezhet saját classpath-tal, amit a MANIFEST.MF fájlban lehet beállítani. Ebben a rétegben történik a verziókezelés is, valamint a publikus szolgáltatásai és függőségei is itt vannak meghatározva. A verziót több módon lehet megadni, ami nem egy konkrét érték, hanem egy halmaz.

Az osztálybetöltés sorrendje a következő:

1. Amennyiben a java.* csomagból kell egy osztály, abban az esetben a szülő Classloader-től kéri be a bundle a megfelelő osztályt. Amennyiben nem találja, akkor egy kivétel keletkezik.
2. Amennyiben egy a bundle által importált csomag egyik osztálya kell, akkor a keret-rendszer a megfelelő exportáló modultól tölti be a kért osztályt. Amennyiben nem találja, akkor egy kivétel keletkezik.
3. A bundle a saját Classpath-jában keresi a megfelelő osztályt, ha megtalálja, használja, ellenkező esetben pedig egy kivétel keletkezik.

Szabvány Java nyelvben úgy használunk egy JAR-t, hogy regisztráljuk a classpath-ba, OSGi-ban pedig installálni kell a JAR-t a keretrendszerbe. Az életciklus réteg feladata ennek a kezelése.



49. ábra: Az OSGi életciklus

A teltvonalas nyíl azt jelenti, hogy azt manuálisan kell hívni kódból, valamilyen konfigurációs fájlból vagy parancssorból. A szaggatott vonalas nyíl pedig azt jelzi, hogy az automatikusan hajtódik végre. Az install művelettel lehet egy új bundle-t behozni a keret-rendszerbe, ezzel még nem történt semmi, a bundle nem használ semmilyen szolgáltatást, és nem is nyújt szolgáltatást a többi batyu felé. A resolve parancs hatására a kertrendszer ellenőrzi, hogy milyen szolgáltatásokat hirdet kifelé a batyu és milyen külső csomagokra

van szükség az adott modul futtatásához. Ha ezt végrehajtotta, akkor kerül a bundle a resolved állapotba. A start művelet hatására a bundle először a starting, majd active állapotba kerül. Az active állapotban az általa nyújtott szolgáltatások is elérhetőek, illetve az általa futtatott program végrehajtódik. Ha a bundle nem fut, lehetőség van az adott JAR újra beolvasására, illetve egy frissebb verzióra való frissítésre is. Amennyiben már nincs szükség adott modulra, az uninstall paranccsal eltávolítható a keretrendszerből.

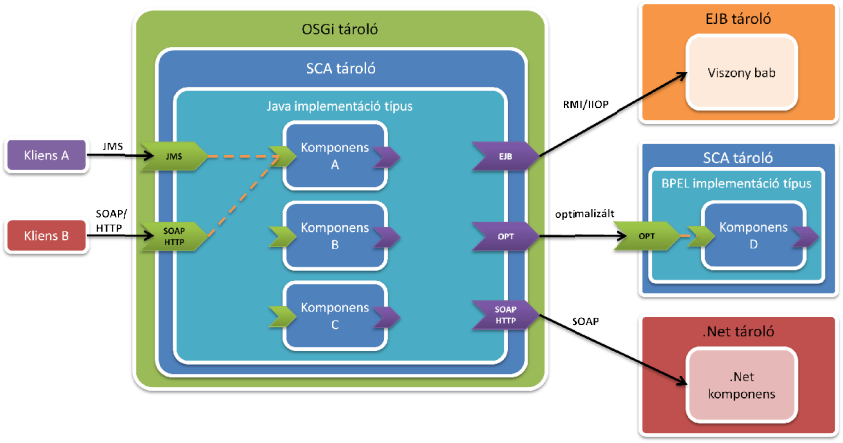
A szolgáltatás réteg a felelős a szolgáltatásorientált modell néven ismert rugalmas alkalmazásfejlesztési modellért. Ez a modul még a SOA fogalom elterjedése előtt az OSGi keretrendszer része volt. A szolgáltatást nyújtó modulok egy szolgáltatástárba felviszik az általuk biztosított szolgáltatásokat, a kliensek pedig fel tudnak iratkozni a tárban levő szolgáltatásokra. A SOA legtöbbször webszolgáltatásokat használ, míg az OSGi esetében egy Java virtuális gépen belül történnek ezek a műveletek. A szolgáltató bundle-ök képesek egyszerű java objektumokat szolgáltatásként regisztrálni a keretrendszer szolgáltatástárjába. A felhasználó bundle-ök képesek a szolgáltatás tárban keresni, és az ott található szolgáltatásokat igénybe venni.

Az OSGi-t a szolgáltatás rétegbeli funkcionalitása miatt Java virtuális gépen belüli SOA-nak is nevezik. Van lehetőség az OSGi kiterjesztésére, hogy több virtuális gépen futó OSGi keretrendszerek között is elérhetőek legyenek ezek a műveletek. Így az OSGi a modularitásának köszönhetően nagyon jó alapja lehet egy SOA keretrendszernek is. Röviden áttekintünk a következőkben pár olyan köztesréteget, amelyek önmagukban, vagy az ESB, SCA köztesrétegeket kiegészítve hatékonyabb SOA alapot nyújtanak alkalmazás-rendszerek számára.

Az R-OSGi (Remoting-OSGi) egy elosztott köztesréteg platform, amely funkcióját tekintve több OSGi környezetet képes összekapcsolni, és a különböző helyeken telepített szolgáltatások az adott OSGI telepítés határain túl is kiajánlani. A R-OSGi egy bundle-ként fut az OSGi keretrendszerben. A szolgáltatást nyújtó keretrendszernek a szolgáltatásait úgy kell regisztrálnia, hogy távolról is elérhetőek legyenek, majd a kliensek kapcsolódnak a megfelelő keretrendszerhez, és hozzáférnek a szolgáltatáshoz. A távoli szolgáltatások teljesen átlátszó módon érhetőek el. Minden távoli szolgáltatáshoz generálódik egy proxy bundle, ami a megfelelő szolgáltatást regisztrálja. A helyi kliensek a helyi szolgáltatásokat azonosan érik el, mint a távolikat. További, hasonló keretrendszer a WSO2 Carbon rendszere, amely egy olyan SOA megoldás, ami OSGi-ra épül, ezáltal építőkocka jelleggel lehet létrehozni a platformot, és működés közben lehetőség van egyes komponensek cseréjére. A Swordfish az Eclipse Equinox OSGi alapú SOA keretrendszer. A Swordfish magasabb szinten JBI-t (Java Business Integration) használ az üzenetkezelésre. Eredetileg a JBI 1.0 saját komponens modellt használt, de az OSGi-nek köszönhetően erre nincs szükség a Swordfish keretrendszerben.

Az OSGi leghatékonyabb felhasználása a SOA világban az SCA keretrendszerekkel történő kombinálása. A SCA elosztott heterogén rendszerekhez lett kialakítva, míg az OSGi arra lett tervezve, hogy mobil vagy beágyazott rendszerek VM-jén is fusson, és az adott virtuális gépen belül lehetővé teszi a moduloknak szolgáltatások meghirdetését, szolgáltatások keresését és azok futtatását. Az OSGi több módon is kommunikálhat az SCA-val. Az SCA-OSGi csatolás (binding) segítségével lehetséges az együttműködés SCA komponensek és OSGi szolgáltatások között. Lehetőség van OSGi alkalmazások (bundle-ök) SCA környezetbe való telepítésére, és ez után SCA komponensként lehet azokat felhasználni. Az OSGi-t lehet az SCA tároló alapjául szolgáló technológiaként is használni,

aminek köszönhetően elérhetőek lesznek az OSGi hasznos tulajdonságai, úgymint a kiterjesztés mechanizmus, a függőségek kezelése és a szolgáltatás jegyzékkezelés.



50. ábra: SCA és OSGi kapcsolata

11.4. Kérdések

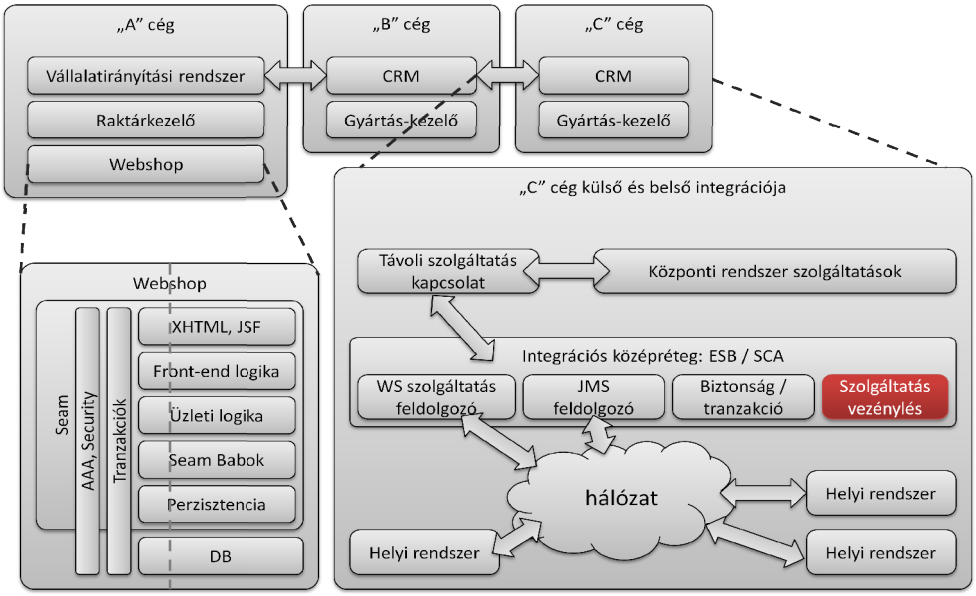
1. Milyen különbségek fedezhetőek fel a hagyományos és RESTful webszolgáltatások között?
2. Mik az üzenetorientált köztesréteg legfontosabb tulajdonságai?
3. Mik az ESB általános jellemzői?
4. Mi az SCA specifikáció négy fő tárgyköre?
5. Milyen problémákat és hogyan próbál megoldani az OSGi?

11.5. Ajánlott irodalom

- Richard Monson-Haefel, “J2EE Web Services: XML • SOAP • WSDL • UDDI • WS-I • JAX-RPC • JAXR • SAAJ • JAXP”, Addison-Wesley Professional, 2003.
- Leonard Richardson, Sam Ruby, “Restful Web Services”, O'Reilly Media; 1st edition, 2007.
- Richard Monson-Haefel, David Chappell, “Java Message Service (O'Reilly Java Series)”, O'Reilly Media; 1st edition, 2000.
- David Chappell, “Enterprise Service Bus: Theory in Practice”, O'Reilly Media, 2004.
- Tijs Rademakers, Jos Dirksen, “Open-Source ESBs in Action: Example Implementations in Mule and ServiceMix”, Manning Publications; 1st edition, 2008.
- Jim Marino, Michael Rowley, “Understanding SCA (Service Component Architecture)”, Addison-Wesley Professional; 1st edition, 2009.
- Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, Simon Nash, “Tuscany SCA in Action”, Manning; Early Access Edition, 2009.
- Richard S. Hall, Karl Pauls, Stuart McCulloch, David Savage, “OSGi in Action: Creating Modular Applications in Java”, Manning; Early Access Edition, 2008.

