*Alkalmazott Matematikai Lapok* **26** (2009), 273-287.

# MODELLVEZÉRELT SZOFTVEREK KÉSZÍTÉSE II.

KILIÁN IMRE

## 1. Bevezetés

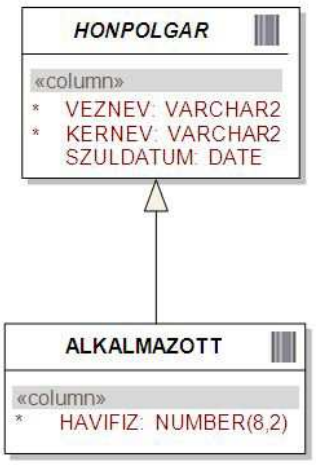
A cikk első részében (Alkalmazott Matematikai Lapok, **26** (2009), 1–7. oldal) bemutattuk a modellvezérelt szoftverek általános felépítését, és elemeztük azok alkalmazási lehetőségeit. Ilyenek a szoftver fejlesztési és tervezési gyakorlatban többféle helyzetben is felmerülnek, ezek kielégítése kétszintű szerkezettel – a modellszint tárolásával könnyebb és kevésbé kockázatos lehet. A cikk jelen, második szakaszában erre vonatkozó példákat mutatunk be.

### 1.1. UML testreszabása

A példákban az UML következő olyan testreszabásait alkalmazzuk, amelyek nem magától értetődőek.


**1. ábra.** UML osztályok megfeleltetése

*Megfeleltetés (correspondence):* A megfeleltetés olyan sztereotípus, amely két osztály közötti 0..1–0..1, vagy szűkebb többszörösségű kapcsolatra alkalmazható, és az egymásnak megfeleltethetőség intuitív fogalmát fejezi ki, vagyis azt, hogy mindkét osztály példányainak van egy olyan részhalmaza, amelyek egymásnak kölcsönösen egyértelműen megfeleltethetők. Ha ez a részhalmaz mindkét esetben a teljes példányhalmaz, akkor a többszörösség éppen 1–1, a megfeleltetés pedig *szoros* lesz. Ha 0..1–1, akkor a bal kapcsolatvég minden egyes példányához létezik megfelelő elem a jobb kapcsolatvégről. Ha 1–0..1, akkor fordítva. Ha a többszörösség mindkét kapcsolatvégen 0..1, akkor a megfeleltetés *laza*, mindkét kapcsolatvégen lehetnek olyan példányok, amelyek nem vesznek részt a megfeleltetésben, vagyis amelyeknek nincs megfelelőjük a másik oldalon.




**2. ábra.** Absztrakt relációs táblák

A megfeleltetést a bemutatott példákban elsősorban egyes háttérben tárolt (perzisztens) UML osztályok és az őket megvalósító relációs táblák összekapcsolására használjuk. A megfeleltetés egyéb megkötéseit a kapcsolatra írt OCL megszorítással adhatjuk meg.

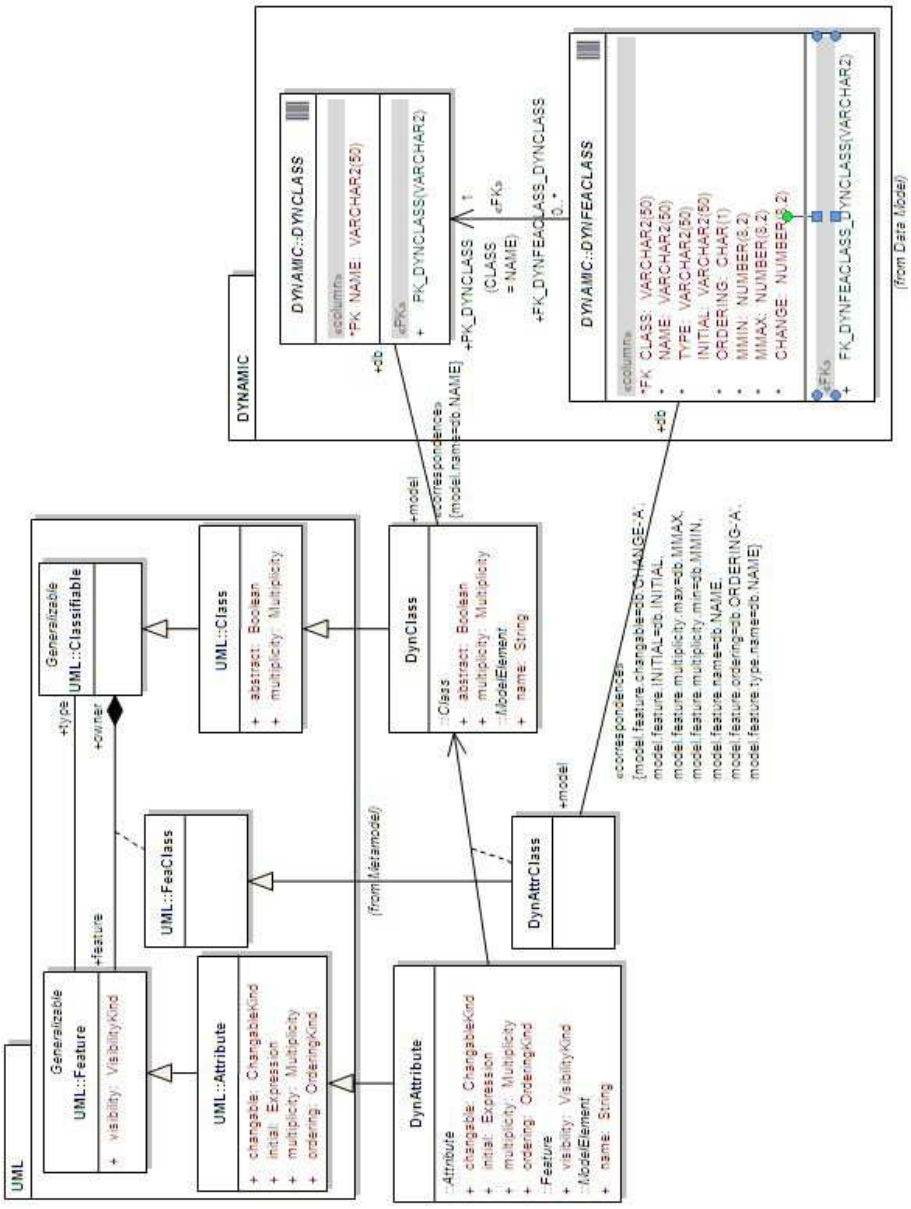
*Öröklődés relációs táblák között, és absztrakt relációs táblák:* Az egyes relációs táblákban ismétlődően megvalósítandó oszlopszekvenciákat absztrakt relációs táblákkal írjuk le. Azokat a konkrét táblákat, amelyekben az absztrakt táblák oszlopait szeretnénk látni, az UML öröklődés kapcsolóval leszármaztatjuk az absztrakt ős-táblából.

