

# **Diplomamunka**

**Miklóssy Tamás**  
**2008**

**Debreceni Egyetem  
Informatikai Kar  
Információ Technológia Tanszék**

**Néprajzi gyűjtemények  
nyilvántartása**

**Témavezető:**

Kollár Lajos  
egyetemi tanársegéd

**Készítette:**

Miklóssy Tamás  
programtervező matematikus

Debrecen  
2008

# **Tartalomjegyzék**

<b>Bevezetés</b>	5
<b>1. Elemzés</b>	6
1.1 Követelmények	7
1.2 Fogalomszótár	9
1.3 Használati eset diagram	10
1.4 Folyamatok ábrázolása	11
1.5 Forgatókönyvek	12
<b>2. Tervezés és megvalósítás</b>	13
2.1 Az adatbázis	14
2.1.1 Az adatbázis felépítése	14
2.1.2 Az objektumok típusokat és táblákat létrehozó szkript	16
2.1.3 Az objektumtípusok között lévő kapcsolatok	21
2.2 Felhasználói felületek	22
2.2.1. AWT-Swing	22
2.2.2. Ablakok	22
2.2.3. Panelek, elrendezési stratégiák	23
2.2.4. Fastruktúra	25
2.2.5. Táblázatok	26
2.2.6. Bementi/kimeneti mezők	27
2.2.7. A megjelenítés stílusa	27
2.2.8. Események kezelése	28
2.2.9. Felhasznált eseményfigyelők	29
1. ActionListener	29
2. MouseListener	29
3. TreeSelectionListener, TreeModelListener	30
4. DocumentListener	31
2.2.10. Nyomtatás	32

2.3 Az adatbázis és a felhasználói felületek összekapcsolása.....	33
2.3.1. JDBC, a kapcsolat felvétele/lebontása.....	33
2.3.2. SQL-Java típusmegfeleltetések.....	34
2.3.3. SQL utasítások végrehajtása Javából.....	35
2.3.4 Adatbázisműveletek a Java oldalon.....	36
 <b>3. Felhasználói dokumentáció.....</b>	<b>37</b>
3.1. Az alkalmazás indítása.....	38
3.2. A fastruktúra bővítése.....	38
3.3. Elem törlése a struktúrából.....	38
3.4. Elem átnevezése.....	39
3.5. Egy adott elemre jellemző adatok megjelenítése.....	39
3.6. Az értékek megadása, módosítása.....	39
3.7. Az adatok mentése.....	40
3.8. Képek kezelése.....	40
3.9. A menü funkciói: nyomtatás, kilépés.....	40
 <b>4. Összefoglalás.....</b>	<b>41</b>
 <b>5. Irodalomjegyzék.....</b>	<b>42</b>
 <b>6. Függelék–Képek.....</b>	<b>43</b>

## **Bevezetés**

Célom egy nyilvántartó rendszer megtervezése és elkészítése volt, amely múzeumi tárgyak, elsősorban szöttesek kezelését teszi lehetővé.

Ezt az alkalmazást egy néprajszakos hallgató számára készítettem el, aki különböző régiségekről, tárgyakról gyűjt információkat. Az egyre bővülő adathalmazt megfelelően már csak számítógéppel lehet kezelni. Az adatok nyilvántartását lehetővé tévő általános célú programok (Microsoft Excel, Access) nem elég hatékonyan alkalmazhatóak. A piacon már meglévő, erre a célra kifejlesztett alkalmazások elsősorban anyagi okokból nehezen elérhetőek, illetve az általuk megvalósított funkciók sem teljesen illeszkednek a leendő felhasználó igényeihez. Ezek a programok általában olyan funkciókkal is rendelkeznek, melyekre nincs szüksége, illetve számos, hasznos funkció hiányzik belőlük. Az adatok minél hatékonyabb nyilvántartására és kezelése érdekében úgy gondolta, hogy egy saját alkalmazást fejlesszet ki, amely egy az egyben megfelel az ő igényeinek, illetve az ő elképzeléseit tükrözi.

A követelmények pontos rögzítése után következhetett a tervezés, majd a megvalósítás. A feladatom három, jól elkülöníthető részből tevődött össze. Először az adatbázis megtervezése, létrehozása, másodszor a felhasználói felületek elkészítése, harmadszor pedig az előbb említett két különálló rész összekapcsolása, az adatbázis és a program közötti kommunikáció, az adatforgalom optimális megvalósítása. A nyilvántartó rendszerek alapja az adatbázis, ezért erre a részre nagy hangsúlyt fektettem. Ezután következhetett a felületek megtervezése, az adatmezők, funkciógombok elhelyezése. Végül az adatbázis-kapcsolat. Mivel az adatbázissal való kommunikáció meglehetősen erőforrás-igényes, meg kellett terveznem, hogy mely időpillanatban történjen a kapcsolat felépítése, az adatok mozgatása, a kapcsolat lebontása.

Az egyetemi tanulmányaim során alaposabban az **Oracle** adatbázis-kezelő rendszerrel, illetve a **Java** programozási nyelvvel ismerkedtem meg, így a diplomamunkámat is ezen eszközök felhasználásával készítettem el. Az adatbázishoz az egyetemi Oracle-szervert, míg a programozáshoz Eclipse fejlesztői környezetet használtam.

# **1. fejezet**

## Elemzés

## **1.1 Követelmények**

### **1. Szoftverkövetelmények**

Az első lépés egy alkalmazás elkészítésekor az elemzés, ezen belül is a követelmények rögzítése. Ezen dokumentáció keretében a programozó és a leendő felhasználó közösen leegyeztetik és pontosan rögzítik, hogy mit várnak el a programtól mind a kinézetet, mind a funkcionalitást illetően.

Először a tárolandó adatokat egyeztettük le. A nyilvántartani kívánt múzeumi tárgyak darabszáma akár a több százat is elérheti, ezért fontos, hogy azokat osztályozni, csoportosítani tudjuk. Az összetartozó tárgyakat egy gyűjtemény fogja össze, minden gyűjtemény pedig egy adott intézményhez tartozik. Követelmény volt annak átláthatósága is, hogy egy adott intézményhez mely gyűjtemények, ezáltal mely tárgyak tartoznak. Közösen megállapodtunk, hogy ezen objektumokat egy fa adatszerkezetben tároljuk, melynek első szintjén az intézmények, második szintjén a gyűjtemények, harmadik szintjén pedig a tárgyelemek helyezkednek el. Ezen fa egy részfája tartalmazza az összes, összetartozó elemeket, így biztosítva az átláthatóságra vonatkozó követelményeket.

Ezután következett a három elemről (intézmény, gyűjtemény, tárgy) tárolni kívánt információhalmaz pontos rögzítése. Mivel a leendő felhasználó régóta foglalkozik a néprajzi adatok gyűjtésével, egyértelmű volt számára, hogy egy tárgyat mely adatok jellemeznek, illetve mely adatokat szükséges, érdemes tárolni. Az adatok egy része minden tárgyra egyaránt jellemző (pl. leltári szám, állapot), illetve vannak olyan adatok is, melyek szótesenként változhatnak (pl. díszítőminta száma, széleldolgozások). Bizonyos tárgyaknál elképzelhető, hogy az adatok egy része nem, vagy nem pontosan ismert (pl. készítésre, használatra vonatkozó adatok, időpontok). A tárgyak sokfélesége miatt hasznos, ha a fix jellemzők mellett egyéb szövegek, megjegyzések is szabadon megadhatóak.

További követelmény volt a nyilvántartott tárgyak minél egységesebb kezelése. Ugyanannak a tárgynak számos különböző neve lehet, máshogy nevezik az ország különböző részein, vagy akár külföldön (pl. Erdélyben) is. Előfordulhat az is, hogy a városok, községek, helységek nevei időről időre változnak. Ennek a követelménynek úgy tudunk eleget tenni, ha a felhasználót bizonyos esetekben korlátozzuk abban, hogy szabadon adhassa meg az adatokat. A kívánt adattagok értékkészletét előre meghatároztuk, majd a halmazokat dinamikusan bővíthetővé tettük. A felhasználó ezen halmazbeli elemek közül választva állíthatja be a kívánt értéket.

Az alkalmazás funkcionális követelményei közé tartozik egy elem nyilvántartásba vétele, valamint az adatok megjelenítése, módosítása. Hasznos, ha az adatok nemcsak a képernyőn, hanem papíron is megjeleníthetők, kinyomtathatóak.

## 2. Rendszerkövetelmények

A szoftverkövetelmények mellett pontosan rögzíteni kell a rendszerkövetelményeket is. Ezen dokumentáció keretében adjuk meg, hogy az alkalmazást milyen környezetben szeretnénk használni.

A programot otthoni, magánjellegű használatra terveztem meg, amikor is az alkalmazás és az adatbázis ugyanazon a számítógépen található (offline használati üzemmód). A választásom – elsősorban az adatbázis-műveletek optimalizálása miatt - az Oracle adatbázis-kezelő rendszerre esett. Mivel az Oracle Express Edition ingyen hozzáférhető, ez a későbbiekben nem ró extra költséget a felhasználó számára.

## 3. Követelmények a továbbfejlesztés esetén

A fent említett követelményeken kívül elhangzottak olyanok is, melyek megvalósítása az alkalmazás továbbfejlesztése esetén valósíthatóak meg. A további követelményeket és a továbbfejlesztés lehetőségeit az összefoglalásban részletesen tárgyalom.



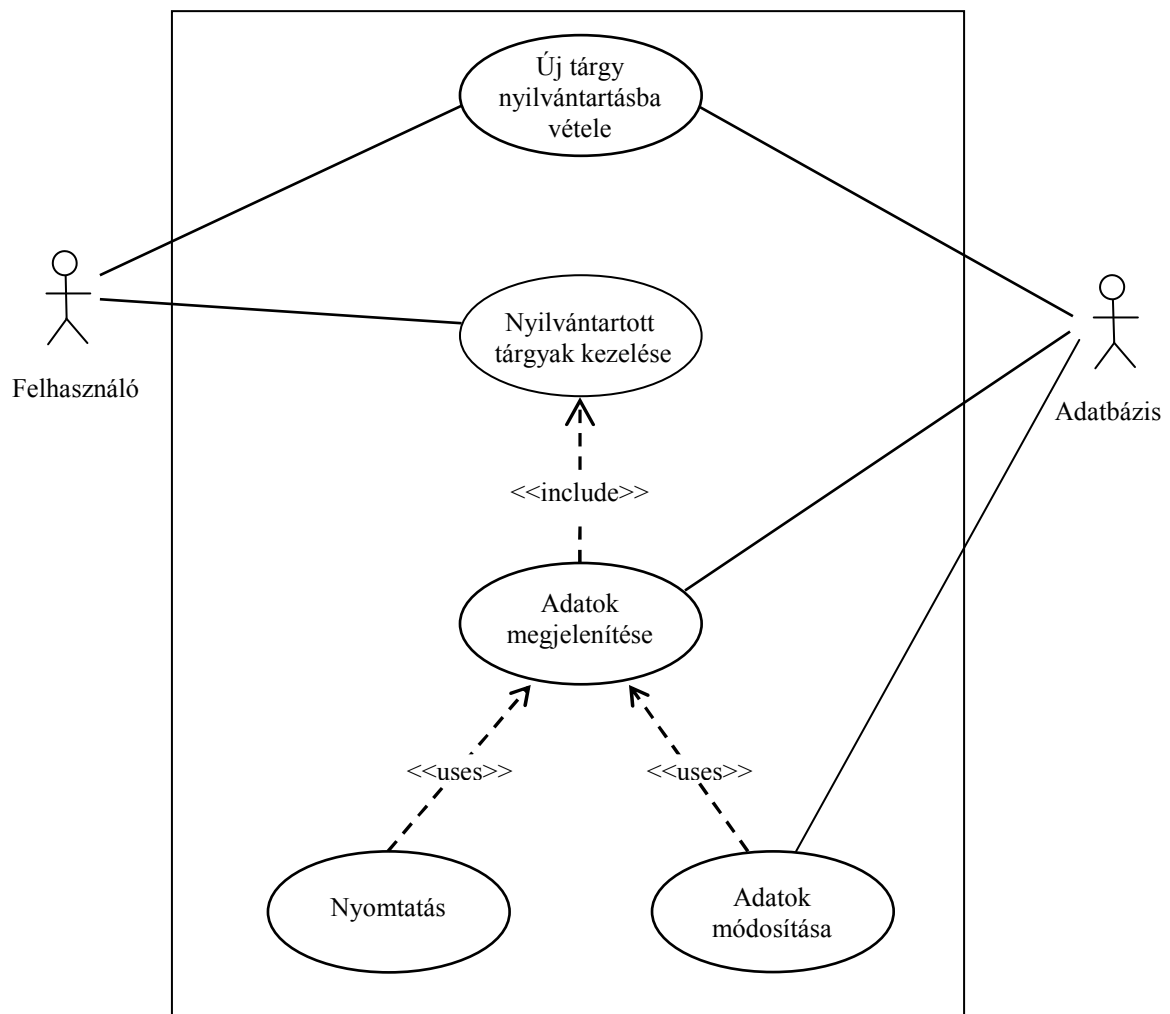
## **1.2 Fogalomszótár**

A fogalomszótár keretén belül a rendszerrel kapcsolatba kerülő fogalmakat definiáljuk. Ezen dokumentum célja, hogy az elolvasása után – akár egy teljesen kívülálló személy számára is – egyértelműek legyenek az eddig, illetve később megjelenő informatikai és néprajzi fogalmak.

