2010. Március

Informatikai Navigátor

Gondolatok a szoftverek használatáról és fejlesztéséről

CreedSoft

3. szám

Tartalomjegyzék

Főszerkesztő: Nyiri Imre (imre.nyiri@gmail.com)

1. SQLite - Lokális adatbáziskezelés

Az *SQLite* egy szoftver library és néhány utility, amik lehetővé teszik a szerver nélküli, lokális SQL adatbázisok létrehozását. Célja, hogy olyan eszközt adjon a programozók kezébe, ami magas szinten támogatja az SQL alapú adatkezelést olyan esetekben is, amikor nem szükséges egy komoly hálózati adatbázis szerver. Jellegénél fogva kiváló komponense lehet sok mobileszköz belső adatkezelésének. Alapelvei között van a zéró konfiguráció, ismeri a tranzakciókezelést.
Webhelye: `http://www.sqlite.org/`

Kik és miért használják?

Az *SQLite* egy beágyazott (embedded) relációs adatbáziskezelő, azaz az őt használó szoftverhez linkelve lehet használni, nem egy külön SQL szerver. Nincs user adatbázisa. Ettől eltekintve nagyrészt megvalósítja az SQL-92 szabványt. Olyan esetekben érdemes megfontolni az alkalmazását, amikor a helyi rendszeren, azaz a localhost-on szeretnénk az adatainkkat tárolni. Az SQLite-ot neves szoftverek és eszközök használják: MAC OS, Solaris 10, Skype, iPhone, Firefox. A fejlesztők szlogenje szerint használjuk ott ezt a könyvtárat, ahol egyébként a *fopen()* parancsot használnánk. Gondoljunk arra, hogy egy hordozható eszközhöz írt szoftvernek általában pont ilyen adatbáziskezelőre van szüksége.

Adatbázisok esetén az *ACID* mozaikszó az Atomicity (atomicitás), Consistency (konzisztencia), Isolation (izoláció), és Durability (tartósság) rövidítése. Ezek az adatbáziskezelő rendszer tranzakciófeldolgozó képességeinek alapelemei. Enélkül az adatbázis integritása nem garantálható, így a tranzakciókezelés támogatott ebben a környezetben is.

Részlegesen megvalósítja a triggereket és a legtöbb komplex/összetett lekérdezést. Az SQLite szokatlan típuskezelést használ az SQL adatbázis-kezelőhöz: egy adattípus nem a tábla oszlopaihoz, hanem egyedi értékekhez van hozzárendelve, más szóval dinamikus típuskezelést használ, gyengén típusos adatkezelés mellett. Amennyiben string típusú adat beilleszthető integer oszlopba, akkor a SQLite először a stringet integerre konvertálja, ha az oszlop preferált típusa integer. Ez nagyobb rugalmasságot ad az oszlopoknak, ami hasznos lehet dinamikus típuskezelésű script-nyelvekben való alkalmazás esetén, azonban ez a technika nem vihető át más SQL adatbázis-kezelőkbe. Az SQLite képtelen a tipikus adatbázisokban található szigorúan típusos oszlopok kezelésére.

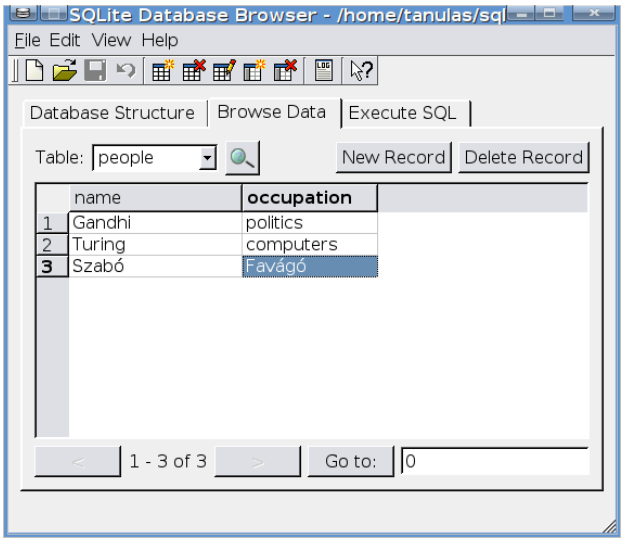
Ugyanazt az adatbázist több processz és szál használhatja egyidejűleg problémamentesen. Az olvasási kérelmek kiszolgálása párhuzamosan történik. Az írási kérelmek végrehajtása akkor történik meg, amikor nincs folyamatban más kérelem kiszolgálása, egyébként az írási kérelem sikertelen lesz és hibakóddal tér vissza, illetve lehetőség van egy beállítható várakozási idő elteltével a kérelem ismétlésére. Ez a konkurens hozzáférési állapot megváltozható ideiglenes táblák használata esetén.

Az SQLite számos programnyelvből használható, így BASIC, C, C++, Common Lisp, Java, C#, Visual Basic .NET, Delphi, Curl, Lua, Tcl, R, PHP, Perl, Ruby, Objective-C (Mac OS X-en), Python, newLisp, Haskell, OCaml, Smalltalk és Scheme nyelvekhez rendelkezik illesztőfelülettel.

Az adminisztrációs kliens felület

Az SQLite rendelkezik parancssoros (neve: *sqlite3*) és GUI-s (neve: *SQLite Database Browser*, 1. ábra) kezelőfelülettel is. A GUI egy minimális

és szokásos adatkezelő felületet ad, amin interraktív módon elvégezhetjük a legfontosabb feladatokat: adatbázisok létrehozása, megnyítása, SQL parancsok kiadása, browse funkciók. Használata nem jelenthet gondot.



1. ábra. SQLite Database Browser GUI

A mindennapi adminisztratív munkát hatékonyan az *sqlite3* parancsoros eszközzel tudjuk elvégezni. Indítása: *sqlite3 <adatbázis-file-név>*. Egy SQLite adatbázis egyetlen file-ként jelenik meg az operációs rendszerben, ami kényelmes kezelhetőséget eredményez a mindennapi munkában.

Hozzunk létre egy *inyiri.sqlite* nevű adatbázis az *sqlite3 inyiri.sqlite* paranccsal! A továbbiakban ezt az adatbázis fogjuk használni példáinkban. Lépjünk be az adatbázisba, aminek az eredményét az alábbiakban látható sqlite prompt megjelenése mutat:

```
$ sqlite3 inyiri.sqlite
SQLite version 3.6.16
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

A *.help* „pontparancs” segítséget arról, hogy ezen a konzolon mit tehetünk. Az adatbázis tábláit például a *.tables* mutatja meg:

```
sqlite> .tables
Actors          Reservations    people
Customers       passwords       phones
sqlite>
```

Látható, hogy jelenleg 6 db tábla van. A táblák szerkezetét (sémáját) a *.schema* paranccsal tudhatjuk meg. Nézzük meg a *Customers* és *Reservations* táblákat!

```
sqlite> .schema Customers
CREATE TABLE Customers(CustomerId integer PRIMARY KEY,
Name text);
sqlite> .schema Reservations
CREATE TABLE Reservations(Id integer PRIMARY KEY,
                          CustomerId integer, Day text);
```

Érdekes lehetőség a .dump parancs, ami egy tábla tartalmát SQL parancsban adja vissza:

```
sqlite> .dump Customers
BEGIN TRANSACTION;
CREATE TABLE Customers
(CustomerId integer PRIMARY KEY, Name text);
INSERT INTO "Customers" VALUES(1,'Paul Novak');
INSERT INTO "Customers" VALUES(2,'Terry Neils');
INSERT INTO "Customers" VALUES(3,'Jack Fonda');
INSERT INTO "Customers" VALUES(4,'Tom Willis');
COMMIT;
sqlite>
```

A *.output customers.sql* parancs olyan, mint a *.dump*, de az eredményt egyből a megadott file-ba menti. Nagyon gyakori, hogy egy külső sql file – mint script – végrehajtása szükséges. A *Customers* és *Reservations* táblákat például így hoztuk létre, erre szolgál a *.read* parancs. Nézzük ezt az sql scriptet (a file neve: *test.sql*):

```
-- SQL for the Customers, Reservations tables

BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS
Customers(CustomerId integer PRIMARY KEY, Name text);
INSERT INTO Customers(Name) VALUES('Paul Novak');
INSERT INTO Customers(Name) VALUES('Terry Neils');
INSERT INTO Customers(Name) VALUES('Jack Fonda');
INSERT INTO Customers(Name) VALUES('Tom Willis');

CREATE TABLE IF NOT EXISTS
Reservations(Id integer PRIMARY KEY,
             CustomerId integer, Day text);
INSERT INTO Reservations(CustomerId, Day)
VALUES(1, '2009-22-11');
INSERT INTO Reservations(CustomerId, Day)
VALUES(2, '2009-28-11');
INSERT INTO Reservations(CustomerId, Day)
VALUES(2, '2009-29-11');
INSERT INTO Reservations(CustomerId, Day)
VALUES(1, '2009-29-11');
INSERT INTO Reservations(CustomerId, Day)
```

```
VALUES(3, '2009-02-12');
COMMIT;
```

A fenti script futtatása:

```
sqlite> .read test.sql
```

Az *sqlite3* konzol tudása ennél természetesen több, itt csak ízelítő és kedvet szerettünk volna adni a használatához.

Az SQL használata

Maradjunk még az *sqlite3* konzolban, de a továbbiakban már nem az *SQLite* parancsokat nézzük, hanem a használható SQL-92 nyelv lehetőségeit mutatjuk be. Az SQLite támogatja a szokásos DDL (Data Definition Language) lehetőségeket:

- CREATE – tábla, index, view, trigger létrehozás
- ALTER TABLE – egy tábla sémájának módosítása
- DROP – tábla, index, view, trigger törlés

Egy tábla oszlopainak típusai ilyenek lehetnek:

- NULL - The value is a NULL value
- INTEGER a signed integer
- REAL - a floating point value
- TEXT - a text string
- BLOB - a blob of data

Ezek az SQL parancsok szabványosak, itt nem ismertetjük emiatt. Például az *Actors* tábla átnevezése ilyen:

```
sqlite> .tables
Actors          Reservations    people
Customers       passwords       phones
sqlite> ALTER TABLE Actors
        RENAME TO Szinesz;
sqlite> .tables
Customers       Szinesz         people
Reservations    passwords       phones
sqlite>
```

Az SQLite ismer egy sor beépített operátort, nézzük meg őket a program webhelyén!

Az ismert constraints-ek támogatottak: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, DEFAULT. Ezek természetesen az oszlopok attributumai itt is.

Nézzük meg a *select* parancs lehetőségeit! Az oszlop nevének lekérdezéskori átkeresztelése itt is lehetséges. Ezt mutatja ez a 2 select parancs:

```
sqlite> select name from Customers;
Name
----------
Paul Novak
Terry Neil
Jack Fonda
Tom Willis
sqlite> select name as nev from Customers;
nev
----------
Paul Novak
Terry Neil
Jack Fonda
Tom Willis
sqlite>
```

A *limit* záradék használható, azaz a *select*-ben megadott *limit n* biztosítja, hogy maximum *n* sora legyen az eredményhalmaznak. Létezik az *offset n* záradék is, ami az eredményhalmaz első *n* sorát kihagyja.

Az SQL egyik érdekes lehetősége a JOIN, ami itt is támogatott. A következőekben áttekintjük őket. Ehhez példaképpen a *Customers* és *Reservations* táblákat fogjuk használni:

```
-- Ez a vevő foglal szobákat
sqlite> select * from Customers;
CustomerId  Name
----------  ----------
1           Paul Novak
2           Terry Neil
3           Jack Fonda
4           Tom Willis
```

```
-- Foglalások
sqlite> select * from Reservations;
Id          CustomerId  Day
----------  ----------  ----------
1           1           2009-22-11
2           2           2009-28-11
3           2           2009-29-11
4           1           2009-29-11
5           3           2009-02-12
sqlite>
```

Az alapértelmezett összekapcsolása a táblának az *Inner Join*, azaz a belső összekapcsolás. Itt csak azokat *Customers* sorokat hozza le, ahol van összekapcsolódás a *Reservations* táblában lévő valamely sorral.

```
sqlite> SELECT Name, Day FROM Customers AS C
        JOIN Reservations
        AS R ON C.CustomerId=R.CustomerId;
Name        Day
----------  ----------
Paul Novak  2009-22-11
Terry Neil  2009-28-11
Terry Neil  2009-29-11
Paul Novak  2009-29-11
Jack Fonda  2009-02-12
sqlite>
```

Látható, hogy Terry Neil 2 foglalt nappal is rendelkezik. Ez az SQL select egyébként ekvivalens a megszokottabb kinézetű következő paranccsal:

```
sqlite> SELECT Name, Day
  FROM Customers, Reservations
  WHERE
   Customers.CustomerId = Reservations.CustomerId;
Name        Day
----------  ----------
Paul Novak  2009-22-11
Terry Neil  2009-28-11
Terry Neil  2009-29-11
Paul Novak  2009-29-11
Jack Fonda  2009-02-12
sqlite>
```

Támogatott a NATURAL INNER JOIN, ami kihasználja az oszlopok névegyezőségét, esetünkben ez most a *CustomerId* oszlop:

```
sqlite> SELECT Name, Day FROM Customers
        NATURAL JOIN Reservations;
Name        Day
----------  ----------
Paul Novak  2009-22-11
Terry Neil  2009-28-11
Terry Neil  2009-29-11
Paul Novak  2009-29-11
Jack Fonda  2009-02-12
sqlite>
```

Csak érdekességképpen említjük meg a CROSS INNER JOIN nevet, amivel mindenki csak a „where nélküli többtáblás” select néven szokott találkozni:

```
sqlite> SELECT Name, Day FROM Customers
CROSS JOIN Reservations;
Name        Day
----------  ----------
Paul Novak  2009-22-11
Paul Novak  2009-28-11
Paul Novak  2009-29-11
Paul Novak  2009-29-11
Paul Novak  2009-02-12
Terry Neil  2009-22-11
Terry Neil  2009-28-11
Terry Neil  2009-29-11
Terry Neil  2009-29-11
Terry Neil  2009-02-12
Jack Fonda  2009-22-11
Jack Fonda  2009-28-11
Jack Fonda  2009-29-11
Jack Fonda  2009-29-11
Jack Fonda  2009-02-12
Tom Willis  2009-22-11
Tom Willis  2009-28-11
Tom Willis  2009-29-11
Tom Willis  2009-29-11
Tom Willis  2009-02-12
sqlite>
```

Ez természetesen ekvivalens ezzel:

```
SELECT Name, Day FROM Customers, Reservations;
```

Most térjünk át a külső összekapcsolások (Outer joins) tesztelésére! Itt nem csak az összeilleszkedő sorok kerülnek bele az eredménytáblába, hanem azok is, amiknek nem találtunk párt a másik táblában. Talán feltűnt, hogy Tom Willis neve nem került be az eredménybe (kivéve a CROSS esetet). Ez azért van, mert az ő CustomerId-je 4, ilyen érték pedig nincs a Reservations táblában. Amennyiben mégis látni akarjuk őt is, úgy a LEFT OUTER JOIN-t kell használni:

```
sqlite> SELECT Name, Day FROM Customers
        LEFT JOIN Reservations
ON Customers.CustomerID=Reservations.CustomerId;

Name          Day
------------  ----------
Paul Novak    2009-22-11
Paul Novak    2009-29-11
Terry Neil    2009-28-11
Terry Neil    2009-29-11
Jack Fonda    2009-02-12
Tom Willis
sqlite>
```

Természetesen ekkor a Tom Willis sorának *Day* értéke *NULL*, de létezik ez a sor is. Ez az SQL parancs ezzel ekvivalens:

```
SELECT Name, Day FROM Customers
LEFT JOIN Reservations USING (CustomerId);
```

A RIGHT és FULL OUTER JOIN típusú lekérdezések jelenleg nem támogatottak.

A select utasításban használhatóak a szabványos SQL függvények (core, agregáló, dátum), például:

```
sqlite> select max(CustomerId) from Customers;
max(CustomerId)
---------------

4
sqlite>
```

Néha fontos, hogy az adattáblákat egy viewn keresztül lássuk, legyen ez most a már ismert LEFT JOIN-os select-ünk:

```
sqlite> create view allcustomer as
SELECT Name, Day FROM Customers
LEFT JOIN Reservations USING (CustomerId);

sqlite> .tables
Customers     Szinesz       passwords     phones
Reservations  allcustomer   people
```

Látható, hogy a nézetek a táblák között listázódnak.

A triggerek bemutatásához létrehozunk 2 új táblát!

```
CREATE TABLE Log
(Id integer PRIMARY KEY,
 OldName text,
 NewName text, Date text
);

CREATE TABLE Friends
(Id integer PRIMARY KEY,
 Name text UNIQUE NOT NULL,
 Sex text CHECK(Sex IN ('M',   'F')));

CREATE TRIGGER mytrigger
UPDATE OF Name ON Friends
BEGIN
INSERT INTO Log(OldName, NewName, Date)
VALUES(old.Name, new.Name, datetime('now'));
END;
```

A bemutatott trigger feladata, hogy loggolja a *Friends* tábla változtatásait.

A tranzakciókezelésre már láttuk a megfelelő parancsokat: *BEGIN TRANSACTION*, *ROLLBACK*, *COMMIT*.

Az SQLite átttekintése után kezdjük el használni programjainkban. A részletekbe nem megyünk bele, inkább csak szeretnénk felvillantani a használat lehetőségét *C*, *Java*, *C#* és *Pascal* (Lazarus) környezetekből.

A C nyelvű felület

A *C* nyelvű SQLite megvalósítás az alap, így természetesen az SQLite programozás bemutatását is ezzel kezdjük. Példaprogramunk (1. programlista) azt fogja csinálni, hogy parancssorból fogad 2 paramétert: az adatbázis file nevét és a kiadandó SQL parancsot, amit mindjárt végre is hajt az adatbázison. A 3. sorban az API használatát tesszük lehetővé az *sqlite3.h* header beemelésével. A 7-17 sorok közötti *callback* nevű függvény feladata, hogy az eredménytábla egy-egy sorát ő dolgozza fel. Az *argc* paraméter kapja meg az aktuális rekord oszlopainak számát. Az *argv* egy stringekre mutató tömb, ami az adott sor konkrét oszlopértékeit tartalmazza, míg az *azColName* az oszlopok neveit adja át. A működés a lehető legegyszerűbb. Az aktuális rekord minden oszlopát egymás után kiirja a képernyőre. A 19. sorban kezdődő főprogram *db* változója reprezentála az adatbázist. A 25-29 sorok azt vizsgálják meg, hogy a parancsorból 3 paraméter jött-e át a programunknak,

amennyiben nem, úgy megjeleníti a helyes használat helpjét. Azért kell 3 paraméter, mert *C* nyelven az elindított program a 0. és utána jön még 2 valódi paraméter. Az *argv[1]* tartalmazza az adatbázis nevét, amit a 30. sorban próbálunk megnyitni, később majd a 45. sorban fogjuk lezárni. A 37. sorban az *argv[2]* paraméterben átvett SQL parancs lesz meghívva. Az *sqlite3_exec()* első paramétere a már említett *db*, azaz az adatbázis, hiszen egyszerre többet is használhatunk. A 4. paraméter az esetleges hibaüzenet szövege, amennyiben a 38-44 sorokban erre szükség lenne. A 3. paraméter pedig a mi *callback()* függvényünk, amit belül az *sqlite3_exec()* minden eredménysorra meg fog hívni.

```
1  // 1. programlista: Egy C nyelvű mini sql commander
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sqlite3.h>
6
7  static int callback(void *NotUsed, int argc, char **argv, char **azColName)
8  {
9        NotUsed=0;
10       int i;
11       for(i=0; i<argc; i++)
12       {
13         printf("%s = %s\n", azColName[i], argv[i] ? argv[i]: "NULL");
14       }
15       printf("\n");
16       return 0;
17 }
18
19 int main(int argc, char **argv)
20 {
21       sqlite3 *db;
22       char *zErrMsg = 0;
23       int rc;
24
25       if( argc!=3 )
26       {
27         fprintf(stderr, "Usage: %s DATABASE SQL-STATEMENT\n", argv[0]);
28         exit(1);
29       }
30       rc = sqlite3_open(argv[1], &db);
31       if( rc )
32       {
33         fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
34         sqlite3_close(db);
```

```
35            exit(1);
36        }
37        rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
38        if( rc!=SQLITE_OK )
39        {
40            fprintf(stderr, "SQL error: %s\n", zErrMsg);
41            /* This will free zErrMsg if assigned */
42            if (zErrMsg)
43                free(zErrMsg);
44        }
45        sqlite3_close(db);
46        return 0;
47    }
```

A C API használatát a `http://www.sqlite.org/cintro.html` helyen tanulmányozhatjuk részleteiben, illetve magát az API-t a `http://www.sqlite.org/c3ref/funclist.html` helyen találjuk meg.

A program fordítása és futtatása így lehetséges.

```
Fordítás:
gcc test.c -I/usr/include -lsqlite3 -o test

Futtatás:
$ ./test inyiri.sqlite" "select * from phones"
datum = 2010-01-01
name = Nyiri Imre
number = 5791832
```

Az *inyiri.sqlite* a már fentiekben ismertetett adatbázis neve (a *phones* táblában tényleg csak 1 record volt!).

Java alapú SQLite adatbáziskezelés

Az SQLite-hoz van JDBC driver, így a Java-ból használni egy adatbázist nem okoz gondot. Ez érdekes lehetőség olyan mobil eszközök esetén, amik Java alapon működnek, ugyanis ez a driver 100%-ig Java nyelven van implementálva, nem igényel semmilyen külső natív könyvtárat. A 2. programlistán láthatjuk a példaprogramunkat. A 17-19 sorokból a connection string formátumát érdemes megnézni, ahogy átadjuk az adatbázis file nevét (*inyiri.sqlite*). A 21. és 22. sorok a DDL műveletet mutatják. Észrevehetjük azt a lehetőséget, hogy egy mező típusát nem kötelező megadni, ekkor *TEXT* lesz. A program a most létrehozott *people* táblába beszúr 3 recordot, majd ki is listázza azokat a képernyőre.

```
1  // 2. programlista: SQLite használata Java nyelven
2
3  package cs.test.sqlite;
4
5  import java.sql.Connection;
6  import java.sql.DriverManager;
7  import java.sql.PreparedStatement;
8  import java.sql.ResultSet;
9  import java.sql.Statement;
10
11 public class Test
```

```
12 {
13
14     public static void main(String[] args) throws Exception
15     {
16         Class.forName("org.sqlite.JDBC");
17         Connection conn =
18                 DriverManager.getConnection(
19            "jdbc:sqlite:/home/tanulas/sqlite/inyiri.sqlite");
20         Statement stat = conn.createStatement();
21         stat.executeUpdate("drop table if exists people;");
22         stat.executeUpdate("create table people (name, occupation);");
23         PreparedStatement prep = conn.prepareStatement(
24                 "insert into people values (?, ?);");
25
26         prep.setString(1, "Gandhi");
27         prep.setString(2, "politics");
28         prep.addBatch();
29         prep.setString(1, "Turing");
30         prep.setString(2, "computers");
31         prep.addBatch();
32         prep.setString(1, "Wittgenstein");
33         prep.setString(2, "smartypants");
34         prep.addBatch();
35
36         conn.setAutoCommit(false);
37         prep.executeBatch();
38         conn.setAutoCommit(true);
39
40         ResultSet rs = stat.executeQuery("select * from people;");
41         while (rs.next())
42         {
43             System.out.println("name = " + rs.getString("name"));
44             System.out.println("job = " + rs.getString("occupation"));
45         }
46         rs.close();
47         conn.close();
48     }
49 }
```

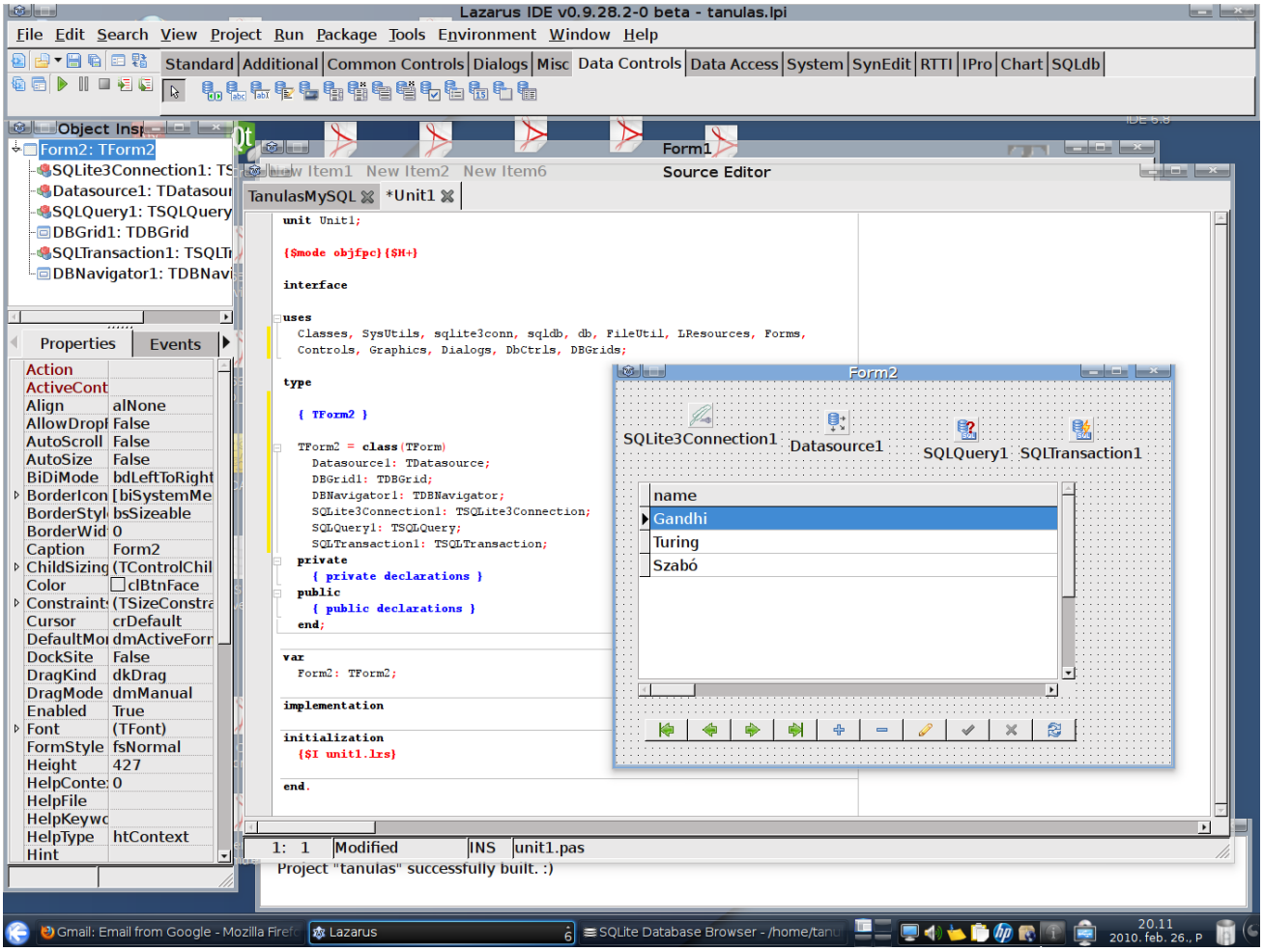
C# (mono) alapú SQLite adatbáziskezelés

A .NET a másik fontos környezet, így nem csoda, hogy innen is használható az SQLite. Aki ismeri a C# (.NET) adatbáziskezelést, annak a 3. programlista valószínűleg csak egyetlen új információt jelent, ami a 15-16 sorokból kiolvas-

ható connection URI. A program futtatásának eredményét a 47-52 sorok között láthatjuk, azaz ez a példaprogram kilistázza a Java-s példában már megismert *people* tábla tartalmát.