## 2. Kétszintű rendszerek: modellek dinamikus kezelése

A kétszintű működés esetében tehát a modellek futásidejű létrehozása, módosítása, esetleg törlése szükséges. Az alapprobléma egyes részeit a következő módon oldhatjuk meg.

### 2.1. Dinamikus tulajdonságok

A leghétköznapibb ilyen igény az, amikor az objektumközpontú felépítmény megengedte rögzített tulajdonságlista nem elegendő, mert a felhasználó megköveteli, hogy egyes objektumok tulajdonságlistája futásidőben módosuljon, ill. kibővülhessen. Az ilyen igények egyedi megoldásokon alapuló kielégítése helyett a modell részleges tárolása alapján az alább vázolt egységes és hatékonyabb megoldás javasolható.



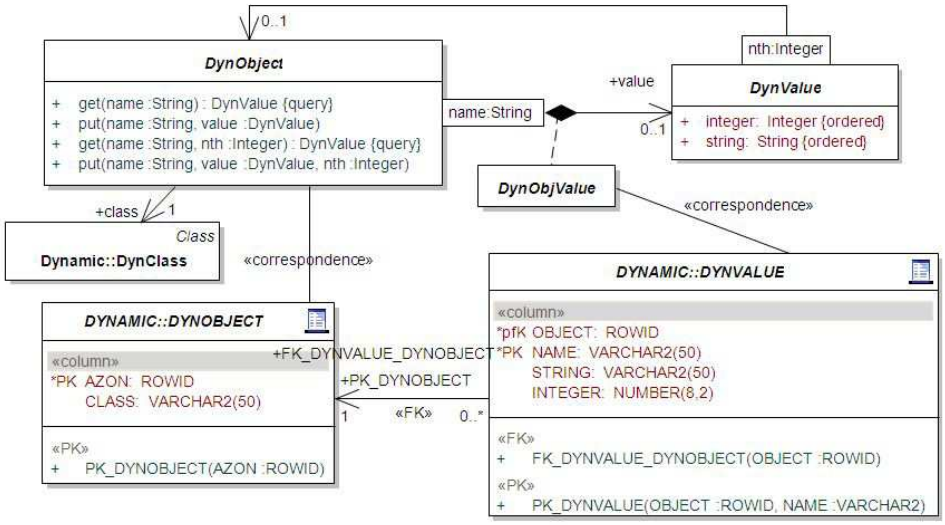

**3. ábra.** Dinamikus tulajdonságlista – modellszint

A megoldás a jelen esetben is kétszintű: a modell tulajdonságleíró részletének tárolása a metamodell egy részének beprogramozását, ill. a metamodell kiterjesztését, leszármaztatott osztályok létrehozását jelenti. Ezen leszármaztatott osztályok példányosításával kapjuk a modellszint objektumait, vagyis azokat az osztályokat, tulajdonságokat és ezek egymáshoz rendelését, amelyeket a szoftverben dinamikusan szeretnénk kezelni. A dinamikus kezelés miatt a tulajdonságok futásidejű felvételét, törlését és lekérdezését is meg kell valósítani.