12. Szolgáltatásvezénylés és koreográfia

Manapság egyre szélesebb körben használnak különböző eszközöket egy alkalmazás üzleti logikájának leírására, ahogy azt már korábbi fejezetekben is tárgyaltuk. Mivel azok a megfontolások, amelyek alapján a rendszernek működnie kell, sokszor a hatályban lévő törvények, vagy éppen a vállalat aktuális üzleti politikáját követik, nem tekinthetők véglegesnek, és folyamatosan tovább kell fejleszteni ezeket, hogy kielégítsék a mindenkori követelményeket. Ezen elvek figyelembe vétele szükségessé tette, hogy az üzleti logikát magasabb szinten írják le a fejlesztők, ami független magától a programtól. Ilyen módon könnyen módosítható a logika, és átalakítható anélkül, hogy az alkalmazáshoz hozzá kéne nyúlni. Az üzleti logika ilyen szintű kiszervezése nem csak egy alkalmazás logikáját érintő elvárás, hasonlóan megjelenik ez a követelmény egy alkalmazásrendszerben található komponensek összehangolt működésének vezénylésénél is. A 51. ábra kiemelt része jelzi azt, hogy jelen fejezet az átfogó rendszer mely részét taglalja.



51. ábra

A SOA paradigmára épülő, szolgáltatások egy halmazából álló rendszerben a szolgáltatásvezénylés által definiált magas szintű logika összetett szolgáltatások, alkalmazások létrehozását segíti elő. Az alkalmazásrendszerek köztesrétegei specializált formában támogathatják az ilyen irányú törekvéseket, mint például az előző fejezetben tárgyalt SCA oldaláról a összetett komponens (Composite Component) összeállítás, továbbá a JBossESB esetén a JBoss saját munkafolyamat motorjára és nyelvére, a jBPM-re építő megvalósítás, amely nyelvét kiterjesztették ESB szolgáltatás-interfészek szinkron és aszinkron hívási lehetőségét biztosító tevékenységekkel. A számos lehetőség ellenére jelen fejezet egy szabványos, a legelterjedtebb megvalósítást tárgyalja, amely a webszolgáltatás interfésszel rendelkező szolgáltatások folyamatba szervezését biztosítja. Ez nem más, mint az üzleti folyamatvégrehajtási nyelv (Business Process Execution Language, BPEL).

Két alapvető fogalmat kell tisztázni a szolgáltatások munkafolyamatba történő szervezése kapcsán. A szolgáltatásvezénylés (orchestration) szolgáltatások egy csoportjára, azok

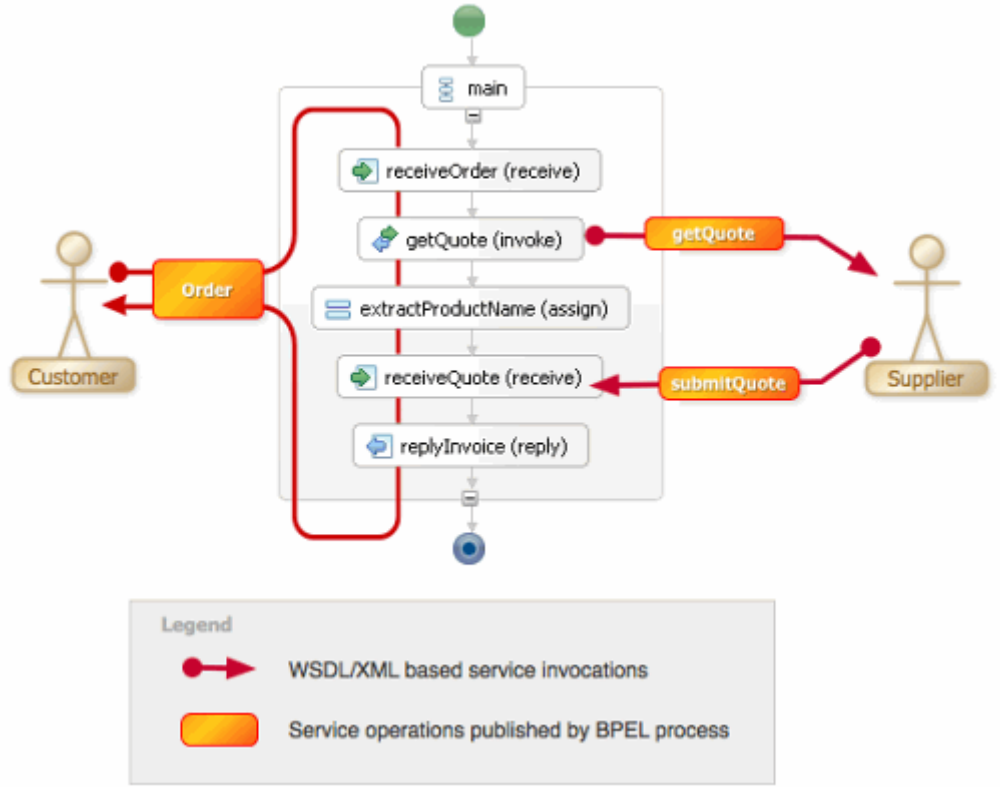
együttműködésének és kommunikációjának definiálására vonatkozik egy közös üzleti vagy rendszerbeli cél elérése érdekében (vagyis az összetett szolgáltatások belső megvalósításával foglalkozik). A koreográfia (choreography) a kívülről látható a szolgáltatások viselkedését definiálja (egy munkafolyamatban) az üzenetváltások leírásával. A szolgáltatásvezénylés a központi vezérlés viselkedését formalizálja egy elosztott szolgáltatásrendszerben, míg a koreográfia olyan szabálykészletnek tekinthető, amely az elosztott rendszer működését határozza meg központi vezérlés nélkül.

12.1. A Business Process Execution Language

A BPEL egy szolgáltatásvezénylő nyelv, nem az együttműködés koreográfiáját írja le. A legfontosabb különbség a kettő között a futtathatóságban és a vezérlésben rejlik. A szolgáltatásvezénylés egy futtatható folyamatot definiál, amely különböző rendszerek vagy szolgáltatások közötti üzenetváltásokat és azok algoritmusba foglalását definiálja. A koreográfia az együttműködésben résztvevő rendszerek, szolgáltatások közötti üzenetváltást, annak a helyes és érvényes sorrendiségét definiálja, így ez nem közvetlenül futtatható. A koreográfiát leíró legfontosabb specifikáció a W3C által gondozott WS-Choreography és az első komolyabb implementáció a webszolgáltatás koreográfia-leíró nyelv (Web Service Choreography Description Language, WS-CDL). A koreográfia fontosságát és létjogosultságát bizonyítja, hogy történtek már kutatások a BPEL nyelv kiegészítését illetően a koreográfiák leírásának támogatására.

A webszolgáltatások együttműködése kétféleképpen definiálható: futtatható és absztrakt üzleti folyamatokként. A futtatható munkafolyamatok a szolgáltatások aktuális viselkedéséhez és interfészeihez igazodva teljes körűen definiálják az együttműködést a webszolgáltatások között. Az absztrakt üzleti folyamatok csak részlegesen definiáltak, így nem is futtathatóak, mivel hiányozhatnak belőlük kötelezően meghatározandó működési követelmények. Az absztrakt munkafolyamatok jellegükből fakadóan több különböző használati esetet, üzleti folyamatot modellezhetnek egyszerre általánosan, ezzel egy munkafolyamat sablont képezve azok számára. A BPEL nyelv mind futtatható, mind absztrakt munkafolyamatok definiálását lehetővé teszi, így kiegészítve a webszolgáltatások együttműködését. Algoritmusok és folyamatok köré szervezi azokat, valamint tranzakciós keretek meghatározását segíti elő.

Maga a BPEL nyelv egyszerű utasításkészletből áll: változókezelés, adatműveletek, külső folyamatok és szolgáltatások hívása. A nyelv hiányosságai is alátámasztják, hogy a BPEL önmagában nem alkalmas semmilyen konkrét feladat elvégzésére, minden egyes állapotában egy külső utasítás, hivatkozás végzi el a feladatot, a BPEL csak a központi, irányító szerepet tölti be. A definícióját XML formátumban lehet megadni, és az XML adja az adatkezelés, valamint a kommunikáció alapját is a külső szolgáltatások felé. A BPEL magas szinten modellezi a folyamatok állapot-átmeneteit.



52. ábra: Egy BPEL folyamat

Egy BPEL folyamat XML struktúrájában a legfelső szinten a process (folyamat) elem áll, amely alapvetően két fontosabb részre bontható. Egyrészt a folyamatban résztvevő szolgáltatások, valamint a folyamat által kiajánlott szolgáltatások azonosítására (parnterlinks), másrészt magára a folyamat logikájának leírására, műveleteire (activities). Ezen kívül lehetőség van változók (variables) és a hibakezelők (faulthandlers) definiálására is.

Partnerlink

Minden webszolgáltatás-hívás (egy BPEL folyamat és egy külső szolgáltatás között) mindkét irányát tekintve, egy-egy partnerlinkhez rendlethető. A két irányt szerepeknek hívják (roles). Mindegyik szerep azonosítja az interfészt (porttype), amin kommunikál az adott irányban. Következésképpen egy partnerlink két szerepet tartalmazhat, ahol definiálni kell, hogy melyik szerep azonosítja a BPEL folyamat oldalát.

Műveletek (activities)

A BPEL folyamat logikáját leíró aktivitásokat két csoportba sorolhatjuk. Az alap aktivitások (base activities) a folyamatban felhasználható elemi építőköveket, műveleteket definiálnak, a strukturált aktivitások (structured activities) a folyamat vezérlését teszik lehetővé, amik rekurzívan tartalmazhatnak alap és strukturált aktivitásokat.

Az alap aktivitások közé tartozó legfontosabb elemek a következőek:

- invoke: egy külső webszolgáltatás meghívása
- receive: a folyamat által kiajánlott webszolgáltatás hívásának fogadása
- reply: a folyamat által kiajánlott webszolgáltatás meghívására válasz küldése

- assign: XML alapú adatmanipuláció és változók értékkezelése
- wait: adott időpontig vagy időintervallumig történő felfüggesztése a folyamatnak
- throw: hiba keletkezésének jelzése hiba esemény létrehozásával a hibakezelők számára
- empty: üres aktivitás, amit legtöbbször például szinkronizációs pontnak alkalmaznak
- exit: a folyamat azonnali félbeszakítása és befejezése

A főbb strukturált aktivitások közé továbbá a következő elemek tartoznak:

- sequence: aktivitás(ok) szekvenciális végrehajtása egymás után
- while: kezdőfeltételes ismétléses vezérlés
- repeat: végfeltételes ismétléses vezérlés
- pick: szelektív esemény bekövetkezésekkor végrehajtandó struktúra
- flow: egy vagy több vezérlési ág végrehajtása párhuzamosan
- if: egy feltétel kiértékelésének megfelelően egyszerű elágazásos vezérlés megvalósítása
- switch: egy feltételnek megfelelően több vezérlési ág közül az egyik végrehajtása

A nyelv további előnyei és hasznos tulajdonságai közé tartozik a korrelációs halmazok (correlation sets) definiálása, amely segítséget nyújt válaszüzenetek utólagos megfeleltetéséhez a korábban kiküldött kérésekhez.

Minden munkafolyamat definíciós nyelv többé-kevésbé támogatja a hatókörök alkalmazását, amellyel a folyamatváltozók, adatok láthatósága is szabályozható egy folyamaton belül. Általánosan ezt üzleti kontextusnak (Business Context) nevezik. Bár a BPEL nem rendelkezik olyan kifinomult üzleti kontextus-kezeléssel, mint a jBPM és a SEAM keretrendszerek párosításából keletkező, perzisztálható kontextus, de lehetőséget nyújt hatókörök definiálására, amelyek a scope elem segítségével adhatóak meg az XML struktúrájában, és pár különbséget leszámítva hasonlóan felépíthető, mint a process elem, vagyis tartalmazhat változókat, aktivitásokat, partnerlinkeket, amik csak az adott hatókörben érvényesek és láthatóak.