1. **intézmény:** egyed, melyhez gyűjtemények, ezáltal tárgyak tartoznak
2. **gyűjtemény:** valamilyen szempont szerint összetartozó tárgyak együttese
3. **tárgy:** bármilyen olyan egyed (elsősorban szöttes), amely néprajzi szempontból érdekes, ezáltal róla nyilvántartást kívánunk vezetni
4. **thesaurus:** a tárgyak osztályozására, besorolására használt segédlet, az egységes kezelést megkönnyítő, előre rögzített értékhalmoz
5. **díszítőminta:** az alapanyagot (szövetet, bőrt ...stb.) díszítő eljárás, melynek során különféle technikával (hímzés, szövés, karcolás, faragás, domborítás, festés, viasz-írás stb.) mintát visznek fel az anyagra. A díszítőminták koronként (archaikus/régi-, és új stílus), vidékenként, felekezetenként, társadalmi pozíciónként stb. változhatnak.
6. **széleldolgozás:** a textília szélét (pl.: szöttes) befejező/eldolgozó technika, amely lehet visszahajtás, elszegés, azsúrozás, különféle rojtkötés...stb.
7. **revízió:** állományellenőrzés, selejtezés. Bizonyos időközönként törvény által szabályozott módon végzett felülvizsgálat a muzeális gyűjteményekben.
8. **leltározás:** az a folyamat, amikor az intézmény a hozzátartozó tárgyakat számbaveszi
9. **eladó:** egy adott tárgyat/tárgyakat az intézmény rendelkezésére bocsátó személy
10. **gyűjtés:** az a folyamat, amikor eddig nem ismert tárgyakról tudomást, információkat, adatokat szerzünk

### 1.3 Használati eset diagram

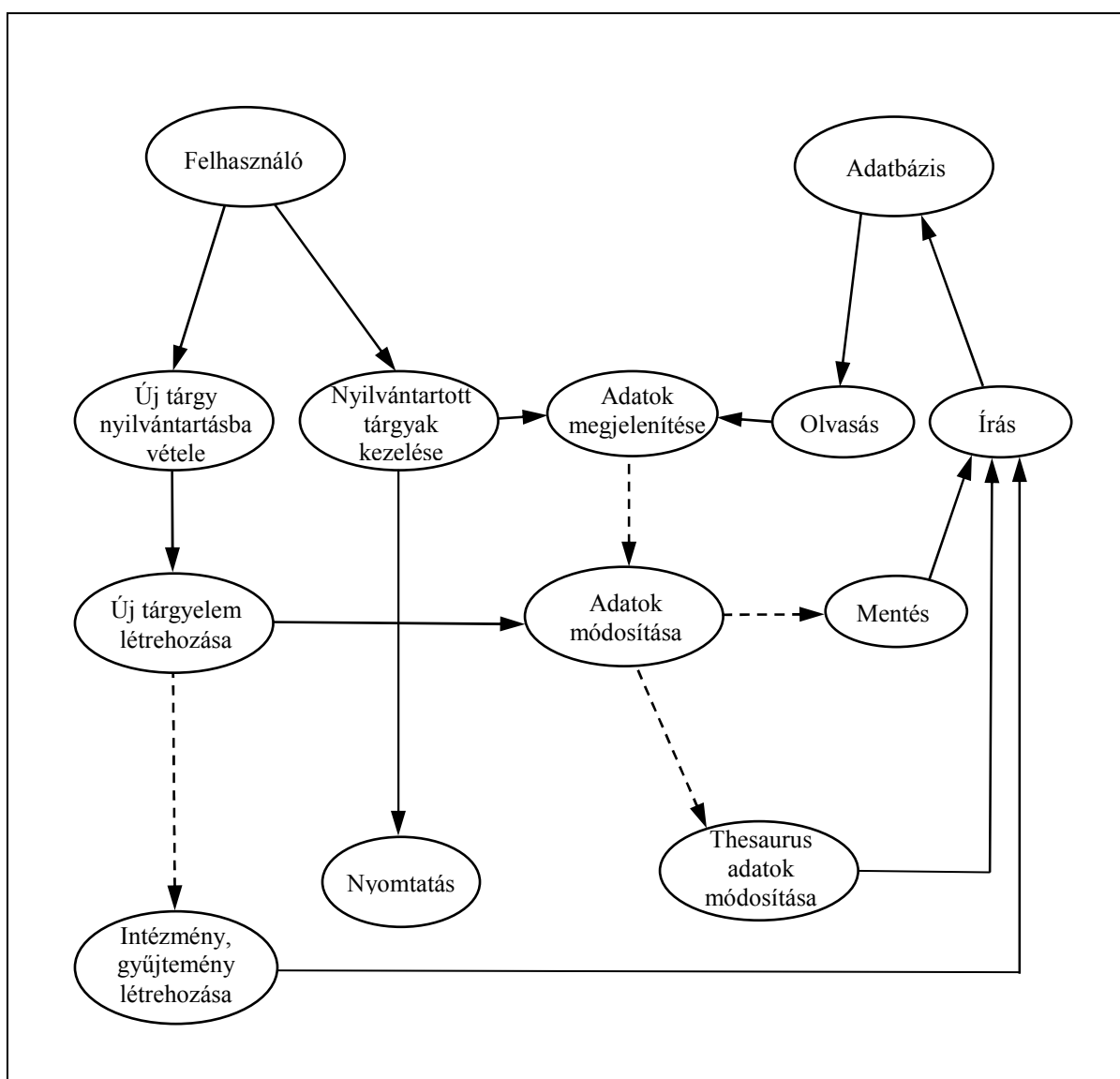
A használati eset diagram egyike a szabványos modellező nyelv (UML) diagramjainak, melynek segítségével a rendszer működését, illetve az általa nyújtott funkciókat távolról mutatjuk be. A rendszeren kívüli, ám szorosan hozzátartozó elemeket (a felhasználót és az adatbázist) aktorokkal ábrázoljuk. Az egyes használati eseteket az ellipszisek jelentik. A használati esetek egymás közötti és az aktorokkal való kapcsolatát is ábrázoljuk. A lényegesebb használati esetek által jelzett folyamatokat a forgatókönyvben tárgyaljuk.



1. ábra Használati eset diagram

## 1.4 Folyamatok ábrázolása

Ezen diagram keretében az alkalmazás működését, illetve a megvalósítandó funkciókat folyamatszerűen mutatjuk be. A főbb lépéseket ellipsziszekkel ábrázoljuk, az egymás utáni lépéseket nyilak kötik össze. Vannak olyan lépések, amelyeket mindig végre kell hajtani, illetve vannak olyanok is, melyek az adott helyzettől függenek. A folytonos nyilak a kötelező, míg a szaggatottak az opcionális lépéssorozatot jelentik. A folyamatokról bővebben a forgatókönyvekben, illetve a felhasználói dokumentációban olvashatunk.



2.ábra A program folyamatszerű működése

## **1.5 Forgatókönyvek**

### 1. Új tárgy nyilvántartásba vétele

A folyamat azzal kezdődik, hogy elhelyezzük a tárgyat a fastruktúrában a megfelelő helyre. Ehhez szükség esetén létrehozuk a megfelelő intézményt, illetve gyűjteményt. Ezután következik az új tárgyelem létrehozása a fában a tárgy azonosítójának (leltári számának) megadásával. Ezáltal az adott tárgy bekerül a nyilvántartásba, ekkor azonban az opcionális adatok még üres értékekkel rendelkeznek. Az attribútumoknak információtartalommal rendelkező értéket a módosítás során adhatunk.

### 2. Meglévő adatok módosítása

Az adatok a nyilvántartás szempontjából két részre tagolódnak. Egyrésről az intézmények, gyűjtemények és a tárgyak neveinek módosítását a fastruktúra megfelelő elemének átnevezésével végezhetjük el. Egy adott elemet (tárgyat vagy intézményt) jellemző adatok módosítására pedig a jobboldali adatpanelen van lehetőség. Az adatpanelen a leírók aktuális értékei jelennek meg. Egy adott attribútum értékét a megfelelő szövegmező átírásával módosíthatjuk. Azoknál az attribútumoknál, melyek értékészlete egy rögzített halmazra korlátozódik, szükséges szerint a kívánt új értéket először az értékhalmban kell elhelyezzük. A kívánt érték csak ezután állítható be. A módosítások csak az explicit (felhasználó által végzett) mentés során véglegesítődnek.

### 3. Nyilvántartott elemek adatainak nyomtatása

Miután megkerestük és kiválasztottuk az adott elemet a fában, a nyomtatást egy menüpont segítségével indíthatjuk el. Az adatok ugyanolyan elrendezésben kerülnek a papírra, mint ahogyan az adatpanelen láthatjuk őket. Egyéb beállítások elvégzésére (egyelőre) nincs lehetőség.

## **2. fejezet**

### Tervezés és megvalósítás

## **2.1 Az adatbázis**

### **2.2.1 Az adatbázis felépítése**

Az Oracle adatbázis-kezelő rendszer lehetővé teszi a relációs és az objektumorientált tervezést is, így választanom kellett a két technika közül. Mivel a program nyelvének választott Java programozási nyelv objektumorientált, ezért az adatbázis alapjának is az **objektumorientáltságot** választottam, hogy a későbbi sql $\leftrightarrow$ java típusmegfeleltetések egyszerűbben megvalósíthatók legyenek. A tárolni kívánt, nagymennyiségű információhalmazt kisebb, összetartozó egységekre bontottam, ezzel is növelve az átláthatóságot, megkönnyítve a karbantarthatóságot, illetve a későbbi fejlesztéseket, kiegészítéseket. A kisebb, összetartozó egységek leírására objektumtípusokat hoztam létre.

Az adatbázis szerkezete lehetővé teszi adatok tárolását nemcsak magáról a tárgyról, hanem arról az intézményről is, amelyhez a tárgy tartozik. Egy intézménynek a leíró adattagjai: a neve (mely egyben az azonosító is), valamint a címe. Egy tárgyról az alábbi információk tárolására van lehetőség: leltározási szám, intézmény, gyűjtemény (ez a három attribútum összetett kulcsként alkotja az azonosítót), hivatalos név, helyi név, darabszám, állapot, leírás, méretek, revízió éve, adattári szám, lelőhely, őrzési hely, képi hivatkozások, szakirodalmi hivatkozások, megjegyzés, restaurálási információk. Ezeken kívül lehetőség van a készítésről is információkat tárolni: a készítés helye, ideje, technikája, a készítő neve, vallása, nemzetisége, társadalmi helye. Egy adott tárgy használatáról a következő adatokat lehet tárolni: használat helye, ideje, egyéb használati adatok. Egy tárgy díszítőmintáját a következő leíró attribútumok jellemzik: anyaga, szakneve, helyi neve, színe, széleldolgozása. Lehetőség van a gyűjtés helyének, idejének és a gyűjtő nevének a tárolására is, valamint a megszerzés módjának, vételárának, pénznemének, illetve az eladó nevének, címének, születési évének rögzítésére is. Mivel az eladó és az intézmény címe azonos összetevőkből áll, ezért mindkét esetben ugyanazt a cím típust használtam (település, megye, kistérség, régió, helység, utca, házszám, irányítószám). Végül pedig a leltározás időpontja és a leltározó személy neve is tárolásra kerülhet.