Ilyenkor az alábbi ábrán látható modellből a `DynAttribute`, `DynClass` osztályokat és a `DynFeatureClass` kapcsolóosztályt kell megvalósítani. Ezek közül az előbbi kettő esetleg el is hagyható akkor, ha a `DynClass` metatulajdonságai érdektelenek, vagyis ha nem akarunk különbséget tenni absztrakt és konkrét metaosztályok között, netán nem érdekel bennünket az, hogy hányszorosan példányosítható a dinamikusan kezelt osztály. Ez esetben azt is feltételezzük, hogy nem akarunk a dinamikus tulajdonságlistát alkalmazó rész-modellben csomagokat használni, ezért az osztálynév (a `name:String` metatulajdonság) egyértelműen azonosítja az osztályt. Ilyenkor ugyanis a `DynAttribute` osztály egy példánya egy osztály egy tulajdonságát jellemzi, de ebből az osztályt csupán a neve egyértelműen azonosítja. A `DynAttrClass` kapcsolóosztály az osztály-tulajdonság kapcsolatok modellezésére szolgál. A fenti ábrán bemutatott megoldás teljes abból a szempontból, hogy a `DynClass` osztályt is tartalmazza. A modell háttérbeli tárolására a modellosztályoknak megfelelő relációs táblák, ill. táblaszegmensek megadásával utalunk. A relációs táblákat absztraktként vettük fel, vagyis csupán a megadott oszlopkészletet kell egy konkrét táblában esetleg megvalósítani.

A megoldást az adat/példányszinten szintén elő kell készíteni. Erre a célra az alábbi ábrán látható `DynObject-DynValue` absztrakt osztályokat biztosítjuk, melyek a dinamikus tulajdonságlistával rendelkező objektumok, ill. az ilyen listában ábrázolt értékek alapműködését adják, és amelyeket az ilyen működésmódot igénylő osztályoknál örökölni kell. A `DynObject` osztály a `class` egyirányban navigálható kapcsolaton keresztül megnevezi a saját dinamikus osztályát, valamint tulajdonságelérő eljárásokat ad. Ezekre tulajdonképpen nincs is igazán szükség, hiszen a tulajdonság-értékek a megadott navigációs szerkezeteken keresztül is elérhetők. A `DynObject` egy névvel indexelt vektorban tárolja a hozzátartozó tulajdonságértékeket. A `DynValue` osztály egyetlen tulajdonság-értéket modellez. Ez diszjunktívan felépített (union) típusú, vagyis bármilyen típust tárolni kell tudnia. Az ábrán jelölt `string:String`, ill. `integer:Integer` osztálytulajdonságok a megfelelő nevű skaláris típusokat jelentik, amelyeket esetleg bővíteni kell, ha újabb skalárisokat akarunk bevezetni. A tulajdonság-értékek esetleges többszörösségét (gyűjtemény jellegét) a fenti két tulajdonság rendezett többszörösségével érhetjük el akkor, ha a fent modellezett skaláris típusokról van szó. Egyéb, összetett értékek esetében az `nth:Integer` kiválasztókifejezés azt jelzi, hogy `DynObject` típusú adatok egy vektoráról van szó.

Az ábrázolás bonyolultságát az „union” típus, valamint a tulajdonságértékek többszörösségének megvalósítása adja. Ha az adott alkalmazásban a többszörösséget egyszeres (vagy üres) értékre (0..1) korlátozzuk, akkor az „nth” indexelés meg-




**4. ábra.** Dinamikus tulajdonságlista – példányok szintje

szűnhet mind a `DynValue`-`DynObject` kapcsolat, mind a `DynValue` osztály többszörös tulajdonságainak tekintetében, Ha viszont az értékek típusait pl. stringgé egyértelműen konvertálható alaptípusokban rögzítjük, akkor az egész `DynValue` osztály egyetlen `String` értékre egyszerűsödhet.

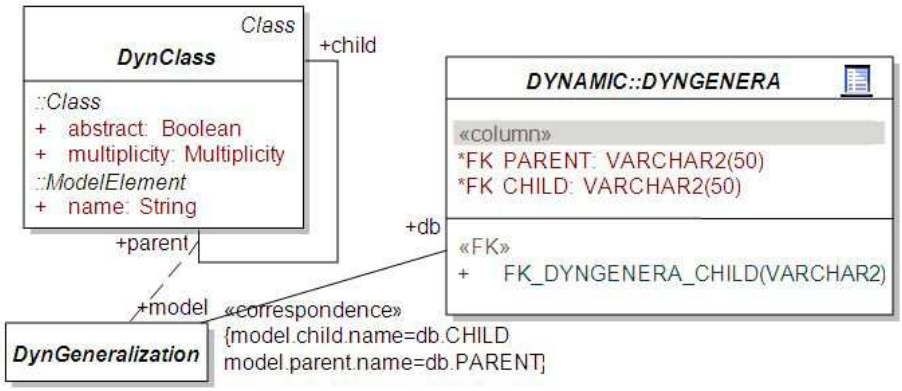
Ha egy osztály dinamikus tulajdonságkészletének megvalósításáról, és az objektum adatbázisban történő tárolásáról van szó, akkor a `DYNOBJECT` absztrakt táblától történő öröklődés figyelembevétele a háttértárban mindössze az osztálynévnek megfelelő új `CLASS` mező felvételét jelenti, hiszen a másik, az `AZON` azonosítómező valószínűleg amúgy is létezik. Ezen túl valamint a `DynObject`-`DynValue` kapcsolatot kell egy újonnan létrehozott táblával modellezni (az ábrán: `DYNVALUE`).