12.2. A BPEL hátrányai és korlátai

A BPEL nyelv, bár a legelterjedtebben használt eszköz a szolgáltatás-vezénylés megvalósítására, rendelkezik több hátránnyal is. A teljes nyelv XML alapú, amely az előnye, mivel nem kell transzformálni sem az adatokat, sem az üzeneteket a szolgáltatások között, az mindig mindenhol egyformán XML. Az XML azonban korlátot is jelent, hiszen így csak és kizárólag XML struktúrán értelmezhető műveleteket végezhetünk az adaton, és aki nem rendelkezik mély ismeretekkel az XPath és az XSD világáról, annak meggyűlhet a baja a BPEL használatával. A kizárólag XML-alapú adatmanipulációs támogatása az egyik legnagyobb korlátja a nyelvnek.

A BPEL egyedül SOAP protokollt alkalmaz a szolgáltatásokkal való kommunikációhoz, vagyis minden komponensnek, amelyet BPEL környezetben történő együttműködésre

szánnak, webszolgáltatás interfésszel kell rendelkeznie, még ha alapvetően nem is ehhez köthető szolgáltatásokat ajánl ki. Ez nehézkes lehet, amikor például egy FTP-t, vagy elektronikus levelezést támogató műveletekhez webszolgáltatás adaptert kell készíteni ahhoz, hogy a BPEL folyamatban felhasználhatóak legyenek.

A BPEL, mivel teljes mértékben blokk struktúrával rendelkezik, nem alkalmas finomhangolt algoritmusok definiálására. A munkafolyamatok segítségével üzleti folyamatok modellezését kívánjuk megvalósítani, ám egy vállalat vagy cég üzleti folyamata alapvetően gráf-orientált módszert alkalmazva modellezhető. A gráf pontjait és az azokat összekötő éleket, amik lehetnek többszörösek, akár köröket tartalmazóak is, szinte lehetetlen átültetni a BPEL nyelvre pár kényszeres megszorítás elfogadása nélkül.

A BPEL folyamatot kizárólag gép-gép közötti automatizált szolgáltatásvezénylésre alkották meg. Ezt az elvárást bár teljesíti, célszerűbb lett volna úgy felépíteni a nyelvet, hogy emberi beavatkozást, humán tevékenységeket is definiálni lehessen, így támogatást adva a konkrétabb, üzleti logikát leíró munkafolyamatok modellezésére és futtatására.

A felvázolt problémák a BPEL nyelvet illetően a folyamatok tervezését, létrehozását nehezítik meg elsősorban. A nyelv programozói ismereteket igényel, így nehéz a vállalati folyamatokat modellező szakértőknek a BPEL használata, ha nem is lehetetlen. A BPEL nyelv sajátosságainak figyelmen kívül hagyása a tervezés során megbosszulja magát, mivel a definiált folyamat tipikusan gráf alapú lesz, valamint a gráf pontjaiban meghatározott műveletek nem fognak megfelelni a rendszerben található szolgáltatásoknak egyértelműen. A nyelv korlátai így kihatással vannak a BPEL szerkesztő felületeire is.

12.3. A BPEL kiegészítései

Az előző részben felvázoltuk a BPEL nyelv fontosabb problémáit, amelyek feloldása érdekében több kiegészítő specifikációt adtak ki, ezek közül tekintünk át néhányat röviden a következőkben. Ahogy fentebb olvasható, maguk a BPEL folyamatok egészen addig megállják a helyüket, amíg az állapotaikban elvégzendő feladatoknál egy-egy külső programkódot kell meghívniuk webszolgáltatásokon keresztül. Nincs lehetőség emberi beavatkozásra, adatbevitelre a folyamat lezajlása közben. Ezt a problémát hivatott megoldani a BPEL egy kiterjesztése, amely neve BPEL4People. A BPEL4People a webszolgáltatások menedzselésén, irányításán kívül képes szerepkör alapú emberi tevékenységet is lekezelni, lényegében bevezeti az emberek és szerepkörök definiálásának lehetőségét BPEL környezetben. Ezzel a megoldással lehetővé vált emberi adatbevitelt, teendőt hozzárendelni egyes BPEL állapotokhoz, így még kényelmesebbé, sokrétűbbé téve az üzleti modell vagy folyamat leírását.

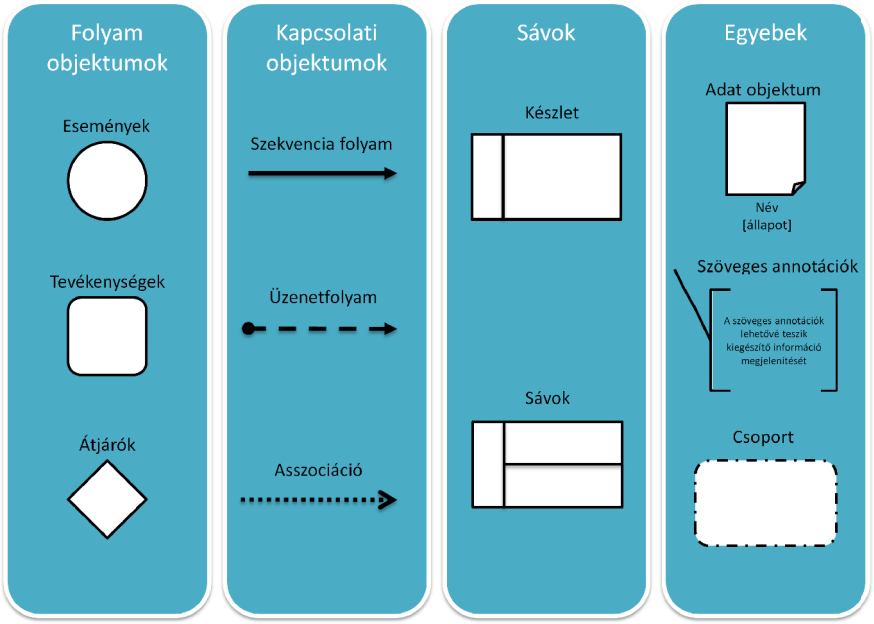
A BPELJ specifikáció a Java nyelv BPEL folyamatba való integrálását célozza meg, amely magába foglalja Java kódrészletek használatát, objektumok változókként kezelését, valamint Java babok (JavaBeans) hívásának támogatását. A Java szabvány létrehozását próbálja megteremteni ebben a témakörben JSR-207 specifikáció. Ezzel a megoldással még sokrétűbb folyamat logika, algoritmus építhető fel a BPEL nyelvvel, és a változók, adattagok kezelését is magasabb szintre emelhetjük az XML szintjéről.

Az üzleti logikát leíró munkafolyamatok, folyamatleíró nyelvek csomópontokból (node) állnak, függetlenül az interfészüktől (amely lehet esetünkben webszolgáltatás). Az egyes csomópontokban feladatokat kell ellátni, amelyek legtöbbször forráskód formájában adottak. A csomópontok egy részhalmaza döntéseket, elágazásokat definiál, leírja hogy az adott

állapotban és kontextusban a munkafolyamatnak mi a következő lépése és feladata. Egy elosztott, szolgáltatás-orientált rendszerben nem csak a munkafolyamatok, hanem a munkafolyamatok csomópontjaiban végzett döntések specifikációja is változhat. Ezeket a döntéseket, üzleti szabályokat, ha forráskódban tároljuk, a változtatás mindig bonyolult, nehézkes és költséges feladat. Ennek kiküszöbölésére a döntési szabályok és tények számítógépes modellezésére széles körben használnak üzleti szabálymotorokat. Csakúgy, mint a munkafolyamatok esetén, így egy magasabb szinten lehet definiálni a szabályokat és a döntési mechanizmust, továbbá könnyen, dinamikusan lehet azokat változtatni és elkülönül a megvalósításuk a rendszer forráskódjától. A munkafolyamatok, vagy a szolgáltatásvezénylés által definiált üzleti logikák kiegészítése szabály-alapú nyelvekkel (a döntési pontokban) megnöveli a rendszer általánosságát, karbantarthatóságát, így a kettő nyelv előnyeit kombinálva hatékonyabb rendszert építhetünk.

A szabály alapú nyelveket a munkafolyamatok kapcsán, így a BPEL esetében is, nem csak a döntési pontokban érdemes felhasználni. Megfelelő keretrendszerre építve a munkafolyamat egyes csomópontjai végrehajtásához elő- és utófeltételek teljesülését is vizsgálhatjuk, így például a BPEL folyamatleírás esetében olyan megvalósítási lehetőségek állnak a rendelkezésünkre, amelyek a munkafolyamat nyelve önmagában nem támogat: logikai kiértékelést, tényeken alapuló döntéseket. Egy ilyen, ESB-re épülő architektúrát mutat be az összefoglaló fejezet, ahol mindezt még kiegészítették egy olyan funkcióval, hogy a BPEL folyamat webszolgáltatás hívásait interceptorok segítségével ágyazzák be az elő- és utófeltételek ellenőrzésének rendszerébe. Az architektúra képes többféle szabálymotor becsatlakoztatására (plug-inek formájában) a szabályok kiértékeléshez, amit egy általános interfész használatára alapoz. Ez az általános interfész a JSR-94 specifikációban lett meghatározva, amely a Java interfésszel rendelkező üzleti szabálymotorok generikus kezelését fedi le.

A fejezet korábbi részeiben már említésre került, hogy nehéz a BPEL folyamatleírások definiálását segítő szerkesztő (BPEL designer) létrehozása. A munkafolyamatok egységesített modellezését specifikálja a üzleti folyamat-modellező jelölésrendszer (Business Process Modeling Notation, BPMN). Az összes komolyabb munkafolyamat-szerkesztő felület, bármely folyamatleíró nyelv esetén a BPMN specifikációt veszi alapul, amely definiálja a folyamatok létrehozásához szükséges grafikus jelölőelemeket és azok kapcsolatait. Az alapelv hasonló, mint az UML aktivitásdiagramokat modellező eszközeinél. Célja, hogy gyorsan átlátható, grafikus szerkesztésen keresztül egyszerűen módosítható, karbantartható üzleti folyamatokat lehessen előállítani költséghatékony módon. Emellett fontos, hogy támogassa mind a fejlesztők, mind az üzleti folyamatokat elemző szakemberek munkáját. A BPMN nem csak a komplex folyamatok grafikus megjelenítését teszi lehetővé, hanem különböző megfeleltetéseket is definiál a vizuális folyamatleírás és konkrét futtatható folyamatleíró nyelv elemei között (így a BPEL számára is). A BPEL esetén ez a megfeleltetés korántsem egyértelműen megadható, és nagy kihívást jelent a szerkesztő felületeket fejlesztőknek a helyes és alkalmazható megvalósítás, amely kellő szabadságot nyújt a felhasználó számára, de képes a BPEL nyelvre jellemző korlátok mindenkori betartására.



53. ábra: BPMN alapelemei

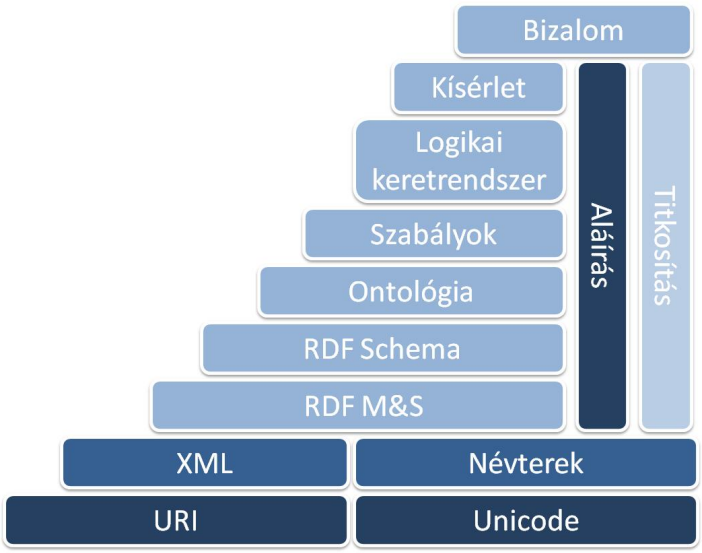
12.4. Jövőkép - Automatizált szolgáltatásvezénylés

A webszolgáltatások, bár betöltik a rájuk szabott feladatot, a technológia mégsem teljes. Túl sok emberi interakció szükséges a szolgáltatások meghajtásához, ami miatt az automatizálás nem lehetséges.