A típusok közötti 1:1 kapcsolatot az objektumtípusok egymásba ágyazásával, míg az 1:N kapcsolatot az objektumtípusba illesztett beágyazott objektumtábla használatával valósítottam meg. Beágyazott táblák esetében az adatok fizikailag a NESTED TABLE utasításrész után megadott táblában tárolódnak, így maga az oszlop csak egy mutató, amely megmondja, hogy a beágyazott adat ténylegesen hol található meg.

Vannak olyan adatok, melyek értéküket csak egy előre megadott értékhalmból vehetik fel (**Thesaurus**). Ennek a módszernek két előnye van. Egyrészt megkíméli a felhasználót a gépelés fáradalmaitól, kizárja az elgépelés lehetőségét, másrészt pl. minden olyan tárgy, melynek azonos a helyi neve, biztos, hogy azonos névvel lesz letárolva. Minden törölköző hivatalos neve meg fog egyezni, nem pedig valahol így, valahol úgy lesz letárolva (törölköző↔törölköző). A két névben lévő csupán egy karakter eltérés a későbbiek folyamán azt eredményezné, hogy a két tárgynak teljesen különböző a neve. Illetve a készítés helyénél egy adott városnak mindenhol egy adott neve lesz megadva és egységesen lesz kezelve még akkor is, ha az adott városnak időközben megváltozott a neve. (Leninváros→Tiszaújváros). Az alábbi leíró attribútumok csak Thesaurusokból vehetik fel az értéküket: megszerzés módja, vásárláskor fizetett pénznem, leltározó neve, készítés helye, díszítőminta anyaga, a méretek megadásánál használt mértékegység, a készítés technikája, egy tárgy hivatalos neve, állapota, lelőhelye, gyűjtésének helye, gyűjtő neve, eladó neve.

Az értékhalmbazok tárolásához különböző adatbázistáblákat hoztam létre, itt tárolódnak a konkrét értékek. Az objektumtípusoknál pedig egy referenciatípussal hivatkozunk az adott konkrét értékre. Ez a referenciatípusos megoldás azt eredményezi, hogy egy konkrét értéket csak egyszer tárolunk az adatbázisban. Ez egyrészt helymegtakarítást jelent (ha pl. 50 tárgynak ugyanaz a helyi neve, akkor azt nem 50-szer, hanem csak 1-szer tároljuk). Másrészt pedig a módosítást könnyíti, gyorsítja meg (nem kell végigmenni az összes elemen és egyenként módosítani a megfelelő attribútumot, a módosítást elég csak egy helyen véghezvinni). A módszer hátránya az előre letárolt értékhalmbazok folyamatos bővítése, karbantartása, amely viszont a felhasználói felületeken keresztül könnyen kezelhető.

### **2.1.2 Az objektumtípusokat és a táblákat létrehozó szkript**

#### **objektumtípusok a Thesaurusokhoz**

```
create or replace type t_mod as object(  
    mod varchar2(30)  
)  
/  
create or replace type t_penznem as object(  
    penznem varchar2(30)  
)  
/  
create or replace type t_leltarozo as object(  
    leltarozo_nev varchar2(50)  
)  
/  
create or replace type t_keszites_helye as object(  
    keszites_helye varchar2(50)  
)  
/  
create or replace type t_anyag as object(  
    anyagnev varchar2(50)  
)  
/  
create or replace type t_mertekegyseg as object(  
    mertekegysegnev varchar2(15)  
)  
/  
create or replace type t_technika as object(  
    technika varchar2(40)  
)  
/  
create or replace type t_hivatalos_nev as object(  
    hivatalos_nev varchar2(40)  
)  
/  
create or replace type t_allapot as object(  
    allapot varchar2(40)  
)  
/  
create or replace type t_lelohely as object(  
    lelohely varchar2(40)  
)  
/
```



```
create or replace type t_gyujtes_hely as object(  
    gyujtes_hely varchar2(40)  
)  
/  
create or replace type t_gyujto_nev as object(  
    gyujto_nev varchar2(40)  
)  
/  
create or replace type t_elado_nev as object(  
    elado_nev varchar2(40)  
)  
/
```

### objektumtípusok a leíró adattagokhoz

```
create or replace type t_koordinata as object(  
    x number,  
    y number)  
/  
create or replace type t_cim as object (  
    telepules varchar2(30),  
    megye varchar2(20),  
    kisterseg varchar2(20),  
    regio varchar2(50),  
    helyseg varchar2(40),  
    utca varchar2(40),  
    hazszam varchar2(10),  
    irányitoszam number,  
    foldrajzi_koordinata t_koordinata)  
/  
create or replace type t_elado as object(  
    nev ref t_elado_nev,  
    cim t_cim,  
    szuletesi_ev number)  
/  
create or replace type t_megszerzes as object(  
    mod ref t_mod,  
    vetelar number,  
    penznem ref t_penznem,  
    elado ref t_elado)  
/  
create or replace type t_leltarozas as object(  
    idopont Date,  
    leltarozo_neve ref t_leltarozo)  
/
```

```
create or replace type t_gyujtes as object(  
    helye ref t_gyujtes_hely,  
    ideje Date,  
    gyujto_neve ref t_gyujto_nev,  
    foldrajzi_koordinata t_koordinata)  
/  
create or replace type t_keszito as object(  
    nev varchar2(30),  
    vallas varchar2(30),  
    nemzetiseg varchar2(30),  
    tarsadalmi_hely_sz_sz varchar2(1000))  
/  
create or replace type t_keszites as object(  
    helye ref t_keszites_helye,  
    helye_szabad_szoveg varchar2(1000),  
    foldrajzi_koordinata t_koordinata,  
    ideje Date,  
    ideje_szabad_szoveg varchar2(1000),  
    technika ref t_technika,  
    technika_szabad_szoveg varchar2(1000),  
    keszito ref t_keszito)  
/  
create or replace type t_gyujtemeny as object(  
    nev varchar2(30)  
)  
/  
create or replace type t_gyujtemenyek IS TABLE of t_gyujtemeny  
/  
create or replace type t_intezmeny as object(  
    nev varchar2(50),  
    cim t_cim,  
    gyujtemenyek t_gyujtemenyek --beagyazott tabla  
)  
/  
create or replace type t_hasznat as object(  
    hely varchar2(30),  
    hely_szabad_szoveg varchar2(1000),  
    ideje Date,  
    ideje_szabad_szoveg varchar2(1000),  
    hasznalati_adatok varchar2(1000))  
/  
create or replace type t_diszitominta as object(  
    anyaga ref t_anyag,  
    szakneve varchar2(20),  
    helyi_neve varchar2(20),  
    szine varchar2(20),  
    szeleldolgozas varchar2(50))  
/
```

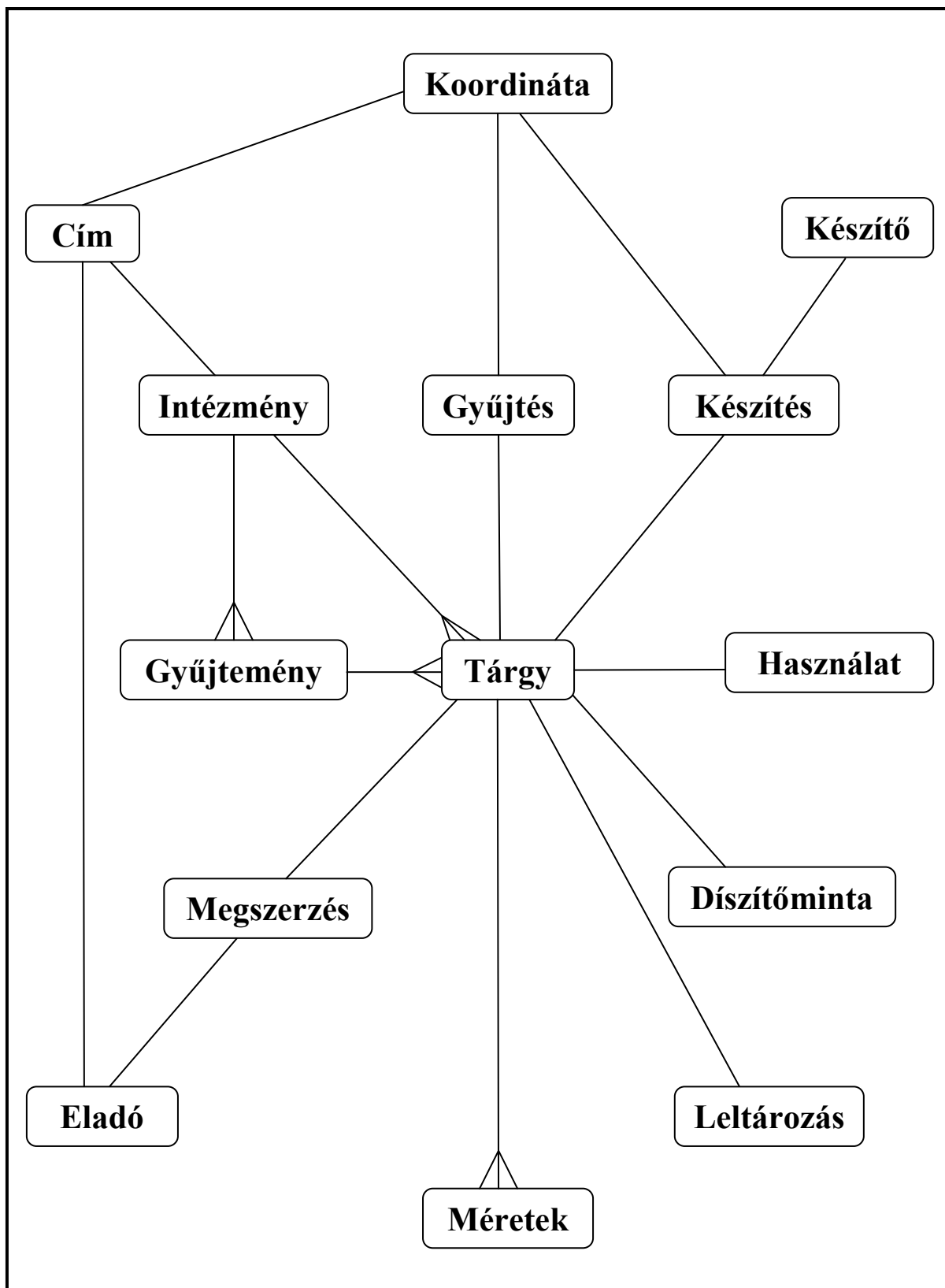
```
create or replace type t_meret as object(  
    dimenzio varchar2(50),  
    meret    number,  
    mertekegyseg ref t_mertekegyseg)  
/  
create or replace type t_meretek IS TABLE of t_meret  
/  
  
create or replace type t_targy as object(  
    leltarozasi_szam varchar2(20), --azonosito  
    intezmeny ref t_intezmeny,    --azonosito  
    gyujtemeny ref t_gyujtemeny, --azonosito  
    hivatalos_nev ref t_hivatalos_nev,  
    helyi_nev varchar2(30),  
    darabszam number,  
    allapot ref t_allapot,  
    leiras_szabad_szoveg varchar2(1000),  
    meretek t_meretek,          --beagyazott_talba  
    revizio_eve number,  
    adattari_szam number,  
    lelohely ref t_lelohely,  
    lelohely_szabad_szoveg varchar2(1000),  
    orzesi_hely varchar2(50),  
    kepi_hivatkozasok varchar2(1000),  
    szakirodalmi_hivatkozasok varchar2(1000),  
    megjegyzes_szabad_szoveg varchar2(1000),  
    restauralas_szabad_szoveg varchar2(1000),  
    keszites t_keszites,  
    hasznalat t_hasznalat,  
    diszitominta t_diszitominta,  
    gyujtes ref t_gyujtes,  
    megszerzes t_megszerzes,  
    leltarozas t_leltarozas)  
/
```

**Adatbázistáblák**

```
create table targyak of t_targy nested table meretek store as meretek_ntab
/
create table intezmenyek of t_intezmeny nested table gyujtemenyek store as
gyujtemenyek_ntab
/
create table gyujtesek of t_gyujtes
/
create table keszitok of t_keszito
/
create table modok of t_mod
/
create table penznemek of t_penznem
/
create table leltarozo_nevek of t_leltarozo
/
create table keszites_helyei of t_keszites_helye
/
create table anyagok of t_anyag
/
create table mertekegysegek of t_mertekegyseg
/
create table hivatalosnevek of t_hivatalos_nev
/
create table allapotok of t_allapot
/
create table lelohelyek of t_lelohely
/
create table gyujteshelyek of t_gyujtes
/
create table gyujtonevek of t_gyujto_nev
/
create table eladonevek of t_elado_nev
/
create table technikak of t_technika
/
```

**--az intézmények és a tárgyak egyediséget biztosító utasítások**

```
alter table intezmenyek add constraint int_nev_uq unique(nev);
/
alter table targyak add constraint targy_leltizam_uq unique(leltarozasi_szam);
/
show errors
```

**2.1.3. Az adatbázis objektumtípusai között lévő kapcsolatok**

## **2.2 Felhasználói felületek**

### **2.2.1. AWT-Swing**

Javában a grafikus felületek a **java.awt** és a **javax.swing** csomaghierarchiában található osztályok és interfészek segítségével valósíthatók meg.