### 2.2. Dinamikus osztályszerkezet

A dinamikus osztályszerkezet megvalósítása esetén az egyik megoldás szerint a létrejövő dinamikus osztályok példányai mind egyetlen rögzített őstípushoz tartoznak, és a dinamikus osztályok futásidőben csupán egy osztálykötődést jelző tulajdonsággal, dinamikusan változó tulajdonságszerkezettel, ill. dinamikusan létrejövő függvénykészlettel vannak megvalósítva. Ezt az ősosztályt az adott programnyelven meg kell valósítani, a többi, dinamikusan létrehozott osztály, valamint a tulajdonságkészletük is már csak képzetes módon, futási időben áll elő, csak dinamikusan hozható létre.

A dinamikus osztályszerkezet megvalósítása a modellszinten a metamodell hasonló nevű kapcsolatát kiterjesztő `DynGeneralization` kapcsolóosztály segítsé-

gével történik. Ez egy kétoldalú kapcsolatot jelent, amelynek mindkét oldala egy-egy (dinamikus) osztály – az ős (`parent`), ill. a gyerek (`child`) osztályok. Ennek ábrázolása a relációs háttértárban könnyű – egyetlen olyan táblával, amelynek két mezője van. Az operatív tárban ennek egyik megvalósítása az osztályokat megvalósító objektumokhoz kötött `parent` és `child` gyűjteményekkel, de megvalósítható az objektumhálótól függetlenül létező, globális hatókörű környezeti változóként is. Ez a két osztálynév, ill. azonosító alapján címezhető gyűjtemény lenne, amelyek közül az egyik az adott osztály gyerekeit, a másik a szüleit tartalmazza. A dinamikus osztályszerkezet-kezelés másik követelménye az öröklődés megvalósítása. Ezzel összefüggésben ésszerű alkunak látszik, de komoly egyszerűsítést jelent, ha az objektumpéldányok típusának futásidejű módosítását nem engedélyezzük. Ez azt jelenti, hogy csak az objektum létrehozásakor gyűjtjük össze a modell alapján az tulajdonságlistát, de példányokra vonatkozólag semmilyen olyan műveletet nem valósítunk meg, amely ezt módosítaná. A leszármazási viszony dinamikus törlését – az adott típusú objektumok állapotának meghatározatlansága miatt – egyébként sem engedélyezzük, de ugyanígy tiltjuk az objektum típusának futásidejű változtatását – a konkretizálást éppúgy, mint a közvetlen típusmódosítást (casting). Tehát a választott modellben a tulajdonság mindig az első hivatkozásakor, az ak-




**5. ábra.** Dinamikus osztályszerkezet megvalósítása

kori osztályszerkezet alapján, a korai típuslekötés elveinek megfelelően jön létre, ami későbbiekben nem változik. Első közelítésben feltételezzük azt is, hogy az eredeti tulajdonságkészletet újabb öröklési utak felvétele sem változtatja.

A leírt egyszerűsítések lényegesen megkönnyítik a vonatkozó megvalósítást, és feltehetően a szóba jövő szoftverek túlnyomó részénél elegendőek lehetnek. Mindazonáltal az egész csupán megvalósítási könnyebbség, vagyis a fent vázolt adatszerkezet alapján a tényleges megvalósítás nem jelenthet elvi akadályt.

### 2.3. Dinamikus kapcsolatok és osztályfüggvények

A kapcsolatok és az osztályfüggvények dinamikus kezelését konkrét terv szintjén nem gondoltuk végig. Ezért ebben a szakaszban csupán az ezzel kapcsolatos alapvető meggondolásokat szeretnénk bemutatni.

Modellünk kapcsolatainak kezelése – csupán a megvalósítás szempontjából – felfogható az osztálytulajdonságokhoz hasonlóan is. Vagyis ha a modellszinten a kapcsolatok kezelése is követelmény, a legkézenfekvőbb azokat tulajdonságokká – összetett típusú objektumértékű tulajdonságokká átalakítva megvalósítani.

Osztályfüggvények kezelése lényegesen komolyabb gondokat vet fel. A futásidőben történő létrehozhatóság követelménye miatt egy futásidőben beprogramozható és beszerkeszthető megoldásra van szükség, ami behatárolja a megvalósítás nyelvét (pl. Java, Python, Prolog alkalmas ilyesmire). Olyan objektumközpontú nyelv esetén, amely ilyesmit nem támogat, a dinamikus osztályszerkezeten keresztül újabb függvények futásidejű hozzászerkesztése nem megoldható.