```
1  // 3. programlista: Az SQLite és a C#
2
3  using System;
4  using System.Data;
5  using Mono.Data.SqliteClient;
6
7  namespace sqlite
8  {
9
10 class MainClass
11 {
12     public static void Main(string[] args)
13     {
14         Console.WriteLine("SQLite Select:");
15         string connectionString =
16             "URI=file:/home/tanulas/sqlite/inyiri.sqlite,version=3";
17         IDbConnection dbcon;
18         dbcon = (IDbConnection) new SqliteConnection( connectionString )
19         dbcon.Open();
20         Console.WriteLine("Name\t\t\tJob");
21         Console.WriteLine("-------------------------------------------");
22                 IDbCommand dbcmd = dbcon.CreateCommand();
23         string sql = "select * from people";
24
25         dbcmd.CommandText = sql;
26         IDataReader reader = dbcmd.ExecuteReader();
27         while(reader.Read())
28         {
29                 string name  = reader.GetString (0);
30                 string occup = reader.GetString (1);
31                 Console.WriteLine(name + "\t\t\t" + occup );
32         }
33
34         // clean up
35         reader.Close();
36         reader = null;
37         dbcmd.Dispose();
38         dbcmd = null;
39         dbcon.Close();
40         dbcon = null;
41      }
42
43 } // end class
44 } // end namespace
45
46 Futtatás:
47 SQLite Select:
```

```
48 Name                    Job
49 ------------------------------------------
50 Gandhi                  politics
51 Turing                  computers
52 Szabó                Almaszedő
```



2. ábra. Az SQLite és a Lazarus

Lazarus (Open source Delphi)

Az utolsó példa egy kiváló desktop alkalmazás fejlesztő környezetből, a Lazarus-ból való használatot mutatja be. Előljáróban 3 fontos URL-t szeretnék kiemelni:

- Lazarus honlap: `http://www.lazarus.freepascal.org/`
- Magyar lazarus közösség: `http://lazarus.freepascal.hu/`
- Free Pascal honlap: `http://www.freepascal.org/`

A *Lazarus* projekt 1999 februárjában indult azzal a céllal, hogy megalkossa a 90-es évek népszerű *Delphi* környezetének a nyílt forráskódú

3. ábra. SQLite, Lazarus és Pascal emblémák

változatát. Együtt fejlődik a *Free Pascal* projecttel, ami egy open source *Object Pascal* implementáció. Ez a vállalkozás mára egy nagyon érett, kiváló környezetet eredményezett, aminek használatára bíztatok mindenkit (2. ábra). Aki ismeri a Delphi-t, annak a 2. ábra ismerős lehet. Az adatbázisok használatának elve természetesen Lazarusban is ugyanaz. A *Form2* mutatja, hogy szükségünk van egy connection komponensre, itt adjuk majd meg az *inyiri.sqlite* adatbázisunk nevét is. Az *SQLQuery1* egy select parancsot reprezentál, aminek az eredménye egy tábla. Ezen komponens *SQL* propertyjében lehet megadni a kérdéses select-et, ahogy azt Delphi-ben is megtanultuk korábban.

A *Datasource1* komponens a híd az adatforrás és a GUI vezérlők között. Most 2 vezérlő van a *Form2*-ön: egy grid és az azt kezelő navigátor. Az *SQLQuery1* komponenst – ahogy láthatjuk – fejlesztési időben is aktívvá tehetjük, ekkor a vezérlőkben megjelennek az élő adatok. A fenti példát nem részletezzük tovább, aki ismeri a Delphi-t, annak ennyi induló munníció is elegendő a használathoz.

Csak röviden megemlítjük a használható típusokat a *TSqliteDataset* és *TSqlite3Dataset* osztályok esetén: Integer, AutoInc, String, Memo, Bool, Float, Word, DateTime, Date, Time, LargeInt és Currency.

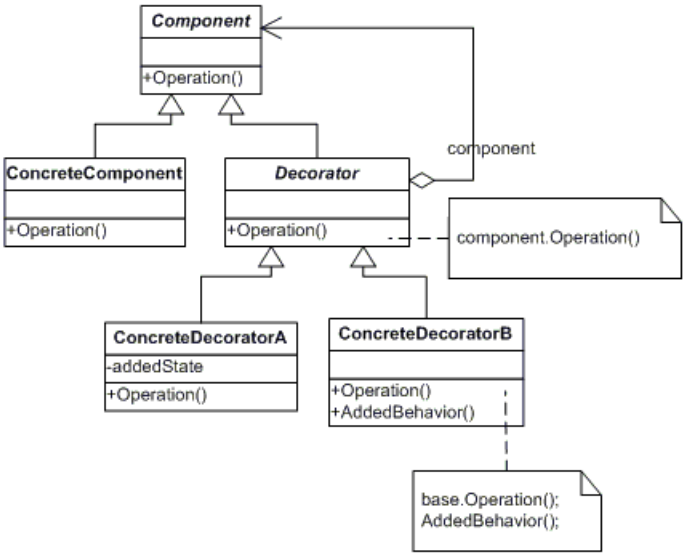
Az *SQLQuery1* SQL property használata: *SQL := 'Select * from TABLENAME'.*

2. Objektumaink tudása – A Decorator minta

Folytatjuk a tervezési mintákról szóló cikksorozatunkat, ezúttal a dekorátor minta lehetőségeiről elmélkedünk. Amikor egy objektumnak csak kicsit kell másképpen viselkednie, akkor természetes ötlet az öröklődés használata. Van azonban egy kis gond. Amennyiben túl sok apró eltérésünk van és azok kombinációja is szerepet játszik, rengeteg osztály jönne létre. Ezen segít ez a tervezési minta, ahol az objektumaink tudását dinamikusan rakjuk össze. Érdekes következménye ennek a szemléletnek, hogy a csatornák fölötti szűrőkre is remek megoldást ad.

Mire jó ez a minta?

Képzeljük el, hogy van egy osztályunk, ami tud valamit. Legyen a neve: *AlapClass* és rendelkezik néhány publikus metódussal: *pm1()*, *pm2()*, ... Nevezzük ezt az osztály használati felületének. Sokszor adódik, hogy ez a felület marad és azért kell egy utód class-t létrehozni, mert kissé (vagy nagyon) változik az a specifikáció, amit az osztály működésétől elvárnánk. Ezek a működésbeli módosítások a tapasztalat szerint csak 1 vagy néhány metódust szoktak érinteni, a többi változatlan marad. Legyen például *M1*, *M2*, *M3* 3 különféle működési módosítási igény, amire *M1AlapClass*, *M2AlapClass*, *M3AlapClass* utód-osztályokat készítettük el és reménykedünk, hogy mindegyiket akkor és ott fogjuk használni, amikor éppen szükséges. Aztán kiderül, hogy olyan osztályokra is szükség van, amik egyszerre több módosítási igénynek is megfelelnek, azaz kellenek ilyen utód osztályok is: *M1M2AlapClass*, *M1M3AlapClass*, *M1M2M3AlapClass*. Ez 6 új osztály, pedig csak 3 módosítási feltételünk van. A helyzet és az osztályok fölösleges elszaporodásának kezelésére egy elegáns megoldás a *decorator* (ismert nevei még: díszítő, toldalék, csomagoló, wrapper) minta.



4. ábra. Decorator design pattern

Egy konkrét példa. Legyen az *AlapClass* most a *Window*, ami egy téglalap alakú terület részért felelős a képernyőn. Az M1=lehessen görgetni, M2=lehessen naplózni tartalmának a változását, M3=lehessen menteni az előző ablaktartalmat. A Window metódusai elég szűkszámúak, de a példánknak ez most megfelel: *show()*, *hide()*, *addContent()*. Az M1 feltétel a *show()* metódusra lesz hatással, ami egy scrollbar-t is elhelyez az ablakon. Az M2 feltétel az *addContent()*-re hat, ami a metódus naplózásra való felkészítését jelenti. Az M3 feltétel lehetséges, hogy szintén csak az *addContent()*-re fog hatni. Már itt is lehet érezni, hogy az M_i feltételeket osztályörökléssel nem lenne gazdasá-

gos megvalósítani, főleg amennyiben ezek gazdag kombinációira is szükség van.

Az M1, M2, ... feltételekre gondolhatunk úgy is, mint az alap class egy-egy aspektussal való tudásbővítési igényei.

A decorator minta UML diagramját a 4. ábra mutatja. A következőkben 2 konkrét példán keresztül elmagyarázzuk a használatát és a konstrukció mögött meghúzódó filozófiát. Ezen az ábrán a *Component* egy interface (milyen metódusok játszanak szerepet), míg az *AlapClass* neve itt a *ConcreteComponent* class. A *Decorator* interface és annak implementációi pedig az M1, M2, ... szempontok megvalósításának felel meg.

A minta használhatóságának alapfeltétele, hogy létezzen egy közös *Component* interface a dekorálandó és a dekoráló osztályok számára, azaz a kliens programot úgy lehessen megírni, hogy annak mindegy legyen a megkonstruált *Component* objektum létrehozásának módja (azaz, hogy mennyi M1, M2, ... feltételt, mint aspektust igyekeztünk figyelembe venni az objektum kliensbeli használatakor).

Első példa: Egy naplózó környezet

A decorator minta bemutatásához nézzünk meg egy egyszerű napló szolgáltatást! A közös interface-t a 4. programlista tartalmazza. A 3 interface metódus szemantikája a következő:

- *getEntries()* – visszaadja a bejegyzések listáját. Az egy bejegyzést reprezentáló *DiaryEntry* class-t a 6. programlista mutatja
- *newEntry()* – egy új naplóbejegyzés elhelyezése a naplóba
- *getEntry()* – az i. naplóbejegyzés lekérése a naplóból.

```
// 4. programlista: A naplót használó kliensek ezt
// az interface-t látják.
// A 3. UML ábra Component interface-ének felel meg.

package cs.test.dp.decorator;

import java.util.List;

public interface IDiary
{
    public List<DiaryEntry> getEntries();
    public void             newEntry(DiaryEntry entry);
    public DiaryEntry       getEntry(int index);
}
```

Nézzük meg ezekután azt a *Diary* class-t, ami egy alapnaplót valósít meg, szerepe a 4. ábra *ConcreteComponent* szerepével egyezik meg. Az implementációt az 5. programlista tartalmazza, annyira triviális, hogy nem is részletezzük. Azt azonban kiemeljük, hogy csak az *IDiary* interface-t használja majd a kliens.

```
// 5. programlista: Az UML diagram ConcreteComponent
// szerepkörét tölti be. Ez az alaptudású napló class és
// implementálja az IDiary interface-t (ahogy a 3. ábra szerint kell)
package cs.test.dp.decorator;
import java.util.ArrayList;
import java.util.List;

public class Diary implements IDiary
{
    List<DiaryEntry> diary = new ArrayList<DiaryEntry>();

    public List<DiaryEntry> getEntries()
    {
        return diary;
    }

    public void newEntry(DiaryEntry entry)
    {
        diary.add(entry);
    }

    public DiaryEntry getEntry(int index)
    {
        return diary.get(index);
    }
}

// 6. programlista: Egy naplóbejegyzés típusa

package cs.test.dp.decorator;

public class DiaryEntry
{
    public String title;
    public String content;
    public String date;

    public DiaryEntry(String title, String content, String date)
    {
        this.title = title;
        this.content = content;
        this.date = date;
    }
}
```

Elérkeztünk oda, hogy meghatározzuk azokat az aspektusokat, amiket a naplónknak időnként tudnia kell, persze nem mindig. Vegyük őket számba, továbbra is jelöljük az egyes dekorációkat M betűvel:

- M1: Olyan napló, amit perzisztálni lehet valamilyen tárolóra
- M2: Olyan napló, ahol a műveletek loggolva lesznek

- M3: Olyan napló, ahol minden bejegyzés automatikusan kap egy időpecsétet
- M4: Olyan napló, aminek a kinézete kissé jobban formázva van

Nézzük először M1-et, amely decorator-nak most a sokkal kifejezőbb *PersistentDiaryDecorator* nevet adtuk. Ez egy jelölési konvenciót követ, aminek a szerkezete: *<az M konkrét neve><DiaryDecorator>*. A kódot a 7. programlista tartalmazza. Van néhány általános, követendő szabálya egy ilyen decorator class elkészítésének:

1. A decorator class-nak implementálnia kell a *Component* interface-t, azaz az *IDiary*-t. Ez teszi lehetővé majd, hogy a dekorált objektumra való hivatkozást tárolhassa el egy *IDiary* osztálybeli objektum.
2. Kell egy tagváltozó, ami *Component* interface szerepkörű, azaz esetünkben *IDiary* típusú. Ezen keresztül tudjuk a dekorálandó objektum metódusait elérni és „toldalék" kódot hozzáadva dekorálni.
3. A decorator class konstruktorában mindig szerepelni kell ez a *Component* interface szerepkörű paraméternek, azaz esetünkben *PersistentDiaryDecorator(IDiary diary)* a konstruktor kinézete.
4. Meg kell vizsgálni, hogy a közös *Component* interface-ben szereplő metódusok között melyek érintettek ebben az aspektusban. Az M1 aspektus perzisztenciát követel, így a listába szúrás mellett egy tárolóba is menteni kell a bejegyzésünket. Ez azt is jelenti, hogy csak a *newEntry()* metódus most az érintett. A 7. programlistából látható, hogy a perzisztálást csak jelöltük egy képernyőre írással, hiszen nem ennek implementálása volt most a célunk, hanem annak megmutatása, hogy hol kéne azt megtenni.

A *newEntry()* metóduson kívül a többi nem változik, azaz egyszerűen továbbhívja a konstruktorban kapott *IDiary* objektum megfelelő metódusait. A *newEntry()* is meghívja a *diary* objektum *newEntry()* metódusát, de annak funkcionalitását kiegészíti még a perzisztálással. Emiatt a kiegészítés miatt hívják még ezt a mintát *toldalék*nak is (vagy hozzácsatolt felelősségnek).

```
// 7. programlista: PersistentDiaryDecorator
package cs.test.dp.decorator;

import java.util.List;

public class PersistentDiaryDecorator implements IDiary
{
    IDiary diary;

    public PersistentDiaryDecorator(IDiary diary)
    {
        this.diary = diary;
    }

    public List<DiaryEntry> getEntries()
    {
        return diary.getEntries();
    }
```

```
    public void newEntry(DiaryEntry entry)
    {
        diary.newEntry(entry);
        System.out.println("The diary was saved into persistent store");
    }

    public DiaryEntry getEntry(int index)
    {
        return diary.getEntry(index);
    }
}
```

Másodjára nézzük meg az M2 aspektust is! Ez a loggolásért felel, ezért adjuk ezt a nevet neki: *LoggedDiaryDecorator*. Milyen metódusokat érint ez az aspektus? A 8. programlistán látható, hogy mindegyiket, ugyanis minden napló műveletet loggolni akarunk (ezt is csak jeleztük egy képernyőre írással).

```
// 8. programlista: LoggedDiaryDecorator

package cs.test.dp.decorator;

import java.util.List;

public class LoggedDiaryDecorator implements IDiary
{
     IDiary diary;

    public LoggedDiaryDecorator(IDiary diary)
    {
        this.diary = diary;
    }

    public List<DiaryEntry> getEntries()
    {
        System.out.println("It is logged. GET List: " );
        return diary.getEntries();
    }

    public void newEntry(DiaryEntry entry)
    {
        System.out.println("It is logged. New entry: " + entry.title);
        diary.newEntry(entry);
    }

    public DiaryEntry getEntry(int index)
    {
        System.out.println("It is logged. GET entry: " );
        return diary.getEntry(index);
    }
}
```

Az M3 az esetlegesen igényelt időpecsételésért felel, ezért a neve *TimeStampedDiaryDecorator* lesz. Látható, hogy csak a *newEntry()* metódust érinti, azaz csak ő kap toldalékot.

```
// 9. programlista: TimeStampedDiaryDecorator

package cs.test.dp.decorator;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.List;

public class TimeStampedDiaryDecorator implements IDiary
{

    IDiary diary;

    public TimeStampedDiaryDecorator(IDiary diary)
    {
        this.diary = diary;
    }

    public List<DiaryEntry> getEntries()
    {
        return diary.getEntries();
    }

    public void newEntry(DiaryEntry entry)
    {
        entry.content += "\nTime: "+getNow() + " \n" + entry.content;
        diary.newEntry(entry);
    }

    public DiaryEntry getEntry(int index)
    {
        return diary.getEntry(index);
    }

     /**
     *  This method will update the internal nowString data field
     */
    private String getNow()
    {
        Calendar calendar = Calendar.getInstance();
        java.util.Date d = null;
        d = calendar.getTime();
        SimpleDateFormat formatter = new SimpleDateFormat("yyyyMMddHHmm");
        return formatter.format(d).substring(2, 12);
    }
}
```

Az utolsó, M4 aspektus – igény esetén – egy kis formázással látja el naplónkat, így a neve a 10. programlista alapján *FormattedDiaryDecorator* lett. A „toldalékolásban” érintett metódusok nyilván a *getEntries()* és *getEntry()* lekérő metódusok.

```
// 10. programlista: FormattedDiaryDecorator

package cs.test.dp.decorator;

import java.util.List;

public class FormattedDiaryDecorator implements IDiary
{

    IDiary diary;

    public FormattedDiaryDecorator(IDiary diary)
    {
        this.diary = diary;
    }

    public List<DiaryEntry> getEntries()
    {
        System.out.println("Format the whole diary");
        return diary.getEntries();
    }

    public void newEntry(DiaryEntry entry)
    {
        diary.newEntry(entry);
    }

    public DiaryEntry getEntry(int index)
    {
        System.out.println("Format the requested Entry");
        return diary.getEntry(index);
    }
}
```

Elkészítettük az összes előre elképzelt decorator class-t. Látható a filozófia, ahogy a beágyazott *IDiary* típusú *diary* objektumot – amit a konstruktor vesz át nekünk – hogyan hívjuk vissza, majd igény esetén toldalékolunk.

Próbáljuk ki az eddigi eredményeinket, amit a 11. programlista mutat. A lényeget a 9. sorban láthatjuk, ahogy felépítjük a nekünk szükséges *IDiary* objektumot, ami így most a loggolást és időpecsételést is tudni fogja.

```
1  // 11. programlista: Tesztprogram
2
3  package cs.test.dp.decorator;
4
5  public class TestDecorator
```

```
6  {
7      public static void main(String[] args)
8      {
9          IDiary d1 = new TimeStampedDiaryDecorator(new LoggedDiaryDecorator( new Diary() ));
10
11         d1.newEntry( new DiaryEntry("Első entry", "b1111labla", "2010.01.10") );
12         d1.newEntry( new DiaryEntry("2. entry", "b222labla", "2010.01.10") );
13         d1.newEntry( new DiaryEntry("3. entry", "b3333labla", "2010.01.10") );
14
15         System.out.println( d1.getEntry(2).content );
16     }
17 }
```

Második példa: A Decorator, mint Filter

A decorator minta lehetőséget ad az adat vagy csatornafilterezés (szűrők) megvalósítására. A Java nyelv *java.io* könyvtára is ezt a mintát alkalmazza. A következőkben mindezt egy példán érdemes megérteni, ami természetesen a decorator mintát követi, azonban a szűrési funkció kihangsúlyozása miatt ilyenkor a névkonvenció szerint a *Decorator* szót mindenütt a *Filter* szó szokta helyettesíteni.

A feladat a következő: Van egy olyan csatornánk, amin *String*ek sorozata jön és az egyetlen csatorna-művelet a *getLine()*, azaz a következő *String* megszerzése. A „Component” szerepkörű interface most az *IFuzer* lesz, amit a 12. programlista mutat.

```
1  // 12. programlista: A közös komponens interface
2
3  package cs.test.dp.decorator2;
4
5  public interface IFuzer
6  {
7      public String getLine();
8  }
```

Az egyes aspektusok – amiket itt inkább szűrőknek nevezünk – legyenek a következőek:

- M1: *ReplaceFuzerFilter* – a bejövő sorok bizonyos karaktereit más karakterekre cseréli
- M2: *MirrorFuzerFilter* – a bejövő Stringeket visszafelé olvasva adja vissza
- M3: *ConcatFuzerFilter* – a következő n db sort összekapcsolja és azt adja vissza
- M4: *CharByCharFuzerFilter* – a csatornát 1 karakteres String-enként olvassa fel
- M5: *CounterFuzerFilter* – a csatorna következő sora helyett annak hosszát adja vissza, mint String
- M6: *FlushFuzerFilter* – a teljes csatorna tartalmat 1 db String-ként adja vissza

Egy lehetséges „*ConcreteComponent*” szerepkörben lévő osztály példánkban a *Fuzer* class (13. programlista). Itt a csatornát belül egy *String* tömb reprezentálja, de természetesen ez lehetne egy text file vagy bármi más is (például HTTP response). A *getLine()* felolvassa az összes *String*et (minden hívásnál 1 darabot), amennyiben már üres lett a csatorna, úgy *null*-t ad vissza.

```
1  // 13. programlista: Fuzer
2
3  package cs.test.dp.decorator2;
4
5  public class Fuzer implements IFuzer
6  {
7      String[] lines;
8      int index = 0;
9
10     public Fuzer(String[] lines)
11     {
12         this.lines = lines;
13     }
14
15     public String getLine()
16     {
17         if ( index < lines.length )
18         {
19             return lines[index++];
20         }
21         else
22         {
23             return null;
24         }
25     }
26 }
```

Tekintsük át a filterek implementációját egyenként. Könnyű dolgunk van annak a megítélésében, hogy az egyes aspektusok mely metódusokat érintik, hiszen csak a *getLine()* az egyetlen, azt pedig kötelezően érinti, hiszen ha nem így lenne, akkor a Filter sem csinálna semmi újat.

A 14. programlista mutatja a *ReplaceFuzerFilter*-t. Az itteni *getLine()* annyit tesz hozzá (azaz annyit toldalékol vagy díszít) az eredetihez, hogy lekérve az eredetit, mindig átereszti azt egy replace filteren, amit a visszaadott *getLine.replaceAll(mit, mire)* sor mutat. A konstruktorban azt is láthatjuk, hogy nem kell mindig csak a kötelezően megadandó közös interface-t – azaz itt az *IFuzer*-t – átvenni, lehet több paraméter is. Itt például értelemszerűen a *mit* és *mire* paraméterek ilyenek.