Az awt komponensek megjelenítését nem a Java virtuális gép, hanem az operációs rendszer ablakozó rendszere végzi. Mivel ebben az esetben a grafikus komponensek kinézete az operációs rendszertől függ, sérül a Java platformfüggetlenségi elve. A swing komponensek az awt alapjaira épülnek, de tisztán Javában íródtak, így függetlenek a grafikus felületet megjelenítő operációs rendszertől. Minden awt komponenshez megtalálható annak Swing megfelelője, melynek osztályneve egy nagy J betű előtaggal különbözik.

A következőkben az implementáláshoz használt Swing komponenseket (JFrame, JDialog, JPanel, JScrollPane, JSplitPane, JTree, JTable, JTextField, JTextArea, JLabel, JButton) részletesen bemutatom.

### **2.2.2. Ablakok**

A grafikus alkalmazások alap építőelemei az ablakok. Egy ablakot egy **JFrame** típusú objektum reprezentál, melynek mérete a void setSize(..), pozíciója a void setLocation(..) metódusok segítségével állítható be, láthatósága pedig a void setVisible(..) metódussal módosítható. Az ablakok egy speciális részhalmaza a párbeszédablakok, melyek **JDialog** típusú objektumok. Jellemzőjük, hogy modálissá tehetőek. Ez azt jelenti, hogy amíg egy modális ablak látható, addig a képernyőn lévő összes többi ablakot megjelenítő programszál blokkolódik, így azok az ablakok csak a modális ablak bezárása után lesznek aktívak. Ilyen típusú ablakokkal szabályoztam, hogy a felhasználó addig ne kezdhessen egy új műveletbe, amíg az aktuálisat teljesen be nem fejezte.

A program alapjául szolgáló főablak JFrame típusú objektum, az összes többi, időközben felbukkanó ablakok (Thesaurus-ablakok, hibaüzenet-ablakok, mentéskor-kilépéskor megerősítést kérő, nyugtázó, információközlő ablakok) pedig JDialog típusúak.

### 2.2.3. Panelek, elrendezési stratégiák

Minden ablakhoz tartozik egy **JPanel** típusú objektum, amelyeken a megjeleníteni kívánt grafikus komponensek, illetve további panelek (egymásba ágyazás, skatulyázás) helyezhetők el. A komponensek egymáshoz viszonyított elhelyezkedését az adott panel elrendezési stratégiája határozza meg, mely a panel konstruktorában, illetve a `setLayout(..)` metódusa segítségével állítható be.

A felhasználói felületek megalkotásánál három különböző elrendezési stratégiát használtam. Alapértelmezett a **FlowLayout** elrendezési stratégia, melynél a komponensek egymás mellé igazodva helyezkednek el. Ott, ahol a komponensek egymás alá igazodására volt szükség, **BoxLayout** elrendezési stratégiát választottam. A **BorderLayout** elrendezési stratégia az adott panelt 5 részre (észak, dél, nyugat, kelet, közép) osztja. A `void add(..)` metódus második paramétereként megadandó nevesített konstans segítségével állítható be, hogy az adott komponenst a panel mely részére kívánjuk elhelyezni.

Az alkalmazás elindításakor megjelenő főablak egy **JSplitPane** típusú panelt tartalmaz, mely két darab alpanelből áll. A baloldali alpanel egy fastruktúra, a jobboldalin pedig a kiválasztott faelemre jellemző adatok láthatóak. Ez az adatpanel (ellentétben a baloldali fastruktúra panellel) nem állandó, attól függően változik, hogy a fában milyen típusú elem (intézmény vagy tárgy) van kijelölve. A jobboldali panel kezdetben nagyjából négyszer szélesebb, mint a baloldali panel, ez a méretarány azonban a két panel között lévő függőleges osztóvonal csúsztatásával megváltoztatható. A kötelezően megadandó adatok (intézménynév, gyűjtemény, leltári szám) mindenféleképpen már értékkel rendelkező leíró attribútumok, ezért (minimum) ezen adatok értékei már látszanak. A többi adat megadása opcionális.

Egy adott intézményről mindössze néhány adatot tárolunk (a nevét és a címét), így ez a mennyiségű adat kényelmesen elfér a rendelkezésre álló helyen. A tárgyaknál viszont már más a helyzet, a nagy adattömeg miatt.

Egyértelmű volt, hogy nem szabad az összes adatot egy kis helyre zsúfolnom, mivel ez az adatok átláthatatlanságát eredményezte volna. Ezért az adatokat 4 csoportra osztottam. Lehetőség van arra, hogy egyszerre csak bizonyos adatcsoportok legyenek láthatóak. Ezeket lenyitható/összecsukható panel segítségével oldottam meg, amely reprezentációjához a **JPanel** osztály kiterjesztésével saját osztályt hoztam létre. Ez az osztály két, egymás alatt elhelyezkedő alpanel tartalmaz. A felső alpanel mindig látható, mivel itt helyezkednek el a

panel nyitáshoz/záráshoz, illetve a panel mentéséhez használandó nyomógombok. Az alsó alpanelen az adatok láthatóak, amelynek láthatóságát a nyomógomb állapota határozza meg (nyitott / zárt). Beállítható, hogy kezdetben mely adatpanelek legyenek nyitva, melyek zárva. A nyitva lévő adatpanelek esetén azonnal látszik, zárt esetben az adatpanelek lenyitásával válik láthatóvá az adathalmaz.

Az alpanelek lenyitásánál előfordulhat, hogy az adatok nem férnek el a számukra előre meghatározott fix méretű helyen, így azok nem teljes egészében látszanak. Ezt a problémát a görgethető panel segítségével orvosoltam, így az adott Swing komponenst egy **JScrollPane** típusú objektumon belül helyeztem el. A görgetést vezérlő görgetősávok JScrollPane típusú objektumok. A vízszintes (horizontális) irányban való görgetést vezérlő görgetősáv a panel alján, a függőleges (vertikális) görgetést vezérlő pedig a panel jobb oldalán helyezkedik el. A `setHorizontalScrollBarPolicy(true)` és a `setVerticalScrollBarPolicy(true)` metódusokkal beállítottam, hogy ezek a görgetősávok csak akkor legyenek láthatóak, ha a görgetésre valóban szükség van, tehát ha az alpanel mérete nagyobb a számára kijelölt helynél. A dinamikus bővíthető adatszerkezetek (fastruktúra, táblázatok) elemeinek számára gyakorlati felső korlát nincs, ezért a bővítések következtében előfordulhat, hogy ezek az elemek is „kinövik” a számukra meghatározott helyet, ezért azokat is görgethető paneleken helyeztem el.

Egy konkrét adathoz (pl. intézmény neve) általában két vagy három **grafikus komponens** tartozik. Először is egy címke (JLabel), mely az adat nevét szövegesen jelzi, valamint egy bemeneti/kimeneti mező, melyben az adott adat értéke látható, módosítható. Azokhoz az adatokhoz, melyek csak egy előre meghatározott értékhalmból vehetik fel az értéküket, tartozik még egy nyomógomb (JButton) is, melynek segítségével a kívánt érték állítható be. Ezt a három komponenst egy panelen helyeztem el BorderLayout elrendezési stratégiát használva. A címke komponens a panel baloldali (BorderLayout.WEST) részére került, az érték komponens az esetleges nyomógommbal pedig FlowLayout elrendezési stratégiával a panel közepére (BorderLayout.CENTER). A FlowLayout elrendezési stratégia azt eredményezte, hogy a beviteli mező és a nyomógomb egymás mellé igazodtak. Az adatpanelek között pedig Y tengellyel igazított BorderLayout elrendezési stratégiát választottam, amely azokat egymás alá igazította.

Az egyes adatpaneleket kerettel (JBorder) vettem körül, ezzel is növelve az egymástól való elkülönülésüket. A keretek színét, vastagságát a konstruktorukban állítottam be.



### **2.2.4. Fastruktúra**

A tárolandó múzeumi tárgyak hierarchikusak, így fába lehet őket rendezni. A képernyő bal oldalán mindig ez a fastruktúra látható. A megvalósításához a **JTree** osztályt használtam, a hozzá kapcsolódó egyéb típusok pedig a **javax.swing.tree** alcsomagban találhatóak. A gyökérelemből indulva minden tárgyhoz egy három hosszúságú úton jutunk el. A fa első szintjén találhatóak az intézmények, a második szintjén a gyűjtemények, a harmadik szintjén pedig maga az adott tárgy (ezek a fa levélelemei). A fa adatszerkezet segítségével átlátható, hogy a rendszerben milyen intézmények vannak nyilvántartva, egy adott intézményben milyen gyűjtemények vannak, illetve egy adott gyűjteményhez mely tárgyak tartoznak. Ez a fastruktúra könnyen módosítható: a harmadik szinten lévő tárgyakat kivéve bármely elem alá létrehozható újabb faelem, illetve bármely levélelem törölhető.

A fastruktúra adatmodelljének, valamint a fában lévő csomópontok reprezentációjához az alapértelmezés szerinti `DefaultTreeModel` és `DefaultMutableTreeNode` osztályokat választottam. Lényeges, hogy a fában egyszerre csak egy elemet lehessen kiválasztani, ezért a `SINGLE_TREE_SELECTION` kiválasztási modellt alkalmaztam. Az elemek átnevezhetőségét a `setEditable(true)` példánymetódussal biztosítottam.

Egy `JTree` típusú objektum elemei tetszőleges objektumok lehetnek. Az intézmények, gyűjtemények és a múzeumi tárgyak reprezentálásához három, különböző objektumtípust használtam fel. Mivel egy gyűjteményről a nevén kívül egyéb leíró adatot nem tárolunk, ezért azt egy `String` típusú objektum reprezentál, az intézmények és tárgyak esetében azonban saját osztályokat hoztam létre (`IntezmenyFaelem.java`, `TargyFaelem.java`), melyek adattagjai – hasonlóan az adatbázis objektumtípusaihoz – az adott elemre vonatkozó jellemzők. A megjeleníteni kívánt értékeket az objektumok `toString()` metódusai által visszaadott értékek határozzák meg. Az intézmények és a gyűjtemények a nevükkel, a múzeumi tárgyak pedig a leltári számukkal jelennek meg.

A faelemeket a `DefaultMutableTreeNode` osztály konstruktorával hoztam létre, melyhez gyermekelemeket a `void add(DefaultMutableTreeNode dmt)` metódusával rendeltem hozzá. Az ábrázolandó objektumot megadni a `DefaultMutableTreeNode` osztály konstruktorában, illetve a `void setUserObject(..)` metódussal, lekérdezni pedig a `getUserObject()` metódussal lehet. A fa felépítésekor (az alkalmazás indításakor) az objektumok attribútumai az adatbázisból értéket kapnak, a jobboldali adatpanelen ezen

attribútumok aktuális értékei jelennek meg. Ha a felhasználó a felületen keresztül egy adott elem adott jellemzőjének értékét megváltoztatja, akkor a háttérben lévő, adott elemhez tartozó objektum állapota is változik, az adott attribútum felveszi az új értéket. Az adatbázisba viszont csak explicit mentés (a mentés gomb megnyomása) után kerülnek be a változások.