További problémák:

- A szolgáltatás tulajdonságok nem megfelelő lefedettsége: az aktuális WSDL csupán a leírást ad az interfészről, viszont nem ad információt a funkcionális (előfeltételek, hatások), nem funkcionális tulajdonságokról (költség, minőség, elérhetőség), belső viselkedésről. Ezek az információk pedig fontosak lehetnek egy alkalmi együttműködés során.
- A szolgáltatás leírások szabad formátumú terminológiája: a WSDL csak szerkezeti felépítést ad. Tehát mindig a fejlesztő feladata elolvasni a dokumentációt és eldönteni, hogy mely szolgáltatást hívja, és hogyan használja azt fel – ez a BPEL folyamatleírások készítésekor is alapvető probléma.
- Az előre meghatározott szolgáltatások statikus kötése: az emberi beavatkozás szükségessége határt szab az M2M együttműködés alapötletének. A szolgáltatások bevezetésének üteme sokkal gyorsabb lehetne, mint az ütemezés, melyben a programozó illeszti hozzá a szolgáltatásokat a rendszerekhez.
- Egyszerű modell: a jelenlegi webes szolgáltatások nagymértékben támaszkodnak az egyszerű, megszokott modellre. Ám ez a modell rengeteg bonyolult fejlesztői fázist rejt el, amelyek egy automatikus alkalmi együttműködés során kifejezetten szükségesek. A modellnek tartalmaznia kell több lépést is, amelyben a szolgáltatás együttműködések folyamata definiálható, így például a definíciót, közzétételt, felderítést, megegyezést, meghívást, monitorozást, stb.

A problémákból látszik, hogy szükség van a webszolgáltatások egy olyan reprezentációs nyelvére, amely a folyamatok automatizálását segíti elő. Ehhez azonban a rendszernek képesnek kell lennie arra, hogy önállóan felismerje a célokat és az általuk elérni kívánt szolgáltatások képességeit, rendelkezésre állását, valamint bizonyos esetekben a kapcsolódó szolgáltatás-modellt létrehozni. Ehhez szükség van egy módszerre, amely az adatok modellezésére képes. Egy erre szolgáló, létező technológia az erőforrás-leíró keretrendszer (Resource Description Framework, RDF), amely képes az adatok gráfszerű strukturálására. Mindez az információk tételszerű megfogalmazásával biztosított, ahol a tétel tárgy-állítmány-érték hármas formában adott. Önmagában az információ strukturálása nem elég, ha az információ beazonosítására is igény van. Az RDF séma (RDF Schema, RDFS) egy olyan technológia, ami osztályok és tulajdonságok definiálását biztosítja az RDF leírások felett. Ezzel gyakorlatilag kapcsolatok állíthatóak fel különböző források között. Bár az információ tárolása és azonosítása már adott volt a szemantikus web számára, mégsem volt lehetséges halmazelméleti sajátosságok definiálása. Ezen sajátosságok leírására képes az ontológia, fogalomhalmazok definiálásával. Az ontológiákkal képesek vagyunk olyan kapcsolatok megfogalmazására, mint: két halmaz részhalmaza egymásnak vagy tulajdonságok által felvehető értékek egy csoportja korlátozott. A W3C dolgozta ki szabvány jelölő nyelvek azon új generációját, amely egyensúlyba hozza a teljesítményt, a rugalmasságot és az ezek felett álló új generációs webes logikát, új utat nyitva ezzel a webes szolgáltatások következő generációjának. Az új nyelv a webes ontológia nyelg (Web Ontology Language, OWL) néven vált közismerté, mely teljes mértékben RDF és RDF sémára épül.



54. ábra

A fentebb vázolt igények és technológiák alapozták meg a web következő verzióját, ahol szemantikus információval láthatjuk el szolgáltatásainkat ontológiák felhasználásával. Ezek eredményeképpen maga a rendszer is „tudatában van” szolgáltatásai tulajdonságainak és sajátosságainak. Következésképpen akár bonyolult folyamatok is elvégezhetőek egyetlen felhasználói interakcióval. A szemantikus web architektúra célja, hogy egy tudásreprezentációt adjon az adatok kapcsolatáról annak érdekében, hogy a globális mértékű gépi feldolgozás lehetővé váljon. A szemantikus webszolgáltatások egy hatalmas ugrást jelentenek előre, hiszen így a fejlesztő képes olyan alkalmazásokat létrehozni, mint a szemantikus kereső, kollektív e-mail szerkesztő, együttműködésen alapuló webes dokumentumszerkesz-

tő vagy automatizált foglalást végző rendszer és a szemantikus web portál. A szolgáltatásokhoz tartozó belső adatstruktúra reprezentálhatósága nem elegendő a robosztus gépi automatizálás megvalósítására. A következő problémák megoldásával együttesen viszont már lehetséges:

- Felderítés: a programnak képesnek kell lennie a megfelelő webszolgáltatások automatikus keresésére, felderítésére. Sem a WSDL, sem az UDDI nem képes arra, hogy információt adjon a kliens számára rendelkezésre álló szolgáltatásokról. A szemantikus webszolgáltatásnak képesnek kell lennie leírni minden tulajdonságát és képességét. Ezáltal a felhasználói célok is megfogalmazhatóak.
- Meghívás: fontos, hogy a program képes legyen automatikusan meghívni vagy végrehajtani a szolgáltatást. Például, ha egy olyan szolgáltatást hajt végre, amely egy összetett eljárás, képesnek kell lennie megmondani, hogy miként kommunikáljon a szolgáltatással annak érdekében, hogy a teljes folyamaton végighaladjon. A szemantikus webszolgáltatásnak részletesen le kell írnia, hogy a résztvevőknek milyen folyamatokon keresztül kell végig haladnia, hogy elérjék céljukat.
- Kompozíció: az alkalmazásnak képesnek kell lennie webszolgáltatások kiválasztására és kombinálására, hogy elvégezzen egy meghatározott feladatot. A szolgáltatásoknak zökkenőmentesen együtt kell működniük, hogy megfelelő eredményt adjanak.
- Monitorozás: a résztvevő szoftvernek képesnek kell lennie a szolgáltatás tulajdonságainak ellenőrzésére és monitorozására.

A problémák orvoslása csak úgy oldható meg, ha további, a szolgáltatáshoz tartozó lényeges szemantikus információval látjuk el szolgáltatásainkat. Ezen információk három alapvető szempont köré csoportosíthatóak: funkcionális, nem-funkcionális és viselkedés.

Egy szolgáltatás legkritikusabb szempontja a funkcionális követelményeinek leírása, hiszen a kívánt funkcionalitás az ok, amiért a felhasználó igénybe kívánja venni a szolgáltatást. A funkcionalitás tehát meghatározza, hogy a szolgáltatás mire képes (pl. egy dokumentum nyomtatására, szöveg fordítására vagy jármű bérlésére). A funkcionalitás általános specifikálása azonban bonyolult tényező, amely legtöbbször több követelményből tevődik össze (ezek: képesség, strukturális képesség, állapot változás, végül be- és kimeneti funkciók). Az állapotváltozást és a be- illetve kimeneti funkciók együttesét általában egy szolgáltatás IOPE tulajdonságainak szokás nevezni (Input, Output, Precondition, Effect). Összehasonlítva egy program hívásával: a be- és kimenetek meghatározzák a program paramétereit, míg az állapotváltozásba tartozó elő- és utófeltételek egy program állításainak és mellékhatásainak feleltethetőek meg.

A viselkedéssel általában a szolgáltatás azon részét azonosítjuk, amely meghatározza, hogy milyen módon lehet elérni a kívánt funkcionalitást. Például egy termék szállításáért felelős kiszolgáló viselkedés szintű leírása: rakománylista átvétele, termékek rakodása, rakomány ellenőrzése, szállítási engedély kérése, célhelyre mozgatás, kapcsolattartó személy keresése, kirakodás, végül átvételi igazolás kérése.

Az előzőeken felül egy szolgáltatás nem funkcionális tulajdonságai definiálják az olyan képességeket, követelményeket, mint rendelkezésre állás, költség, minőség, megbízhatóság, biztonság, tulajdonos, jogok, stb. E funkcióknak hála a szolgáltatás kiválasztás, megegyezés és monitorozási folyamatok jelentősen könnyebbé válnak.

A felsorolt tulajdonságok meghatározása nélkülözhetetlen egy szemantikus szolgáltatás használatához. Ezek megvalósítása csak egy mérföldkövet jelent a szemantikus web világában, mivel legfőbb célja komplex szolgáltatások végrehajtása. Számos technológia létezik összetett szolgáltatások kezelésére. E technológiák mindegyike feltételezi, hogy az elérni kívánt szolgáltatáshalmaz a saját modelljében definiált. Előfordulhat, hogy szemantikai ütközés lép fel a komplex webes szolgáltatás összeállítása során, mivel az egyes elemek különböző területről származnak.

Tekintsünk egy háromszereplős architektúrát, ahol egy szolgáltatást igénybevevő, egy végrehajtó modul és egy illesztő (matchmaker) helyezkedik el. Mindenekelőtt a webszolgáltatást szemantikus információkkal ellátva kell regisztrálni, tehát ontológiával kell leírni. Ezután a végrehajtó modul kinyeri a szükséges kapcsolati információkat, úgymint név, leírás, kapcsolat más példányokkal, be- és kimenet, stb. és egy folyamat ontológiában tárolja le. Híváskor az illesztő a saját jegyzékében keres a kompatibilis szolgáltatások után, összeillesztve őket, majd visszatér a kiszámolt optimális gráffal és egy generált illesztési tervet javasol a végrehajtó modulnak, amely a gyakorlatban például egy végrehajtható BPEL folyamatot jelent. Végül a modul végrehajtja a kapott tervet és meghívja az így összefűzött webszolgáltatásokat. Ez a használati eset jól példázza, hogy különböző területek ellenére hogyan lehet összefüggő láncot létrehozni komplex szolgáltatások végrehajtására. Mindehhez a technológiának biztosítania kell a már tárgyalt két – weben közismert – fogalmat: szolgáltatásvezénylés (orchestration) és koreográfia (choreography).

A szemantikus webszolgáltatások térhódításával nem a BPEL leírásokat cserélhetjük le, hanem gyakorlatilag a BPEL leírások összeállításának, emberi beavatkozás nélküli létrehozásának lehetőségét teremtjük meg.

A szemantikus web paradigma jelenleg sem tekinthető széles körben elterjedt, felhasznált technológiának. A szolgáltatások szemantikus megközelítése és felhasználása komplex, automatizált rendszerek felépítésére még csak prototípusok keretében létezik, bár egyre több fejlesztői keretrendszer és futtatókörnyezet, szemantikus szolgáltatásokat támogató köztesréteg jelenik meg, amelyek közül a legfontosabbak az OWL-S és a WSMX.

12.5. Összefoglaló

A SOA paradigmára épülő elosztott alkalmazásrendszerek alap építőkövei a szolgáltatások. Ezek üzleti logika mentén meghatározott együttműködéséből összetett szolgáltatások képezhetőek a rendszer számára, amely együttműködést a szolgáltatásvezénylés definiálja. Ahogy a SOA alapú rendszerek terjednek mind a beágyazott, mobil eszközök, mind a felhők, számítási rácsok nyújtotta elosztott számítási rendszerek felé, a szolgáltatásvezénylés megfelelő környezetre való átültetése is szükségessé válik. A BPEL nyelv esetén sincs ez másképpen, és különböző fejlesztések eredményeiképpen már futtatható csökkentett erőforrású, mobil eszközökön, valamint számítási rácsok elosztott számításait vezérlő munkafolyamatok összehangolt vezénylésére is alkalmazzák.