```
1  // 14. programlista: ReplaceFuzerFilter
2
3  package cs.test.dp.decorator2;
4
5  public class ReplaceFuzerFilter implements IFuzer
6  {
7      IFuzer line;
8      String mit, mire;
9
10     public ReplaceFuzerFilter(IFuzer line, String mit, String mire)
11     {
12         this.line = line;
```

```
13          this.mit  = mit;
14          this.mire = mire;
15      }
16
17      @Override
18      public String getLine()
19      {
20          String gotLine = line.getLine();
21          if ( gotLine == null ) return null;
22          return gotLine.replaceAll(mit, mire);
23      }
24  }
```

Az M2 Filter a tükrözés, azaz a kapott sort visszafelé olvasva adjuk vissza, ahogy ezt a 15. programlista is mutatja. Itt semmilyen működést definiáló további paraméter nem szükséges, a feladat egyértelmű, ezért a konstruktor is csak az *IFuzer* paramétert kapja meg. A *toMirror()* private metódus végzi a tükör sztring előállítást, amit az itteni *getLine()* a „toldalék” részben fog használni.

```
1   // 15. programlista: A tükrözés
2
3   package cs.test.dp.decorator2;
4
5   public class MirrorFuzerFilter implements IFuzer
6   {
7       IFuzer line;
8
9       public MirrorFuzerFilter(IFuzer line)
10      {
11          this.line = line;
12      }
13
14      @Override
15      public String getLine()
16      {
17          String gotLine = line.getLine();
18          if ( gotLine == null ) return null;
19          return toMirror(gotLine);
20      }
21
22      private String toMirror(String s)
23      {
24          StringBuffer sb = new StringBuffer("");
25          for (int i=s.length()-1; i>=0; i--)
26          {
27              sb.append( s.charAt(i)  );
28          }
29          return sb.toString();
30      }
31  }
```

A következő, M3 aspektusú filter kicsit más, mint az előző M1 és M2, ugyanis ez már nem csak az aktuális, örökölt soron végez valamilyen transzformációt, hanem a paraméterként megadott következő sorokon (a számát a *koteg* paraméter adja meg) dolgozva, „összeragasztja” azokat. A 16. programlista egy olyan filtert mutat, aminek a konstruktorában azt is megadtuk, hogy hány sort ragasszon össze az ő *getLine()* metódusa. Persze elképzelhető, hogy kevesebbet fog konkatenálni, amennyiben a csatorna már nem rendelkezik *koteg* számú sorral.

```
1  // 16. programlista: Több sort visszadó filter
2
3  package cs.test.dp.decorator2;
4
5  public class ConcatFuzerFilter implements IFuzer
6  {
7      IFuzer line;
8      int koteg;
9
10     public ConcatFuzerFilter(IFuzer line, int koteg)
11     {
12         this.line  = line;
13         this.koteg = koteg;
14     }
15
16     @Override
17     public String getLine()
18     {
19         StringBuffer sb = new StringBuffer("");
20         String gotLine = line.getLine();
21         int cnt = 1;
22         if (gotLine == null) return null;
23         while ( gotLine != null )
24         {
25             sb.append( gotLine );
26             if (   cnt == koteg ) return sb.toString();
27             gotLine = line.getLine();
28             cnt++;
29         }
30         return sb.toString();
31     }
32
33
34 }
```

A 17. programlista által mutatott filter karakterekre szedi szét a teljes csatorna tartalmat. A működése egyszerű, csak akkor hívja meg az örökölt *getLine()*-t, ha elfogyott az aktuális sor összes karaktere. Amennyiben az örökölt sorolvasó *null*-t ad vissza, úgy ezen filter sorolvasója is ezt továbbítja a hívónak, jelezve, hogy elfogytak a karakterek.

```
1  // 17. programlista: 1 sor = 1 karakter
2
3  package cs.test.dp.decorator2;
4
5  public class CharByCharFuzerFilter implements IFuzer
6  {
7      IFuzer line;
8      String gotLine;
9      int    index = 0;
10
11     public CharByCharFuzerFilter(IFuzer line)
12     {
13         this.line = line;
14         gotLine = line.getLine();
15     }
16
17     @Override
18     public String getLine()
19     {
20         if ( gotLine == null ) return null;
21         if ( gotLine.length() > index )
22         {
23             return "" + gotLine.charAt(index++);
24         }
25         else
26         {
27             gotLine = line.getLine();
28             if ( gotLine == null ) return null;
29             index = 0;
30             return "" + gotLine.charAt(index++);
31         }
32     }
33 }
```

A 18. programlista talán a legegyszerűbb szűrők egyike. A *getLine()* az aktuális sor karakterekben mért méretét adja vissza.

```
// 18. programlista: egy sor = a hossza karakterekben

package cs.test.dp.decorator2;

public class CounterFuzerFilter implements IFuzer
{
    IFuzer line;

    public CounterFuzerFilter(IFuzer line)
    {
        this.line = line;
```

```
    }

    @Override
    public String getLine()
    {
        String gotLine = line.getLine();
        if ( gotLine == null ) return null;
        return "" + gotLine.length();
    }
}
```

A 19. programlista szűrője az input sorok csatornájáról felolvassa az összes sort és a *getLine()* egyetlen sorként adja vissza. Ez azt is jelenti, hogy itt a *getLine()* a 2. hívástól már mindig a null értéket fogja visszaadni.

```
1  // 19. programlista: Egyszere visszaadja a teljes csatornát
2
3  package cs.test.dp.decorator2;
4
5  public class FlushFuzerFilter implements IFuzer
6  {
7      IFuzer line;
8      boolean endOfChannel = false;
9
10     public FlushFuzerFilter(IFuzer line)
11     {
12         this.line = line;
13     }
14
15     @Override
16     public String getLine()
17     {
18         if ( endOfChannel ) return null;
19         StringBuffer sb = new StringBuffer("");
20         String gotLine = line.getLine();
21         while ( gotLine != null )
22         {
23             sb.append( gotLine );
24             gotLine = line.getLine();
25         }
26         endOfChannel = true;
27         return sb.toString();
28     }
29
30
31 }
```

Az eddigiekben létrehoztunk több szűrőt, hogy érzékeltessük a lehetőségeket és azok implementációjában jártasabbak legyünk. Most befejezzük ezt a tevékenységet és megnézzük egy teszt program segítségével a szűrőink használatát. Megjegyezzük, hogy a szűrők számának csak a képzelet és a racionalitás szab határt. Létrehozhattuk volna például az *UppercaseFuzerFilter*, *HuEnFuzerFilter*, stb... filtereket, amennyiben a feladatban fontos a nagybetűs kezelés vagy a magyarról angolra fordítás. A tesztprogramot, azaz a Filterek használatát a 20. programlista mutatja. Fontos itt is észrevenni, hogy az *f* változó *IFuzer* típusra van felvéve, hiszen ez az egyik lényege a tervezési mintának. Az aktuális változat futási eredményét is láthatjuk. A *CounterFuzerFilter* mindegyik sort annak hosszával helyettesít, ami ezt eredményezné, ha csak ő működne: *865544*. Mindegyik számjegy egy-egy *getLine()* hívás eredménye. Itt azonban van egy másik szűrő is, a *ConcatFuzerFilter*. Ez kettesével kötegeli a sorokat, így kapjuk meg a látható futási eredményt.

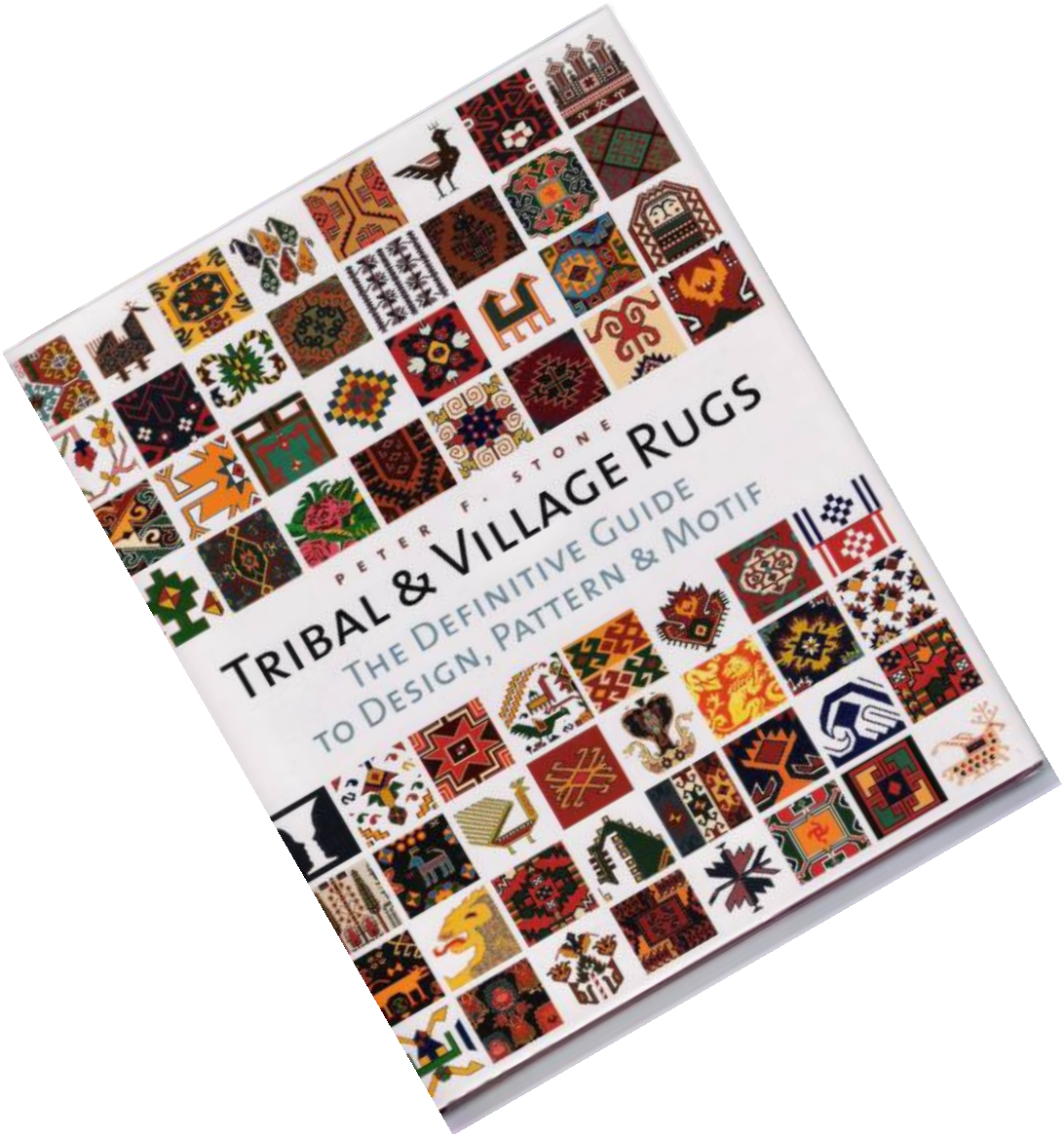
```
1  // 20. programlista: A Filter-ek tesztelése
2
3  package cs.test.dp.decorator2;
4
5  public class Test
6  {
7
8      public static void main(String[] args)
9      {
10         String[] s = {"abcdefgh", "ikllmn", "pqrst", "12345", "6789", "0000"};
11
12         //IFuzer f = new MirrorFuzerFilter(
13         new ReplaceFuzerFilter(new Fuzer(s), "alma", "körte"));
14         //IFuzer f = new MirrorFuzerFilter(new Fuzer(s));
15         IFuzer f = new ConcatFuzerFilter(new CounterFuzerFilter( new Fuzer(s)), 2);
16
17
18         String line;
19         while ( (line = f.getLine()) != null  )
20         {
21             System.out.println( line );
22         }
23
24     }
25 }
26
27 Futási eredmény:
28 86
29 55
30 44
```

A példánkban lévő *Fuzer* class (ami a *ConcreteComponent* szerepkörben van) egy *String* tömbből veszi a sorokat. Persze készíthetnénk egy olyan osztályt is, ami textfile-ból veszi. Lehetne ennek a neve például: *TextFileFuzer*, ami persze implementálja az *IFuzer* interface-t, a

konstruktora paraméterként kapná a textfile nevét. Ennek a design patterneknek a nagy előnye, hogy filtereink minden további nélkül itt is használhatóak maradnak.

Összefoglalás

Áttekintettük a decorator mintát. Befejezésül szeretnénk kiemelni, hogy a gyakorlatban sok helyen lehet ezeket használni. A Java környezetben főleg az *java.io* csomag csatorna megvalósításaiban vagy a GUI framework-ök környékén használatosak. Példa: *OutputStream out = new DataOutputStream (new FileOutputStream ("test.txt"));*



3. A Java biztonsági rendszere - Authentikáció, Authorizáció

Indítunk egy cikksorozatot, ami a Java biztonsági kérdéseivel és megoldásaival foglalkozik. Az első rész témája annak a vizsgálata, hogy ez a technológiai környezet milyen módon oldja meg a kliensek azonosítását és jogosultság kezelését. Elöljáróban szeretnénk kiemelni, hogy a Java is a jól ismert PAM (Pluggable Authentication Module) rendszert alkalmazza – nem véletlenül. Egyrészt ezt a megoldást a SUN fejlesztette ki a Solaris operációs rendszerhez, másrészt a Java is a SUN gyermeke. Megjegyezzük, hogy a PAM rendszer meglehetősen elterjedt, például a Linux disztribúciók – a szabványosság miatt – is ezt használják.

Miről fog szólni a cikksorozat?

A több részből álló cikksorozatunk egy fontos témát szeretne górcső alá venni: a biztonságot. Segítségül a Java környezetet hívjuk, így gyakorlatilag a Java biztonsági megoldások áttekintéséről írunk. A mai korszerű szoftverek nem elszigetelten működnek egymástól, sőt inkább együttműködnek. Egy elosztott rendszerre maga az Internet a legjobb példa, de a modern többrétegű alkalmazások is hasonló modellt követnek. Milyen feladatai vannak a biztonságnak? Vegyünk példaképpen egy képzeletbeli rendszert: repülőjegy foglalás az Interneten. Nézzük meg milyen biztonsági problémák adódhatnak, azaz miket kell megoldanunk:

1. **Utólagosan meghamisított tranzakció (azaz jegyfoglalás és fizetés) előállításának veszélye.** Ez érintheti a jegyvásárlót és a jegyszolgáltatót is, ugyanis mindkét fél rosszindulata lehetséges (például más típusú jegyre vagy dátumra hamisítás). Megoldás: a tranzakció digitális aláírása és időpecsételése.

2. **Más nevében feladott tranzakciók**, azaz példánkban a nevünkben rendel valaki repülőjegyet. Megoldás: a kölcsönös hitelesítés (authentication).

3. **Tranzakciók lehallgatása, módosítása, törlése.** Például valaki drágább jegyre módosítja vagy törli menet közben az igényünket. Megoldás: a tranzakció rejtjelezése.

4. **A szolgáltató letegad egy tranzakciót**, azaz esetünkben úgy hisszük, hogy lesz repülőjegyünk, de utólag ezt az eladó letagadja. Megoldás: digitális, aláírt és időpecséttel ellátott nyugta. Ez azt jelenti, hogy a tranzakcióról a szolgáltató egy nyugtát (visszaigazolást) készít, amit aláírásával és időpecsétjével hitelesít.

5. **Az elindított hálózati program nem az eredeti.** Megoldás: a program eredetének vizsgálata úgy, hogy azt a gyártó digitálisan aláírja, a kliens pedig tárolja, hogy mely aláírókban bízik meg.

6. A program különféle funkcióit, illetve az amögött lévő erőforrásokat (a programunk egy szolgáltatása, adatbázis, hálózati szolgáltatás, nyomtató, képernyő, stb...) eltérő jogosultsággal használhatják a klienseink. Megoldás: jogosultság kezelés és access management használata (authorizáció).

A cikksorozatban a fenti szempontokat fogjuk végigtekinteni. Az első cikkben a Java JAAS rendszer alapjait nézzük meg.

3.1. Alapfogalmak

Mielőtt elmerülnénk a Java alapú hitelesítés világában, fontos megértenünk néhány alapfogalmat.

1. Meghatározás (Authentikáció). *Az authentikáció (más szóval: hitelesítés) egy állítólagos személyazonosság ellenőrzésének folyamata, mely során egy publikus (példa: username) és egy privát (példa: password) információdarab alapján dől el az érvényesség (azonosság) úgy, hogy a hitelesítő rendszer ezeket összehasonlítja a saját adatbázisában tárolt információval.*

A privát és publikus információkat hitelesítési adatoknak nevezzük, amelyek a gyakorlatban 3 fő formában szoktak megjelenni: egy password, egy birtokláson alapuló eszköz (hardverkulcs) vagy egy biometria azonosítás (ujjlenyomat, retina, ...).

A Java PAM megvalósítás a JAAS (Java Authentication and Authorization Service) része, ami a Java 1.4 óta kötelező része az SDK-nak. A központi eleme a *LoginModule* interface, ami egy authentication képes objektum felülete.

Függetlenül attól, hogy egy program használója természetes személy vagy egy másik program, az mindig egy *entity* (entitás)[1]nevében fut, amely rendelkezik valamilyen felismerhető azonosítóval (pontosabban: azonosító halmazzal), azaz identitása (*identity*=azonosság) van. Ez az *entity* általában maga a felhasználó, egy másik program, vagy egy szervezet, csoport lehet.

2. Meghatározás (Subject). *A Subject (alany, a hitelesítés célszemélye) az Entity (entitás) objektum reprezentálása a programban.*

Általában 1 *entity* több *identity*-vel rendelkezhet, amelyekre példák: azonosítója, nick neve, e-mail címe, csoportjainak a nevei, személyi kódjai, stb.

3. Meghatározás (Principal). *A principal az identity fogalom programbeli objektum-reprezentációja. Egy Subject-hez természetesen több fajta Principal tartozhat.*

A *Principal* interface – ahogy nemsokára látni is fogjuk – meglehetősen egyszerű, hiszen feladata leginkább csak az, hogy a különféle identity típusok neveit, értékeit tárolja (példa: nick: morci, törzsszám: 2345634, e-mail: morci@gmail.com, ...).

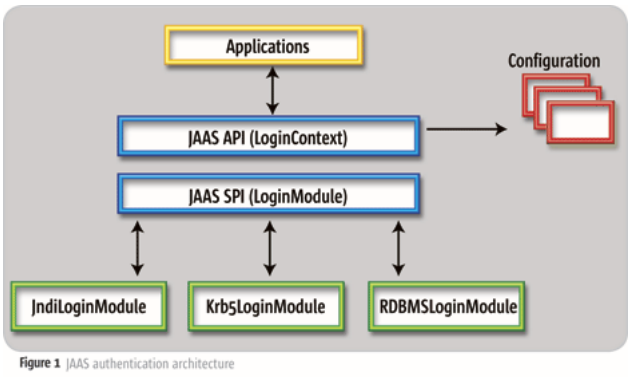
4. Meghatározás (Credential). *Credential (rokonértelmű szavak: tanúsítvány, igazolvány, igazoló valami). A való világban ilyen a személyi igazolvány, útlevél, stb. Ezek hosszabb idejűek, de lehet például a megszerzett színházjegy, tömegközlekedési jegy. A Credential mindig valamit bizonyít. Lehet ujjlenyomat, kép (biometrikus információk). Sokszor jelszó, vagy más hitelesítési eszköz vagy megbízólevél. A hitelesítést adó Credential tehát egy adategyüttes, aminek általában publikus és private (csak a hitelesítést kérő alany ismeri)része is van. Például egy user/password alapú hitelesítésnél a username a publikus, a password pedig a privát része a Credential-nak.*

Authentication alatt tehát egy *entity* azonosítását értjük, ami a gyakorlatban az identitásainak meghatározását jelenti. Ez más szavakkal azt jelenti, hogy az *entity* (*Subject*, alany) a sikeres azonosítás után (amihez *Credential*-t használ) összekapcsolódik az általunk ismert összes *identity* azonosítókkal (*Principálokkal*, identitásokkal), azaz olyan nevekkel, ami az alanyra jellemző és így tőle bármikor lekérdezhető.

[1]Valaminek a létszerűsége, ahol a hangsúly azon van, hogy létezik valami, és nem azon, hogy mi létezik.

3.2. A kliens hogyan használ egy JAAS LoginModule-t?

Ebben a pontban röviden tisztázzuk, hogy miképpen tudunk használni egy más által már elkészített *LoginModule*-t. Mit jelent egyáltalán az, hogy bejelentkeztünk? A JAAS szoftveres felépítését a 5. ábra mutatja.



5. ábra. A JAAS architecture

Az ábráról leolvasható, hogy alkalmazásaink a *LoginContext* osztály segítségével tartják a kapcsolatot a JAAS hitelesítő, elérés és jogosultságkezelő alrendszerrel. A *LoginModule* class pedig a hitelesítést adó háttér modulokkal működik együtt. Ez azt is jelenti, hogy a *LoginModule* egy infrastruktúra szoftver komponens, azzal emiatt nem is kerül vele kapcsolatba az a szakember, aki az alkalmazást készíti. Persze az is lehet, hogy szükséges egy új hitelesítő modulet is készíteni, de akkor azt is csak a *LoginContext* class-on keresztül fogjuk használni. A java az 1.4 verzió óta több előre elkészített, beépített *LoginModule*-t is tartalmaz:

- *JndiLoginModule*: ezen keresztül user adatbázisként egy tetszőleges JNDI (Java Naming and Directory Interface) kompatibilis címtár használható
- *Krb5LoginModule*: Az ismert Kerberos protocol alapján működik
- *NTLoginModule*: Az operációs rendszertől veszi át az azonosított user-t
- *UnixLoginModule*: Az operációs rendszertől veszi át az azonosított user-t

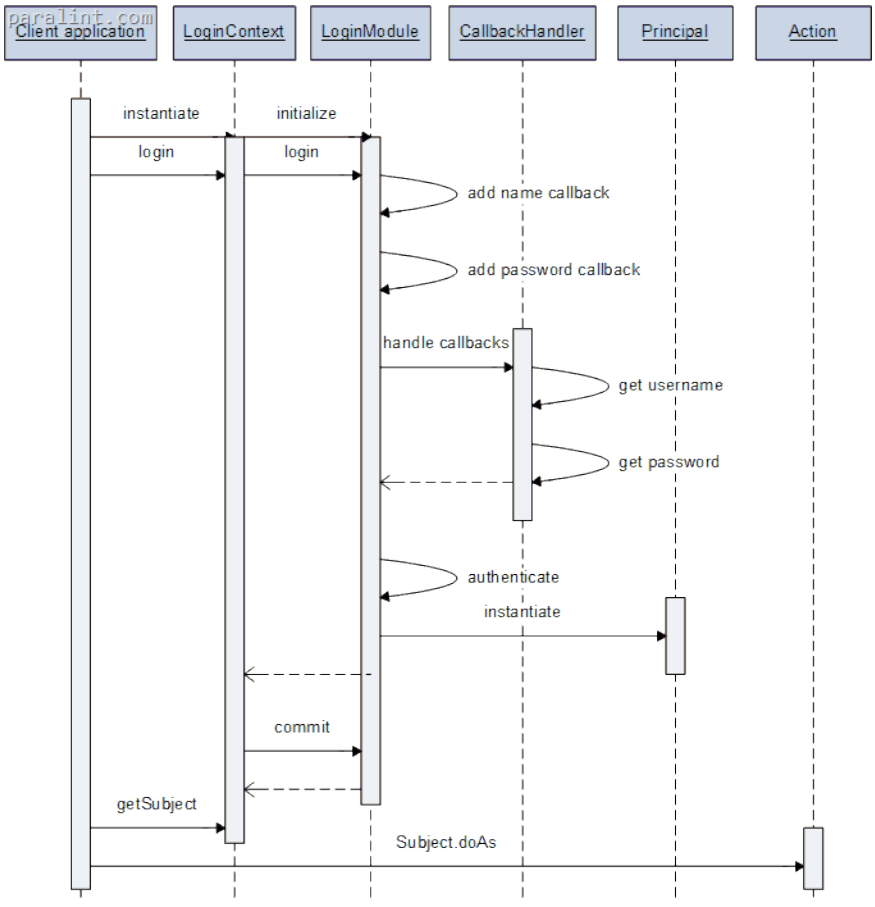
Az utolsó 3 LoginModule használatát ezen cikk későbbi részében a gyakorlatban is be fogjuk mutatni.

Amikor egy *LoginModule*-t használni szeretnék, akkor a java virtuális gépnek ezt meg kell mondanunk néhány egyszerű konfigurációs lépésben, amit szintén a 3.2 pontban mutatunk meg. Most fogadjuk el, hogy bekonfiguráltunk egyet és szeretnénk használni. Ennek módját a 21. programlista mutatja.