### **2.2.5. Táblázatok**

A táblázatok megjelenítésére **JTable** típusú objektumokat használtam, a kezelésükhöz szükséges típusok eléréséhez importálnom kellett a **javax.swing.table** csomagot is. Az adatmodelljüket az **AbstractTableModel** absztrakt osztály kiterjesztésével létrehozott saját osztályokkal valósítottam meg. A következő 6 absztrakt metódus implementációját kellett megadnom:

1. `int getRowCount()` a táblázat sorainak számát adja meg
2. `int getColumnCount()` a táblázat oszlopainak számát adja meg
3. `Object getValueAt(int arg0, int arg1)` a táblázat `arg0` sorának és `arg1` oszlopának metszésében lévő objektumot adja meg
4. `Class getColumnClass(int c)` a `c`-edik oszlop típusát adja meg
5. `void setValueAt(Object value, int row, int col)` beállítja a `value` objektumot a táblázat `row`-edik sorának és `col`-edik oszlopának metszéspontjába
6. `boolean isCellEditable(int row, int col)` megadja, hogy a táblázat `row`-edik sorának és `col`-edik oszlopának metszésében álló táblázatcella szerkeszthető-e

Az adatmodellhez – a folyamatos bővítések miatt – dinamikusan bővíthető adatszerkezetre volt szükségem, így az oszlopszámnak megfelelő darabszámú **Vector** típusú objektumot használtam. A táblázat sorainak számát ezen **Vector**ok aktuális mérete határozza meg. Hogy a táblázat értékeit módosítani tudjuk, a cellákat szerkeszthetőre kellett állítanom.

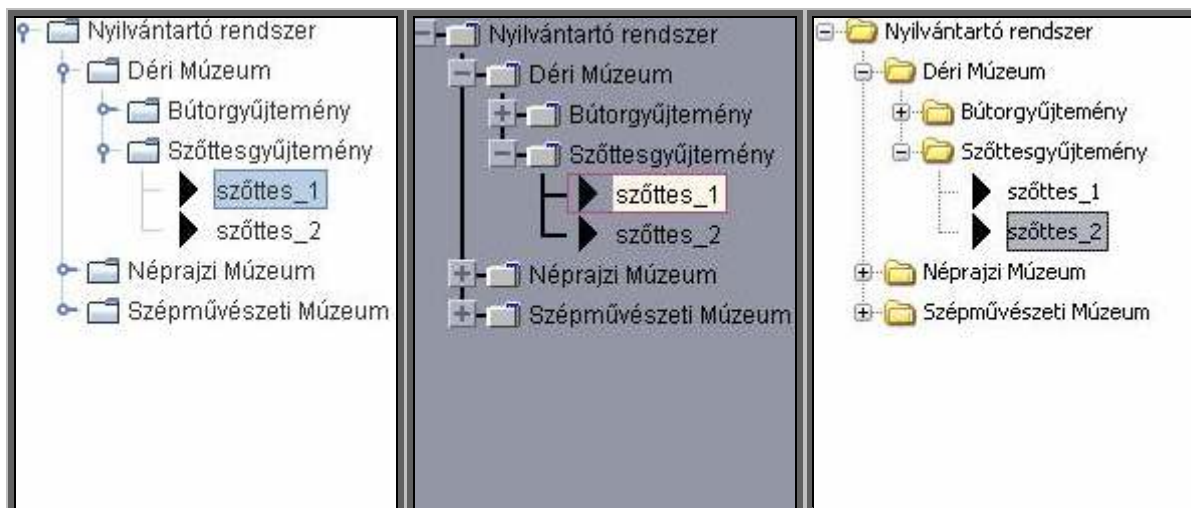
Táblázatokat egy múzeumi tárgy méret adatainak megadásánál, illetve a **Thesaurus**ok megjelenítésénél használtam. A mérethez tartozó táblázat oszlopai: a **Dimenzió**, a **Méret** és a **Mértékegység**. Az értékek egy előre meghatározott értékhalmból való kiválasztását lehetővé tévő táblázatok pedig két oszlopból állnak. A táblázatok első oszlopa egy kiválasztható komponens (**JCheckBox**), a második oszlopában pedig maga az érték jelenik meg.

### **2.2.6. Bemeneti/kimeneti mezők**

Az adatok megadása illetve megjelenítése JTextField illetve JTextArea típusú objektumokkal valósíthatóak meg. Az előbbi csak egysoros, míg az utóbbi többsoros szövegek megjelenítését teszi lehetővé. Méretük a konstruktorban adható meg. Mind a kétfajta mezőhöz saját osztályokat hoztam létre a JTextField és a JTextArea osztályok kiterjesztésével. Így tudtam nyilvántartani, hogy az adott mezőhöz mely mentésgomb tartozik, illetve eseménykezelőket regisztrálni a mező dokumentumtartalmának megváltoztatásának figyelésére. Adott esetben szükség van ezen eseményfigyelők be/kikapcsolására is.

### **2.2.7. A megjelenítés stílusa**

A felhasználói felületek és a rajtuk lévő komponensek megjelenítését elsősorban a megjelenítési stílus szabályozza. A megjelenítés stílusát az absztrakt LookAndFeel osztály reprezentálja. Leszármazott osztályai a MetaLookAndFeel, a MotifLookAndFeel, valamint a WindowsLookAndFeel osztályok, melyek már implementációt adnak minden Swing komponens megjelenítéséhez. A megjelenítés stílusa akár futási időben is megváltoztatható, ez azonban a felhasználó felületeken keresztül nem állítható át. A három beépített megjelenítési stílus közül a WindowsLookandFeel-t használtam.

**MetaLookAndFeel****MotifLookAndFeel****WindowsLookAndFeel**

### 2.2.8. Események kezelése

Grafikus felhasználói felületek esetén a Java a felhasználó és az alkalmazás közötti kommunikációt eseménykezeléssel valósítja meg, melynek eszközei a **java.awt.event** csomagban érhetőek el. Ilyen esetben nem alkalmazható a szekvenciális programozási technika, mivel a felhasználó tetszőleges sorrendben adhatja meg az adatokat (egyszerre akár több adatmező kitöltésével), majd egy vezérlőkomponens segítségével (általában egy nyomógomb megnyomásával) egy eseményt hoz létre, így értesíti az alkalmazást. Eseményvezérelt programok esetében mindig van a programban legalább két programszál, ami fut: az eseményeket az eseménysorba berakó és az azokat feldolgozó programszál. Mivel ezek a programszálak nem démon jellegűek, a program futása nem fejeződik be a `main(..)` metódus végének elérésekor. A grafikus alkalmazásunkat csak explicit módon, a `System.exit()` statikus metódus meghívásával fejezhetjük be.

A különböző grafikus komponensek különböző eseményeket válthatnak ki, melyek közös ősosztálya a **java.awt.AWTEvent** absztrakt osztály. Egy eseménypéldány információkat tartalmaz a keletkezéséről és a program keletkezés pillanatában lévő állapotáról. Minden eseményobjektum osztályának neve az „Event” szóra végződik.

Az események elkapása és feldolgozása (a kivételekhez hasonlóan) figyelő osztályok segítségével valósítható meg, melyek különböző interfészeken keresztül kezelhetők. Adott XXXEvent eseményre figyelő objektumokat az XXXListener nevű interfész reprezentálja. Az interfész metódusai a figyelt esemény bekövetkezésének különböző módzatait írják le. Ezen metódusok paraméterként megkapják az eseményt leíró XXXEvent eseményobjektumot, melyből különböző információk (pl. az eseményt kiváltó komponens) nyerhetők ki. Az adott XXXListener interfész metódusait implementálva adhatjuk meg, hogy mit történjen egy adott esemény bekövetkezésekor.

Minden olyan objektumhoz, amely eseményforrás lehet, hozzárendelhetőek a figyelő objektumok az `addXXXListener()` metódus segítségével. Ezen metódusok olyan objektumpéldányt várnak aktuális paraméterként, mely implementálja a megfelelő XXXListener interfészt.

A program futása során többfajta esemény is kialakulhat, melyek keletkezését különböző eseményfigyelőkkel lehet figyelni, illetve azokra reagálni. A következőkben az implementálás során felhasznált eseményfigyelőket részletesen tárgyalom.

### **2.2.9. A felhasznált eseményfigyelők**

#### **1. ActionListener**

A legáltalánosabb vezérlő komponens, a nyomógomb (JButton), illetve a menük (JMenu) által kiváltott eseményekre az ActionListener interfész segítségével reagálhatunk. Ez az interfész a **void actionPerformed(ActionEvent e)** metódus implementálását teszi kötelezővé. A paraméterként megkapott eseményobjektumból a String `getActionCommand()` metódussal egy, az adott eseményt kiváltó komponensre (nyomógomb vagy menüpont) jellemző String kapható meg, mely a komponens `setActionCommand(String s)` metódusa segítségével állítható be. Egy adott komponenshez figyelő objektum a `void addActionListener(...)` metódussal adható hozzá, melynek paramétere kötelezően egy olyan objektumpéldány, mely implementálja az ActionListener interfészt.

Az alkalmazásban nyomógombokat több helyen használtam: az előre rögzített értékalmazok megjelenítéséhez, a panelek mentéséhez, a táblázatok sorainak bővítéséhez, csökkentéséhez, a felhasználót értesítő párbeszédablakok (JDialog) bezárásához. Felbukkanó menü (JPopupMenu) segítségével módosítható a fastruktúra (elem hozzáadása/törlése), illetve a főablak felső részén látható főmenüvel valósítható meg a nyomtatás, valamint a kilépés.

#### **2. MouseListener**

Az egéreseemények a MouseListener interfész implementálásával kezelhetőek. Az egéreseemények nagyon sokfélék lehetnek, ezért ez az interfész összesen 5 metódus implementálását írja elő. Megadható, hogy mit történjen, ha egy komponensre rákattintunk (**mouseClicked(..)**), ha egy komponensen lenyomjuk, illetve felengedjük az egérgombot (**mousePressed(..)** , **mouseReleased(..)**), ha az egérmutatót a komponensre visszük (**mouseEntered(..)**) illetve elvisszük róla (**mouseExited(..)**). Ezen metódusok formális paramétere egy MouseEvent típusú objektum. Egy adott komponensnél nem volt szükségem arra, hogy az összes különböző egéreseeményre reagáljak, így nem akartam (nem is tudtam) implementálni az összes előírt metódust. Így az adott interfész implementálása helyett egy másik utat választottam. Minden XXXListener osztályhoz tartozik egy XXXAdapter osztály,

mely üres törzzsel implementálja az összes, interfészben előírt metódust. A figyelő létrehozásánál egy névtelen osztályt példányosítottam, mely a `MouseAdapter` leszármazottja és melyben csak a használni kívánt metódust implementáltam újra (overriding).

Lehet, hogy egy egéreseményre különbözőképpen szeretnénk reagálni attól függően, hogy a felhasználó melyik egérgombot nyomta meg. A paraméterben megkapott egéresemény int `getModifiers()` metódusának visszatérési értéke egy bitmaszkot határoz meg, melyet a `MouseEvent` osztályban található, előre definiált bitmaszkokkal (`BUTTON1_MASK`, `BUTTON2_MASK`, `BUTTON3_MASK`) összehasonlítva meghatározható, hogy a baloldali, középső vagy jobboldali egérgommbal váltottuk-e ki az adott egéreseményt. A kattintás darabszámát pedig az egéresemény int `getClickCount()` metódus meghívásával kaphatjuk meg.

Egéresemény figyelőt a fastruktúránál használtam. Amint egy faelemre a bal egérgommbal kétszer rákattintunk, a jobboldali adatpanelen az adott faelem által reprezentált tárgyra vagy intézményre jellemző adatokat láthatjuk, illetve a jobb egérgommbal kattintva egy, a fastruktúra módosítását (bővítést/törlését) lehetővé tévő felbukkanó menü jelenik meg.

### **3. TreeSelectionListener, TreeModelListener**

A fastruktúra módosítása által kiváltott események kezeléséhez két interfészre, a `TreeSelectionListener`-re és a `TreeModelListener`-re volt szükségem.