12.6. Kérdések

1. Mi a különbség a szolgáltatásvezénylés és koreográfia között?
2. Mi a BPEL alapvető feladata és milyen részekre bontható egy BPEL folyamatdefiníció?

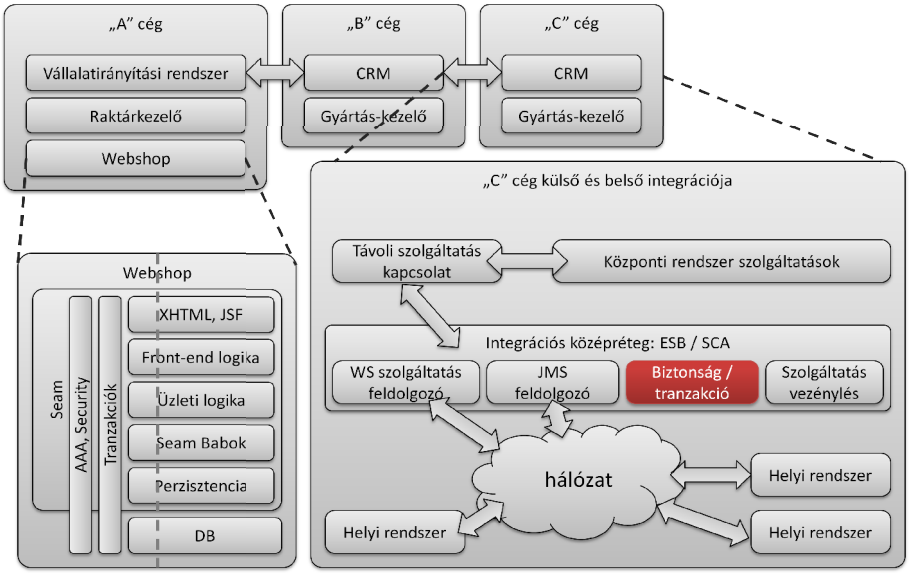
3. Mik a BPEL hátrányai és az ezeket megoldó kiegészítései?
4. Miért nem lehetséges a szűk értelemben vett webszolgáltatások automatizálása?
5. Milyen területeket és miért kell kiegészíteni a szemantikus információk értelmezésének lehetőségével a szemantikus webszolgáltatások alkalmazásához?

12.7. Ajánlott irodalom

- Abdaladhem Albre Shne, Patrik Fuhrer, Jacques Pasquier, "Web Services Orchestration and Composition", University of Fribourg, 2009.
- Benny Mathew, Matjaz B. Juric, Poornachandra Sarang, "Business Process Execution Language for Web Services 2nd Edition", Packt Publishing, 2006.
- Seppo Törmä, Jukka Villstedt, Ville Lehtinen, Ian Oliver, Vesa Luukkala, "Semantic Web Services - A Survey", Helsinki University of Technology, 2008.

13. Keresztülívelő problémák integrált rendszerekben

A 2. fejezetben már találkozhattunk keresztülívelő problémákkal, azaz olyan, jellemzően *nem funkcionális* jellegű kérdésekkel, amelyek *rendszerek között átívelnek* de az egyes rendszereket külön-külön is mélyen érintik. Két ilyen területet említettünk: a *biztonság-kezelést* és a *tranzakció-kezelést*. Ebben a fejezetben megismerkedhetünk azokkal az elterjedt megoldásokkal és technológiákkal, amelyek használatával szabványosan tudunk elosztott rendszerek egyes egységei között biztonságkezeléssel kapcsolatos információt közvetíteni. A fejezetben kizárólag az *azonosítás* (*authentikáció*) és a *jogosultságellenőrzés* (*authorizáció*) területeivel foglalkozunk. A 55. ábra kiemelt része jelzi azt, hogy jelen fejezet az átfogó rendszer mely részét taglalja.



55. ábra

13.1. Elosztott azonosítás és egyszeri bejelentkezés

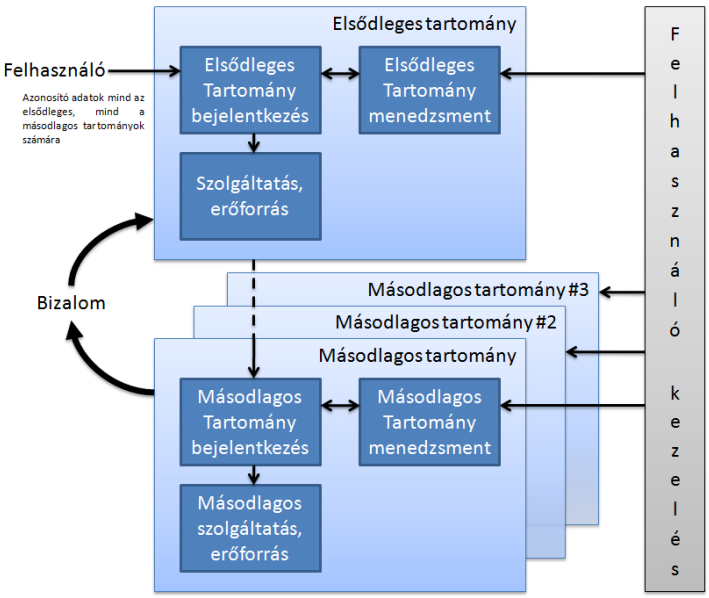
A *személyazonosság-kezelés* meglehetősen fontos tényezővé vált az információtechnológia területén az utóbbi években: az elosztott webes alkalmazások mostanra számos üzleti folyamatban szerepelnek, így megnövelték a felhasználók, csoportok és szerepkörök kezeléséhez szükséges *adminisztrációs terheket*. Minél több rendszer kerül összekapcsolásra, annál inkább megkerülhetetlenné válik ez a kérdéskör. A vállalatok is egyre inkább igyekeznek a működésük hatékonyságukat növelni azáltal, hogy kiterjesztik belső rendszereiket egy szélesebb közönség számára. Ezzel elérhetővé teszik azokat más, külső rendszerek felé, így munkavállalók, ügyfelek és beszállítók számára is. Csak egy konzisztens és megbízható azonosító-infrastruktúra teheti lehetővé, hogy egy vállalat a belső folyamatait megnyissa beszállítói láncok, külső partnerek felé, és nyisson a *gép-gép* (*M2M*) *tranzakciók* világa felé is. Egy közös, egységes felhasználói nyilvántartó rendszer kulcsfontosságú a biztonság tekintetében, és elengedhetetlen kellék az alkalmazás infrastruktúra számára egy ily módon nyitott vállalkozás esetében.

Egy összetett rendszernél a szereplőknek integráció híján minden egyes szolgáltatáshoz külön azonosítóval kell rendelkezniük. Ez azt jelenti, hogy a szolgáltatások igénybevételéhez újabb és újabb bejelentkezési és azonosítási folyamatok szükségesek, amelyeknél különböző felhasználói neveket és azonosítási információkat szükséges megadni. A rendszer szolgáltatásait nyújtó komponensek tehát különálló tartományokként viselkednek, ahol a felhasználónak a teljes rendszertől függetlenül kell azonosítania magát. Ilyen esetekben az egyes komponenseknek nincs információjuk arról, hogy az éppen azonosított felhasználó a rendszer más komponenseiben miképpen szerepel és ott milyen jogosultságokkal rendelkezik. Ez megnehezítheti a rendszerelemek közötti együttműködést. Természetesen ez a rendszer üzemeltetőire is terhet ró: a felhasználói fiókokat, jogosultságokat az egyes rendszerek között manuálisan kell szinkronban és konzisztensen tartani a biztonsági szabályok betartásához.

A használhatóság és a biztonság szempontjából ilyen esetekben indokolt lehet az azonosítási és jogosultságkezelési folyamatok összevonása a különböző tartományokra vonatkozóan. Egy ilyen megoldás, amely az együttműködést segíti, az alábbi előnyökkel járhat:

- Csökkenti a felhasználók által a bejelentkezési és azonosítási folyamatokra szánt időt.
- Növeli a biztonságot, mivel kevesebb azonosítási információt szükséges karbantartani és kezelni mind a felhasználóknak, mind az adminisztrátoroknak.
- Csökkenti az adminisztrátorokra háruló terheket és az egyes adminisztrációs folyamatokra szánt időt a *központosított menedzsmentnek* köszönhetően.
- Az adminisztrátoroknak könnyebb a jogosultságokban történő változtatásokat keresztülvinni az egész rendszeren.

A fent vázoltak egy lehetséges megvalósítása az egyszeri bejelentkezés (*Single Sign-On*, *SSO*) – amely a nevét felhasználók által érzékelt változásról kapta, azaz, hogy az összes komponens eléréséhez elegendő egyetlen egyszer azonosítaniuk magukat.



56. ábra: Egy egyszeri bejelentkezést támogató rendszer felépítése

Ezt a megközelítést az 56. ábra illusztrálja. SSO használata esetén a rendszer az elsődleges tartományba való bejelentkezéskor az összes olyan információt begyűjti, amely szükséges lehet a további tartományokba való azonosításhoz (ahová a felhasználónak hozzáférési jogosultsága van). Valahányszor egy másodlagos tartomány igényli az azonosítást, az SSO szolgáltatáson keresztül a már begyűjtött információk alapján ez megtörténhet. Maga az eljárás többféleképpen is lefolyhat:

- *direkt*: a felhasználó által az elsődleges tartományba való bejelentkezéskor megadott információk kerülnek átadásra a másodlagos tartománynak. (A bejelentkezési adatok tehát mindenhol érvényesek.)
- *indirekt*: a felhasználó által megadott információk az elsődleges tartományba való bejelentkezéskor ahhoz szükségesek, hogy a másodlagos tartományba való bejelentkezéshez szükséges adatokat kinyerjék egy központi nyilvántartó rendszerből. (A bejelentkezési adatok tehát csupán alapul szolgálnak a továbbiakhoz.)
- *azonnali*: az elsődleges tartományba való bejelentkezéskor a munkafolyamat az összes lehetséges másodlagos tartományban is megkezdődik.
- *ideiglenes*: a másodlagos tartományok számára szükséges bejelentkezési információk egy gyorsítótárba kerülnek, és a másodlagos tartományba való bejelentkezéskor kerülnek felhasználásra.

Az SSO megoldásoknak – felhasználási területüket tekintve – három típusát különböztetjük meg:

- A *Web SSO* (*W-SSO*) webes biztonsági szerver és back-end webes alkalmazások között biztosítja az összeköttetést. Segítségével elkerülhető az újbóli bejelentkezés, ha a kliens egy webes erőforrást akar elérni, amely saját felhasználói nyílvántartásából azonosít.
- A *Föderatív SSO* (*F-SSO*) hasonlóan webes erőforrások azonosítását nyújtja, de külső, megbízható webes alkalmazások számára, sokszor a vállalat partnerei irányába.
- Az *Enterprise SSO* (*E-SSO*) vagy más néven *Desktop SSO* kiterjeszti az SSO hatáskörét a webes alkalmazásokon túl a vastag kliensekre, e-mail kliensekre, Java alkalmazásokra valamint egyéb tetszőleges üzleti alkalmazásra.

Természetesen többféle SSO implementáció létezik, melyek különböző technológiákat vesznek igénybe a feladataik elvégzéséhez. A technológiák közül érdemes megemlíteni az alábbiakat: *Central Authentication Service* (*CAS*), *Kerberos*, *X.509*, *SAML*, *OAuth*. A konkrét implementációk között találhatjuk: *Distributed Access Control Systems* (*DACS*), *JBoss SSO*, *JOSSO*, *OpenSSO* (Sun), *OpenAM*, *OpenID*. A népszerűbb webes megoldások között vannak: *Facebook Connect*, *Windows Live ID*, *Ubuntu SSO*, *OneLogin*, *CoSign*.