### 2.4. A modellszint és az alkalmazói szint összefüggése

Ezen a ponton egy pillanatra álljunk meg, hogy megkötést tegyünk a futásidőben a metamodell példányaként változó modell és a változó modell példányaként változó objektumháló összefüggésére, arra, hogy a modell módosítása milyen kihatással van az adatszintre. Erre a következő lehetőségeink kínálkoznak:

- A modellre vonatkozóan minden változtatást a program betöltésének és futtatásának ideje között egy *modellrögzítési fázisban* teszünk meg. Vagyis bár a modell egy része az alkalmazói programban van tárolva, de mégsem tekinthető dinamikusan változónak, hiszen egy programindítás utáni betöltési, ill. esetleges módosítási fázis utáni állapot befagy, és tovább már nem módosítható.
- A modellre vonatkozóan a *futásidőben is megengedjük a bővítést*, vagyis újabb modellobjektumok felvételét, *de a törlésüket nem.* Ez megfelel az ún. *monoton logikai* megközelítésnek, amikor is a kezelt tudásanyag csak nőhet, de sosem csökken. Gyakorlati szempontból ezzel a megkötéssel megtakaríthatjuk a modellelemek törléséből fakadó különbözőféle következményhatások (pl. törölt típusú objektumpéldány) figyelembevételét és megvalósítását.
- *Bármilyen modellműveletet megengedünk,* a nem monoton törlési művelet összes tovagöngyölődő hatását figyelembe vesszük, és megvalósítjuk (pl. tulajdonság törlésekor az összes hivatkozó tulajdonságpéldányt töröljük). Ez általánosságban véve az összes törölhető elemre (`DynAttribute`, ill. `DynClass`) mutató, csak egy irányban navigálható hivatkozási kapcsolat kétirányúvá tételét jelenti, amelyen keresztül a hivatkozó objektumokat a hivatkozás megszűntéről értesíteni lehet.

Az első változat melletti döntés valószínűleg túlságosan erős megszorítás. A modell- és az alkalmazói szint közötti kapcsolat túlságosan befagyott, merev, egy ilyen kapcsolat nem is igazán indokolja a modell- és az adatszint együttes tárolását és kezelését. A harmadik változat viszont túlságosan bonyolultnak, áttekinthetetlennek tűnhet, különösen akkor, ha – mint a következő szakaszban olvasható lesz – nemcsak az osztály-tulajdonság kapcsolatot, hanem ennél több modellelemet is dinamikusan kívánunk kezelni.

A két véglet között jó alkunak tűnik a középső változat, különösen úgy, hogy ebből kiindulva bizonyos megszorításokkal a harmadik változat is megvalósítható. Eszerint a törölt adatot nem távolítjuk el, csupán egy töröltséget jelző tulajdonságot állítunk át bennük. A törlésről a hivatkozó objektumokat nem értesítjük szinkron módon, hanem csak késleltetve: minden műveletnél ellenőrizzük a töröltségi állapotot is, és amennyiben ezt a műveletek beállítva találják, akkor egyrészt ezt az állapotot átveszik, másrészt elvégzik a szükséges hibakezelési vagy egyéb műveleteket.

### 2.5. Modellvezérelt lekérdezés

A modellszintről vezérelt adatcsere legtipikusabb/legáltalánosabb megvalósítása egy objektum-orientált lekérdezési nyelv alkalmazása. Ezek – objektumközpontú programnyelvekhez hasonlóan – igen hasonlóak egymáshoz, lényegében mindegyik az ODMG OQL nyelv valamilyen változatának tekinthető [4]. Az OQL maga – nyitott magú nyelv, csak a legalapvetőbb nyelvtani szerkezetekkel, amely a hagyományos SQL-hez hasonlóan a következő résznyelveket tartalmazza:

- *Lekérdező nyelv*: A résznyelv valójában alapinformációkból – osztálypéldányokból – történő adat-átalakítás absztrakt leírására alkalmas, és a használatát tekintve kétféleképpen hajtható végre:
  - *Hagyományos, szinkron értelemben:* Az információigény fellépésének időpontjában a lekérdezés kiértékelésre, végrehajtásra kerül úgy, hogy a programvégrehajtás megvárja a lekérdezés kiértékelését.
  - *Aszinkron értelemben*, amikor az alapadatok változásához kapcsolt triggerrel indítjuk a lekérdezés kiértékelését, amely a programhoz magához szintén aszinkron – eseményvezérelt módon érkezik.
- *Módosító nyelv*: A résznyelv a háttérben tárolt adatok módosítására – törlésére, cseréjére és létrehozására alkalmas.

### 2.6. A kétszintű/dinamikus működés további eszközei

A dinamikus működés megoldásához néhány további eszköz szükséges.

Mivel a kezdetben ismert osztály- és tulajdonságszerkezet dinamikus, vagyis nem a tervezés-fordítás ciklus folyamán, hanem a program futása során jön létre, ill. a háttérbeli tárolás miatt esetleg a program indulásakor betöltődik, ezért esz-

közt kell adnunk a modell kezdeti betöltésére (populációjára), valamint a modell futásidejű bővítésére is.