```
1  // 21. programlista: Bejelentkezés egy LoginModule segítségével
2
3  LoginContext lc = new LoginContext("MyExample");
4  try
5  {
6      lc.login();
7  } catch (LoginException)
8  {
9      // Authentication failed.
10 }
11 // Authentication successful, we can now continue.
12 // We can use the returned Subject if we like.
13 Subject sub = lc.getSubject();
14 Subject.doAs(sub, new MyPrivilegedAction());
```

A 3. sorban megszerezzük a *LoginContext* objektumot, amit a *MyExample* paraméterrel gyártunk le. Ez a *LoginModule* konfiguráció neve is egyben, így itt véglegesül, hogy melyik PAM modul szerint akarjuk a hitelesítést elvégezni. Ezután a 6. sorban tehetünk egy próbát a bejelentkezésre. Amennyiben ez sikerült és nem dobódik kivétel, úgy a hitelesítés és egyben a bejelentkezésünk is sikerült. A 13. sorban már megtehetjük, hogy lekérdezzük a *Subject*-et, azaz a bejelentkezett entity-t (egyedet, ami egy személy vagy egy másik program), amelyen keresztül az alkalmazásunk számára egyértelművé válik, hogy éppen ki használja ezen a kapcsolaton keresztül az alkalmazást. Emiatt a 14. sorban látható authorizációs műveletet is képessek vagyunk immár elvégezni. A folyamatot az 6. ábrán lévő szekvencia-diagram érzékelteti, érdemes áttanulmányozni.

Összefoglalásként megállapíthatjuk, hogy akkor vagyunk bejelentkezve, ha a futó programunk tartalmaz egy *Subject* objectumot, amelyen keresztül lekérhetjük a fontosabb tulajdonságainkat, azaz a principáljainkat (például mely nevű csoportok tagjai vagyunk, milyen a szerepkörünk vagy éppen mi a lakcímünk, stb...).



6. ábra. A JAAS alapú hitelesítés folyamata

3.3. Hogyan készítsünk el egy saját JAAS LoginModule-t?

A hitelesítési folyamat használatának megértését mélyítsük úgy el, hogy készítünk egy saját *LoginModule*-t! A neve legyen *TestLoginModule* és a hitelesítésnél használt credential most egyedül csak egy titkos jelszó lesz. Emlékezzünk rá, hogy a hitelesítéshez használt credential-ok többen lehetnek, egy részük publikus, más részük privát, azaz titkos.

A JAAS környezet az előzőekben már definiált 2 fontos fogalomhoz (*Principal* és *Subject*) reprezentációt is készített. Nézzük először a *Principal*-t, ami triviális módon egy interface, ugyanis bármelyik objektum lehet ebben a szerepben, amely identity-t tud adni magáról. A beépített Java *Principal* interface-t a 22. programlista mutatja, azaz látható, hogy ez lényegében tényleg csak egy névvel rendelkező objektum interface-e, ahol az összehasonlítás és hash táblában való tárolhatóság a követelmény. Bármi lehet principál, ami ezt tudja, ami egy nevet tud tárolni a *Subject*-ről.

```
1  // 22. programlista: A Principal interface
2
3  package java.security;
4
5  public interface Principal
6  {
7      public boolean equals(Object another);
8      public String toString();
9      public int hashCode();
10     public String getName();
11 }
```

A *Principal* interface csak a legalapvetőbb dolgot követeli meg az őt implementáló objektumtól. Legfontosabb a *getName()* metódus, hiszen ez adja vissza az azonosság konkrét értékét, mint karaktersorozatot. A *Principal* gyakorlatilag egy azonosító konkrét értékének eltárolására képes osztály. Egy új principal class készítésénél javasolt egy elnevezési konvenció, ami ilyen formátumú: *<TTT>Principal*, ahol a *TTT* a principal típusa. Például egy e-mail jellegű principál javasolt class neve: *EMailPrincipal*.

A Java-ban az entity-t a *final Subject* osztály valósítja meg, azaz a programozó ezen már nem változtathat. A metódusai közül néhány a principál-okhoz (*getPrincipals()*) és credential-okhoz való hozzájutást támogatja. A másik nagy metóduscsoportja az authorizációt segíti (*doAs()* metódusok), de erre csak később térünk ki.

Tekintettel arra, hogy egy saját, új *LoginModule*-t szeretnénk készíteni, ezért nézzük meg ennek a „gyári” interface-ét is, amit a 23. programlista mutat!

```
// 23. programlista: LoginModule interface

package javax.security.auth.spi;

import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

public interface LoginModule
{
  // Megszakítja az azonosítási folyamatot
  boolean abort() throws LoginException;
  // Jóváhagyja az azonosítási folyamatot
  boolean commit() throws LoginException;
  // Inicializálja a LoginModule-t
  void initialize(Subject subject, CallbackHandler handler,
                  Map<String, ?> sharedState, Map<String, ?> options);
  // Ez indítja el a bejelentkezési folyamatot
  boolean login() throws LoginException;
  // Kijelentkezés
  boolean logout() throws LoginException;
}
```

Az egyes metódusok szerepét rövid megjegyzésekben adtuk meg a 23. programlistán. Egy PAM azonosító modul olyan objektum, ami implementálja ezt a *LoginModule* interface-t.

Egy *LoginModule* konfigurációját egy olyan külső config file-ba lehet elhelyezni, aminek a pontos elérhetőségét a *java.security.auth.login.config* Java környezeti változó mondja meg. Ez azért ilyen közvetett, mert a *LoginModule*-t (PAM modult) sohasem használjuk közvetlenül, helyette a programjainban a *LoginContext* osztály objektumaival dolgozunk, amit a 21. programlistán láthattunk, de ebben a példában a későbbiekben is bemutatunk.

Most „pihenésképpen” készítsünk el egy Principal-t, aminek a neve legyen *TestPrincipal*! Minden komolyabb filozófiát mellőzve, ez az „azonosság tároló” csak visszaadja a nevet, amivel létrehoztuk. A megvalósítása egyszerű, ahogy a 24. programlistán is látható. Az osztály alig tud annál többet, mint a konstruktorban kapott és eltárol név visszaadása a *getName()* metódussal.

```
1 // 24. programlista: Egy teszt Principal
2
3 package test.jaas;
4
5 import java.security.Principal;
```

```
6
7  public class TestPrincipal implements Principal
8  {
9      private final String name;
10
11     public TestPrincipal(String name)
12     {
13         if (name == null)
14         {
15             throw new IllegalArgumentException("Null name");
16         }
17         this.name = name;
18     }
19
20     public String getName()
21     {
22         return name;
23     }
24
25     public String toString()
26     {
27         return "Az identitásom : : " + name;
28     }
29
30     public boolean equals(Object obj)
31     {
32         if (obj == null)
33         {
34             return false;
35         }
36         if (obj == this)
37         {
38             return true;
39         }
40         if (!(obj instanceof TestPrincipal))
41         {
42             return false;
43         }
44         TestPrincipal another = (TestPrincipal)obj;
45         return name.equals(another.getName());
46     }
47
48     public int hasCode()
```

```
49      {
50          return name.hashCode();
51      }
52  }
```

Az azonosítási folyamat egy párbeszéd az azonosítást végző és az azonosítást kérő között. Ezt szokták „chat scenario”-nak (chat script) is nevezni és ahhoz hasonlít, ahogy az igazoltató rendőr (esetünkben a *LoginModule*) kérdezget és várja rá a válaszokat. Ezeknek a válaszoknak az absztrakcióját valósítja meg a *CallbackHandler* interface (25. programlista), aminek egyetlen *handle()* metódusa a kérdésre adott válaszreakció. Például a „kérem a jelszót” kérdésre, a jelszó megszerzése valahonnan (például a konzolról). Az 25. programlistán látható az „üres” *Callback* interface is, aminek csak az a feladata, hogy a meghívandó visszahívások tömbjét a *handle()* metódus átvehesse. A *Callback*-ra ezek szerint semmilyen megkötés nincs, majd látni fogjuk a példa *CallbackHandler*-nél, hogy mindegyik callback-en végig kell menni és azokat minden ágon egyedileg kezelni. A példánkban csak a *NameCallback* class szerepel a *for* ciklusban, így a „minden” esetünkben most csak egy 1 elemű callback tömböt jelent.

```
1  // 25. programlista: A CallbackHandler interface
2
3  package javax.security.auth.callback;
4
5  public interface CallbackHandler
6  {
7    void handle(Callback[] callbacks)
8      throws java.io.IOException, UnsupportedCallbackException;
9  }
10
11 package javax.security.auth.callback;
12 public interface Callback { }
```

Írjunk egy *TestCallbackHandler* nevű osztályt és remélhetőleg minden megvilágosodik! A forráskódját a 26. programlista mutatja (és a hozzá használt *NameCallback* class-t, amit a 27. programlista tartalmaz). A *NameCallback* egyébként csak egy egyszerű 3 mezős tároló osztály, ahol a mezők szerepét a kommentekbe írtuk be. A *TestCallbackHandler.handle()* esetünkben annyit csinál, hogy az átvett *NameCallback* paraméter alapján kiteszi a konzolra a prompt-ot és vár egy karaktersorozat begépelésére, amely értéket utána elmenti a *NameCallback.inputName* mezőbe (a *setName()* metódussal).

```
1  // 26. programlista: Egy saját CallbackHandler
2
3  package test.jaas;
4
5  import java.io.*;
6  import javax.security.auth.*;
7  import javax.security.auth.callback.*;
8
9  public class TestCallbackHandler implements CallbackHandler
10 {
```

```
11
12      public TestCallbackHandler()
13      {
14      }
15
16      public void handle(Callback callbacks[]) throws IOException, UnsupportedCallbackException
17      {
18          for (int i = 0; i < callbacks.length; i++)
19          {
20              if (callbacks[i] instanceof NameCallback)
21              {
22                  NameCallback nc = (NameCallback) callbacks[0];
23                  System.err.print(nc.getPrompt());
24                  System.err.flush();
25                  String name = (new BufferedReader(new InputStreamReader(System.in))).readLine();
26                  nc.setName(name);
27              } else
28              {
29                  throw (new UnsupportedCallbackException(callbacks[i], "Callback handler not support"));
30              }
31          }
32      }
33  }
```

```
1   // 27. programlista: NameCallback
2
3   package javax.security.auth.callback;
4
5
6   public class NameCallback implements Callback, java.io.Serializable {
7
8       private static final long serialVersionUID = 3770938795909392253L;
9
10      private String prompt; // kiírja a konzolra ezt a promptot
11      private String defaultName; // alapérték
12      private String inputName; // ide kerül a megadott név
13
14      public NameCallback(String prompt) {
15          if (prompt == null || prompt.length() == 0)
16              throw new IllegalArgumentException();
17          this.prompt = prompt;
18      }
19
20      public NameCallback(String prompt, String defaultName) {
21          if (prompt == null || prompt.length() == 0 ||
22              defaultName == null || defaultName.length() == 0)
23              throw new IllegalArgumentException();
24
25          this.prompt = prompt;
26          this.defaultName = defaultName;
27      }
28
29      public String getPrompt() {
30          return prompt;
31      }
32
33      public String getDefaultName() {
34          return defaultName;
35      }
36
37      public void setName(String name) {
38          this.inputName = name;
39      }
40
41      public String getName() {
```

```
42          return inputName;
43      }
44  }
```

Ennyi előzmény és tudás birtokában végre megírhatjuk az első JAAS *LoginModule*-unkat (28. programlista), aminek adjuk a *TestLoginModule* class nevet! A feladat az, hogy implementáljuk az összes metódust, amit a *LoginModule* interface megkövetel. Nézzük át együtt ezt az osztályt, hiszen ennek a megértése a legfontosabb! A 27-34 sorok *initialize()* metódusa a *LoginModule* objektum inicializálását végzi el. Itt még nem ismert az *identity*, így azt *null*-ra állítja, ezzel együtt a *status* is *NOT* lesz. Beállításra kerül a használt callback handler (azaz a kérdez-felelek játék lebonyolításának módja), valamint a *LoginModule*-t hívó (ez *LoginContext* objektum) program *Subject* objektumának a referenciája.

A *login()* metódus megvalósítása a 36-66 sorok között található. A 44. sorban beállítjuk az egyetlen callback-ünket, amit a 48-49 sorokban használunk. Mivel itt *NameCallback* használatos, ezért tőle válaszként egy *String*-et kapunk, amit a *name* változóban tárolunk el. Ez a név egy titkos karaktersorozat, ami private credential szerepkörben van és a billentyűzetről gépeljük be. A 57-60 sorok a beléptetést valósítják meg. Nincs user adatbázisunk, így az egyetlen érvényes titkos jelszót az 57. sor feltételébe „égettük" be. Amennyiben a user ezt, azaz a *123456* szót adta meg, úgy az 59. sorban beállítjuk az egyszem (mert lehetne sokkal több is!) principálunkat, valamint a *status*-t OK-ra billentjük.

A *commit()* és *abort()* megvalósítása szinte mindegyik *LoginModule* típusban ugyanaz. A 103-109 sorokban lévő *logout()* hasonlít az inicializálásra, nem véletlenül, hiszen a célja a *login()* előtti állapot visszaállítása, azaz a kiléptetés.

```
1   // 28. programlista: A saját TestLoginModule
2
3   package test.jaas;
4
5   import java.io.*;
6   import java.util.*;
7   import java.security.Principal;
8   import javax.security.auth.Subject;
9   import javax.security.auth.callback.*;
10  import javax.security.auth.spi.LoginModule;
11  import javax.security.auth.login.LoginException;
12
13  public class TestLoginModule implements LoginModule
14  {
15
16      public TestLoginModule()
17      {
18      }
19      private Subject subject;
20      private TestPrincipal identity;
21      private CallbackHandler callbackhandler;
22      private static final int NOT = 0;
23      private static final int OK = 1;
24      private static final int COMMIT = 2;
25      private int status;
26
27      public void initialize(Subject subject, CallbackHandler//
28                      callbackhandler, Map state, Map options)
29      {
30          status = NOT;
```

```
31          identity = null;
32          this.subject = subject;
33          this.callbackhandler = callbackhandler;
34      }
35
36      public boolean login() throws LoginException
37      {
38
39          if (callbackhandler == null)
40          {
41              throw new LoginException("No callback handler is available");
42          }
43          Callback callbacks[] = new Callback[1];
44          callbacks[0] = new NameCallback("Mi az ön titkos kódja? ");
45          String name = null;
46          try
47          {
48              callbackhandler.handle(callbacks);
49              name = ((NameCallback) callbacks[0]).getName();
50          } catch (java.io.IOException ioe)
51          {
52              throw new LoginException(ioe.toString());
53          } catch (UnsupportedCallbackException ce)
54          {
55              throw new LoginException("Error: " + ce.getCallback().toString());
56          }
57          if (name.equals("123456"))
58          {
59              identity = new TestPrincipal("inyiri");
60              status = OK;
61              return true;
62          } else
63          {
64              return false;
65          }
66      }
67
68      public boolean commit() throws LoginException
69      {
70          if (status == NOT)
71          {
72              return false;
73          }
74          if (subject == null)
75          {
76              return false;
77          }
78          Set entities = subject.getPrincipals();
79          if (!entities.contains(identity))
80          {
81              entities.add(identity);
82          }
83          status = COMMIT;
84          return true;
85      }
86
87      public boolean abort() throws LoginException
88      {
89          if ((subject != null) && (identity != null))
90          {
91              Set entities = subject.getPrincipals();
92              if (entities.contains(entity))
93              {
94                  entities.remove(identity);
95              }
```

```
96          }
97          subject = null;
98          identity = null;
99          status = NOT;
100         return true;
101     }
102
103     public boolean logout() throws LoginException
104     {
105         subject.getPrincipals().remove(identity);
106         status = NOT;
107         subject = null;
108         return true;
109     }
110 }
```

Van egy *TestLoginModule* osztályunk, de vajon, hogyan kell használnunk? A 29. programlista egy *ProbaLogin.conf* nevű file tartalmát mutatja. A jelenleg érvényesnek hagyott (a többit megjegyzésbe tettük) egyetlen PAM modulunk éppen a *TestLoginModule*. A *required* jelentését lásd lentebb.

```
// 29. programlista: A LoginModule konfigurálása
ProbaLogin
{
  test.jaas.TestLoginModule required;
  //com.sun.security.auth.module.UnixLoginModule required;
  //com.sun.security.auth.module.NTLoginModule required;
  //com.sun.security.auth.module.Krb5LoginModule required;
};
```

Az egyes *LoginModule*-ok lehetséges opciói:

- *REQUIRED*: Az authentikáció csak akkor lehet sikeres, ha minden így jelölt modul sikeresen azonosította a klienst. Egy ilyen modul esetleges sikertelensége nem szakítja meg a hitelesítési folyamatot – azaz több modul esetén folytatódik –, azonban a hitelesítés végeredménye ekkor más nem lesz sikeres.
- *REQUISITE*: Az authentikáció csak akkor lehet sikeres, ha minden így jelölt modul sikeresen azonosította a klienst. Itt azonban egy modul sikertelensége az egész authentikációs folyamat leállítását is jelenti, azaz ekkor nem megy végig, mint az előző esetben.
- *SUFFICIENT*: Az így jelölt modulnak nem kötelező sikeresnek lennie, azonban ha az, akkor a teljes hitelesítési folyamat is sikeres lesz és az esetleges további modulok nem hajtódnak végre. Ezt az esetet használjuk jellemzően, amikor a user-eket több helyen is tároljuk és azt akarjuk biztosítani, hogy mindenütt végignézze az authentikációs folyamat, az esetleges siker reményében. Például van pár címtárunk és egy helyi */etc/password* jellegű file-unk.
- *OPTIONAL*: Közömbös ennek a modulnak a sikeressége vagy sikertelensége, mindkét esetben az authentikációs folyamat folytatódik.

A 30. programlista a *TestLoginModule*-t tesztelő kliens programot mutatja. A 20. sorban lévő *System.setProperty()* beállítással konfigu-

ráljuk be, hogy ez a kliens – amennyiben authentikálni akar – milyen konfigurációt használjon.

Természetesen használhattuk volna a -Djava.security.auth.login.config=ProbaLogin.conf parancssori beállítást is. A 26. sorban hozunk létre egy *LoginContext* objektumot, ahol paraméterül megadjuk, hogy a *ProbaLogin* konfigurációs szekciót (mint input String paramétert) használjuk a *ProbaLogin.conf* file-on belül és ehhez a *TestCallbackHandler* forgatókönyvvel szerezzük meg a credential-t. A 39. sorban lévő *ctx.login()* szerepét már ismerjük.

Lefuttatva ezt a kliens programot, megjelenik a *Mi az ön titkos kódja?* prompt, ahol az *123456* beírása beléptet (más kód most természetesen nem jó!), majd a kliens kiírja a frissen azonosított alany principáljait, ami esetünkben csak az *inyiri*.

```
1  // 30. programlista: A TestLoginModule-t tesztelő kliens
2
3  package test.jaas;
4
5  import java.util.Hashtable;
6  import java.util.Iterator;
7  import java.util.Properties;
8  import javax.security.auth.Subject;
9  import javax.security.auth.login.LoginContext;
10 import javax.security.auth.login.LoginException;
11
12 public class TestClient
13 {
14
15     public TestClient()
16     {
17     }
18
19     public static void main(String argv[])
20     {
21         System.setProperty("java.security.auth.login.config",
22 "/opt/netbeans/myprojects/TestProjects/TesztelesekProject/src/ProbaLogin.conf");
23
24         LoginContext ctx = null;
25         try
26         {
27             ctx = new LoginContext("ProbaLogin", new TestCallbackHandler());
28         }
29         catch(LoginException le)
30         {
31             System.err.println("LoginContext cannot be created. "+ le.getMessage());
32             System.exit(-1);
33         }
34         catch(SecurityException se)
35         {
36             System.err.println("LoginContext cannot be created. "+ se.getMessage());
37         }
38         try
39         {
40             ctx.login();
41         }
42         catch(LoginException le)
43         {
44             System.out.println("Authentication failed. " + le.getMessage());
45             System.exit(-1);
46         }
47         System.out.println("Authentication succeeded.");
48
49         Iterator p = ctx.getSubject().getPrincipals().iterator();
50         while ( p.hasNext() )
```

```
51          {
52              System.out.println( "--->" + ((Principal)p.next()).getName()   );
53          }
54      }
55  }
```

Amennyiben a 29. programlistában a *UnixLoginModule*-t állítjuk be (és a *TestLoginModule*-t megjegyzésbe tesszük), úgy a Linux-ba való bejelentkezést veszi át a Java programunk. Ez 2 dolgot jelent. Az egyik az, hogy itt a *login()* hatására a chat forgatókönyv némán lejátszódik a háttérben, átvéve a Subject-et az operációs rendszertől. A másik az, hogy ilyenkor a konfigurációt (esetünkben a *ProbaLogin.conf* nevű file-t) az operációs rendszer szinten kell védeni, különben bárki átírhatná. A *TestClient* class futási eredménye a következő (a user neve inyiri volt, az utána jövő számok a unix csoportok azonosító számai, ahova az inyiri user tartozik):

```
Authentication succeeded.
--->inyiri
--->1000
--->1000
--->4
--->20
--->24
--->25
--->29
...
```

Ezzel a módszerrel olyan vastag Java programokat készíthetünk, amik az operációs rendszertől veszik át az azonosítást. Az eljárás természetesen Windows-on is működik, mely esetben az NTLoginModule class-t kell bekonfigurálni. Ekkor a saját Windows XP gépemre bejelentkezve és a klienst futtatva a következő eredményt kaptam:

```
Authentication succeeded.

--->inyiri
--->S-1-5-21-796063269-1865366272-2249723920-39659
--->MOL
--->S-1-5-21-796063269-1865366272-2249723920
--->S-1-5-21-796063269-1865366272-2249723920-513
--->S-1-1-0
--->S-1-5-32-544
--->S-1-5-32-547
--->S-1-5-32-555
--->S-1-5-32-545
--->S-1-5-14
--->S-1-5-4
--->S-1-5-11
--->S-1-5-5-0-1443760
--->S-1-2-0
--->S-1-5-21-796063269-1865366272-2249723920-23303
--->S-1-5-21-796063269-1865366272-2249723920-23112
--->S-1-5-21-796063269-1865366272-2249723920-41976
...
```

Érdekes lehetőség a Kerberos bekonfigurálása, amihez a *Krb5LoginModule* szükséges. A JVM indítási környezete ekkor ezt a 3 környezeti paramétert kapja meg.

```
-Djava.security.krb5.realm=CEG.SYS.CORP
-Djava.security.krb5.kdc=cegdc02.ceg.sys.corp
-Djava.security.auth.login.config=ProbaLogin.conf
```

Az első a kerberos domain vagy realm neve. A második a domainhez tartozó KDC (key distribution center) szerver neve. A harmadik a már ismert authentikáció konfigurációs file-unk használata felőli rendelkezés. A bejelentkezés ilyenkor 2 körös kérdés-válasz forgatókönyv szerinti. Az egyikben a username, a másikban a password kerül bekérésre, majd a *Krb5LoginModule* beléptet bennünket. A példában szereplő *CEG.SYS.CORP* egy Windows domain, a *cegdc02.ceg.sys.corp* pedig ennek a domain-nek egy domain controllere, ami definíció szerint egy KDC is, ugyanis a Windows 2000 óta a Windows első helyen támogatott hitelesítési módszere a kerberos.

```
Kerberos username [inyiri]:
Kerberos password for inyiri: xxxxxxx
Authentication succeeded!
--->inyiri@CEG.SYS.CORP
```

3.4. A Java authorizáció

Az authorizáció használatához egy újabb konfigurációs file-ra van szükségünk, ami a példánkban a *test.policy* file lesz. Tesztprogramunkat a 31. programlista tartalmazza.