A **`TreeSelectionListener`** mindössze csak a **`void valueChanged(...)`** metódust tartalmazza. Ezen metódus implementálásával adható meg, hogy mi történjen, ha a fastruktúrában egy új elemet jelölünk ki (rákattintunk). Nagyon fontos az éppen kiválasztott faelemnek a nyilvántartása, hiszen ez határozza meg, hogy a jobboldali panelen milyen adatok jelennek meg, illetve a nyomtatás menüpont meghívása után is a kijelölt faelem adatai kerülnek kinyomtatásra. Egy `TreeSelectionListener` a fastruktúrát jelképező `JTree` típusú objektumhoz az `addTreeSelectionListener(...)` metódussal adható hozzá.

Egy **`TreeModelListener`** figyelőobjektumot az `addTreeModelListener(..)` metódus segítségével a fastruktúra adatmodelljéhez kell hozzáadni, melyet a `JTree` típusú objektum `getModel()` metódusa szolgáltatja.

A `TreeModelListener` a következő 4 metódust tartalmazza: **`void treeNodesInserted(..)`**, **`void treeNodesRemoved(..)`**, **`void treeNodesChanged(..)`**, **`void treeStructureChanged(..)`**. Az első metódus akkor hívódik meg, amikor egy új elemet helyezünk el a fában, a második, amikor egy meglévőt törölünk, a harmadik, ha egy faelem értéke (neve), a negyedik pedig, mikor a fastruktúra szerkezete változik meg. Ezen metódusok közül csak az első hármat használtam, egy adott elem beszúrásakor, törlésekor, illetve átnevezésekor az adatbázis karbantartásához, a megfelelő adatbázis-műveleteket (`insert`, `delete`, `update`) reprezentáló metódusok meghívásához.

#### **4. DocumentListener**

Az adatok megadására szolgáló bemeneti mezők (`TextField`, `TextArea`) tartalmát dokumentumoknak nevezzük, melyek megváltozása egy, a `DocumentListener` interfészt implementáló objektummal figyelhető. Dokumentum-esemény keletkezik, amint a bemeneti mezőbe egy karaktert írunk, illetve abból egy karaktert kitörölünk. Ezekre az eseményekre a **`void insertUpdate(..)`**, illetve a **`void removeUpdate(..)`** metódusokkal reagálhatunk. Egy bemeneti mezőhöz tartozó dokumentumot a `getDocument()` metódussal kapjuk meg, melyhez egy dokumentumfigyelő objektumot az `void addDocumentListener(..)` metódussal rendelhetünk.

A keletkezett dokumentum-események segítségével követtem nyomon, hogy egy adott attribútum értékét a felhasználó megváltoztatta-e vagy sem, mivel azokat a paneleket, amelyeken az értékek nem változtak meg, szükségtelen menteni, így jelentősen csökken az alkalmazás és az adatbázis közötti adatforgalom. Azokon a paneleken, amelyen valamely adat megváltozott, a MENTÉS gomb aktívvá válik, így a felhasználó is tájékozódhat, hogy mely panelen lévő adatokat változtatta meg. Dokumentum-esemény nemcsak akkor keletkezik, ha a felhasználó módosítja a bemeneti mezők tartalmát, hanem akkor is, amikor a rendszer írja bele az adatokat az adatbázisból. Az ilyen esetekben bekövetkezett dokumentum-eseményekre nem kívántam reagálni, így ekkor (egy logikai típusú változó átállításával) kikapcsoltam az eseményfigyelőt.

### **2.2.10. Nyomtatás**

A Java a nyomtatási modellt a **java.awt.print** csomag segítségével valósítja meg. A nyomtatási képek elkészítéséhez egy külön osztályt (Nyomtato.java) készítettem, amely konstruktorában meg kell adni a kinyomtatni kívánt faelemet reprezentáló objektumot. Ez az osztály implementálja a Printable interfészt, mely mindössze csak a print metódust tartalmazza. Ennek a metódusnak három paramétere van: a nyomtatási képet a Graphics típusú paraméter drawXXX(...) metódusai segítségével készítettem el hasonlóképpen ahhoz, mintha azokat egy panelre rajzolnám. Az adatokat a drawString(...) metódus segítségével „rajzoltam” le, x és y koordináta paraméterekkel megadva az egyes szövegrészek pontos elhelyezkedését. Egy konkrét múzeumi tárgyra vagy intézményre jellemző adatok hasonló elhelyezkedés szerint jelennek meg, mint a képernyőpaneleken, azzal a különbséggel, hogy az üres, értékkel nem rendelkező adatok nem kerülnek kinyomtatásra. A print metódus második paramétere egy PageFormat objektum, mely az oldalbeállításokat (pl. margók mérete) tartalmazza. A harmadik paraméter pedig az oldalszám. Az első oldal a nullás indexű. Amikor az utolsó oldal is kinyomtatásra került, a print metódus a NO\_SUCH\_PAGE konstanssal tér vissza.

Ezután a PrinterJob osztály statikus getPrinterJob() metódusával létrehoztam egy PrinterJob objektumot, majd ennek az objektumnak a setPrintable(..) metódusa segítségével beállítottam a nyomtatás módját, illetve a print metódus meghívásával elindítottam a nyomtatást.

A print() metódus ellenőrzött kivételeket válthat ki, melyek figyelése kötelező. Az itt említett kivételek közé tartozik a PrinterException, melynek keletkezése végzetes hibára utal, így ebben az esetben a nyomtatás azonnal felfüggesztésre kerül.

#### **A nyomtatást végző kódrészlet:**

```
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(new Nyomtato(TreePanel.kivalasztott));
try{ job.print();
    }catch(Exception e){
        JOptionPane.showMessageDialog(Foablak.ablak,e.getMessage(),"ERROR",
            JOptionPane.ERROR_MESSAGE); }
```



## **2.3 Az adatbázis és a felhasználói felületek összekapcsolása**

### **2.3.1. JDBC, a kapcsolat felvétele, lebontása**

A Java programok adatbázissal való kommunikációját a JDBC (Java Database Connectivity) programozói interfész biztosítja. A JDBC API típusait a **java.sql** és a **javax.sql** csomagok tartalmazzák. A JDBC API szolgáltatásait három csoportba soroljuk: összekapcsolódás az adatbázissal, SQL utasítások végrehajtása, SQL lekérdezések eredményeinek feldolgozása. A JDBC használatával a Java adatbázis-kezelő programok nemcsak platformfüggetlenekké, hanem még adatbázis-kezelőktől is függetlenekké válnak. A program közvetlenül az adatbázis-kezelő rendszerrel kommunikál. Maga az adatbázis akár másik gépen is elhelyezkedhet, mint ahol a program fut, az adatforgalom pedig a hálózaton keresztül történik. Ez az esetet kliens-szerver konfigurációnak nevezzük, ahol az adatbázist tároló gép a szerver, a programot futtató gép pedig a kliens. A JDBC hívások végrehajtásakor mindig fizikailag is fel kell venni a kapcsolatot az adatbázissal. A hívások megfelelő értelmezését és kiszolgálását a JDBC meghajtóprogramok végzik el azáltal, hogy implementálják a Driver interfészt. A használni kívánt meghajtóprogramot a DriverManager osztály registerDriver() metódusával regisztrálni kell. Ezután következhet a kapcsolat felvétele.

A program és az adatbázis közötti kapcsolatot egy Connection objektum reprezentálja. Egy program egyszerre több kapcsolatot is fenntarthat ugyanazon, vagy akár több különböző adatbázissal. Egy kapcsolat a kiadott SQL utasításokat és azok végrehajtásának eredményeit foglalja magába. A kapcsolat felvételéhez szükséges az adatbázis URL megadása, mely az elérni kívánt adatbázist jelöli ki. Szintaxisa a következő:

*jdbc : alprotokoll : adatforrás leírása*

A protokoll neve jdbc. Az *alprotokoll* nevét a megfelelő meghajtóprogram forgalmazója határozza meg. Az *adatforrás leírása* a kért adatbázis eléréséhez szükséges további adatokat (pl.: az adatbázis neve és hálózati címe, a felhasználó neve és jelszava) tartalmazza. A program elkészítéséhez és teszteléséhez az egyetemi Oracle adatbázisszervert használtam, a következő adatbázis URL-lel: **jdbc:oracle:thin:@oracle.inf.unideb.hu:1521:oracle.**

Az adatbázis-kapcsolat felvétele a DriverManager getConnection() metódusának meghívásával történik, amelynek paramétere az elérni kíván adatbázis URL címe. Egy új adatbázis-kapcsolat alapértelmezés szerint automatikus nyugtázási móddal jön létre, azaz minden SQL utasítás befejeződése után automatikusan meghívódik a commit() metódus. Ha az automatikus nyugtázási módot kikapcsoljuk (a setAutoCommit(false) metódussal), akkor a programnak magának kell vezérelnie a tranzakció-kezelést a commit() és a rollback() metódusok segítségével. Az alkalmazás elkészítésénél az automatikus nyugtázási módot használtam.

Ha az adatbázis-kapcsolat során bármiféle hiba történik, akkor egy SQLException kivétel váltódik ki, melynek szövegét (és némi magyarázatot) egy felbukkanó ablakban (JDialog) látja a felhasználó.

A kapcsolatot lezáró close() metódus szabadítja fel az adatbázis-kapcsolat által lefoglalt JDBC erőforrásokat. Ez a metódus a kapcsolatobjektum megsemmisítésekor és bizonyos fatális hibák fellépésekor automatikusan is meghívódik.

### **2.3.2. sql ↔ java típusmegfeleltetés**

Mivel a java és az sql típusok nem azonosak, ezért biztosítanom kellett annak a lehetőségét, hogy az alkalmazás írni/olvasni tudja az adatbázis sql típusok adatait. Minden sql típusnak megfeleltethető valamely java osztály, alap- vagy összetett típus. Ez a megfeleltetés akkor jut szerephez, amikor a java típusú adatokat ki kell írni az adatbázisba, vagy az sql típusú értékeket be kell olvasni az adatbázisból. Vannak automatikus konverziók is, de mivel a felhasználói SQL adattípusokat az adatbázis szintjén saját magam definiáltam, ezért ezek pontos leképezése a megfelelő felhasználói java típusra is az én feladatom volt. A használandó megfeleltetéseket egy java.util.Map típusú objektum tartalmazza, amely a felhasználói SQL típusok nevéhez egy java.lang.Class objektumot rendel. Ezen Class objektum a felhasználói SQL típus értékét reprezentáló osztályt adja meg, amelynek implementálnia kell az **SQLData** interfészt. Ez az interfész a következő metódusok implementálását írja elő:

readSQL: akkor hívja meg a JDBC driver, ha be kell olvasni az adatbázisból egy reprezentált típusú objektumot. Paraméterként megkapjuk a reprezentált típus adatbázisbeli SQL nevét, valamint egy SQLInput objektumot, melynek segítségével beolvashatjuk a típust felépítő sql adatokat.

writeSQL: akkor hívja meg a JDBC driver, ha ki kell írni az adatbázisba egy reprezentált típusú objektumot. Paraméterként kapunk egy SQLOutput objektumot, melynek segítségével kiírhatjuk a típust felépítő SQL adatokat.

Tehát egy SQLData-val reprezentált felhasználói SQL típus olvasásakor a JDBC driver meghívja annak readSQL, míg kiírásakor a writeSQL metódusát. A paraméterekben kapott SQLInput, illetve SQLOutput objektumok tulajdonképpen az adatbázis-kapcsolat SQL adatfolyamát reprezentálják.

### **2.3.3. SQL utasítások végrehajtása java programnyelvből**

Az SQL utasításokat Java környezetből a következő három interfész segítségével lehet végrehajtani:

- Statement: egyszerű SQL utasítások végrehajtására használható.
- PreparedStatement: bemenő paraméterekkel is rendelkező SQL utasítások végrehajtására használható.
- CallableStatement: ki/bemenő paraméterekkel is rendelkező tárolt SQL eljárások végrehajtására használható.