13.1.1. Címtárak

Az elosztott rendszerekben történő egységes azonosítás egy lehetséges módja címtárak alkalmazása. A címtár gyakorlatilag egy központi tárhely, ami arra hivatott, hogy egy adatforrásba koncentrálja a felhasználói adatokat. A legtöbb vállalat ugyanis, miközben az üzleti rendszerét bővíti, az alkalmazott platformok és alkalmazások számát is növeli. Ezek mindegyike saját formátummal, tárhellyel és protokollal rendelkezik. Az eredmény, hogy a

felhasználók adatai, azonosításukhoz szükséges információk növekvő számban és különböző, izolált helyeken bukkannak fel. Ez azt vonja maga után, hogy az egyes felhasználóknak különböző helyeken, különféle, és nem szinkronizált fiókjaik lehetnek.

Ez a probléma igen gyakran elkerülhető megfelelő címtár alkalmazásával. A címtárak fő előnye, hogy szabványos protokollokon keresztül használhatók, így a különböző gyártók alkalmazásai, operációs rendszerei és köztesréteg termékei mind támogathatják megfelelő hozzáférési protokollt. A legismertebb és legelterjedtebb ilyen címtár-hozzáférési protokoll a könnyűsúlyú címtár-hozzáférési protokoll (*Lightweight Directory Access Protocol*, *LDAP*), amely nagyban támaszkodik az *X.509* nevű szabványra. Mindkettő az *X.500* nevű – címtárakkal kapcsolatos – szabványcsalád fejlesztése eredményeképpen került definiálásra. Az alábbiakban áttekintjük, hogy mit is tud ez a két technológia.

- Az *LDAP* definiálja azt a *kommunikációs protokollt*, amely felhasználásával szabványos módon lehet egy címtárszerverrel kommunikálni, azaz meghatározza hozzáféréskor a kliens által a címtár szolgáltatás irányába és szolgáltatás által ezekre adott válaszként küldött üzenetek formátumát. Az LDAP nem határozza meg magát a címtár szolgáltatást, ugyanakkor sok helyen *LDAP címtárként* emlegetik azokat a címtárakat, amelyek képesek LDAP üzenetekkel kommunikálni. (Máshol úgy tartják, az LDAP csupán a protokoll, és nincs olyan, hogy LDAP címtár.)
- Az *X.509* egy nyilvános kulcs alapú *ITU-T* (az *International Telecommunication Union* telekommunikációs szabványosítási területe) szabvány nyílt rendszerek összekapcsolására. A szabvány többek között a *nyilvános kulcs tanúsítványok* felépítését, és ezen tanúsítványok kiadását (*issue*), ellenőrzését (*check*) és visszavonását (*revocation*) specifikálja. Ebben a rendszerben a *tanúsító hatóság* (*Certificate Authority*, *CA*) bocsát ki tanúsítványokat nyilvános kulcsot rendelve egy egyedi azonosítóhoz (ami lehet egy *név* az X.500 szabvány szerint, egy *DNS-cím* vagy akár egy *e-mail cím*). A tanúsítvány és a tanúsító hatóság megbízhatósága a *gyökértanúsítványtól* függ. A gyökértanúsítványok implicit megbízhatóak. Egy X.509 (v3) digitális tanúsítványnak három fő részeleme van: a *tanúsítvány*, az *aláírási algoritmus* és az *aláírás*. Az X.509 tanúsítványok használatát az LDAP mellett számos protokoll támogatja. Ilyenek – többek között – az *SSL/TLS*, az *IPSec*, az *S/MIME*, a *SmartCard*, az *SSH*, a *HTTPS* és az *EAP*.

Az egységes azonosítást nagyban segíti az imént említett szabványok használata. Van azonban egy fontos következménye a hagyományos (LDAP) címtárak használatának, mégpedig az, hogy ha az elosztott rendszerünk különböző komponensei a felhasználói azonosítást egy ilyenen keresztül végzik, akkor tudniuk kell, hogy melyek azok a megbízható címtárak, amelyek a rendelkezésre álló információt biztosítják.

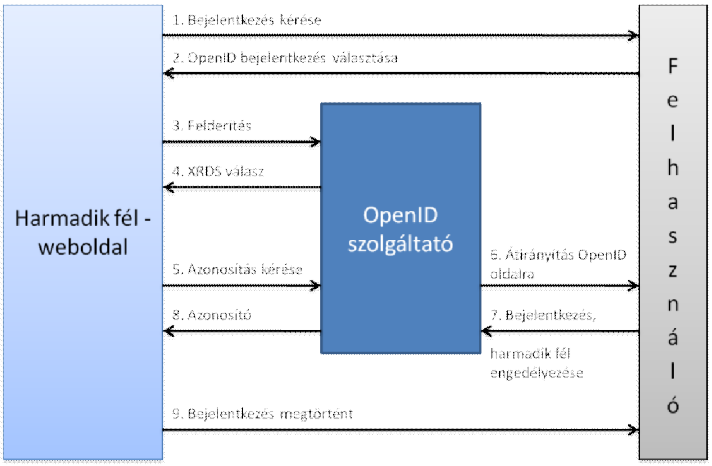
13.1.2. Elosztott azonosítás

Manapság egyre inkább elterjedőben van az *elosztott azonosítással* egybekötött egyszeri bejelentkezésre való igény. Az alábbiakban az *OpenID* és az *OAuth* technológiákat ismertetjük röviden. Ez a két technológia a mai, elosztott azonosítást támogató webes alkalmazások meglehetősen nagy részét lefedi.

13.1.2.1. OpenID

Az implementációk és szabványok között az egyik legelterjedtebb az OpenID. Az OpenID szabvány célja, hogy levegye a felhasználók válláról a többféle fiók kezelésének nehézségeit a különböző szolgáltatásokhoz. Más szemszögből nézve a szolgáltatások üzemeltetőit is segíti: nem szükséges számukra a felhasználói bejelentkezéseket és biztonságot szavatoló funkciókat egyedileg implementálni. Az OpenID mindezt úgy valósítja meg, hogy egy keretrendszert biztosít, ahol a felhasználók egy fiókot hoznak létre valamilyen OpenID szolgáltatónál, és ezt a fiókot használják olyan szolgáltatások esetén, amelyek elfogadják az OpenID azonosítókat.

Amikor egy harmadik fél (alkalmazás szolgáltató) kérelmet intéz az OpenID szolgáltatóhoz, amely bejelentkezteti a felhasználót a már létező fiókkal, és a harmadik fél számára elküldi azt az egyedi azonosítót, amely alapján az felismerheti a felhasználót. Ez az azonosító konzisztens, így több munkafolyamaton keresztül is biztosított a felhasználó felismerése.



57. ábra: OpenID példafolyamat

A folyamat szereplői tehát a harmadik fél webes alkalmazása, vagyis alkalmazás szolgáltató (SP), az OpenID szolgáltató, vagyis identitás szolgáltató (IdP) és maga a felhasználó – természetesen valamilyen kliensen keresztül, ami legtöbb esetben egy web böngésző.A folyamat főbb lépései az alábbiak:

1. Az első lépésben a harmadik fél (SP) felkínálja többek között az OpenID bejelentkezést, amikor a felhasználó védett tartalomhoz akar hozzáférni, vagy magát a szolgáltatást igénybe venni.
2. Második lépésben a felhasználó kiválaszthatja az OpenID bejelentkezést.
3. Ha ez a mód kerül kiválasztásra, a harmadik fél felderítést végez. Ez akkor történik, amikor a harmadik fél még nem ismeri a felhasználó OpenID azonosítóját, de már ismert a szolgáltató (IdP).
4. A harmadik fél egy XRDS dokumentumot kap, amely hitelesítve és aláírva van és információt szolgáltat a további lépésekhez.
5. A harmadik fél ezt követően kérheti a felhasználó azonosítását.

6. A kliens átirányítja a felhasználót az OpenID szolgáltatóhoz.
7. Itt a felhasználó megadja bejelentkezési adatait, - ha még nincs aktív munkamenete az OpenID szolgáltatóval -, és ha szükséges, megszabja az engedélyeket a harmadik fél számára.
8. Ezt követően a szolgáltató átadja az egyéni azonosítót a harmadik félnek a szükséges és engedélyezett további információval egyetemben.
9. Ezzel megtörténik a bejelentkezés a harmadik fél weboldalára.

13.1.2.2. OAuth

Az *OAuth* (*Open Authorization*) szintén egy nyílt szabvány engedélyezési folyamatokra. Az OAuth kiegészíti az OpenID megoldást, de mint különálló egység. A megoldás lényege, hogy úgy lehessen erőforrásokat, adatokat megosztani harmadik féllel, hogy ne legyen szükséges az azonosítási adatokat kiadni, mint a felhasználói nevet és jelszót. Ezt úgy éri el, hogy a szolgáltatónál tárolt adatok eléréséhez tokeneket készít. Egy token hozzáférést biztosít tehát egy kimondott tartományon belül egy vagy több erőforráshoz, anyaghoz egy bizonyos időtartamig.

13.2. Elosztott jogosultságellenőrzés

Az elosztott jogosultságellenőrzéssel kapcsolatos általános technológiák manapság kezdenek elterjedni. Ebben az alfejezetben a biztonsági állításjelölő nyelv (*Security Assertion Markup Language*, *SAML*) és a hozzá tartozó jogosultságellenőrzési modell alapján vizsgálunk meg egy tipikus elosztott ellenőrzési folyamatot, majd nagy vonalakban áttekintünk egy olyan jogosultságellenőrzési modellt, amelyben tisztán elkülönülnek az egyes felelősségek, az XACML-t.

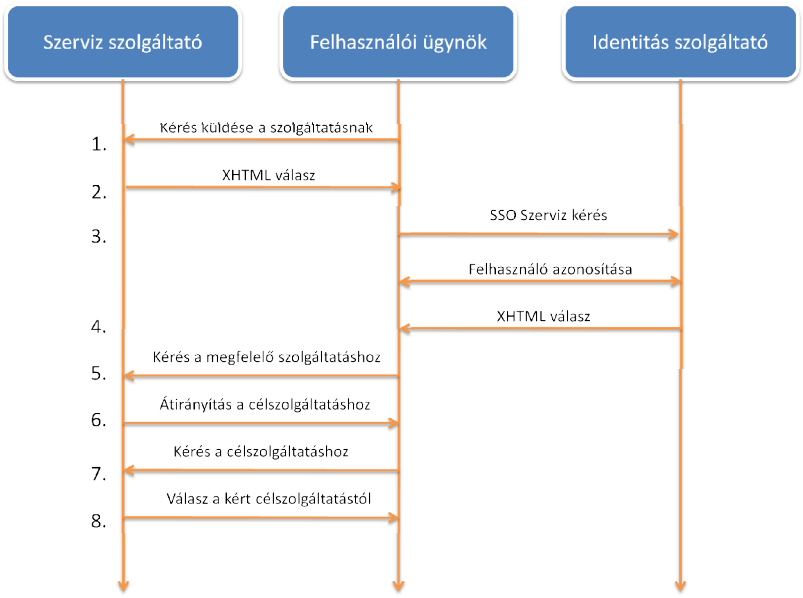
13.2.1. SAML

A SAML egy, az OASIS által kifejlesztett XML-alapú nyílt szabvány, amely azonosítási és engedélyezési adatok cseréjét teszi lehetővé különböző biztonsági tartományok (azaz identitás szolgáltatók és alkalmazás szolgáltatók) között. A SAML előtt nem volt olyan XML alapú szabvány, amely segítségével egy biztonsági rendszer azonosítási adatokat továbbíthatott volna egy olyan harmadik fél felé, amely megbízik benne. Használata során feltételezzük, hogy a felhasználó rendelkezik azonosítóval legalább egy identitás szolgáltatónál (amely azonosítási szolgáltatásokat nyújt a felhasználónak). Ugyanakkor a SAML nem specifikálja ezeknek a szolgáltatásoknak az implementációját, azaz a SAML számára mellékes, hogy az identitás szolgáltatók miképp vannak megvalósítva.