```
1  // 31. programlista: Authorizációs teszt Linux alatt
2
3  package test.jaas;
4
5  import java.io.File;
6  import java.io.FileNotFoundException;
7  import java.io.IOException;
8  import java.security.Principal;
9  import java.security.PrivilegedAction;
10 import java.util.Iterator;
11 import java.util.logging.Level;
12 import java.util.logging.Logger;
13 import javax.imageio.stream.FileImageInputStream;
14 import javax.security.auth.Subject;
15 import javax.security.auth.login.LoginContext;
16 import javax.security.auth.login.LoginException;
17
18 public class TestClientUnix
19 {
20         public static void main(String argv[]) throws FileNotFoundException, IOException
21     {
22         System.setProperty("java.security.auth.login.config",
23 "/opt/netbeans/myprojects/TestProjects/TesztelesekProject/src/MOLLogin.conf");
24
25         LoginContext ctx = null;
26         try
27         {
28             ctx = new LoginContext("MOLLogin", new TestCallbackHandler());
29         }
30         catch(LoginException le)
31         {
32             System.err.println("LoginContext cannot be created. "+ le.getMessage());
33             System.exit(-1);
34         }
35         catch(SecurityException se)
36         {
37             System.err.println("LoginContext cannot be created. "+ se.getMessage());
38         }
39         try
40         {
41             ctx.login();
42         }
43         catch(LoginException le)
44         {
45             System.out.println("Authentication failed. " + le.getMessage());
46             System.exit(-1);
47         }
48         System.out.println("Authentication succeeded.");
49
50         Iterator p = ctx.getSubject().getPrincipals().iterator();
51         while ( p.hasNext() )
52         {
53             System.out.println( "--->" + ((Principal)p.next()).getName()  );
54         }
55
56
57         PrivilegedAction action = new PrivilegedAction() {
58
59             public Object run()
60             {
61                 //A file
62                 String fn = "/opt/netbeans/myprojects/TestProjects/TesztelesekProject/src/egyfile.txt";
63                 File file = new File(fn);
64                 try
65                 {
66                     FileImageInputStream fis = new FileImageInputStream(file);
67                     System.out.println("Méret:" + fis.length());
68                 } catch (FileNotFoundException ex)
69                 {
70                     Logger.getLogger(TestClientUnix.class.getName()).log(Level.SEVERE, null, ex);
71                 } catch (IOException ex)
72                 {
73                     Logger.getLogger(TestClientUnix.class.getName()).log(Level.SEVERE, null, ex);
74                 }
75                  return file;
76             }
77
78         };
79
80         Subject.doAsPrivileged(ctx.getSubject(), action, null);
81
82     }
83
84
85 }
```

A program a Linux hitelesítést veszi át, így a username és annak group principáljai is onnan jönnek. A 60. sorban látható az *egyfile.txt* file, aminek a méretét a program a képernyőre szeretné írni, azonban ezt csak az inyiri teheti meg a *test.policy* file-ban lévő konfiguráció miatt. A program az 52. sorig ismerős, a *login()* művelet fut le, megszerezve a *Subject*-et is. Az 55-76 sorok között egy *PrivilegedAction* objektumot hozunk létre, ez a Java szabványos authorizáción keresztül való működési mechanizmusa, ugyanis ennek a *run()* metódusához lesz vagy nem lesz jogunk futáskor. A 78. sorban futtatjuk a *Subject.doAsPrivileged()* metódust, ami azt a kódot futtatja, ami a file méretének megállapítását és kiírását csinálja. Természetesen csak akkor, ha az azt megelőző hitelesítés során kapott principál megfelelő, amit a *test.policy* file 20. sorában láthatunk.

A tesztprogram indítása:

```
# A -Djava.security.manager:
#     bekapcsolja a security alrendszert
# -Djava.security.policy:
#     A policy leírásának helye
#     Lásd: test.policy file

$ java
-Djava.security.manager
-Djava.security.policy=./test.policy
-cp ../build/classes test.jaas.TestClientUnix
```

```
// test.policy file: Authentikáció konfiguráció
grant
{
    //permission java.security.AllPermission;
    permission javax.security.auth.AuthPermission "createLoginContext.MOLLogin";
    permission javax.security.auth.AuthPermission "getLoginConfiguration";

    permission javax.security.auth.AuthPermission "modifyPrincipals";

    permission javax.security.auth.AuthPermission "modifyPublicCredentials";
    permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
    permission javax.security.auth.AuthPermission "doAs";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
    permission java.security.SecurityPermission "setPolicy";
    permission java.security.SecurityPermission "getPolicy";

    permission java.util.PropertyPermission "java.security.auth.login.config", "read,write";
};