Az *adatbázis kezdeti feltöltése* esetleg egy megfelelően előkészített adatbázis importtal helyettesíthető akkor, ha a kezdeti modell kicsi és egyszerű. A dinamikus módosíthatóság azonban csak úgy biztosítható, ha erre valamiféle eszközt adunk. Ha csak igen kismérvű módosítások képzelhetők el, akkor valami egyszerű grafikus felület is elegendő lehet. Ha a módosítási igényeket nem lehet igen-igen egyszerű mértékre korlátozni, akkor célszerű egy külső *UML részmodellt leíró programnyelv*, ill. a hozzá tartozó *formátumátalakító szoftver*_eszközök (kigeneráló, ill. elemző) használata célszerű, amely egyben a kezdeti modellmegadás feladatát is megoldja. Ehhez mintául szolgálhat a SILK projekt SILAN nyelve [6]. Egy ilyen nyelv általános megadását olvashatjuk az OMG kibocsátott Human-Usable Textual Notation [5] szabványában is. CASE eszközzel (pl. Rational Rose) történő modellcseréhez az XML alapú XMI modell-leíró nyelv használata célszerű.

## 3. Kétszintű szoftverfelépítmény természetvédelmi alkalmazásokban

Természetvédelmi tárgyú alkalmazói programok térképi adatokat tárolnak, amelyekhez a legkülönbözőbbféle szöveges, ill. számszerű adatokat rendelnek: elsősorban a terepi munkát végző kutatók által végzett észlelések ezek az adatok, amelyekhez dokumentumfelvételeket (álló és mozgó fényképeket, hangfelvételt stb.) is kapcsolhatnak.

A természetvédelemben jobbára a legtöbb üzleti alkalmazásban használatos feladatok merülnek fel. Ami mindezeken túlmutat, az egyrészt a földrajzi adatok kezelése, másrészt pedig az élőlények Linné-féle rendszertanának tárolására vonatkozó igény. Az előbbi a földrajzi rendszerek kialakítási stratégiájának megfelelően részben modellvezérelt módon történik, de mindezek megoldása készen van, vagyis legfeljebb testreszabási, ill. modellkészítési és alkalmazásgenerálási feladataink maradnak.

A Linné-féle rendszertan taxonszerkezetére viszont kiválóan illeszkednek a dinamikus tárolás követelményei. Emellett azonban egy további kikötés is van. Mivel a biológus kutatók fáradhatatlan munkája eredményeképpen a taxonszerkezet változik – fajok összeolvadnak, szétválnak, esetleg újakat fedeznek fel, a rendszerben az *időfüggés* megvalósítása is szükséges. A korábban rögzített adatok értelmezéséhez ezért meg kell oldani azt, hogy a rendszert bármilyen múltbeli időpontra vonatkoztatva is működtetni tudjuk.

### 3.1. Időfüggés/időgép

Az időgép egy olyan működést jelent, amelyben a rendszer állapotát múltbéli időkre visszamenőleg is le lehet kérdezni. Az időgép általánosságban többféle módon is megvalósítható attól függően, hogy az időfüggés az adatok milyen körére

terjed ki, és milyen finomság a követelmény. Az alábbi leírást tekinthetjük egy programtervezési mintának is: annyiban mégis csak „elő-minta" (proto-pattern), hogy széles alkalmazási körről (legalább 3 sikeres és működő alkalmazás) nem számolhatunk be.

Az időfüggés finomságától függően az alábbi kétféle időfüggést lehetséges megkülönböztetni:

- ha az időfüggő adatok folytonosak, de legalábbis az értékváltozás nagyon sűrű, akkor *folytonos időfüggésről* beszélünk;
- ha az adatok értékei bizonyos események hatására változnak meg, ami után bizonyos ideig (a következő esemény bekövetkeztéig) az értékük állandó marad, akkor *diszkrét időfüggésről* beszélünk, ilyen megoldást tárgyal a jelen tanulmány is.

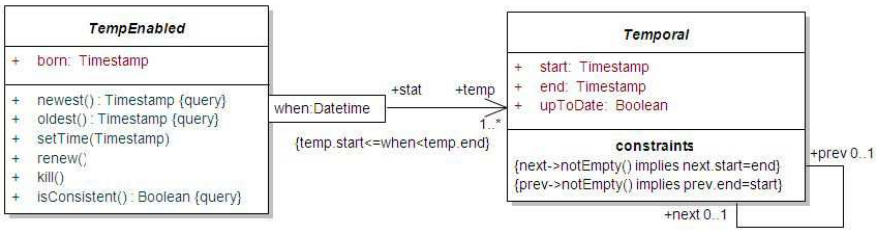
Az időfüggés ritkán terjed ki az adatok teljességére. Attól függően, hogy mely adattartományok időfüggőek, az alábbi megoldások képzelhetők el:

- ha az időfüggés mégiscsak közel teljes körű (minden változik), és sok érték változik egyidejűleg, de nem túl gyakran, akkor valószínűleg a teljes adatállomány történeti mentése a célszerű, vagyis visszamenőlegesen több generációs teljes adattár-állományokat érdemes mentenünk, ezt *állomány szintű időfüggésnek* nevezzük, egy ilyen megoldás a konfiguráció- és verziókezelő rendszerekéhez hasonló működést jelent;
- ha az időfüggés egyes objektumokra, azok összes, de legalábbis több tulajdonságára vonatkozik, ilyenkor *objektum szintű időfüggésről* beszélünk;
- ha az időfüggés csupán egyes objektumok egyes tulajdonságaira vonatkozik, akkor *tulajdonság szintű időfüggésről* beszélünk.