Az SAML folyamat jellemzően a következőképpen néz ki. A felhasználó kérésére az alkalmazás szolgáltató egy *SAML kérelmet* intéz az identitás szolgáltató irányába. Az identitás szolgáltató azonosítás után egy *SAML állítást* (*assertion*) küld az alkalmazás szolgáltatónak, aki ennek alapján hoz hozzáférési döntést. Általánosságban elmondható, hogy az ilyen SAML üzenetek a következő alakúak: „*Jelen állítás t időben lett kiadva R által, S tárgyban, C feltételekkel*”.

A szolgáltatók különféle módszereket használhatnak a SAML üzenetek cseréjére, mint például SOAP, HTTP átirányítás (GET), HTTP POST, SAML URI.

A kidolgozott és elterjedt implementációk (mint pl. a *Shibboleth* vagy a *simpleSAMLphp*) esetében lehetőség van annak szabályozására, hogy egy IdP mely SP-knek milyen, a felhasználókról tárolt adatot adhat ki, továbbá hogy egy SP mely IdP-től milyen felhasználói adatot fogad el.



58. ábra: SAML ellenőrzési folyamat (példa)

A folyamat (58. ábra) kezdeményező szereplője tehát a *felhasználó* egy kliensprogrammal (web böngésző), amely a *szolgáltatótól* (SP) egy erőforráshoz vagy szolgáltatáshoz akar hozzáférni. Az alkalmazás szolgáltatónak ekkor szüksége van a felhasználó azonosítására, így SAML azonosítási kérelmet intéz az *identitás szolgáltatóhoz* (IdP) a kliens segítségével.

1. A felhasználó védett tartalomhoz akar hozzáférni.
2. A szolgáltató egy XHTML űrlapot küld válaszként, ami tartalmazza a szolgáltató által legyártott SAML azonosítási kérelmet (*SAML Authentication Request*)
3. A kliens a kapott XHTML űrlapot (a benne lévő SAML kérelemmel együtt) elküldi az identitás szolgáltatóhoz, ahol megtörténik a felhasználó azonosítása
4. Az identitás szolgáltató egy XHTML űrlapot küld vissza a kliensnek, amely tartalmazza a legyártott SAML válaszüzenetet is (*SAML Authentication Response*)
5. A kliens a kapott űrlapot elküldi a szolgáltató megfelelő (azonosítási) címére
6. A szolgáltató ellenőrzi az űrlapon keresztül kapott SAML azonosítási információt, és ha ez megfelelő, akkor átirányítási üzenetet küld a kliensnek
7. A kliens ezen az új (átirányított) címen keresztül kéri az erőforrást
8. A szolgáltató most már (a biztonsági kontextus megléte miatt) szolgáltatni tudja a kívánt erőforrást

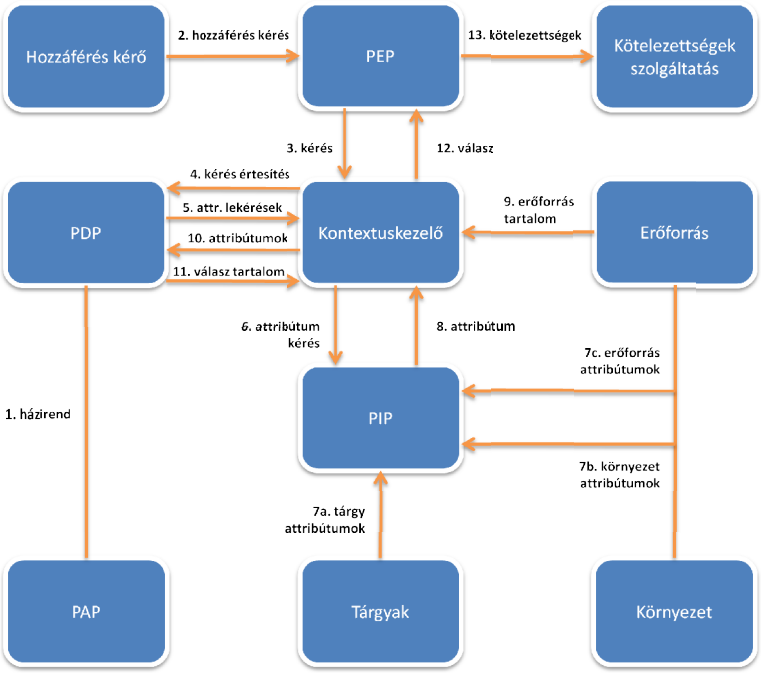
Vegyük észre, hogy a SAML ilyen módú használata kiválóan beépül a meglévő webes infrastruktúrába, ezáltal egyszerű webböngésző is használható kliensprogramként, az azo-

nosítási folyamat pedig úgy történhet meg, hogy a felhasználó esetleg tudatában sincs annak, hogy éppen SAML azonosítást használ.

13.2.2. XACML

Láthatjuk tehát, hogy a SAML felhasználásával kiválóan végezhetünk elosztott azonosítást és bizonyos mértékig jogosultságellenőrzést is. Igazán kifinomult elosztott jogosultságellenőrzést a SAML azonban nem tud biztosítani. Ha ilyenre van szükségünk, használhatjuk például a szintén XML-alapú XACML szabványt és a hozzá tartozó biztonsági modellt.

Az XACML modell egy sokszereplős elosztott biztonsági modell, amely rugalmas jogosultságellenőrzést tesz lehetővé. Az 59. ábra mutatja be e modell szereplőit (érdemes megjegyezni, hogy egy szoftvermodul egyszerre több szerepet is betölthet, tehát fizikailag nem feltétlenül kell az ábrán látható számú elemnek szerepelnie).



59. ábra: Az XACML modell és folyamat

Az XACML ellenőrzési folyamata az alábbi lépésekből áll:

1. A *házirend-adminisztrációs pont* (*Policy Administration Point*, *PAP*) tartja nyilván a *házirendeket* és elérhetővé teszi azokat a *házirend döntési pont* (*Policy Decision Point*, *PDP*) számára. Ezek a házirendek egy-egy adott erőforrásra vonatkoznak.
2. A hozzáférés kérő entitás a kéréseit minden esetben a *házirend-végrehajtó pontnak* (*Policy Enforcement Point*, *PEP*) küldi el.
3. A *PEP* a kérést a *Kontextuskezelő* felé továbbítja, amely ettől a ponttól kezdve gyakorlatilag közvetít a többi modul között az ellenőrzési folyamat során.
4. A *Kontextuskezelő* egy XACML kérést generál a *PDP* felé.

5. A *PDP* – ismervén, hogy a házirend milyen tulajdonságok alapján rendelkezik, – a megfelelő attribútumokra vonatkozó információkérést visszaküldi a *Kontextuskezelőnek*.
6. A *Kontextuskezelő* a *PDP* által kért attribútumokra vonatkozó kérést elküldi a *házirend információs pontnak* (*Policy Information Point*, *PIP*). A *PIP* tudja, hogy az egyes *attribútumokat* honnan és milyen módon lehet lekérdezni.
7. A *PIP* összegyűjti a megfelelő helyekről (*erőforrások*, *környezet*, *személyek*) a kért attribútumokat.
8. A *PIP* visszaküldi a kért attribútumokat a *Kontextuskezelőnek*.
9. Ha szükséges, a *Kontextuskezelő* az attribútumokkal együtt a megcélzott erőforrás *tartalmának* egy részét is begyűjtheti (*tartalomalapú döntések* esetén szükség lehet erre).
10. A *Kontextuskezelő* továbbítja a kért *attribútumokat* (és az esetleges *tartalmat*) a *PDP*-nek.
11. A *PDP* meghozza a *döntést* az ismert házirendek alapján, és a döntést visszaküldi a *Kontextuskezelőnek*.
12. A *Kontextuskezelő* a *PEP* felé továbbítja a döntést.
13. A *PEP* eleget tesz a *kötelezettségeinek*. Ezek olyan további szükséges tevékenységek, amelyeket a *PDP* a döntéssel együtt előír a *PEP* számára. A kötelezettségek jellemzően olyan formális követelményeknek tesznek eleget, amelyeket más módon nehéz megfogalmazni hozzáférési szabályok mentén.
14. Amennyiben a hozzáférés engedélyezett, úgy a *PEP* lehetővé teszi az erőforráshoz való hozzáférést, máskülönben tiltja azt.

Látható tehát, hogy az XACML folyamatnak számos szereplője van, és az ellenőrzési folyamat indokolatlanul hosszadalmasnak és nehézkesnek tűnhet. Ez a felépítés azonban lehetővé teszi, hogy az ellenőrzési modellben minden egyes résztvevőnek világos és egyértelmű felelősségi köre legyen. Ilyen módon teljesen szétválaszthatók és külön-külön szabályozhatók a folyamat egyes részterületei.

Amint a fentiekben már szerepelt, egy XACML folyamat megvalósításakor nem szükséges minden egyes szerepkört külön modullal megvalósítani, sokszor a különböző szerepkörök összekapcsolhatók, így a modell komplexitása csökkenthető.

13.3. Kérdések

1. Mi az egyszeri bejelentkezés? Milyen előnyökkel jár a felhasználók és a vállalatok számára?
2. Mire használható egy címtár? Milyen módon tudunk hozzáférni?
3. Milyen lehetőségeket biztosít az OpenID és mi a működési elve?
4. Milyen biztonsági folyamatokban tudjuk használni az SAML-t és milyen módon?
5. Mik az XACML modell legfontosabb szereplői és hogyan működnek együtt egy jogosultságellenőrzési folyamat során?

13.4. Ajánlott irodalom

- Steel, C. és mások: Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management. Prentice Hall, 2005. ISBN 0-131-46307-1
- Hammer-Lahav, E. és mások: The OAuth 1.0 Protocol. RFC 5849. Elérhető: http://tools.ietf.org/html/rfc5849
- ITU-T: X.500 - The Directory: Overview of concepts, models and services (5. kiadás). ITU-T Recommendation, 2005. Elérhető: http://www.itu.int/rec/T-REC-X.500-200508-I/dologin.asp?lang=e&id=T-REC-X.500-200508-I!!PDF-E&type=items
- ITU-T: X.509 - The Directory: Public-key and attribute certificate frameworks (5. kiadás). ITU-T Recommendation, 2005. Elérhető: http://www.itu.int/rec/dologin.asp?lang=e&id=T-REC-X.509-200508-I!!PDF-E&type=items
- Maler, E.: Web Services and Identity: 2.2 Federated Identity Technologies. Sun Microsystems, 2007. Elérhető: http://www.xmlgrrl.com/publications/fed-id-tech-27jul2007.pdf
- Schumacher, M. és mások: Security Patterns - Integrating Security and Systems Engineering. Wiley, 2005. ISBN 978-0-470-85884-4.
- The OpenID Community: OpenID Authentication 2.0 - Final. 2007. Elérhető: http://openid.net/specs/openid-authentication-2_0.html
- Moses, T. és mások: *eXtensible Access Control Markup Language (XACML) Version 2.0.* OASIS Standard, 2005. Elérhető: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

13.5. Összefoglaló

Jelen jegyzet egy felületes áttekintést adott az alkalmazások, alkalmazásrendszerek fejlesztésének és integrációjának legfontosabb alapelveiről, megoldási lehetőségeiről. Több megközelítés is létezik implementáció szintjén az ilyen összetett rendszerek fejlesztésére, amelyek különböző metodológia mentén próbálják leküzdeni a fejlesztésben és integrációban rejlő kihívásokat. Mindehhez egy tervezési és megvalósítási alapelvet nyújt a SOA, amit elterjedten alkalmaznak és napjainkban a legtöbb elosztott alkalmazásrendszer felépítése követ.