grant codebase "file:/opt/netbeans/-" , principal com.sun.security.auth.UnixPrincipal "inyiri"
//com.sun.security.auth.UnixPrincipal ""
//test.jaas.TestPrincipal "inyiri"
{
    permission java.io.FilePermission "/opt/netbeans/myprojects/TestProjects/TesztelesekProject/src/egyfile.txt", "read";
};
```

4. Session replikációs alternatívák

Az olyan nagyméretű és összetett infrastruktúrák, mint a SOA esetében a túlterhelés miatt lassulások, leállások jelentkezhetnek. Erre a problémára számos megoldás lehetséges, melyek közül az egyik leghatékonyabb a session replikáció. Ebben a cikkben az Oracle WebLogic környezetben vizsgáljuk meg ezt a lehetőséget.

Portál session replikáció

Az olyan nagyméretű és összetett infrastruktúrák, mint a SOA esetében a portálfelületeken a túlterhelés miatt lassulások, leállások jelentkezhetnek. A megoldást a portál processz memóriájának bővítése jelentené, de sok esetben ez nem lehetséges. Erre a problémára számos megoldás lehetséges, amely közül az egyik leghatékonyabb a session replikáció. A session replikációhoz a rendszert két részre kell vágnunk, ami egyben megoldja a clusterben futás kérdését is. Első lépésben meg kell vizsgálnunk, milyen feltételekkel tehetjük alkalmassá a portálalkalmazást a session replikációra. A legfontosabb, hogy a replikációhoz szükséges szerializálhatósági követelményt teljesítse. Azoknál a részeknél, ahol ez nem lehetséges - mint a BPM-folyamatazonosítónál -, meg kell oldanunk a replikáció utáni minél zavartalanabb működést. Következő feladatunk a session replikáció bekapcsolása. Fontos, hogy a felhasználók úgy indíthassák újra a portált, hogy ne kelljen ismételten bejelentkezniük, így érdemes megoldani, hogy az egy gépen futó portálszerverek között történjen a session replikáció.

WebLogic HttpSession kezelés

A munkamenet (session) kezeléssel kapcsolatos beállításainkat webalkalmazásunk WEB-INF/WebLogic.xml leírójának, session-descriptor szekciójában tudjuk végrehajtani, s ennek megfelelően azok szállító (vendor) specifikusak. WebLogic alkalmazásszerver környezetben öt különböző http session állapotkezelési és nyomkövetési (track and store) sémát különböztetünk meg:

1. In Memory (Single Server Not Replicated)
2. JDBC Based
3. File Based
4. In Memory Replication
5. Cookie Based

A fenti lista első négy eleme a munkamenet állapotkezelését szerver oldalon valósítja meg, s csak a megvalósítás módjai különböznek egymástól, az ötödik viszont kliens oldali megoldást nyújt. Az azonosításhoz az első kliens-kérés alkalmával a WebLogic szerver egy munkamenetazonosítót tartalmazó session cookie-t állít össze, és küld el a kliensnek. A kliens, kérései során ezzel a session cookie-val azonosítja majd magát a szerver számára. A session cookie szerkezetileg három részre bontható, a munkamenetazonosítóra (sessionid), az elsődleges szerver azonosítójára (primary_server_id), valamint a másodlagos szerver azonosítóra (secondary_server_id).

In Memory séma

Ez az alapértelmezett munkamenet-kezelési séma. A HttpSessionben levő felhasználói adatok ebben az esetben a kiszolgáló szerver memóriájában tárolódnak. A munkamenet élettartamát a timeout-secs paraméterrel állíthatjuk a

kívánt értékre. A szerver ezeket periodikusan ellenőrzi, és amennyiben lejárt munkamenetet talál, felszabadítja a hozzá tartózó, használaton kívüli erőforrásokat. A megoldás hátránya, hogy függetlenül a munkamenet élettartamától, a HttpSessionben tárolt adatok elvesznek a szerver újraindítását, nem tervezett leállását követően.

JDBC Based séma

JDBC Based Session Persistance séma esetén a munkamenet-perzisztenciát adatbázis tábla írásával biztosítjuk. Az adatbázis-kapcsolat kialakításához létrehozunk egy nem XA-s JDBC adatforrást. A munkamenet adatai bináris formátumban (BLOB) tárolódnak a wl_session_values mezőben. A wl_access_time mező tartja nyilván az utolsó hozzáférés időpontját. Ezt a mezőt akkor is módosítja a szerver, ha csak olvassák a munkamenetet. A munkamenet adatainak írásakor a wl_session_values és a wl_access_time is módosul. A JDBC-megoldásnál a szerver átmeneti tárolóban tárolja az utoljára használt munkamenetek adatait az optimális elérés érdekében. A megoldás előnye, hogy a munkamenet életciklusa nem ér véget a szerver újraindításával, tartalma újraépíthető az adatbázisban tárolt adatok alapján, sőt a munkamenet állapotának replikálására is felhasználhatjuk ezt a megoldást. Természetesen a pertisztálásnak ára van, az adatbázis tábla folyamatos írása/olvasás költséges művelet, ami megnöveli a kliens kérések válaszidejét.

File Based séma

A File Based Session Persistence séma esetében a munkamenet adatai egy natív fájlba íródnak a szerver oldalon. A tároláshoz használt állományok helyét a persistent-store-dir paraméterrel tudjuk meghatározni. Klaszteres környezetben valamennyi tagszervernek írnia/olvasnia kell tudnia ezt a könyvtárat. A helyet adó tárolónak magas rendelkezésre állást és gyors válaszidőt kell biztosítania. A gyakorlatban SAN-hálózatra kötött disk-alrendszeren található tároló egységet használnak erre a célra. A memória alapú megoldáshoz képest a fájl alapú megoldásnál is magasabb válaszidővel kell számolnunk, cserébe a munkamenetünk adatai nem vesznek el a kiszolgáló szerver leállása esetén. Hasonlóan a JDBC-alapú megoldáshoz, ez is használható munkamenetünk állapotának replikálására.

In Memory Session Replication séma

A memóriában végzett munkamenet replikálás esetén a WebLogic másolatot készít a munkamenet állapotáról a klaszter egy másik szerverére. Munkamenetünk állapota így két helyen található meg, azon a szerveren ahová a kérés beérkezett (elsődleges szerver) és azon, ahova a WebLogic átmásolta az állapot-adatokat (másodlagos szerver). Az állapotok szinkronban vannak, így ha az elsődleges szerver valamilyen okból leáll, vagy nem elérhető, akkor a másodlagos szerver veszi át szerepét. A kliens számára a szerepátvétel transzparens, az első alkalommal tapasztalható megnövekedett válaszidőtől eltekintve nem vehető észre a változás. Ahhoz, hogy teljes képünk legyen a WebLogic által nyújtott replikációs lehetőségekről, meg kell ismerkednünk a Replication Group funkcióval és az általa nyújtott lehetőségekkel. Alapértelmezésként a WebLogic a munkamenet-kezelésnél a különböző fizikai gépek (machine) közötti replikálást részesíti előnyben. Ez érthető is, hiszen csak így biztosítható a fizikai gép leállása esetén a munkamenet állapotának megőrzése. A Replication Group használatával a WebLogic szerver példányokat logikailag csoportosíthatjuk, befolyásolva ezzel a munkamenet-állapot másolatok helyét. A replikáció beindításához a persistent-store-type paramétert kell replicated vagy replicated_if_clustered értékre állítani. A szerver példányokat replication grouphoz és preferred

secondary replication grouphoz rendeljük. Amikor beérkezik egy kérés valamelyik szerverpéldányhoz, és annak el kell döntenie, hogy a munkamenet állapotot hova replikálja, akkor először a preferred secondary replication group beállítását nézi meg, majd megvizsgálja, hogy a klaszterben van-e olyan, tőle különböző szerverpéldány, aminek a replication group értéke megegyezik ezzel az értékkel. Ha van ilyen szerver, akkor az lesz a másodlagos szerver (secondary server), oda kerül a munkamenet-állapot másolat. Találat hiányában arra a szerverpéldányra esik a választás, amelyik a kérést kiszolgáló szervert futtató fizikai géptől eltérő fizikai gépen fut. Több találat esetében is a fizikai gép különbözősége dönt. A memóriában tárolt munkamenet példányok élete véget ér, amint a klaszter valamennyi tagját újraindítjuk. Időnként szükség van a munkamenet állapotának megtartására függetlenül a szerverek életciklusától. Ebben az esetben valamilyen külső cache szerver megoldást szoktunk alkalmazni. Ebben nyújt segítséget az Oracle Coherence termékének http session kezelésért felelős modulja (Coherence*Web), mely az alkalmazások számára transzparens módon veszi át a session állapotok kezelését.

Cookie based

A WebLogic lehetőséget biztosít munkamenetünk állapotának kliens oldali tárolására is, cookie formájában. Kisméretű munkamenet esetén használhatjuk ezt a megoldást, akkor, ha valamennyi kliensünk böngészője engedi a cookie tárolását. A gyakorlatban ritkán alkalmazzák, mivel biztonsági kérdéseket vet fel, ezen kívül ennél a megoldásnál csak szöveges (string) tartalmat adhatunk a munkamenetünkhöz.

Terhelés elosztás és Failover

laszteres környezetben a webes erőforrások (Servlet, JSP) terhelés-elosztása megoldható a WebLogic szerver saját proxy plug-in megoldásával vagy külső, nagy teljesítményű hardver eszközzel egyaránt. Most a WebLogic saját proxy plug-in megoldását mutatjuk be. WebLogic környezetben a proxy plug-in komponens feladata a kliensektől érkező http kérések továbbítása a nyilvántartásában szereplő szerverpéldányok felé. A proxy plug-in működését tekintve reverse proxy, azaz a kliens nem közvetlenül a szervert, hanem a proxy plug-int címzi meg. A szerver példányok közötti választás alapértelmezésben körbeforgó (Round Robin) algoritmus alapján történik. A proxy plug-in folyamatosan figyeli a szerver példányok elérhetőségét, és amennyiben valamelyik szerver elérhetetlenné válik, arra a példányra nem küld további kéréseket. A leállások detektálásán felül a plug-in, session replikáció használata esetén képes a kliens http session állapot replikáját kezelő szerverpéldányra átirányítani a kéréseket. A másodlagos szervert a http kérésben megkapott session cookie-ban található secondary_server_id alapján azonosítja be. A munkamenet nyomon követését megoldhatjuk URL rewriting segítségével is. Ez abban az esetben jön jól, ha alkalmazásunk kliensei között van olyan, amely nem támogatja a cookie-kezelést. URL rewriting esetén mind az elsődleges mind a másodlagos szerver azonosítóját az URL kódolja.

Coherence*Web

Az Oracle Coherence termékének Coherence*Web modulja a http session állapot kezelést oldja meg klaszteres környezetben. Lehetőséget biztosít különböző webalkalmazások közötti munkamenet-megosztásra, akár heterogén környezetben is. WebLogic szerverhez illesztése a Coherence*Web SPI-vel már nem

igényli a WebLogic.jar és a webalkalmazások instrumentálását. Telepítéséhez alkalmazásszerver oldalon csak a coherence/lib könyvtárban található coherence-web-spi.war library-t kell telepíteni a WebLogic szerverre, klaszterünk valamennyi szerverpéldányára targetálva. Alkalmazás oldalon a coherence/lib könyvtár alatt lévő coherence.jar-t kell bemásolnunk a WEB-INF/lib könyvtárba, valamint a coherence-web-spi-t felvenni a WebLogic.xml-ben. Környezettől függően többféleképen konfigurálhatjuk a Coherence Web modulját, ezeket a lehetőséget nézzük át a fejezet további részében.

Coherence klaszter tagok izolációja

A Coherence*Web modul telepítéséhez több lehetőség is rendelkezésünkre áll. Ezek az opciók alapvetően meghatározzák a Coherence*Web modul hatáskörét az alkalmazásszerveren belül. Application Server Scoped Cluster Nodes Ez esetben az alkalmazásszerverre telepített valamennyi webalkalmazás Http session kezelését átveszi a Coherence*Web. Az alkalmazásszerveren belül egy Coherence klaszter csomópont jön létre, ehhez kapcsolódnak a webalkalmazások.

EAR Scoped Cluster Nodes Telepíthetjük a coherence.jart az alkalmazásunk (EAR) classpath-ára is. Ekkor alkalmazásonként jön létre egy-egy coherence klaszter csomópont az alkalmazásszerveren belül. Az EAR-ban található valamennyi webalkalmazás egységesen ehhez a csomóponthoz csatlakozik, és delegálja a Http sessionkezelést a Coherence felé.

WAR Scoped Cluster Nodes Végül mód van a Coherence*Web modul webalkalmazásonkénti (WAR) izolációjára is. Ebben az esetben a coherence.jart a webalkalmazásunkba csomagolva telepítjük. Valamennyi alkalmazás a számára dedikáltan rendelkezésre álló coherence klaszter csomópont felé delegálja Http session kezelését.

Session Modellek

A session modell meghatározásával szabályozhatjuk a munkamenet-állapot fizikai reprezentációjának és tárolásának módját a coherence szerveren belül. A munkamenet állapot kezeléséért a HttpSessionModel objektum, a munkamenet állapotok tárolásáért a HttpSessionCollection objektum felel a Coherence*Web modulján belül.

Traditional Model

A munkamenet állapota ez esetben egy entitásban tárolt, de a session attribútumok sorosítása (serialization) és visszaállítása (deserialization) attribútumonként történik. Ezt a megoldást kisméretű (<=10KB) session esetében javasolt használni, ahol nincs olyan session objektum, amelyre több különböző session attribútum is hivatkozik.

Monolithic Model

A munkamenet állapota egy entitásban kerül tárolásra, és a session attribútumok sorosítása (serialization) és visszaállítása (deserialization) egyszerre, egy lépésben történik.

Split Model

Ez a megoldás a Traditional Model lehetőségeit terjeszti ki azzal, hogy a nagyméretű session attribútumokat fizikailag önálló entitásban tárolja el. Egy-egy nagyméretű objektum módosítása így nem igényli a teljes munkamenet-állapot frissítését, elég csak a tároló entitást szinkronizálni, ez pedig kisebb hálózati forgalmat fog generálni.

Nagyméretű munkamenetek esetén a Split Model a javasolt megoldás.

Session Scoping

A Coherence*Web modul használatával megoszthatjuk a munkamenet adatait a webalkalma-

zások között akár akkor is, ha ezek az alkalmazások különböző webkonténerekben futnak. Konténeren belüli megosztásnál a coherence-session-cookie-path paramétert kell beállítani a kontextusoknak megfelelően. Például ha két alkalmazás között kell megosztani session adatokat és az egyiknek a /web/behajtas a másiknak pedig /web/szamlazas a kontextusa, akkor a /web lesz a coherence-session-cookie-path helyes értéke. Különálló web konténerek esetén, ahol a konténerek ráadásul külön fizikai szerveren futnak, a közös sessionelérés érdekében be kell állítani a coherence-session-cookie-domain paramétert is.

Cache topológiák

Replicated Cache Service

A Coherence szerver replicated módban a cashben tárolt adatokat átmásolja a klaszter valamennyi tagjára, és szinkronban tartja azokat. Mivel az adatok minden tagon megvannak, ezért hozzáférésük nagyon gyors, gyakorlatilag az alkalmazásszerver JVM-ből történik az adatok elérése, minimális késleltetéssel. Ennek a megoldásnak éppen ezért az olvasás intenzív felhasználás esetén van előnye a többi megoldással szemben. A cashben tárolt adatok módosítása viszont költséges művelet, mivel a klaszter valamennyi tagjára át kell másolni a változásokat. Tervezéskor figyelembe kell venni ennek a topológiának a skálázhatósági korlátait. Új tagok klaszterbe vonásával ugyanis mindegyik klaszertagnál növelni kell a java heap méretét, ami jelentősen leronthatja a teljesítményt (gc overhead). Továbbá update intenzív esetben a skálázhatóság nem nő lineárisan a klaszter növekedésével együtt, mivel az update-ek válaszideje csökkeni fog egy-egy új tag klaszterbe állításával.

Partitioned Cache Service

Ez esetben nem a teljes adatot másolják át a klasztertagokra, hanem annak csak egy részét, egyenletesen, azonos méretű darabokban szétosztva. Ezt a megoldást ezért Distributed Cashnek is hívják. A megoldás jellemzői:

- Mivel az adatok ez esetben egyenletesen szét vannak ”kenve” a klaszteren belül, ezért a terhelés szétosztási lehetőség (Load Balanced) természetes velejárója e megoldásnak.
- A kliensek mindig ugyanazt az API-t használják az adatok eléréséhez, az API a kliens számára ugyanúgy viselkedik, bárhonnan is kelljen előbányászni klaszteren belül az adatokat (Location Transparency).
- Konfigurálhatóan tudjuk meghatározni az adatmásolatok számát a klaszteren belül. Ha a másolatok száma egy vagy több, akkor valamelyik klasztertag leállásakor az átirányítás (Failover) adatvesztés nélkül történik meg.
- A klaszteren belül egy adat kezeléséért mindig egyetlen tag felel. Ebből következően a kommunikáció – legyen az olvasás (get) vagy írás (put) –, point-to-point jellegű lesz, így nem okoz kommunikációs overhedet újabb tagok bevonása a klaszterbe. A skálázhatóság tehát a klaszter bővítésével együtt lineárisan növelhető (Linear Scalability).
- A local storage paraméterrel szabályozhatjuk, hogy a klaszteren belül egy tag kezeljen-e adatokat. Ennek akkor van jelentősége, ha az alkalmazásszerver JVM-jét nem akarjuk tovább terhelni a cash klaszter adatkezelési teendőivel.

Near Cache

Ez a megoldás a Partitioned Cache megoldást egészíti ki a legutoljára (MRU) vagy a leggyakrabban (MFU) használt objektumok alkalmazás oldali, azaz lokális cache-ben tárolásával, a jobb performancia érdekében. A lokális cache méretét és kezelésének algoritmusát paramérterezéssel szabályozhatjuk. Valójában az adatok elérését gyorsíthatjuk fel e megoldással, azon az áron, hogy az alkalmazás JVM-jéből kell feláldoznunk területet.

Coherence*Web konfiguráció

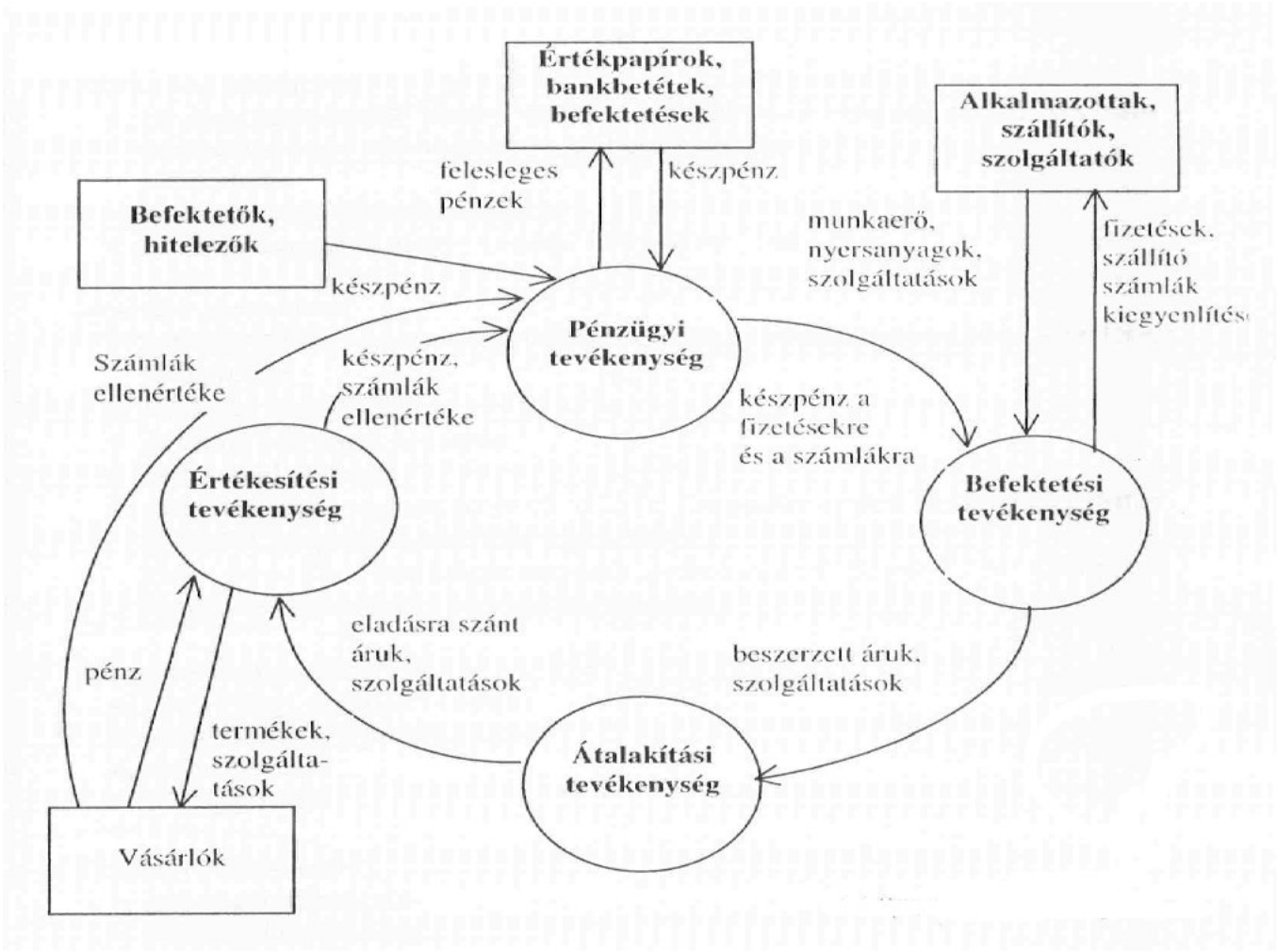
A konfigurációs állományok a coherence-web-spi.war állományban találhatóak. A WEB-INF\classes\session-cache-config.xml állományban tudjuk megadni az alkalmazni kívánt cache sémákat és azok paraméterezését. A WEB-INF/web.xml állományban további paraméterezési lehetőségre van mód.

Rácz Imre
Alerant Informatikai Zrt.

5. Az üzleti informatikai rendszerek fejlődése és típusai

A számítógépes informatika az 1950-es évektől indult el hódító útjára, azóta sokféle üzleti informatikai rendszer bukkant fel. Ebben a cikkben arra vállalkozunk, hogy áttekintjük a fejlődés főbb mérföldköveit, illetve felvázoljuk az egyes rendszertípusok legfontosabb jellemzőit.

Amikor szeretnénk megérteni az üzleti információs rendszereket (továbbiakban: ÜIS), akkor előbb egy általánosabb fogalmat, az üzleti rendszer fogalmát kell tanulmányoznunk. A termelő vagy szolgáltató vállalkozások nyílt rendszerek, azaz ezt a sémát követik: INPUT→ transzformáció→OUTPUT (termék vagy szolgáltatás).



7. ábra. Az üzleti körforgás

Az üzleti tevékenység lényegét az *üzleti körforgás* mutatja (7. ábra). Ez egy olyan absztrakció, ami mindenfajta vállalatra igaz, az ábra önmagáért beszél. Az üzleti tevékenységek egy másik megközelítési lehetősége az *értéklánc modell*. Itt a hangsúly az egymáshoz kapcsolódó alaptevékenységek (beszerzés, logisztika, termelés, marketing, értékesítés, ...) láncolatán van, célfüggvényként megjelölve az egyes tevékenységek hozzáadott értékének maximalizálását, a költségek optimális szinten tartása mellett. Az üzleti rendszerek ellátási láncokba szerveződnek, azaz az egyik vállalat (vagy egy vállalaton belüli részleg) outputja a másik vállalat (vagy részleg) inputja. Emiatt az INPUT és az OUTPUT menedzsmentje fontos, az előbbit CRM (Customer Relationship Management), az utóbbit pedig SRM (Supplier Relationship Management) néven emlegeti a szakirodalom. A teljes ellátási lánc menedzsmentjét pedig SCM-nek (Supply Chain Management) nevezzük. Az Internetes korszakkal az SCM-nek egy magasabb szerveződési szintje alakult ki, az elektronikus kereskedelem, amit kicsit kitágított értelemben e-business-nek hívunk. Ennek legfontosabb kategóriái:

- *B2B* (Business to Business): vállalatok közötti e-kereskedelem
- *B2C* (Business to Customer): a vállalatok és fogyasztók közötti e-kereskedelem
- *C2C* (Customer to Customer): az egyének egymás közötti e-kereskedelme (gondoljunk olyan rendszerekre, mint a világméretű eBay vagy a magyar Vatera)
- *B2A* (Business to Administration): a vállalatok és a hivatalos szervek közötti e-kapcsolat

Az eddigi modellek mellett a vállalkozások üzleti informatikáját azok szervezeti struktúrája és annak jellege (hierarchikus, mátrix), valamint a szervezeti kultúra is alapvetően meghatározza. A mai gazdálkodó szervezetekben az IT stratégiai szerepet kap, ami azt jelenti, hogy az IT szervezetek által nyújtott szolgáltatások előnyt nyújtanak (vagy csökkentik a fennálló hátrányokat) a versenytársakkal szemben. A gyakorlatban ez azt jelenti, hogy az üzleti folyamatokat folyamatosan tökéletesíteni kell, ahol lehet és értelmes, ott célszerű az IT megoldások használata. Ez sokszor a folyamatok újratervezését igénylik, amit *BPR*-nak (Business Process Reenginering) nevezünk.

Az informatikai rendszerek adatbázisai kiváló lehetőséget adnak a vezetői döntések meghozatalához, sőt manapság már olyan rendszerek készülnek, amiknek ez az elsődleges céljuk. A döntéselmélet mára egy külön közgazdasági diszciplína lett, itt most azt szeretnénk csak kiemelni, hogy az informatikai rendszerek a különféle vezetői szintek (politikai, stratégiai, taktikai, operatív) eltérő igényeit egyaránt igyekeznek kielégíteni.

5.1. Rövid áttekintés

Az üzleti rendszerek nagyon vázlatos áttekintésére építve tekintsük át röviden az üzleti informatikai rendszerek fejlődését és típusait. A dobozos információs rendszerek folyamatosan változnak, közben újabb típusok is jelennek meg a piacon. A régi szemléletű rendszerek szabványosodnak, beolvadnak egy korszerűbb felfogású megoldásba vagy eltűnnek.

Üzleti információs rendszer: Olyan formalizált számítógépes rendszer, amely különböző forrásokból gyűjt adatokat, feldolgozza, tárolja azokat, majd információkat szolgáltat a felhasználói számára.

Az alábbiakban összefoglaljuk az üzleti informatikai rendszertípusokat, ahogy azt az üzleti informatika jelenleg osztályozza (forrás: *Kacsukné Dr. Bruckner Lívia, Kiss Tamás: Bevezetés az üzleti informatikába, Akadémia Kiadó*):

- 1950-es évek: *TPS* (Transaction Processing System)
- 1960-as évek: *MIS* (Management Information System)
- 1970-es évek: *DSS* (Decision Support System), *PPS* (Production Planning and Scheduling), *MRP* (Materials Requirements Planning)
- 1980-as évek: *EIS* (Executive Information System), *ES* (Expert System), *GDSS* (Group Decision Support System)
- 1990-es évek: *ERP* (Enterprise Resource Planning), *BI* (Business Intelligence), *CRM*, *SRM*, *SCM*, *KM* (Knowledge Management)
- 2000-es évek: *EPM* (Enterprise Performance Management), *Business Suite*

Mi jellemzi az egyes rendszertípusokat? Próbáljuk meg röviden összefoglalni!

TPS. Látható, hogy az üzleti életben először ezek a tranzakciókezelő rendszerek jelentek meg. Feladatuk az elemi üzleti események rögzítése, feldolgozása. A tranzakció fogalma azt jelenti, hogy az egyik konzisztens állapotból egy másikba visz az üzleti esemény kezelése. A konzisztencia adott szituációban való pontos meghatározása mindig kulcskérdés. Vegyünk például egy töltőállomást, ahol az üzleti esemény az, hogy valaki 50 litert tankolt. A pénztári rendszer ekkor megnyit egy új tranzakciót és rögzíti az 50 litert, de ez egyből inkonzisztens állapotba teszi a tranzakciót, aminek ez az első elemi művelete. Ezután a 2. művelet a fizetés, ami logikailag helyrehozza a konzisztenciát, így kiadható a COMMIT (jóváhagyás) az egész tranzakcióra. Egy tranzakciót általában több úton is konzisztenssé lehet tenni. Amennyiben nincs fizetés, úgy felveszünk egy jegyzőkönyvet és COMMIT. Lehet persze az is a végeredmény, hogy a benzint visszajuttatjuk a tartályba, amit az esemény visszagörgetésének nevezünk (ROLLBACK). A TPS rendszerek tehát a napi üzletmenettel (számlák kifizetése, eladások, bérkifizetések, megrendelések fogadása, áttárolás raktárak között, stb.) kapcsolatos adatok gyűjtésével, tárolásával, feldolgozásával foglalkozik. Ugyanakkor az egyik legfontosabb adatforrása a magasabb szintű rendszereknek.

A technikai adottságok miatt ezek a rendszerek először batch feldolgozású megoldások voltak, off-line működéssel. Ebben az időben az üzleti emberek csak „távolról" találkoztak a számítástechnikával. A bizonylatokat eljutatták a GAF-ba (Gépi AdatFeldolgozás), ahonnan vastag tablókat kaptak vissza. Ezek között voltak olyanok, mint a készletelszámolás vagy számlázás, azaz a tényleges manuális munkát jelentősen csökkentette. Később a technika egyre közelebb hozta a felhasználót a rendszerhez (például SAP R/2 CICS), aki emiatt egyre hatékonyabban használta azt, már magában a valós üzleti folyamatban (tranzakciók lebonyolításában) is számítógépes terminált használhatott. Ezt a módszert hívjuk *OLTP*-nek, azaz OnLine Transaction Processing-nek.

MIS. Egy idő után felmerült annak az igénye, hogy a rendszerek ne csak a tranzakciók kezelésében segítsenek, hanem a menedzserek információigényét is elégítsék ki. Ezeket hívjuk Vezetői Információs Rendszereknek (VIR=MIS). Itt jellemzően előredefiniált jelentések készülnek, valamilyen időközönként. Az is gyakran előfordul, hogy valamilyen különleges esemény vagy igény esetén alkalmi riport készül. Jellegénél fogva a MIS típusú rendszerek jól meghatározott scope-ra vonatkozóan adnak információt, főleg operatív, kisebb mértékben taktikai szinten, azaz magasabb vezetői döntésekhez általában már nem elég hatékonyak.

DSS. A döntéstámogató rendszerek a MIS gondolatának természetes továbbfejlesztéseként

főleg a taktikai szint számára készített, adott problémára koncentráló megoldások. A legnagyobb előrelépést az ad-hoc lekérdezések sokkal jobb támogatása jelenti. Az interaktív terminálhasználat bevezetése lehetővé teszi, hogy a rosszabbul strukturált döntési helyzetekben is támogatást adjon.

GDSS: A csoportos döntéstámogató rendszerek a DSS fejlesztései, ahol a csoportos (testületi) döntések támogatása a fő cél, emiatt a beépített kommunikációs megoldásokon (közös mappa, e-mail, stb.) nagy hangsúly van.

EIS: A felsővezetői információs rendszerek összegzett, grafikus, a legfontosabb tényezőkre koncentráló információkat nyújtanak. Persze lehetőséget adnak a részletekbe való betekintésre is. Alapelve az intuitív, könnyen kezelhető GUI biztosítása. A DSS, GDSS és EIS rendszerek kialakulásával jelent meg az *OLTP* fogalommal szemben meghatározott *OLAP* (Online Analytical Processing) kezdetleges fogalma is, ami a későbbi *BI* rendszerekben teljesedik ki.

ERP: A vállalati erőforrás-tervező rendszerrel tipikusan a 90-es évek terméke. Alapgondolata a termelés és a hozzá kapcsolódó erőforrások (pénz, humán) integrált tervezése. Kezdetleges formában tartalmazza a CRM, SRM és SCM csíráit is. Az ERP rendszerek a 2000-es években kiteljesedtek, így ma már próbálják támogatni az operatív szintű feladatok egy részét is. Jellemzően moduláris szerkezetűek, amit az ismert 8. ábra is mutat.

CRM: Az ügyfélkapcsolati rendszerek az ügyfelekkel kapcsolatos operatív feladatokat támogatják, segítik ezzel a marketing és ügyfélszolgálat munkáját. Taktikai, kisebb részben stratégiai szinten információkat adhat a termékfejlesztéshez és marketing irányokhoz is.

SRM: A beszállítóikapcsolat-kezelő rendszerek a beszállítókkal és beszerzésekkel foglalkoznak, eközben főleg operatív és taktikai szinten nyújtanak döntéstámogatást.

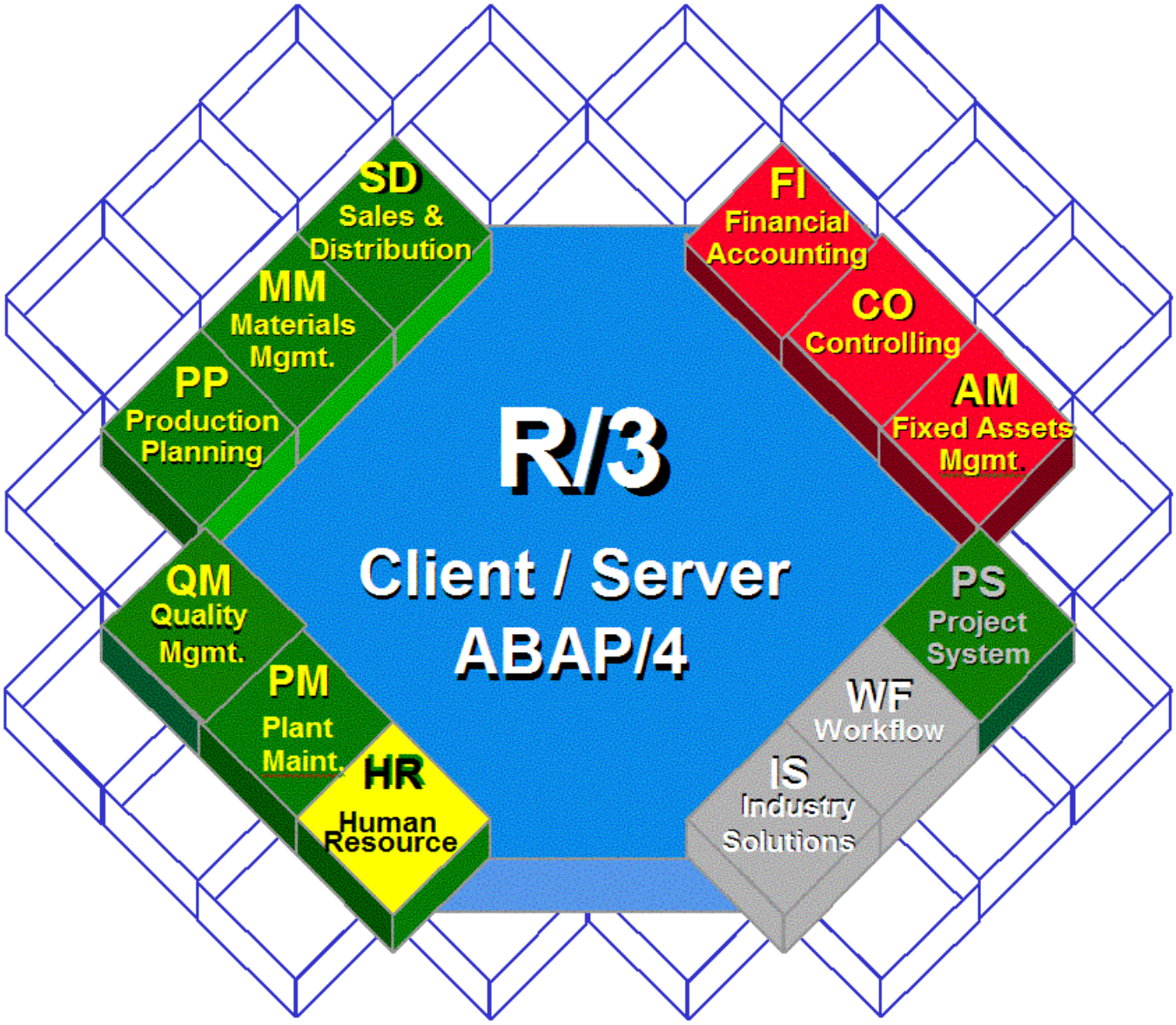
BI: Az üzleti intelligencia rendszerek a döntéstámogató rendszerek bármelyikét tartalmazhatják. Az OLAP kiteljesedik, online elemzésekre alkalmas. Jól átgondolt felépítése van, melyek közül az alapvető részek a következőek: *ETL* (Extract, Transform, Load), *Reporting* (minden formában, grafikus dashboard is van), *Planning* és *adatbányászat*. Az ETL rétegben megjelenik az adattárház (Datawarehouse), ahol tisztított, előfeldolgozott adatokat tárol, többdimenziós kockákban. Itt jelent meg legtisztább formában az un. multidimenzionális adatbáziskezelő fogalma is. Ezekben a rendszerekben a legkritikusabb dolgok a kockák dimenzióinak (adatszerkezet) jó eltalálása, az adatforrásokból származó hiteles adatok átvételi lehetőségének kialakítása. A kockák „töltése” során fontos szerepet kap a transzformáció, hiszen a tranzakcionális input adatokat össze kell vonni, megfelelő alakra kell hozni. A *BI* reporting rétege felelős az adatokhoz való hozzájuthatóságért. Tipikus példa erre a rétegre a *Business Objects*, ami a riportozás összes statikus és dinamikus módját támogatja, kiváló dashboard grafikus felületet is nyújt. A tervezés (planning) funkció egy természetes alkalmazása a *BI*-nak, hiszen a vállalati tervtáblák egy az egyben olyanok, amit a többdimenziós kockákba le lehet képezni. Az adatbányászat érdekes lehetőséget ad a vezetői döntésekhez, kutatásokhoz. Különféle matematikai (például regresszió számítás) és nem matematikai eszközökkel mutatja az OLAP adatbázist, amiben nem triviális összefüggések megtalálását segíti.

EPM: A vállalati teljesítménymenedzsment rendszerek új szemléletű megoldások, amik KPI-kat definiálnak, figyelnek. Képesek ezen mutatószámokat hierarchikusan is kezelni, emiatt mindhárom döntési szintet átfogják. Jelenleg még nem annyira szabványosak, mint a kiérlelt korábbi rendszertípusok, amiatt még további jelentős fejlődés várható ezen a fronton.

KM: A tudásmenedzsment rendszer egy gyűjtőfogalom, különféle altípusai vannak,

amikben az közös, hogy a vállalati tudás összegyűjtése és szétosztása az alapfeladat. Nem kötődik egyik vezetői szinthez sem.

ES: A szakértői rendszerek (Expert System) végigkísérik a teljes informatika történelmet, a kor adta lehetőségeken belül mindig az volt a feladatuk, hogy valamely speciális szakmai terület munkájához adjon informatikai támogatást (példa: szeizmika az olaj-kutatásban). Itt a mesterséges intelligencia is komoly szerephez jut, tényeket és szabályokat tárol, ezekből következtetéseket von le.



8. ábra. SAP modulok

A továbbiakban néhány érdekesebb gondolatot szeretnénk összefoglalni az egyes főbb rendszertípusokról, kitérve az új kihívásokra is, amikre az integráció adja meg az egyik lehetséges választ.

5.2. Operatív szintű információs rendszerek

Ezen rendszerek a vállalat funkcionális területeinek (marketing, gyártás, pénzügy, könyvelés, HR) napi munkáját támogatják. Nézzük meg ezek közül a legfontosabb részterületeket, amik csak kiragadott példák!

Marketing: Értékesítés, piackutatás, eladások elemzése, előrejelzése, piacbefolyásolás (hirdetések).

Gyártás: készletgazdálkodás (nyersanyag, félkésztermékek, késztermékek), termeléstervezés, termelésirányítás, minőségellenőrzés.

Könyvelés: vevő és szállító oldali számlák kezelése, naplófőkönyv, bérszámfejtés.

Pénzügy: pénzügyi tervezés, ellenőrzés, előrejelzés, készpénzforgalom (készpénz és hitelgazdálkodás) menedzsment, költségvetés.

HR: munkakör tervezés, felvétel és elbocsátás, bérelszámolás, továbbképzések kezelése, a munkaerő ráfordítások hatékonyságának elemzése, tervezése.

Az operatív szintű rendszerek emiatt általában *TPS, MIS, ERP, CRM, SRM és SCM* rendszerek.

A kifejezett cél az, hogy az egyes területeken jelentkező nagyszámú üzleti eseményeket biztonságosan kezeljék. Az üzleti események rögzítése maguk is tranzakciók, azonban a tárolt adatok feldolgozása (például a vevőszámlák elkészítése) is tranzakcionális. Régebben a kötegelt (batch) feldolgozás volt jellemző, mára egyre elterjedtebbek a valós idejű (real-time) feldolgozások. A *TPS* rendszerek adatmodellje tipikusan 2 féle adatot tartalmaz: törzs (egyik típusa a paraméter táblák) és tranzakcionális adatokat.

A *MIS* rendszerek elsősorban a *TPS* adatbázisokra épülve az operatív szintű döntésekhez adnak segítséget. A *TPS* az adatok biztonságos kezelését és feldolgozását oldják meg, így az adatokból való információk kinyerésével csak a legszükségesebb mértékben foglalkoznak. Ezt a hiányt pótolják a *MIS* rendszerek, amikben a riportozási feladatokat lehet megoldani. Jellemzője, hogy a riportok statikusak, nem biztosítják az alternatívák lekérdezésének dinamikus lehetőségeit.

Az *ERP* rendszerek a *TPS* és *MIS* megoldások egy „nagy” közös csomagba történő továbbfejlesztései, ahol az új paradigma az integráció. Lefedi a *TPS* rendszereknél már említett funkcionális modulokat, közöttük interface kapcsolatokat alakít ki. Ezzel párhuzamosan a *MIS* igényeit is igyekszik – kezdetlegesen szinten – kielégíteni.