Az alábbiakban az objektum szintű adatfüggés megvalósítására mutatunk be javaslatot. Ebből a tulajdonság szintű adatfüggés már könnyen származtatható, a fenti megjegyzések miatt az állomány szintű adatfüggés viszont más jellegű megoldást igényel.

### 3.2. Objektum szintű időfüggés

Az objektum szintű időfüggés megvalósítására a fent bemutatott absztrakt osztályokat vezetjük be. Az ilyen objektumok képzeletben két részre bonthatók: a teljesen időfüggetlen jellemzőket (`TempEnabled`) és a teljesen időfüggőket (`Temporal`) tartalmazóra. A kettő egymással a `stat-temp` kapcsolaton keresztül kommunikál, amelynek az időfüggő (`temp`) oldala 1-nél nem kisebb többszörösségű. Ez egy gyűjtemény, amely a diszkréten időfüggő adatokat tartalmazza, és a `when:Datetime` kiválasztókifejezésen keresztül indexelhető. A gyűjtemény elemei rögzítik a saját érvényességi tartományukat (`start-end`), amelyek között átfedés nem lehet, és a következő intervallum kezdete mindig megegyezik a megelőző intervallum végével (ld. a `Temporal` osztály megszorítását). A gyűjtemény indexelésénél azt az idő-




**6. ábra.** Időgép megvalósítása

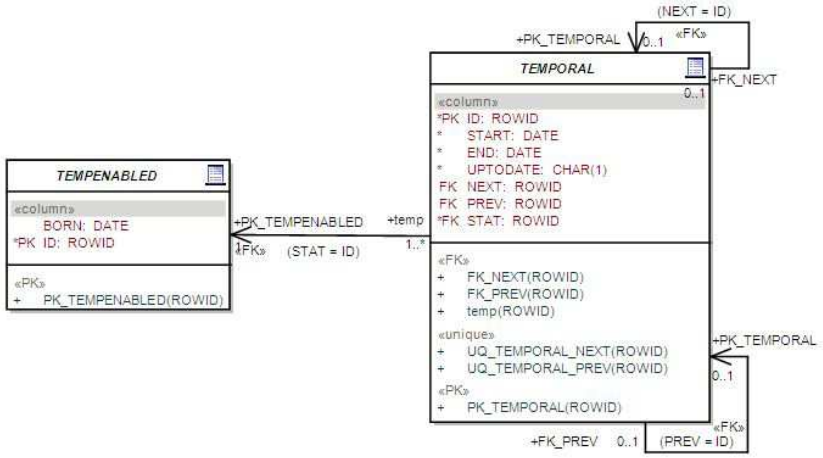
függő elemet keressük meg, amelyre a kiválasztókifejezés beleesik az érvényességi tartományba (`{start<=when<end}`).

Az időfüggő részek a prev és next mutatókon keresztül kétszeresen láncolt listába szervezhetők: ez nem tartozik szigorúan a követelményekhez, de lényegesen megkönnyítheti pl. történelmi összefoglalók készítését.

A `TempEnabled` osztály tartalmazza az objektum időfüggetlen törzsét és az időfüggés kezelésére alkalmas eljárásokat:

- Időlekérdező eljárások (`newest()`, `oldest()`). Ezek a megadott időpontok lekérdezésére alkalmasak. Az adott időpontbeli értékek a kiválasztókifejezés szerinti indexeléssel érhetők el (OCL2).
- referenciaidőpont beállítása (`setTime()`). A rendszer alapértelmezés szerint az aktuális időpontban működik. Az időpont-beállító eljárást akkor használjuk, ha a rendszert mégis visszamenőleges üzemkódban kívánjuk működtetni.
- Az objektum frissítése (`renew()`). Az egyes tulajdonság-értékek változtatása egyöntetű módon történik mind az időfüggetlen, mind az időfüggő objektumszegmens esetén. Amennyiben az időfüggő tulajdonságokat átállítanánk, az `upToDate` tulajdonság önműködően `False` értéket vesz fel. A frissítés eljárás meghívásával az objektumnak egy újabb időfüggő példányát (a `Temporal` gyűjteménynek újabb elemét) hozzuk létre, és az `upToDate` bitet egyidejüleg `True` értékre állítjuk.
- Az objektum élettörténetének lezárása (`kill()`).
- (`isConsistent()`) Az objektumra és a belőle navigációval elérhető társobjektumokra időbeli konzisztencia-ellenőrzést hívunk meg. A megszüntető művelet következtében ugyanis előfordulhat, hogy a társobjektumok az adott (aktuális vagy a beállított) időpontban nem léteznek már, vagy nem léteztek még.