A **Statement** interfész tartalmazza az SQL utasítások végrehajtásához és a visszaadott eredmények feldolgozásához szükséges alapmetódusokat. Egy Statement objektum a fennálló kapcsolatot reprezentáló Connection objektum createStatement() metódusával hozható létre. Az SQL utasítást azonban nem az objektumot létrehozó, hanem az azt végrehajtó egyik metódusnak kell megadni. Az SQL utasítást lezáró adatbázis specifikus jelsorozatot (pontosvesszőt) a végrehajtandó SQL utasítás szövegének a végén nem kell kitenni, ezt a meghajtóprogram automatikusan megteszi helyettünk. Egy Statement objektumot három metódus segítségével is végre lehet hajtatni. (executeQuery(...), executeUpdate(...), execute(...)).

A **PreparedStatement** interfész a Statement interfész kiterjesztettje, attól két dologban különbözik. Először is ezen interfész egy példánya már tartalmaz egy SQL utasítást, méghozzá előfordított formában, másodsor pedig tartalmazhat bemenő paramétereket is. Ezen paramétereket az SQL utasításon belül kérdőjelek jelölik, mivel értékük az utasítás létrehozásakor még nem ismert. Az utasítás végrehajtása előtt minden bemenő paraméternek értéket kell adni a megfelelő set(..) metódusok valamelyikével. Mivel ez az sql utasítástípus előfordított, ezért gyorsabb a végrehajtás, mint a Statement objektumok esetén. Egy

PreparedStatement interfészt megvalósító objektum a fennálló kapcsolatot reprezentáló Connection objektum prepareStatement() metódusával hozható létre. A végrehajtandó SQL utasítást az objektumot létrehozó, nem pedig valamelyik, az azt végrehajtó metódusnak kell megadni. Az előfordított SQL utasítás létrehozásakor maga az utasítás egyből átadódik az adatbázisnak. Végrehajtás előtt minden bemenő paraméternek be kell állítani az aktuális értékét. Egy bemenő paraméter értékét a *setTípusnév* metódusokkal lehet beállítani. Ezen metódusok első argumentuma mindig a bemenő paraméter sorszáma (1-től kezdve), második argumentuma pedig a beállítandó értéket adja meg. Bemenő paraméternek NULL érték a setNull() metódussal adható. Végrehajtásra a Statement-nél ismertetett három metódus használható. Ezen metódusoknak nem kell paramétert megadni, mivel a végrehajtáskor már ismert az elvégzendő SQL utasítás.

#### **2.3.4. Adatbázisműveletek a Java oldalon**

Az adatbázis-műveleteket végrehajtó metódusokat az Adatbazis.java osztályban helyeztem el, melyek publikusak és statikusak, mivel a program bármely pontjáról elérhetőeknek kell lenniük. Ezen metódusok segítségével kezeltem a végrehajtandó DML utasításokat.

A SELECT lekérdező utasításokat végrehajtó metódusok visszatérési értéke egy Vector típusú objektum, mely a lekérdezés eredményét reprezentáló objektumokat tartalmazza. Az INSERT, DELETE és UPDATE utasításokat végrehajtó metódusok visszatérési értéke void. Azon adatbázis-műveletek végrehajtásánál, melyek általam definiált adatbázis objektumtípusokat érintenek, PreparedStatement objektumokat használtam. A többi adatbázis-műveletet pedig Statement objektum segítségével hajtottam végre.

A Statement és a PreparedStatement objektumok execute() és executeQuery() metódusai ellenőrzött kivételeket válthatnak ki. A kivételek kezelésére két lehetőség közül választhattam. Vagy továbbadom a kivételt a hívó környezetnek, vagy a keletkezés helyén lekezelem őket. Az utóbbit választottam, így ezeket a metódusokat egy kivételkezelő (try-catch) blokkban helyeztem el. Egy kivétel bekövetkezéséről és a hiba okáról a felhasználó egy felbukkanó ablakon keresztül értesül.

A Java és az Oracle megfelelő kommunikációjához két jar fájlt kellett a programhoz csatolnom. Az **ojdbc14.jar** az adatbázis elérését, míg az **orai18n.jar** az adatok mozgathatóságát szükséges karakterkódolásokat biztosítja.

## **3. fejezet**

### Felhasználói dokumentáció

### 3.1. Az alkalmazás indítása

Az alkalmazásunkat a nyilvanto.jar fájl segítségével indíthatjuk el, melynek eredményeképpen egy ablak jelenik meg. Az alkalmazás ezen az ablakon keresztül kezelhető. Az ablak bal oldalán álló fastruktúrában a nyilvántartott intézmények láthatóak, a jobboldali adatpanel egyelőre üres, rajta még semmi sem látható (később itt jelennek majd meg részletesen az adott elemre jellemző adatok). Minden intézmény neve előtt található egy + jelű gomb, mely megnyomásával az adott intézményhez tartozó gyűjteményeket jeleníthetjük meg. A gyűjtemények lenyitása után pedig megjelennek a hozzátartozó konkrét tárgyak. Ez a fastruktúra szélességében tetszőlegesen, míg mélységében 3 szintig bővíthető.

### 3.2. A fastruktúra bővítése

Ha egy elemre a jobb egérgombbal rákattintunk, akkor egy menü jelenik meg, mely az ÚJ és a TÖRLÉS menüpontokat tartalmazza. Az ÚJ menüpontra kattintva a kijelölt elem alá új elemet hozhatunk létre, melynek kezdeti neve egy előre beállított érték lesz, ez az érték azonban az átnevezés során tetszőleges értékre módosítható. Egy konkrét tárgy (harmadik szintű elem) alá újabb elem már nem hozható létre, ebben az esetben az ÚJ menüpont nem választható.

### 3.3. Elem törlése a struktúrából

Egy elemet eltávolítani a rendszerből a TÖRLÉS menüpont segítségével lehet. A struktúrából csak olyan elemek törölhetőek, amelyek alá további elemek nem tartoznak. A többi elemnél a törlés menüpont nem választható. Ha olyan elemet szeretnénk törölni, amely alá még további elemek tartoznak, akkor először az alsó szinteken lévő elemeket ki kell törölnünk. A Törlés menüpont kiválasztása után egy figyelmeztető ablak hívja fel a felhasználó figyelmét, melyen lévő OK gomb megnyomása után a faelemet véglegesen kitöröljük az adatbázisból. **Visszaállításra lehetőség nincs!**

### 3.4. Elem átnevezése

Ha egy faelemre a baloldali egérgombbal kétszer rákattintunk, akkor a faelem átnevezhetővé válik. A faelem neveinek megadásakor (létrehozáskor és átnevezéskor) figyelni kell arra, hogy egyetlen faelemnek sem lehet két ugyanolyan nevű gyermeke. Amennyiben mégis ez a helyzet alakulna ki, akkor a név megváltoztatására egy üzenet figyelmezteti és kötelezi a felhasználót.

### 3.5. Egy adott elemre jellemző adatok megjelenítése

Az ablak jobb oldalán láthatóak a kiválasztott faelemre jellemző adatok. Az adatok akkor jelennek meg, amint egy faelem kiválasztásra kerül (rákattintunk). Amennyiben egy intézményt választunk ki, akkor a jobboldali adatpanelen az összes adat megjelenítésre kerül, mivel a rendelkezésre álló hely elegendő. Tárgyaknál viszont más a helyzet a rendelkezésre álló nagy adattömeg miatt. A tárgyra jellemző adathalmaz 4 csoportból áll, melyek 4 különböző, egymás alatt elhelyezkedő panelen jelennek meg. Mindegyik panel lenyitható, illetve visszacsukható a +/- gomb megnyomásával, így az adatok egyszerűen megjeleníthetők vagy elrejtethetők. Amennyiben az adatok nem férnek el a rendelkezésre álló helyen, úgy görgetésre a jobboldali görgetősáv segítségével van lehetőségünk.

### 3.6. Az értékek megadása, módosítása

A paneleken szereplő adatok közül az intézmény neve, a gyűjtemény és a leltári szám megadása kötelező, tehát ezen jellemzők már biztos, hogy rendelkeznek értékkel. A többi adat megadása opcionális. Ha egy jellemzőnek még nem adtunk értéket, akkor a hozzá tartozó megjelenítő mező üres, melyre kattintva konkrét értéket adhatunk meg. Ha a jellemző már rendelkezik értékkel, akkor ez az érték a megjelenítő mezőben megjelenésre kerül, mely szintén átírható.

Azoknál a jellemzőknél, melyek csak egy előre megadott értékhalmból vehetik fel értéküket, található egy nyomógomb (T) is. A nyomógombra kattintva egy ablak jelenik meg, ahol az értékhalmoz elemei láthatóak. Ez az értékhalmoz a + gomb megnyomásával bővíthető. Csak olyan új értéket adhatunk meg, amely eddig még nem szerepelt az értékhalmban. Az átnevezés az elemre való kétszeri kattintással lehetséges. Egy adott érték a – gomb megnyomásával törölhető. Törlésre nem mindig van lehetőség. Ha egy olyan adatot akarunk

törölni, amelyre egy másik elem már hivatkozik, akkor a törlés nem végrehajtható. Miután kiválasztottuk a szükséges értéket az értékhalmból (az előtte álló négyzet bepipálásával), két lehetőség közül választhatunk. Az OK gomb megnyomásával a kiválasztott érték a megjelenítő mezőbe kerül, a MÉGSE gomb esetén a megjelenítő mező változatlan marad.

### 3.7. Az adatok mentése

Mentésre a panelek jobb felső sarkában elhelyezkedő MENTÉS gomb megnyomásával van lehetőségünk, mely kezdetben inaktív (nem választható). Amennyiben egy adat módosításra került, a mentés gomb aktívvá válik, melynek megnyomása után a panelen lévő összes adat mentésre kerül. A sikeres mentésről egy üzenet tájékoztatja a felhasználót. Ezután a mentés gomb ismét inaktívvá válik, mivel újabb mentés csak akkor válik szükségessé, amint valamely jellemző értéke ismét megváltozik. A 4 adatpanel külön-külön menthető.

### 3.8. Képek kezelése

A 4. adatpanel legalján egymás mellett, kicsinyítve az adott tárgyat bemutató képek jelennek meg. A kicsinyített képekre kattintva a kép teljes méretében egy külön ablakban látható. Új kép hozzáadására az ikonok felett lévő gomb megnyomásával van lehetőségünk. Ekkor egy fájlkieválasztó ablak jelenik meg, mely segítségével megkereshetjük, kiválaszthatjuk, és nyilvántartásba vehetjük az új képet.

### 3.9. A menü funkciói: nyomtatás, kilépés

A főablak felett elhelyezkedő menü két menüpontot tartalmaz. A NYOMTATÁS menüpontra kattintva kinyomtathatjuk egy adott intézmény vagy tárgy adatait. A nyomtatás csak akkor lehetséges, ha a baloldali fastruktúrában egy intézmény vagy egy tárgy van kijelölve. Egyébként hibaüzenetet kapunk. A nyomtatás során az összes nem üres jellemző kinyomtatásra kerül.

A programból való kilépésre két lehetőségünk van. Megnyomhatjuk a főablak jobb felső sarkában lévő X jelet, illetve a KILÉPÉS menüpont segítségével. Mindkét esetben szükség van még egy megerősítésre. A megerősítés visszavonható, ekkor a nem lépünk ki a programból, megerősítés után pedig a program befejezi a működését.



# 4. fejezet

## Összefoglalás

Az implementációt a leendő felhasználóval közösen megfogalmazott követelményekhez igazodva készítettem el. Az alkalmazásfejlesztés utolsó, ám legfontosabb lépése a tesztelés. Igyekeztem az alkalmazást minél jobban bolondbiztossá tenni, hogy az esetlegesen fellépő hibák még futás közben korrigálhatóak legyenek, ne okozzák a program működésének befejezését. Az elkészült programon különböző, ám korántsem elegendő teszteseteket alkalmaztam. Az elkövetkezendő időkben további – általam és a leendő felhasználó által végzett - tesztelések következnek, mielőtt az alkalmazást elkészültnek minősíthetjük.