A *CRM* és *SRM* rendszerekről már volt szó. Az *SRM* több fontos részfeladatot megold a vállalat üzleti körforgásában: beszállítók értékelése (felkutatása), a beszállítási feltételek menedzselése, keret-szerződések megkötése, ajánlatkérések és azok kiértékelése, a beszerzési workflow, beérkeztetés.

Az *SCM* megoldások kezelik az ellátási láncban lévő vevő-szállító kapcsolatokat, a láncban lévő vállalatok közötti termelési és logisztikai koordinációt.

5.3. Döntéstámogató rendszerek

A DSS és GDSS rendszerek tartoznak ide, amik különféle döntési modelleket ismernek, mindehhez adatbázisokat használnak. A döntéshozó saját ítéletalkotó képességét egészítik ki ezek az interaktív szoftverek, amik képesek segíteni a programozható (példa: lineáris programozás) vagy csak részben programozható döntések meghozatalában. A *DSS* a taktikai és startégia szint – nem jól strukturált – döntéseit támogatja (példa: milyen befektetési csomagot vá-

lasszunk?). A DSS rendszerek ugyanakkor képesek alacsonyabb szintű döntések támogatására is, erre jó példa egy fuvarszervező program. Egy *DSS* rendszer igazi értéke az a modell, ami jó működést eredményező válaszokat adhat kérdéseinkre. A *GDSS* rendszerek a modelleket csoportok számára teszik elérhetővé, leküzdve ezzel a térbeli és időbeli távolságokat is.

5.4. Az üzleti intelligencia

Az adatbázisok növekedtek és egyre világosabb lett az a felismerés, hogy a vállalat adatai a vállalati vagyon része, amit ki kell használni. Ezzel párhuzamosan jelentkezett egy új fogalom: *diszparát adat*. Alapvetően a vállalati adatokkal több probléma is van, ami nehezíti az adatvagyon jó „forgatását”:

- az adatok formátuma és jelentése változó
- az adatok pontossága nem ismert, kezelésükre nincsenek mindig világos szabályok
- az adatok időszerűsége nem megfelelő
- az adatok egyes részei csak különböző adatbázisokból szedhetők össze
- az adatok nem „tiszták”, azaz azokat logikai elemzésnek alávetve inkonzisztenseknek találhatjuk
- egy adat egzisztenciája (létezik-e) is egy fontos megoldandó feladat

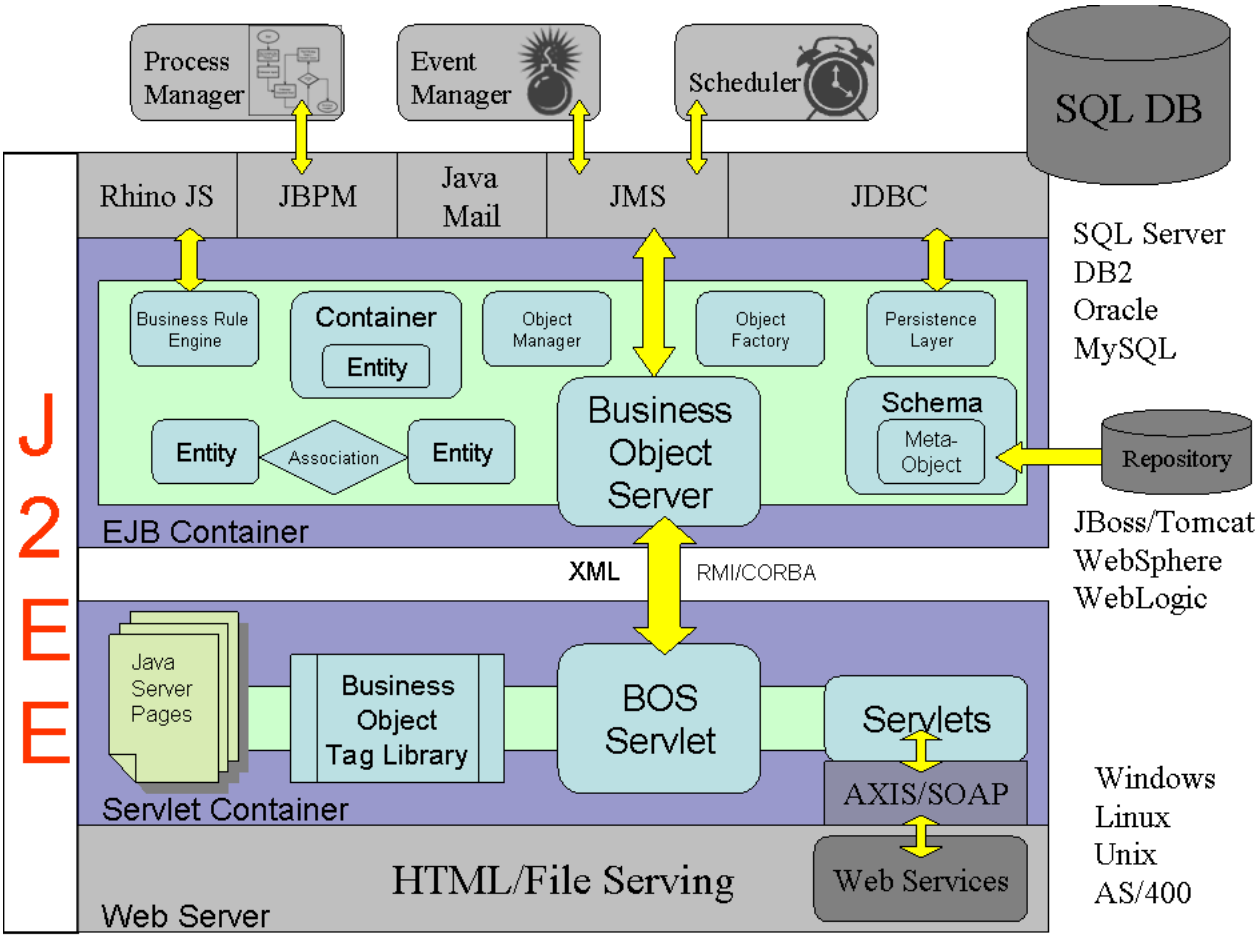
Az információs adatbázisok jelenleg legfejlettebb formája az adattárház (DW – Data Warehouse) koncepciója. A DW adatbázisok már jó minőségű és feldolgozott adatokat tartalmaznak OLAP kockákban. Ezt az adatraktározást persze nem kapjuk ingyen, a szakirodalom szerint a következő jól megtervezett lépések szükségesek hozzá, amit a már korábban is említett *ETL* kifejezéssel szoktunk illetni:

- az adatok összegyűjtése különböző forrásokból
- másolat készítése az adattisztás előtt
- adattisztítás, a tisztított adatok tárolása az előre definiált struktúrákba
- katalogizálás

A jó minőségű DW adatbázisokra építve a mai BI rendszerek (9. ábra) már nagyon fejlett információ kinyerési technikákat alkalmaznak, amit *reporting*-nak nevezünk. Érdekes lehetőség az un. adatbányászat. Itt az elemzőnek van egy hipotézise, ennek megfelelően megfogalmaz egy kérdést. Az adatbázis vizsgálatával 2 kimenetele lehet a hipotézisnek: elvetjük vagy elfogadjuk. A *BI* szoftverekben a klasszikus *DSS* és *GDSS* eszköztárak is megtalálhatóak.

Tágabb értelemben a 2000-es években megjelent *EPM* rendszereket is a *BI* részeként szokták tekinteni, hiszen hozzájárul az adatok elemzésével a teljesítmények elemzéséhez. A terv és tényadatok kezelésével a *BI* rendszerek a controlling kiváló támogatói. Az *EPM*-ben is előre tervezett KPI-ok vannak, amiket a mért adatokkal lehet összevetni.

Az *EIS* rendszerek is integrálódtak a *BI*-ba, azaz a reporting funkcióknak van egy olyan felülete, ami kifejezetten a felső vezetők igényeinek kiszolgálását célozza meg.



9. ábra. A Business Objects felépítése

5.5. Tudásmenedzsment

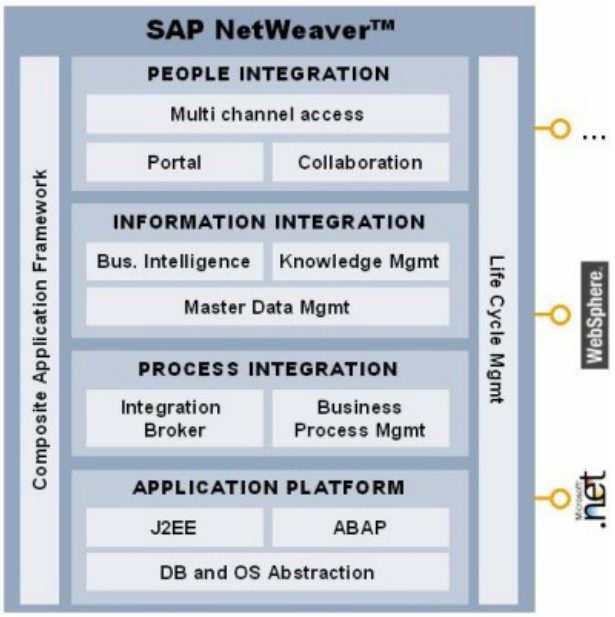
A tudásmenedzsment rendszerek mások, mint az eddig tárgyaltak, azok nagy része nem is a *TPS* adatokra épülnek. Ezek a programok támogatják az információk összegyűjtését, tárolását, szétosztását. Egyik fontos eszköztára a kollaboratív tartalom-menedzsment (*CM*=Content Management). Manapság a portál technológiák felé tolódik a hangsúly, erre a 4.6 pontban térünk ki.

A tudásmenedzsment részeként tekintik a szakértői rendszerek bizonyos moduljait, a mesterséges intelligencia megoldásokat (kézírás felismerés, elektronikus dokumentumfeldolgozás és scannelés, robottechnológia, stb...) és a már említett felsővezető ES rendszereket is.

5.6. Az integráció szerepe

Az eddigiekben sokféle információs rendszertípust vizsgáltuk meg röviden. Ugyanakkor ezek a rendszerek térben szétosztva, különféle üzleti tulajdonosok számára működnek. Az alkalmazott technológiák száma is sokrétű, néha már aligalig tudunk kiigazodni a technológiák „dzsungelében”. Mi az ami ebben a szituációban minden vállalatnál előbb-utóbb jelentkezni fog? Az integráció. Ezek a megoldások ugyanis nem a „holdon”, önmagukban működnek, hanem egy-

egy konkrét üzleti környezetbe kell ágyazni őket. Az integrációnak több aspektusa van, röviden nézzük meg őket!

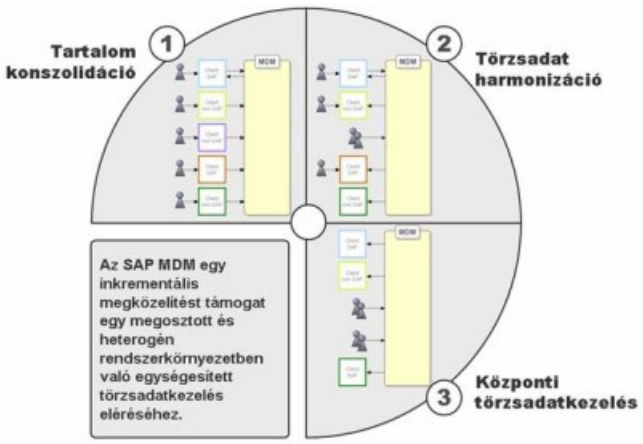


10. ábra. SAP NetWeaver

A rendszerek közötti kommunikáció (EAI). A 2000-es évek elején jelent meg az *EAI* (Enterprise Application Integration) fogalma. Addig a rendszerek között olyan alacsony szintű volt az adatkommunikáció mennyisége, hogy ilyen eszközre nem volt szükség. A mostani fejlett informatika kiköveteli, hogy erre külön eszközöket alkalmazzunk. A feladat technikailag nehéz, jelenleg az informatikai piacon ezen tevékenységnek az egyik legmagasabb a díja. Manapság egyre inkább üzenetkezelő rendszereket és adaptereket alkalmazunk ezen feladat megoldására. Az EAI rendszerekkel a vállalatoknál is tért hódít az elosztott számítástechnika, ami a tranzakciókezelés egy magasabb szintjét is jelenti (elosztott tranzakciókezelés). A technikai nehézségek közül csak egy példát szeretnék kiemelni. Tegyük fel, hogy A és B rendszerek (például adatbázisuk van) is képesek tranzakcióban működni. Amikor azonban A→B között adatmozgás akarunk megvalósítani, akkor a COMMIT műveletét egyszerre kell kiadni A-ra és B-re, így a hagyományos megközelítés nem alkalmazható. Miért? Amennyiben külön COMMIT van A-ra és B-re, akkor vagy adatot vesztünk, vagy duplázzuk azt.

MDM. A Master Data Management. A vállalatok gyakran több különböző, esetleg össze sem kapcsolt informatikai rendszerben, adatbázisban tárolják adataikat. Ennek eredményeképpen a vállalat tevékenységéhez kapcsolódó törzsadatok nincsenek összhangban, ábrázolásuk pontatlan, redundáns, és az egyes példányok között eltérések is lehetnek. A probléma főként a termékorientált, azaz termelő vagy kereskedő vállalatok esetében jelentős, hiszen fő értékteremtő folyamatuk a termék köré szerveződik, így bármely pillanatban naprakész termékkatalógussal kell rendelkezniük. Ez mind a belső szereplők (beszerzés, gyártás stb.), mind a külső szereplők (szállítók, vevők stb.) számára alapvető fontosságú. Egy jó MDM szoftver az említett problémák kezelése érdekében lehetőséget biztosít a következőkre:

- általános és rugalmas adatmodell kialakítására
- törzsadatok importálására különböző forrásokból
- adatok konszolidálására (duplikátumok keresése, hasonlóságok keresése, transzformációs szabályok megadása által)
- törzsadatok szinkronizálására különböző rendszerek között
- könnyen használatba vehető, skálázható megoldás kialakítására
- az adatkarbantartási költségek csökkentése



11. ábra. Az MDM felépítése

A törzsadatkezelési forgatókönyv részei:

- Törzsadat-konszolidáció (Master-data consolidation). Egyező törzsadatobjektumok azonosítása különböző informatikai rendszerekben, a megfelelő objektumok központi egyesítése és a megfelelő kulcsleképezések létrehozása a további üzleti folyamatokban történő felhasználás céljából.

- Törzsadatok harmonizálása (Master-data harmonization). A cikkek, szállítók, ügyfelek stb. törzsadatainak tisztítása, rendbe tétele, majd ezek visszajuttatása a kliensrendszerekbe, az üzleti folyamatok feljavítására a jobban használható törzsadatok révén.

- Központi törzsadatkezelés (Central master-data management). A konszolidáció után a törzsadatok központilag kezelhetők, a logikailag összetartozó objektumok és attribútumaik változatásai és mozgatásai is egyszerre, egy helyen végezhetők el. Mindez az erre vonatkozó házirendek és vállalati szabványok felállítása után az újonnan létrejövő törzsadatok egységes, szabályozott kezelését is jelenti.

Portál. A rengetek alkalmazás, az információk és a hírek elérése komoly kihívás elé helyezi a felhasználókat, így gyorsan megfogalmazódik a User Integration, azaz a portál igénye. Itt egységes felületen lehet igénybevenni (vagy legalábbis az elérését kezdeményezni) a vállalati informatikai szolgáltatásokat. A tudásmenedzsmentnek megjelennek az új perspektívái: wiki, blog.

BPM (Business Process Management). A vállalat egyik értéke az üzleti folyamatai. A gyakorlatban olyan üzleti folyamatok is megfogalmazódnak, amik rendszereket és szervezeti egységeket (sőt SCM esetén vállalatokat is) átfognak. Egyre jobb BPM eszközök jelennek meg, amik új, érdekes kiegészítő szolgáltatásokkal is rendelkeznek. Ilyen például a *Business Rules Engine*, ami lehetővé teszi az üzleti szabályok kiemelését az egyes alkalmazásokból, így azokat a továbbiakban több alkalmazás is használhatja. Az előny nyilvánvaló, csak két fontosat említünk meg: költségtakarékosság az újrahasznosítás miatt, de talán ennél is fontosabb az üzleti szabályok konzisztenciája.

B2B. A vállalatok vagy azok ügyfelei közötti kommunikáció alapjait valósítja meg, legfontosabb szolgáltatásai: titkos és megbízható csatorna az Internet és intranet között, digitális aláírás, időpecsét, archiválás.

5.7. Irodaautomatizálás

Szinte mindenki találkozik ezekkel az eszközökkel, aki irodában dolgozik. Főbb témái a dokumentum menedzsment köré csoportosulnak:

- Az iratok és dokumentumok rendszerezett, biztonságos központi tárolása és visszakeresése

- üzleti és szervezeti folyamatok automatizálása, szabályozása (dokumentum központú munkafolyamatok)

- az egész cég egyesített információs, dokumentációs adatbázisának létrehozása
- a vezető hatékonyságának emelése, áttekintési és ellenőrzési lehetőségeinek többszörözése.

Ezekre a feladatokra alkalmas például az SPS+Nintex páros (`http://www.nintex.com/en-US/Pages/default.aspx`).

5.8. A jövő - ami már itt van (SOA, Cloud Computing)

A cloud computing mindmáig a kis cégek körében volt népszerű, de a globális vállalatok és az egyéb nagy szervezetek félreteszik majd a gátlásaikat és belevágnak a cloud-szolgáltatások használatába, hiszen mostmár megtehetik, megéri nekik. A vállalati szintű menedzsment eszközök, az adatintegrációs technológiák elérhetővé váltak, és válnak majd egyre gyorsabban. Az előfizetéses árazás ráadásul sokkal szimpatikusabb a cégek számára, mert jobban testreszabható, pontosan annyit kell rászánni egy feladatra, amekkora igény van a cégen belül, nem kell felesleges kapacitásokat telepíteni.

A cloud, azaz felhő, az internet metaforája, arra utalva ahogyan ábrázolni szokás a világhálót diagramokon, illetve kifejezve azt a komplexitást, amivel a modern hálózatok rendelkeznek. A cloud computing pedig az internetes felületen alapuló technológiai megoldások, melyek lényege a valós-időben skálázhatóság, a szoftverek "szolgáltatásként" való megjelenítése, illetve a gyakorlatban eddig csak desktop verzióban elképzelhető alkalmazások átköltöztetése a böngészőablakba.

A "Software as a Service" (SaaS), a web 2.0 és egyéb jelenleg jól csengő technológiai trendek befogadásából és a meglévő rendszerekbe illesztéséből összeálló egésznek a célja, hogy különösebb erőforrások és befektetések nélkül folyamatosan ki lehessen elégíteni a felhasználók igényeit. Cél, hogy eközben a lehető legkevesebb helyen kelljen belenyúlni a helyi, meglévő rendszerekbe, illetve, hogy a felhasználók ne szoruljanak oktatásra.

A legismertebb példa a cloud computingra talán a Google Apps, ahol a levelek, fotóalbumok és egyéb információk, tartalmak a központi szervereken tárolódnak, böngészőn keresztül érhetőek el, de a funkcionalitás inkább egy asztali szoftverére emlékeztet.

6. SOA – a jövő üzleti és architekturális modellje

A cégek áttekinthetetlen informatikai rendszereire a megoldás nem az IT-költségvetés növelése, hanem a szemléletváltás az irányításban. Az elmúlt időszakot az jellemezte, hogy a vállalatok az egyes üzletágak bizonyos feladatainak ellátására vásároltak rendszerek, amelyek aztán IT-részleg többnyire csak üzemeltetett. Így alakultak ki a szigetrendszerek vagy silók. Ezek a rendszerek különböző platformokra, különböző technológiával készültek, és adott üzleti területeket csak a "bedrótozott" folyamat szerint támogatnak, nincsenek felkészítve a folyamatok módosítására és más üzleti folyamatokhoz kapcsolására.

A szükségessé váló alkalmazásintegrációt többnyire pont-pont alapon, sokféle egyedi technikával oldották meg, tovább növelve a komplexitást. De nemcsak a funkciók és folyamatok vannak silókba zárva, hanem az adatok is, amelyek heterogén formájúak és inkonzisztensek, ami rendkívül megnehezíti azok egységes vállalati szintű kezelését.

Ebben a környezetben a "visszapillantó tükör" jellegű IT-költségvetés a jellemző: a költségek nagy része az üzemeltetésre és karbantartásra megy el, kevés jut stratégiai célokra, a versenyképességet javító új fejlesztésekre. Ugyanakkor napjaink üzleti elvárásai lényegesen megváltoztak az IT-vel szemben.

Mit várunk az IT-től?

- Gyors, rugalmas reagálást a piaci változásokra (time to market).
- Vállalati szintű és a vásárlókra, partnerekre kiterjesztett folyamatok automatizálását/informatikai támogatását, pl. az elektronikus értékesítési csatornák kiszolgálását.
- Átlátható, rugalmasan alakítható, mérhető üzleti folyamatok kialakítását.
- Egységes vállalati adatnézetek és törzsadatok biztosítását.
- Az IT-költségek szinten tartását és ésszerűbb felhasználását
- Az IT-beruházások üzleti prioritások szerinti megvalósítását
- Meglévő rendszereinket a kockázatok és költségek miatt csak indokolt esetben cseréljük le.
- Nem akarunk hosszú ideig tartó, kockázatos, nagy átalakításokat.

Ezek az elvárások erősen függenek az IT rugalmasságától, hatékonyságától, és jelzik, hogy a meglévő architektúrával, illetve irányítási megközelítéssel ezen üzleti célok nem érhetők el. A problémák elsősorban a folyamatok, alkalmazások és adatok integrációja, valamint a fejlesztés és üzemeltetés hatékonysága körül jelennek meg. Mivel az üzleti igények általánosak és sürgetők, az IT-iparág kialakította választ: a SOA (Service Oriented Architecture), magyar terminológiával Szolgáltatás Alapú Architektúra, az a megközelítés, ami a fenti elvárásokat biztosítani képes, és az informatikai ipar fejlődésének hajtóerejévé vált.

A SOA informatikai stratégia

Technikai megközelítésben a SOA egy architektúra és technológiai modell, amely lehetővé teszi, hogy különböző alkalmazásokban (alkalmazáscsomagok, egyedi fejlesztésű alkalmazások stb.) megvalósított különálló üzleti és technikai funkciókból, szolgáltatásokból üzleti folyamatokat

megvalósító alkalmazásokat állítsunk össze. Tágabb megközelítésben a SOA informatikai stratégia, amely irányítja az egységes SOA alapú vállalati architektúra kialakítását és a vállalati alkalmazásokban található diszkrét üzleti és technikai funkciók együttműködő, szabványokon alapuló szolgáltatásokká szervezését, azaz meghatározza az IT működési és megvalósítási folyamatait és előírásait.

A SOA stratégiának két, egymással szorosan összefüggő része van, a szolgáltatás-infrastruktúra (SOA architektúra) vagy másképpen a működtető technológia és a SOA irányítás (SOA Governance).

Szolgáltatás-infrastruktúra

A SOA nyílt, szabványos komponens technológia, amelynek építőelemei a szolgáltatások. A szolgáltatások önállóan is működőképesek, platform- és eszközfüggetlenek (tetszőleges technológiával készülhetnek), szabványos, jól definiált interfésszel rendelkeznek, és szabványos adatcsere- és kommunikációs protokollokkal érhetők el az elosztott hálózatokban.

A szolgáltatások két alapvető típusát különböztetjük meg. Az üzleti szolgáltatások teljes vagy rész üzleti funkciókat valósítanak meg (ügyfélkezelés, számlakezelés stb.). Az üzleti szolgáltatások megvalósításához az egységesítés és újrafelhasználás érdekében nélkülözhetetlenek a technikai szolgáltatások, amelyek alacsonyabb szintű építőelemek, és az üzleti felhasználók számára nem láthatók. Technikai szolgáltatás például a naplózás, az archiválás, a dokumentumtárolás, a megjelenítési szolgáltatások stb. A szolgáltatások összekapcsolásából úgynevezett kompozit alkalmazásokat építhetünk kódolással vagy dinamikus láncolással, utóbbit üzleti folyamatkezelő eszközök segítségével.

A szolgáltatásokra alapozott alkalmazások megvalósításához azonban nem elegendő a komponens alapú architektúra - még akkor sem, ha a szolgáltatások mint üzleti logika egységek vannak megvalósítva - mert egyrészt a nagy számú szolgáltatás tervezési és végrehajtási szintű kezelése speciális megoldásokat igényel, másrészt a hagyományos komponenstechnológiában a folyamatvezérlés, az integrációs mechanizmusok és a szükséges adattranszformációk bele vannak kódolva a komponensekbe. Komoly gondok forrása az egységes, közös törzsadatok hiánya is. Ezért a Szolgáltatás Alapú Rendszerek kialakításához és működtetéséhez az erre a célra szolgáló SOA architektúrára van szükség.

A SOA infrastruktúra részei

Szolgáltatások: a SOA alapú IT-működés esetén a fejlesztők a projektekben monolit alkalmazások helyett önállóan is életképes technikai és üzleti szolgáltatásokat hoznak létre, alakítanak ki, és azokat szabványos interfészen keresztül publikálják. Az alap üzleti és technikai szolgáltatások összekapcsolásával építhetők fel az üzleti folyamatokat támogató összetett, illetve folyamatcentrikus szolgáltatások.

Folyamatszolgáltatások (Business Process Management): a folyamatok vezérlését, automatizálását megvalósító szolgáltatások, amelyeket a SOA architektúrában üzleti folyamatvezérlő eszközök biztosítanak, amelyek lehetővé teszik a humán feladatok támogatását (workflow) és a folyamatok automatizálását a folyamatelemeket megvalósító szolgáltatások láncolásával, továbbá lehetővé teszik a folyamatok szüntelen monitorozását. A BPM motorok és az üzleti aktivitást monitorozó modulok (BAM) a SOA architektúra fontos elemei, hiszen a SOA megközelítés alapvető célja a rugalmas, könnyen módosítható üzleti folyamatok megvalósíthatóságának a támogatása.

Üzletialkalmazás-szolgáltatások: az újonnan fejlesztett üzleti szolgáltatások futtató környezete. Célszerűen Java J2EE vagy .NET alkalmazásszerveren futó szolgáltatások, de min-

den más olyan környezet is lehet, amelyben lehetőség van szabványos interfészekkel és kommunikációs felülettel szolgáltatásokat publikálni.

Szolgáltatásbusz (ESB): a SOA alapú IT-infrastruktúra idegrendszerként funkcionál; egységes, központilag menedzselt kommunikációs és biztonsági szolgáltatásokat nyújt a szolgáltatások és SOA infrastruktúra elemeinek az összekapcsolására, valamint a szolgáltatáshoz kapcsolódó szabályzatok és nem funkcionális követelmények érvényesítésére.

Szolgáltatástár (Registry-Repository): a SOA alapú IT-működés és irányítás a szolgáltatásdefiníciók és a kapcsolódó követelmények központi tárolásával lehetővé teszi a szolgáltatás tervezési és fejlesztési ciklusának hatékony menedzselését, beleértve a publikálást, újrafelhasználást, a függőségek, verziók és követelmények kezelését.

Elérési (access) szolgáltatások: azokat a technikai szolgáltatásokat soroljuk ebbe a kategóriába, amelyek segítségével biztosíthatjuk a meglévő alkalmazáscsomagokban (ERP, CRM, SCM stb.) és egyedi fejlesztésekben található funkciók szolgáltatásként való publikációját.

IT Service Management: monitorozza, menedzseli és biztosítja a szolgáltatásokat, alkalmazásokat és erőforrásokat. A SOA alapú IT-működés nagyon fontos része, hogy a szolgáltatások különböző aspektusait mérni tudjuk, beleértve a szolgáltatás felhasználásának mértékét, a nem funkcionális követelmények teljesülését, az adatforgalmat, a SOA infrastruktúra komponenseinek a rendelkezésre állását stb. A folyamatos mérések lehetővé teszik a hibák gyors elhárítását és a hatékony üzemeltetést, sőt az üzleti folyamatok optimalizálását is.

Üzleti monitorozás, dashboard: a SOA megközelítés egyik alappillére a rugalmas, átlátható és mérhető üzleti folyamatok biztosítása. Ezért a SOA architektúra az üzleti vezetés számára is biztosít üzleti szemléletű monitorozási és beavatkozási felületeket, ahol az üzleti teljesítménymutatók figyelemmel kísérhetők, és lehetőség van a beavatkozásra.

Front-end rendszerek, a szolgáltatások felhasználói: a szolgáltatásokat különféle alkalmazások használhatják, amelyek közül a legfontosabbakat emeljük ki. A Business Process Management Solution (BPMS) segítségével szolgáltatásokból üzleti folyamatokat lehet felépíteni akár az IT bevonása nélkül is - így ezek a folyamatok rugalmasan, gyorsabban és kisebb költséggel módosíthatók, hiszen a központi folyamatvezérlés megszünteti a fejlesztés és működtetés redundanciáit. Az egységes vállalati intra/internet Portál biztosítja az alkalmazottaknak és az ügyfeleknek az egységes, de személyre és szerepkörre szabható, több csatornán elérhető felhasználói felületet. A B2B/B2C megoldásokkal a szolgáltatások biztonságosan elérhetővé tehetők az üzleti partnereknek és ügyfeleknek. A felhasználói felületek kialakítását a prezentációs és kollaborációs szolgáltatások teszik gyorsabbá és egységesebbé.

SOA-irányítás

A SOA infrastruktúra, a szolgáltatások és az ezekre épülő üzleti alkalmazások megtervezése, megvalósítása és működtetése a korábbi silók üzemeltetéséhez képest gyökeresen új megközelítést igényel. A SOA előnyei csak megfelelő irányítás mellett érhetők el. A SOA-irányítás az IT-irányítás kiterjesztését jelenti a szolgáltatások teljes életciklusára a tervezéstől a megvalósításon és tesztelésen át az üzemeltetésig. A SOA-irányítás feladata nem a silók, hanem a szolgáltatások és a szolgáltatásokból épülő alkalmazások fejlesztése és üzemeltetése, másrészt a szigetszemlélet helyett a teljes IT-infrastruktúrában, szolgáltatatás- és alkalmazás portfólióban mint egy virtuális egészben kell gondolkodni. Ugyanez igaz az üzleti folyamatokra is, a helyi üzletági folyamatok mellett a teljes vállalaton átívelő és sok esetben azon

túlnyúló folyamatokban kell gondolkodni.

Az SOA-irányítás fő alkotóelemei a következők:

SOA irányítási modell: meghatározza a szerepköröket, a döntési jogokat, a kapcsolódó mérési és ellenőrzési mechanizmusokat és folyamatokat, amelyek az IT-döntések előkészítéséhez, meghozatalához és végrehatásához szükségesek. A döntési és végrehajtási mechanizmusuk szabályzatokon (policy) alapulnak, amelyek meghatározzák pl. az alkalmazások architekturális szabályait vagy a csomagalkalmazások kiválasztási szempontjait. Az irányítási modellnek fontos eleme a projektfinanszírozási szabályzat, valamint a folyamattulajdonosok és -szponzorok meghatározása. Az irányítási modellben el kell helyezni a SOA kompetenciaközpont feladatait és felelősségi körét is.

SOA referencia-architektúra: ez írja le teljes vállalati SOA architektúra tervét teljes kiépítés esetén. A referenciamodell meghatározza az alkotóelemeket, azon belül is a modulok szétválasztását és kapcsolódását, ami lehetővé teszi a SOA infrastruktúra lépcsőzetes, inkrementális fejlesztését. Az izoláció és a kapcsolódás az ESB-n keresztül történik. Mivel a referenciaarchitektúra kialakítása hosszú távra szól, a beruházás értékállósága és későbbi módosíthatósága végett célszerű nyílt szabványokat alkalmazni. A SOA architektúra komponens alapú fejlesztést tesz lehetővé oly módon, hogy a komponensek, azaz szolgáltatások üzleti feladatokat valósítanak meg, a folyamatvezérlés, biztonsági elemek és az integrációs/kommunikációs funkciókat nem kell és nem is szabad a szolgáltatásokba bekódolni, azokat egységes, újrafelhasználható módon a SOA infrastruktúra biztosítja.

SOA ütemterv (Roadmap): ez a a SOA infrastruktúra kialakításának terve, amely a referencia-architektúra teljes kiépítéséhez szükséges lépéseket és azok várható ütemezését tartalmazza. Fő területei: alap infrastruktúra (ESB, BPM, Registry stb. bevezetése), üzleti funkció típusú szolgáltatások (pl. hitelezési vagy ügyfélszolgálati folyamatok elemei), információelérési szolgáltatások (pl. központi dokumentumkezelő szolgáltatásainak publikálása), közös IT-szolgáltatások (egységes logolás, felhasználó azonosítás stb.). A referencia-architektúrát és az ütemtervet az üzleti célok és megszerzett tapasztalatok alapján rendszeres felül kell vizsgálni!

Metrikák: a SOA metrikák és siker kritériumok felállítása és bevezetése vállalatonként változó, egyedi és nem feltétlenül egyszerű feladat. A SOA sikerességét három fő szempont alapján lehet talán a legjobban mérni: üzleti előnyök, IT-előnyök, továbbá szolgáltatások és kapcsolódó erőforrások újra felhasználásának a szempontjából.

Az irányítási modell és az ütemterv tervezéséhez el kell végezni az aktuális IT-környezet és a vállalati üzleti folyamatok nagyvonalú felmérését, valamint az aktuális IT-irányítási képességek és irányítási struktúra részletes felmérését és figyelembe kell venni a vállalat üzleti terveit és prioritásait.

A SOA bevezetésének egyik leginkább hangsúlyozott előnye az újrafelhasználhatóság. Ugyanakkor fontos megjegyezni, hogy nem minden vállalatnak egyforma fontosságú és jelentésű az újrafelhasználás. Ez jelentheti egyrészt a meglévő alkalmazás silókba zárt szolgáltatások újrahasznosítását, illetve az újonnan kifejlesztett szolgáltatások, törzsadatok képességét arra, hogy későbbi projektekben újra lehessen hasznosítani őket. Mindkét esetben igaz, hogy az újrafelhasználást igen jól lehet mérni: hány projektben lett hasznosítva egy adott szolgáltatás, hány meglévő szolgáltatást sikerült a SOA rendszerbe integrálni és felhasználni.

A SOA másik, sokszor emlegetett előnye az agilitás, amely már sokkal absztraktabb, nehezebben mérhető fogalom. Az agilitásnak sok oldala van, ezekből talán a következők a legfontosabbak:

- Üzleti vagy IT-igény megvalósításához szükséges idő.
- SOA rendszer toleranciája; üzleti igények szintje, amelyeket a SOA, illetve IT-architektúra jelentős változtatása nélkül meg lehet valósítani.
- Az üzleti igények száma, amelyeket a SOA rendszer képes gyorsan kielégíteni.

A fejlesztési és az üzemeltetési költségek csökkenése azok a további területek, ahol az előnyök megjelennek. Itt az összehasonlíthatóság érdekében a korábbi és az új projektekben is szükség van az összehasonlításhoz megfelelő részletezettségű ráfordításmérésekre, ami általában a korábbi időszakokra nem áll rendelkezésre, de bevezetésük is ellenállásba ütközik.

Befejezésként a SOA alapú IT-működést szeretnénk érzékeltetni néhány fontos kérdés kiemelésével. A SOA architektúra és a szolgáltatások kialakítása lépésekben, az üzleti prioritásokat figyelembe vevő, a szolgáltatásokat felhasználó üzleti projektekben történik. A projektek tervezését és irányítását SOA-irányításban meghatározott folyamatok mentén és az érvényes szabályzatoknak megfelelően kell végrehajtani. A tervezés során, a tervezés alatt álló projekt igényei, a SOA ütemterv és a rendelkezésre álló szolgáltatás katalógus alapján a felelős üzleti és IT-szakértőknek el kell dönteni, hogy a projekt során milyen SOA infrastruktúra- és irányítási elemeket kell bevezetni vagy módosítani, milyen új üzleti és technikai szolgáltatásokat fejlesztenek ki és milyen meglévő szolgáltatásokat használnak fel a projektben készülő alkalmazáshoz. Ugyancsak határozni kell a projektfinanszírozásról. A projekt során újonnan fejlesztendő közös szolgáltatásokat lehet pl. az üzleti projekt költségvetéséből vagy részben/teljes egészében központi SOA költségvetésből finanszírozni. A SOA infrastruktúra- és irányítási elemek bevezetését a SOA kompetenciaközpont végzi. Ugyancsak az ő feladatuk az üzleti projektekben folyó szolgáltatásfejlesztés ellenőrzése és szakmai támogatása.

A SOA előnyei

- Az üzlet és az IT hatékonyabban képes együttműködni: az IT hatékonyabban és gyorsabban ki tudja szolgálni a gyorsan változó üzleti igényeket. Az üzleti folyamatokban és szolgáltatásokban gondolkodás biztosítja a közös nyelvet az Üzlet és IT között.
- Átfogó, a teljes vállalati IT-t és üzleti folyamatokat egységben kezelő, az üzlettel szorosabban együttműködő IT-irányítás alakítható ki.
- Rugalmas, átlátható, mérhető, monitorozható vállalati és kiterjesztett üzleti folyamatok alakíthatók ki.
- Rövidül a piacra jutás átfutási ideje (time to market).
- Az IT-alkalmazások konszolidációja jellemzi. A szolgáltatásokba szervezéssel, a törzsadatok egységes kezelésével hosszú távon megszüntethetők a redundáns alkalmazások.
- Inkrementális, üzleti prioritások szerint, lépésenként alakítható ki az új SOA architektúra, a technikai és üzleti szolgáltatások, új üzleti folyamatok, illetve végezhető el az alkalmazáskonszolidáció.
- Csökkennek az üzemeltetési és adminisztratív költségek. Az alkalmazáskonszolidáció miatt kevesebb alkalmazást kell üzemeltetni. Az üzleti folyamatok és törzsadatok, szolgáltatásközpontú megközelítése radikálisan csökkenti az operátorok napi adminisztratív teendőit, illetve egy hiba elhárításához jóval kevesebb időre van szükség.

- Egységes fejlesztési és üzemeltetési módszertant lehet bevezetni, ezáltal mind csökkenthetők a külső és a belső erőforrások, illetve javul a tervezhetőség.
- Szabványok alkalmazása jellemzi. Így lehetővé válik, hogy az egyes szolgáltatásokat a funkcionális és nem funkcionális, illetve egyéb követelményeknek legmegfelelőbb szállító biztosítsa. Emellett elkerülhetővé teszik, hogy a cég csak egy beszállítótól függjön, ráadásul a kiélezett versenyhelyzet jobb és költséghatékonyabb megoldásokat biztosít.
- A fejlesztési költségek csökkentésének három fő forrása van: egyrészt lehetővé válik a kód-újrafelhasználás az üzleti és technikai szolgáltatások újrahasznosításával, másrészt nő a kódolási hatékonyság. A folyamatok vezérlését, biztonsági szolgáltatásokat, a törzsadatkezelést és az integrációs/transzformációs logikát a SOA infrastruktúra biztosítja, és itt jelentkezik a költségcsökkentés harmadik forrása, az egységes fejlesztési módszertan és joggyakorlat.

Kovács András
Igazgató
IQSYS ZRt.

7. Informatikai szilánkok - tippek és trükkök

Az *Informatikai Navigátor* állandó rovata a szilánkok, aminek célja az, hogy minden érdekes, apró vagy elgondolkodtató dolgot feljegyezzen, leírjon. A mindennapok során sűrűn találkozunk egy-egy érdekes, ötletes megoldással, amiket a továbbiakban igyekszünk ebben a rovatban mindenki számára közkinccsé tenni. Minden kedves olvasó saját érdekes feljegyzését is várjuk abban a reményben, hogy egyszer a Te írásod is megjelenik itt. Az írásokat erre az e-mail címre várjuk: *creedsoft.org@gmail.com*.

7.1. Folyamatábrák készítése - a flow értelmező

A tanulmányainkat sokszor szeretnénk ellátni egyszerű, hagyományos folyamatábrákkal. Egy kellemes és kézhezálló eszköz a *flow* program azoknak, akik szövegesen szeretnék leírni a folyamatábrák kinézetét, majd a LaTeXhasználatával legeneráni annak képét. Az eszköz webhelye: `http://www.ctan.org/pub/tex-archive/support/flow/`. Innen letölthető egy teljesen szabványos *C* program, amit minden *C* fordítóval le lehet fordítani. Mi a *gcc*-ét használtuk ubuntu linuxon. A kapott futtatható program arra képes, hogy egy egyszerű flow nyelven leírt folyamatábra definícióból legenerálja azt a LaTeXrészletet, amiből már kép is generálható vagy beszúrható bármely dokumentumba, mint picture környezet. A fenti webhelyen egy tömör és hasznos flow nyelvi dokumentáció is található.

Nézzünk egy egyszerű flow leírást, amit mentsünk el egy *test.flow* file-ba!