A jegyzet bevezető részében megismerkedhettünk az elosztott rendszerek mögött húzódó alapelvekről, amelyek életre hívták az alkalmazásrendszerek és integrációs köztesrétegek elméleti és gyakorlati kidolgozását. Az elosztott rendszerek jól definiált tulajdonságokkal rendelkeznek, amik a hibatűrés, skálázhatóság, nyíltság és a heterogenitás. Az elosztott rendszerek, – ahogy nevük is mutatja – több, akár fizikailag is különálló rendszeren futhatnak, de együttesen a felhasználó számára egy egységként szolgálják ki az üzleti folyamatokat. Az elosztottság ilyen szintű kezeléséhez elengedhetetlen a gépek, szerverek, rendszerek magas szintű szervezése, amelyek végül, ahogy egy adott méretet

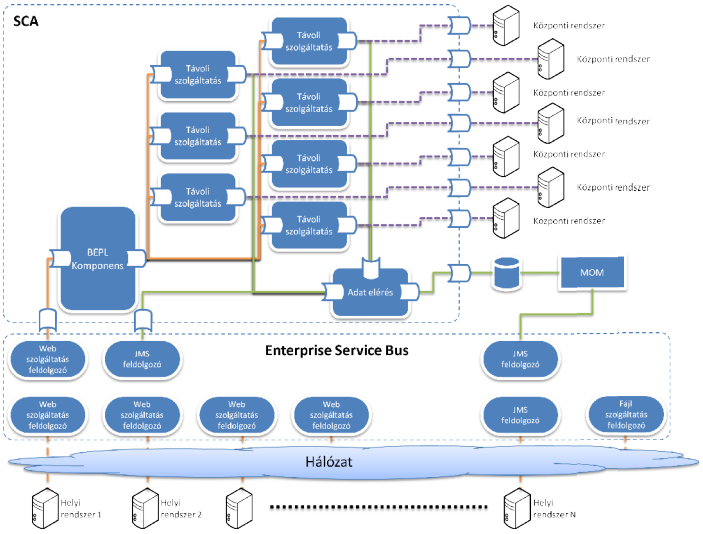
elérnek, fürtöket, rácsokat és felhőket eredményeznek, ami meghatározza a rendszer teljesítményét és laza csatolását.

Az elosztott rendszerek alapköveit képező összetett szolgáltatások, alkalmazások esetében a legfontosabb feladat – függetlenül a megvalósítástól és a felhasznált keretrendszerektől, programozási nyelvektől – az adott célfelhasználók, vállalkozások üzleti folyamatainak támogatása és a számukra releváns adatok kezelése, tárolása. E két alapvető cél hátteréről, fontosságáról, alapfogalmairól és stratégiai felépítéséről egy összefoglalást adtunk a jegyzet második részében. Az üzleti logikát és az adatábrázolást célzó megvalósítások, elméletek fejlődését áttekintve jól megfigyelhető, hogyan próbálják az újabb és újabb generációs lépcsőkön a korábbi koncepciók hiányosságait megoldani, a korlátozásait feloldani.

A jegyzet harmadik és negyedik részében konkrét keretrendszerek és köztesrétegek kerültek bemutatásra, amelyek a teljes rendszerfejlesztés adott célfeladataira optimalizált megoldások. Az alkalmazásfejlesztést és rendszerintegrációt támogató implementációk közül a számunkra megfelelő eszközök kiválasztását a fejlesztői probléma vagy cél maga definiálja (adott körülmények, adott üzleti alkalmazások és folyamatok esetén melyik köztesréteg a leghatékonyabb). Az alkalmazásfejlesztés terén részletesebben csak egy-egy keretrendszer került ismertetésre a felhasználói interakciókat, az üzleti logikát és az adatkezelést illetően, amely három terület az alkalmazásfejlesztés alapjait adja. Az alkalmazásfejlesztésen túl, a szolgáltatások, rendszerek összekötésére alkalmazott integrációs köztesrétegek felhasználása lehetővé teszi a szolgáltatások felett magas szintű üzleti logika definiálását, ezzel egy újabb szintre emelve az alkalmazásfejlesztést. Generikus, a mindenkori elvárásokhoz és követelményekhez könnyen igazítható alkalmazáscsomagok létrehozását segítik elő. Az integrációs köztesrétegek jól azonosítható, általánosan megfogalmazható problémákat oldanak meg, amelyek a szolgáltatás-becsomagolás és üzenet-transzformáció, szolgáltatásvezénylés és koreográfia, valamint a keresztülívelő kontextusok támogatása (ezek minden köztesrétegre jellemzőek). Fontos kritérium a megfelelő integrációs eszköz kiválasztásánál az üzenetküldés formája, amely lehet szinkron vagy aszinkron. További befolyásoló tényező a teljesítmény, valamint a karbantarthatóság.

A szolgáltatásorientált alkalmazásrendszerek megvalósításához felhasznált köztesrétegek bár egyedi sajátosságokkal rendelkeznek, nem kell szükségszerűen egymást kizáró megoldásoknak tekinteni őket. Kombinálva azokat (azaz a teljes rendszerfejlesztés és rendszerintegráció bizonyos területeire optimalizált típusait összekapcsolva) még hatékonyabb rendszert építhetünk. E megoldás egyetlen követelménye, hogy a felhasznált keretrendszerek jól körülhatárolt és pontosan definiált szolgáltatást nyújtsanak a teljes rendszer számára. Így a köztesrétegek is szabadon integrálhatóak egymással, amely a SOA világ szépségét és bonyolultságát is adja egyben.

Különböző keretrendszerek, köztesrétegek kombinációjából felépített rendszerintegrációs környezetet mutat be az 60. ábra.



60. ábra: Összetett rendszerintegrációs környezet

A rendszerben az alap kommunikációs és integrációs réteget az ESB szolgáltatja. Az ESB köztesréteg kétfajta adapterrel rendelkezik, hogy interfészt biztosítson egyrészt web-szolgáltatás, másrészt JMS alapú üzenetváltáshoz. A MOM és SCA komponensek szintén ilyen adapterekkel kerültek illesztésre. A keretrendszer szükség esetén kiegészíthető továbbá JCA és EJB adapterekkel.

A keretrendszerbe integrált SCA specifikációt kielégítő futtatókörnyezet lehetővé teszi az üzleti folyamatok rendszerhatárokat átívelő módon történő kezelését. A rendszerben az interakciók tulajdonképpen a következetesen kialakított, SCA alapú interfészek és hivatkozások mentén értelmezhetőek, valósíthatóak meg. Emellett a komponens BPEL alapú implementációja lehetővé teszi a független szolgáltatások rendszerezését üzleti folyamatok köré és a szolgáltatásvezénylést. Egy dedikált adatbázis is helyet kapott a rendszerben, amely elengedhetetlen a globális adatok és az üzleti folyamatok állapotának tárolásához.

A B2B interakció a következőképpen valósul meg a rendszerben. Minden lokális, vállalat-specifikus rendszer képes a saját üzleti folyamatait végrehajtani. Amikor külső rendszerek által nyújtott szolgáltatást kell igénybe vennie, egy kérést küld az ESB számára. Az ESB elvégzi a szükséges üzenet-transzformációkat és kommunikációs protokoll átalakításokat, továbbítja a kérést az SCA-alapú BPEL motor felé. A BPEL-ben implementált munkafolyamat végrehajtódik, meghívva az szükséges külső rendszerszolgáltatásokat, majd visszatér az eredménnyel az ESB számára. Az ESB elvégzi az üzenet transzformációját és a folyamat végső állomásaként az eredeti hívó félnek továbbítja azt.

Az adatok megosztása, és az adatforgalom biztosítása további szolgáltatása a keretrendszernek, amelyet üzenetalapú megoldással valósít meg. A rendszerbe integrált MOM köztesréteg feladata az ehhez szükséges üzenetszűrés, elő- és utófeldolgozás, valamint az adatokat tartalmazó üzenet továbbítása. A MOM köztesréteg szintén az ESB-n, adaptereken keresztül érhető el, amely a szükséges transzformációkat végzi a keretrendszerek között. Kétféle adatforgalomra képes a keretrendszer, a közzététel/feliratkozás (Publish/Subscribe) és a kérés/válasz (Request/Response) módszeren alapuló implementációra, amelyeket korábban már ismertettünk. Az első esetben az adatot szolgáltató komponens nem közvetlenül küldi az adatot igénylők számára az üzenetet, hanem a MOM köztesrétegen keresztül. Itt a köztesréteg feladata, hogy értelmezze az üzenetet és továbbítsa azt az érde-

keltek felé az ESB-n keresztül. Ebben a formában az adatforgalmat az adatszolgáltató kezdeményezi. A másik esetben (kérés/válasz) az adatforgalmat az igénylő kezdeményezi, mégpedig úgy, hogy ESB-n keresztül egy kérést küld a MOM köztesréteg felé, amely továbbítja azt a megfelelő adatszolgáltató számára, majd a választ visszaküldi a kérelmezőnek.

A felvázolt rendszer életképességét jól példázza, hogy sikeresen adaptálták több terülten is 2008-ban, azóta is számos városban üzemeltetik főként a munkavállalói szerződések kezelésére. A teljesítmény terén szintén elmondható az architektúra létjogosultsága a SOA világán belül, hiszen egy valós, e mintát követő elosztott rendszer átlagosan napi 4-5 ezer kérést fogad és dolgoz fel hatékonyan úgy, hogy a válaszidő soha nem haladta meg az egy másodpercet még a csúcsidőszakban sem.

A rendszer ábráján a „lokális rendszerek” mutatják azokat az alkalmazásokat, amelyek célspecifikus üzleti logikát tartalmaznak, felhasználói felületet szolgáltatnak, és tárolják a nem közösségi felhasználásra szánt adatokat, így megvalósítva a tipikus háromrétegű felépítést. A „központi rendszerek” az általános szolgáltatásoknak adnak helyet az elosztott rendszerben, ezek közül is megkülönböztetett figyelmet kapott a központi perzisztencia-szolgáltatás.

A bemutatott rendszer jól ábrázolja az elosztott, szolgáltatás-orientált alapon megvalósított alkalmazásrendszerek működését. Manapság egyre több hasonló rendszer fog napvilágot látni, és ezek mind jobban lesznek konfigurálhatóak, karbantarthatóak, optimalizálhatóak az egyedi üzleti elvárások számára. A magas szinten definiált üzleti logikák és kooperációk egyszerűséget és újrafelhasználhatóságot nyújtnak a teljes rendszer számára, és ahogy a szemantikus szolgáltatások elterjednek, még komplexebb, automatizált szolgáltatásrendszerek fognak megjelenni. Ezek a fejlesztők válláról is leveszik a terhet a szolgáltatásvezénylés megvalósítása terén.

Animációk

1. animáció:
SOA

2. animáció:
Fürt

3.animáció:
Rács

4. animáció:
Kétfázisú Commit 1.

5. animáció:
Kétfázisú Commit 2.

6. animáció:
Kétfázisú Commit hiba 1.

7. animáció:
Kétfázisú Commit hiba 2.

8. animáció:
Szemantikus adatmodell

9. animáció:
JSP

10. animáció:
JSF-életciklus

11. animáció:
Háromrétegű architektúra

12. animáció:
Kérés–válasz

13. animáció: Hub and Spoke

14. animáció: Bus architektúra

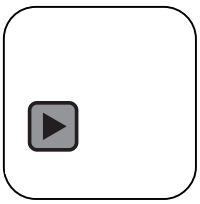
15. animáció: SDO

16. animáció: OSGi

17. animáció: BPEL

18. animáció: OpenID

19. animáció: SAML

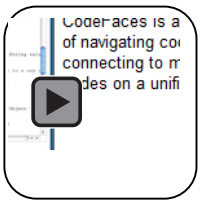


20. animáció: XACML

21. animáció: Perzisztens objektumok átmenetei

22. animáció: RAP demo 1.

23. animáció: RAP demo2.



24. animáció: RAP demo3.

25. animáció: RAP demo 4.

Ábrák, animációk jegyzéke

Ábrák

Animációk