A megvalósítottakon kívül még számos igény, illetve követelmény született meg, melyek megvalósítására az alkalmazás továbbfejlesztése remek lehetőséget biztosít. Hasznos lenne, ha a megjegyzések, szabadon megadható szövegek szövegszerkesztő program segítségével megformázhatóak lennének, mivel így a főbb részek különböző betűtípusok, betűméretek, színek segítségével kiemelhetőek, hangsúlyosabbá tehetőek. Egy múzeumi tárgyat szemléteesebben lehetne ábrázolni, illetve bemutatni, ha a leíró adatok, szövegek, számok mellett multimédiás adatok (képek, hangok, videófájlok) tárolására, illetve megjelenítésére is lehetőségünk lenne. A felmerült igények közül a legegyszerűbb, extra funkció a tárgyak készítésének, illetve a gyűjtés helyének térképen való megjelenítése. Ezzel a módszerrel könnyen szemléltethető lenne, hogy az adott tárgyak az ország mely területeiről származnak.

Végül, de nem utolsó sorban itt szeretném megragadni az alkalmat, hogy köszönetet mondjak mindazoknak, akik segítettek a diplomamunkám elkészítésében. Köszönöm témavezetőmnek, Kollár Lajosnak, hogy bármikor, bármilyen kérdéssel fordulhattam hozzá. Köszönet illeti továbbá az ötlet kigondolóját és az alkalmazás leendő, remélhetőleg elégedett felhasználóját, Kiri Editet is.

# 5. fejezet

## Irodalomjegyzék

1. Nyékiné G. Judit: Java 2 útikalauz programozóknak 1.3 I-II kötet
2. L. Nagy Éva: SQL röviden
3. Gábor András - Juhász István: PL/SQL programozás
4. Java API: <http://java.sun.com/j2se/1.5.0/docs/api/>

# 6. fejezet

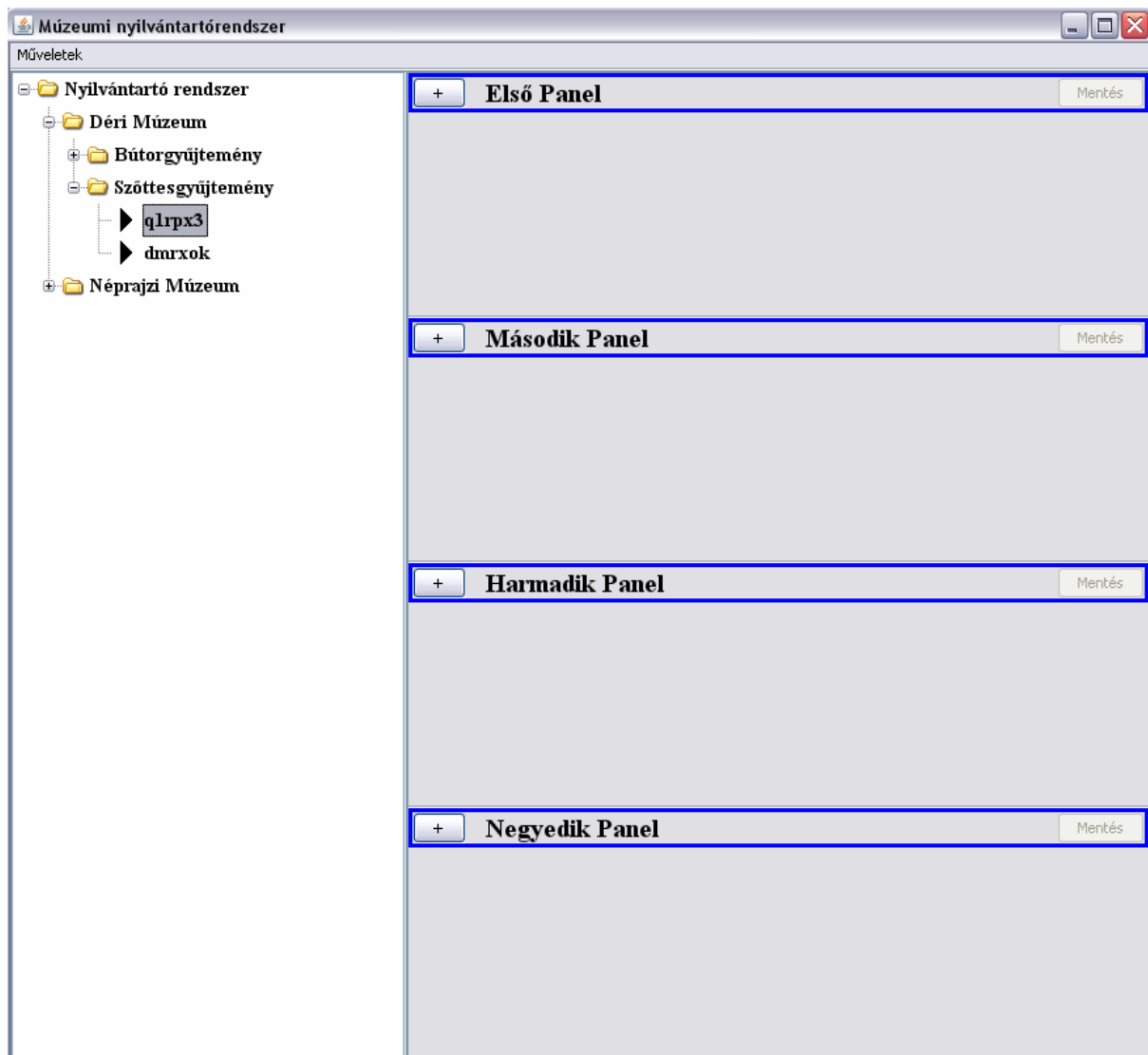
## Függelék - Képek

Az alábbiakban az alkalmazást egy szöttezen keresztül, néhány kép segítségével mutatom be.

Intézmény Panel	
NÉV	Déri Múzeum
Megye	Hajdú-Bihar
Település	
Kistérség	
Régió	
Helység	Debrecen
Utca	Déri tér
Házsám	1
Irányítószám	4026

1. kép Egy intézményről nyilvántartott adatok

A jobboldali adatpanelen a baloldalon kiválasztott intézmény adatai jelennek meg



**2. kép A szóttessről nyilvántartott adatok**

**A jobboldali adatpanelen a 4 alpanel látható, egyelőre még zárt állapotban**

Múzeumi nyilvántartórendszer

Műveletek:

- Nyilvántartó rendszer
  - Déri Múzeum
    - Bútorgyűjtemény
    - Szóttessgyűjtemény
      - q1rpx3**
        - dmxok
    - Néprajzi Múzeum

**Első Panel** [Módosít]

Leltári szám: q1rpx3

Szintek: Déri Múzeum  
Szóttessgyűjtemény  
q1rpx3

Hivatalos név: abrosz [T]

Helyi név: feliratos abrosz

**Második Panel** [Módosít]

**Harmadik Panel** [Módosít]

**Negyedik Panel** [Módosít]

3. kép Az első alpanelhez tartozó adatok

The screenshot shows a web application titled "Múzeumi nyilvántartórendszer". On the left is a tree view of the system structure:

- Nyilvántartó rendszer
  - Déri Múzeum
    - Bűtesgyűjtemény
      - Szövegszűjtemény
        - qlrpx3
        - dmrxok
  - Néprajzi Múzeum

The main area displays the "Második Panel" with the following fields:

Darabszám	1								
Leírás	abrosztöredék 1882-es szőtt dátummal								
Készítés helye	Endröd								
Készítés ideje	1930-as évek								
Készítő	<table border="1"><tr><td>Neve</td><td>Szabó Bólinté Tímea Margit</td></tr><tr><td>Vallása</td><td>katolikus</td></tr><tr><td>Nemzetisége</td><td>magyar</td></tr><tr><td>Társadalmi helye</td><td></td></tr></table>	Neve	Szabó Bólinté Tímea Margit	Vallása	katolikus	Nemzetisége	magyar	Társadalmi helye	
Neve	Szabó Bólinté Tímea Margit								
Vallása	katolikus								
Nemzetisége	magyar								
Társadalmi helye									

4. kép A második alpanelhez tartozó adatok (1.rész)

**Múzeumi nyilvántartásrendszer**

Művelők:

- Nyilvántartó rendszer
  - Értéi Múzeum
    - Bútorgyűjtemény
      - Széltessgyűjtemény
        - díszítőminta
          - díszek
- Néprajzi Múzeum

**Díszítőminta**

anyaga: panut T

szakneve: ☐ panut ☐ váson ☐ félpánutos váson ☐ kendővászón ☐ panukvászón + -

helyi neve:

színe:

széleldolgozás: OK Cancel

**Technika**

szedett szöveg: T

**Méret / Terjedelem**

A	B	C
szélesség	110	cm
hosszúság	81	cm

+

-

T

**Állapot**

lyukas, kopott T

**Lelőhely**

tulajdonosnál T

5. kép A második alpanelhez tartozó adatok (2.rész)

A díszítőminta anyaga Thesaurus táblázat segítségével kap értéket

The screenshot shows a web application titled "Múzeumi nyilvántartórendszer". On the left is a tree view of the system structure:

- Nyilvántartó rendszer
  - Déri Múzeum
    - Bútergyűjtemény
      - Szállésgyűjtemény
        - qlrpx3
        - dmrxok
  - Néprajzi Múzeum

The main area displays three panels. The third panel, "Harmadik Panel", is active and contains the following data fields:

Megszerzés módja	ajándék	[T]
Gyűjtés ideje	1999.04	
Gyűjtés helye	Endröd	[T]
Gyűjtő neve	Kiri Edit	[T]
Eladó neve		[T]
Eladó születési éve		
Vételár	0	
Pénznem	forint	[T]
Adattári szám:	1011	
Leltározó neve:	Kiss Viktória	[T]
Leltározás ideje:	2006-04-28	
Örzési hely:	Magyarország, Debrecen	

6. kép A harmadik alpanelhez tartozó adatok

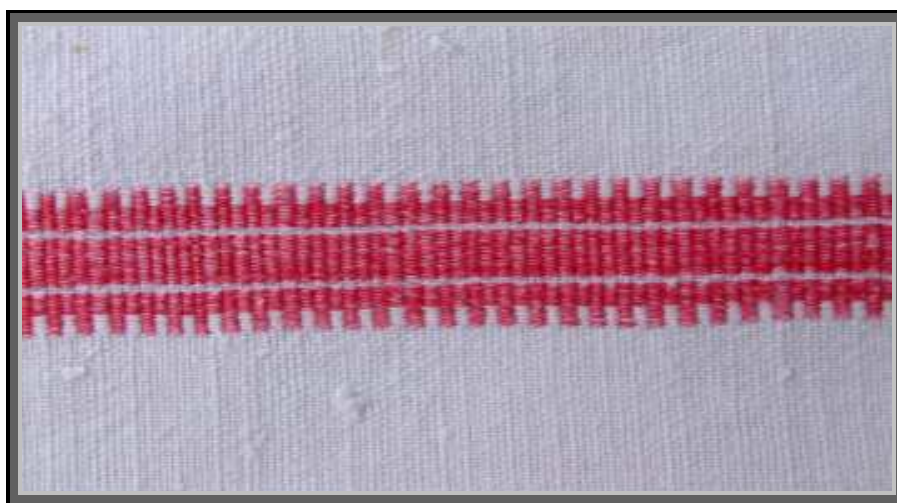


The screenshot displays the 'Múzeumi nyilvántartórendszer' (Museum Inventory System) interface. On the left is a tree view of the system structure, including 'Déri Múzeum', 'Bátorgyűjtemény', 'Szóttessgyűjtemény' (with sub-items 'qlrpx3' and 'dlorxok'), and 'Neprajzi Múzeum'. The main area shows the 'Negyedik Panel' (Fourth Panel) with the following fields:

- Revízió éve** (Revision year): 2002
- Restaurálás** (Restoration): nincs restaurálva eredeti állapot (not restored, original state)
- Megjegyzés** (Note): [Empty text area]
- Szakirodalmi hivatkozás** (Literature reference): Kiri Edit (2007): Szóttések a Békés megyei Endrődi Tájház és Helytörténeti Gyűjteményben
- Képi hivatkozás** (Image reference): [Empty text area]

At the bottom of the panel, there is a 'Tárolás' (Storage) button and a 'Galleria' section displaying a row of six small thumbnail images of textile samples.

**7. kép A negyedik alpanelhez tartozó adatok**  
A panel alján a szóttessről készült képek kicsinyítve, melyek teljes nagyságban is megtekinthetők



8. kép A fent bemutatott szőttesről készült képek teljes nagyságban