```
Right 0
Choice . . N Y
  Kész vagy?
Tag
Choice . . Right Down
  Jobbra vagy Le?
Tag
Right 1
Box
  Jobbra menve...
ToTag
Down
Box
  Lefelé menve...
ToTag
Down
Oval
  STOP

Ezt így fordítjuk le:
./flow test.flow
```

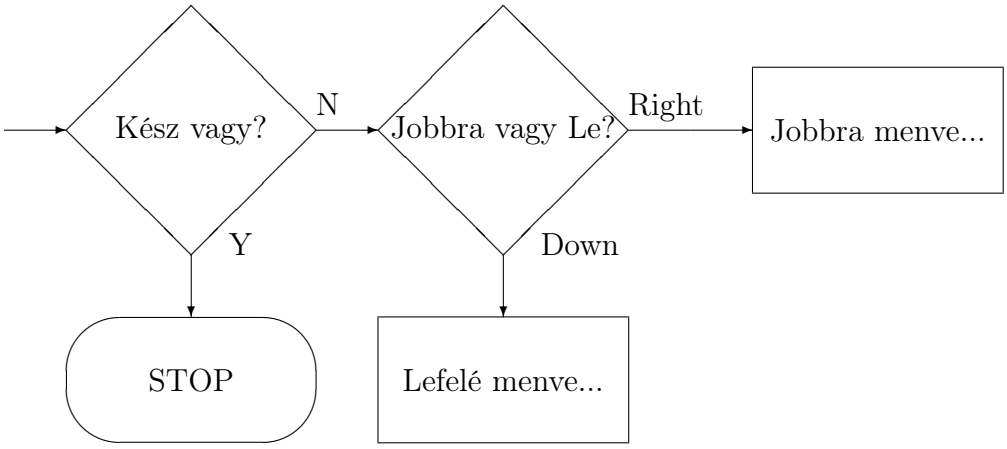
A kimenetre ez a LaTeXprogramrészlet generálódik:

```
\begin{picture}(16.000000,7.000000)(0.000000,-5.000000)
% picture environment flowchart generated by flow 0.99f
\put(0.0000,0.0000){\line(1,0){0.0000}}
\put(0.0000,0.0000){\vector(1,0){1.0000}}
\put(1.0000,0.0000){\line(1,1){2.0000}}
\put(1.0000,0.0000){\line(1,-1){2.0000}}
\put(5.0000,0.0000){\line(-1,-1){2.0000}}
\put(5.0000,0.0000){\line(-1,1){2.0000}}
```

```
\put(1.0000,-2.0000){\makebox(4.0000,4.0000)[c]{\shortstack[c]{
Kész vagy?
}}}
\put(5.0000,0.6000){\makebox(0,0)[lt]{N}}
\put(3.6000,-2.0000){\makebox(0,0)[lb]{Y}}
\put(5.0000,0.0000){\vector(1,0){1.0000}}
\put(6.0000,0.0000){\line(1,1){2.0000}}
\put(6.0000,0.0000){\line(1,-1){2.0000}}
\put(10.0000,0.0000){\line(-1,-1){2.0000}}
\put(10.0000,0.0000){\line(-1,1){2.0000}}
\put(6.0000,-2.0000){\makebox(4.0000,4.0000)[c]{\shortstack[c]{
Jobbra vagy Le?
}}}
\put(10.0000,0.6000){\makebox(0,0)[lt]{Right}}
\put(8.6000,-2.0000){\makebox(0,0)[lb]{Down}}
\put(10.0000,0.0000){\line(1,0){1.0000}}
\put(11.0000,0.0000){\vector(1,0){1.0000}}
\put(12.0000,-1.0000){\framebox(4.0000,2.0000)[c]{\shortstack[c]{
Jobbra menve...
}}}
\put(8.0000,-2.0000){\vector(0,-1){1.0000}}
\put(6.0000,-5.0000){\framebox(4.0000,2.0000)[c]{\shortstack[c]{
Lefelé menve...
}}}
\put(3.0000,-2.0000){\vector(0,-1){1.0000}}
\put(3.0000,-4.0000){\oval(4.0000,2.0000)}
\put(1.0000,-5.0000){\makebox(4.0000,2.0000)[c]{\shortstack[c]{
STOP
}}}
\end{picture}
```

Amennyiben ezt a picture[2] környezetet beszúrjuk dokumentumunkba, megkapjuk a következő kis folyamatábra részletet:

[2]A latex rendszerbe egy kiemelkedő képességű grafikus környezet van beépítve, amit így érünk el.



A *latex2png* paranccsal természetesen egyből png kép is készíthető.

A Terry Brown által készített *flow* generátor dokumentációja (*flowdoc.pdf*) 9 oldal, ajánlom mindenkinek elovasára. Könnyen érthető és mindent tartalmaz, ami egy hagyományos folyamatábra elkészítéséhez szükséges. Bizonyos esetekben sokkal hatékonyabb lehet ezzel a módszerrel folyamatábrákat készíteni, például, ha sok hasonlót kell készíteni vagy nincs annyi időnk, hogy rajzoljunk. A flow nyelve annyira egyszerű, hogy azt egy program kimenetével is generálni lehet, így egy „kettős” generálási eljárással viszonylag jó kinézetű folyamatábrákhoz jutunk, kevés időráfordítással.

7.2. Weblogic ejb[3] készítése NetBeans alatt

Ez a rövid recept azoknak szól, akik ismerik az ejb technológiát és Weblogic környezetben is szeretnék azt használna, miközben egy hatékony java fejlesztőkörnyezetben dolgoznak. A forráskódot *NetBeans* (esetleg *eclipse*) környezetben lehet elkészíteni, a kérdés csupán annyi, hogy mi módon készítsük el a lefordított, telepíthető binárist? Nézzük meg a lépéseket!

(1) New EJB projektet létrehozni NetBeansben.

(2) New message-driven bean létrehozása ebben a projectben (32. programlista)

(3) Az ejb java fájlnak így kell kezdődnie, természetesen valamilyen *xxx* csomagban (a java program első sora: package xxx;)

```
1  // 32. programlista: Message Driven Bean
2
3  import javax.ejb.MessageDrivenBean;
4  import javax.jms.Message;
5  import javax.jms.MessageListener;
6  import javax.jms.TextMessage;
7  import javax.naming.InitialContext;
8  import org.apache.log4j.Logger;
9  import weblogic.ejb.GenericMessageDrivenBean;
10 import weblogic.ejbgen.MessageDriven;
11
12 @MessageDriven(ejbName = "EventProcessor", destinationJndiName = "HU.MOL.EAI.FILE_CONNECTOR_JMSQ",
13 destinationType = "javax.jms.Queue", initialBeansInFreePool = "1", maxBeansInFreePool = "1")
14 public class EventProcessor extends GenericMessageDrivenBean implements MessageDrivenBean, MessageListener
15 {
16    private static final long serialVersionUID = 1L;
```

[3]Enterprise JavaBean

```
17      private static final Logger LOGGER = Logger.getLogger(EventProcessor.class);
18
19      /**
20       *
21       * @param message
22       */
23      public void onMessage(Message message)
24      {
25          LOGGER.info("START: onMessage(...)");
26      }
27  }
```

A fordításhoz *ant* scriptet használhatunk, ami egy olyan taskot használ, amit a weblogic fejlesztői környezet biztosít számunkra. Az *ant build.xml* így néz ki, egy példán keresztül megmutatva:

```
<project name="ejbgen_fileConnectorEjb" default="run-ejbgen">
    <taskdef name="ejbgen" classname="com.bea.wls.ejbgen.ant.EJBGenAntTask"
classpath="ejbgen.xml com.bea.core.ejbgen_1.0.0.0.jar" />
    <target name="run-ejbgen">
        <ejbgen source="1.5" outputDir = "src\conf" descriptorDir = "src\conf" forceGeneration = "true">
            <fileset dir="src\java\hu\mol\eai\fileconnector\processor" includes="EventProcessor.java" />
        </ejbgen>
    </target>
</project>
```

(4) A NetBeans projekthez hozzá kell adni ezeket:
wls-api.jar, *com.bea.core.ejbgen_ 1.0.0.0.jar*

(5) Az *ejbgen.xml*-t be kell másolni a *build.xml* mellé.

(6) Természetesen implementálni kell java az *onMessage()* metódust, hiszen ebben lesz az üzenetfeldolgozó logika

(7) Futtatni a weblogic *ejb-gen*-t: *setDomainEnv.sh*, utána *ant -f ejbgen.xml*

(8) Build a NetBeans-ben, ami a telepíthető végeredményt adja (egy *jar* file)

7.3. Seession cookie nevének beállítása Weblogic környezetben

Amikor 1-nél több web alkalmazást érünk el egy konkrét Weblogic környezetből, akkor a java előre beállított *JSESSIONID* session cookie-ja mindig felülíródik az utolsónak használt (az URL-ben hivatkozott) alkalmazásával. Ez azt jelenti, hogy elromlik a session kezelés, azaz ebben a formában az alkalmazás használhatatlanná válik. Ez egy biztonsági funkció, azaz nem felesleges, ugyanakkor emiatt biztosítani kell, hogy minden web alkalmazás egyedi nevű session cookie-val rendelkezzen. Ezt a problémát a *web.xml* mellé csomagolt *weblogic.xml* file használatával oldhatjuk meg, ahogy a példa is mutatja. A lényeg a *CookieName* paraméterben van, aminek az értékét most BEKISESSION-ra állítottuk, így ez az alkalmazás ezt a nevet fogja használni a *JSESSIONID* helyett.

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems,
Inc.//DTD Web Application 8.1//EN" "weblogic810-web-jar.dtd">
 <weblogic-web-app>
   <session-descriptor>
     <session-param>
       <param-name>CookieMaxAgeSecs</param-name>
```

```
        <param-value>-1</param-value>
      </session-param>
      <session-param>
        <param-name>InvalidationIntervalSecs</param-name>
        <param-value>60</param-value>
      </session-param>
      <session-param>
        <param-name>TimeoutSecs</param-name>
        <param-value>3600</param-value>
      </session-param>
      <session-param>
        <param-name>CookieName</param-name>
        <param-value>BEKISESSION</param-value>
      </session-param>
    </session-descriptor>
</weblogic-web-app>
```

7.4. Webservice kliens proxy generálás Weblogic környezetben

A világon valaholt publikált soap alapú webszolgáltatásokat nagyon egyszerűen használatba tudjuk venni weblogic környezetből is. A lenti *build.xml* egy minta ant script, ami a `http://molszhbits01:8080/jaxrpc-MOLAgent/agent?WSDL` URL-en lévő szolgáltatásból generál kliens proxy-t. A scripthez nem fűzünk magyarázatot, annyira egyszerű, feltéve, ha ismerjük az *ant* építő eszközt.

```
webservice_clients/order$ cat build.xml
<project name="static" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property environment="env"/>
  <property file="my.properties"/>
  <property name="build.compiler" value="${compiler}"/>
  <property name="source" value="."/>
  <property name="build" value="${source}/build"/>

  <taskdef name="clientgen"
           classname="weblogic.ant.taskdefs.webservices.clientgen.ClientGenTask">
          <classpath>
                  <pathelement path="/bea/weblogic81/server/lib/webservices.jar"/>
                  <pathelement path="/bea/weblogic81/server/lib/weblogic.jar"/>
                  <pathelement path="/bea/weblogic81/server/lib/xmlstream.jar"/>
                  <pathelement path="/bea/weblogic81/server/lib/wsclient81.jar"/>
            </classpath>
  </taskdef>


  <target name="all" depends="clean,␣build"/>

  <target name="clean" >
    <delete dir="${build}"/>
  </target>

  <target name="build">
    <mkdir dir="${build}"/>
    <clientgen wsdl="http://cegszhbits01:8080/jaxrpc-CEGAgent/agent?WSDL"
        autotype="True"
        overwrite="False"
        useServerTypes="False"
        packageName="idsservice"
        clientJar="${build}"/>

    <javac srcdir="${source}" destdir="${build}" includes="Main.java">
            <classpath>
                    <pathelement path="/bea/weblogic81/server/lib/webservices.jar"/>
                    <pathelement path="/bea/weblogic81/server/lib/weblogic.jar"/>
                    <pathelement path="/bea/weblogic81/server/lib/xmlstream.jar"/>
                    <pathelement path="/bea/weblogic81/server/lib/wsclient81.jar"/>
            </classpath>
    </javac>
```

```
  </target>

 <target name="makejar">
  <jar destfile="./idsservice.jar">
          <fileset dir="${build}"/>
  </jar>
</target>


  <target name="run">
    <java classname="Main" fork="true" >
      <classpath>
        <pathelement path="${build};${ex.classpath}"/>
      </classpath>
    </java>
  </target>

</project>
```

A példában keletkezett *idsservice.jar* proxy könyvtárat a programunkhoz kell linkelni. A webservice proxy agent (ez reprezentálja a webservice-t Java oldalról) megszerzését, illetve annak *sendData()* metódusának meghívását a következő programrészlet mutatja.

```
MOLLLESXMLService service = new MOLLLESXMLService_Impl("http://cegbudalldm01:8080/jaxrpc-CEGAgent/agent?WSDL");
AgentIF agent = service.getAgentIF();
wsResult = agent.sendData( xmlObject.toString() );
```

7.5. Amikor a CPU „túlpörög”

Előfordul, hogy egy Weblogic server túlpörgeti a processor-t és szeretnénk tudni, hogy mi az oka, de legalább azt, hogy melyik szálon (Thread-en) történt a „baleset”. A *ps -mp <WLSpid> -o THREAD* paranccsal megkereshetjük a gyanús szálat. A *<WLSpid>* az operációs rendszerszintű PID (process ID). A parancs listázza a thread-eket, így kikereshető a kritikus szál. A *kill -3 <WLSpid>* parancs egy thread dump-ot készít. Képesek vagyunk használni a java debugger-t is: *dbx -a <WLSpid>*.

A *dbx*-ben lekérhető „list of all threads” használva a *thread command*-ot. Az *output thread list*-ben megkereshetjük a CPU intenzív *TID*-ünket (formátuma: $t<num>). A dbx *th info <num>* parancsával részletesebb információt is kaphatunk a kérdéses szálról (output find the : pthread_t (hexadecimal) value).

Fontos : leszedés a processről : detach.