Az említett két osztály mindegyike absztrakt, vagyis önmagukban nem példányosíthatók. Konkrét esetben az időfüggést megkövetelő osztályokat a fenti osztályokból származtatjuk le.



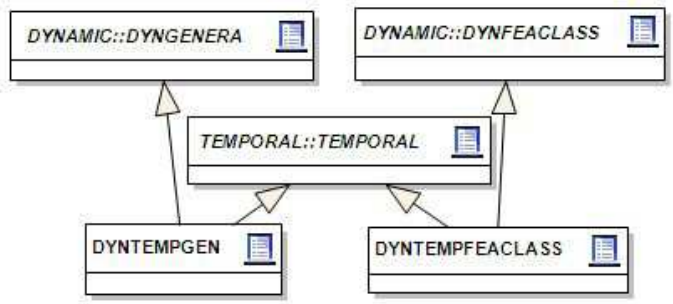

**7. ábra.** Időgép leképezése relációs adatbázisba

Az objektum szintű időfüggést az osztálydiagramnak megfelelő adatbázis-táblaszerkezettel lehet megvalósítani.

A `TEMPENABLED` absztrakt tábla az objektumok statikus részét tárolja, míg a `TEMPORAL` tábla a dinamikus részüket. A dinamikus táblából egy külső kulccsal (`FK STAT`) mutatunk a statikus táblába, valamint egy-egy külső kulcs valósítja meg a kettős láncolást is, de ezek a saját táblában található megelőző, ill. következő rekordra mutatnak.

A táblaszerkezet most is absztrakt, vagyis konkrét adatmodell esetében a hasonló működéseket az absztrakt tábla rögzített mezőinek örököltetésével (beillesztésével) lehet biztosítani.

### 3.3. Időfüggő modellszint megvalósítása




**8. ábra.** Időfüggő, dinamikus modellszint táblái

A kétszintű rendszerben időfüggő modellszint megvalósításához a kétféle működést kombinálni kell. Ez azt jelenti, hogy az alkalmazói osztályszerkezetnek örökölnie kell mind a dinamikusan változtatható modellelemek esetében, mind az időfüggést megvalósító modellelemek esetében meghatározott absztrakt osztályoktól.

Az adatbázis szintjén lényegében ugyanez történik: az alábbi ábra bemutatja, hogy hogyan származtathatók a kétszintű időfüggést megvalósító táblák (`DYNTEMPGEN`, `DYNTEMPFEACLASS`) a kétszintű dinamikus működést, valamint az időfüggő működést megvalósítókból. Az ábrán a két leszármaztatott táblát álló betűkkel, konkrét táblaként jeleztük. Ez azt fejezi ki, hogy egy időfüggő, dinamikus modellszintet használni kívánó alkalmazás esetében a táblák közvetlen relációs adatbázisbeli megvalósítása javasolt, ahol az egyes mezők az ős-absztrakt táblák definíciójából öröklődéssel állnak elő.

### 3.4. Dinamikus taxonszerkezet megvalósítása

Természetvédelmi célú rendszerben szükséges a Linné-féle rendszertant leíró ún. „biológiai taxonszerkezet" ábrázolása is. Mivel a tárgyra vonatkozó ismeretek forrongók, ezeket mindenképpen dinamikusan változtatható módon kell megvalósítani. Másrészt viszont, korábbi időpontokra vonatkozó adatok kiértékeléséhez az egésznek időfüggőnek kell lennie, hogy bármilyen korábbi időpont szimulálható legyen. Ez azt jelenti, hogy a taxonokat leíró osztályszerkezet futási időben változhat, és a változást az időben követni kell. Az időfüggő, dinamikus modellszint az alábbi igényekre kézenfekvő megoldást ad.

Megjegyezzük azonban, hogy a fenti, általános elvekből levezetett szerkezet mellett szükséges egy mindezek felett álló „jogfolytonossági" viszony tárolása is. Ez azt írja le, hogy egy taxon jogelődje, ill. jogutódja melyik mások taxon lenne. Az alább részletezett műveletek a szimmetrikus összeolvadást és szétválást is megengedik, tehát a jogfolytonossági viszony $n \circ m$-es többszörösségű. Emiatt egyszerű külső kulccsal nem tárolható valamelyik már meglevő táblában, hanem számára külön tábla felvétele szükséges.

A dinamikus taxonszerkezettel összefüggésben a következő műveletek megvalósítása szükséges:

1. *Taxon átnevezése:* Az eredmény az eredetivel – a megváltozott névtől eltekintve – teljesen megegyezik. A művelet végrehajtása a név módosítását és új időváltozat létrehozását vonja maga után. A jogfolytonossági viszonyban semmilyen változás nem történik.
2. *Taxon átnevezése és más tulajdonságok módosítása:* Az eredmény részben módosult. A művelet végrehajtása során a szóban forgó tulajdonságok módosulnak, és új időváltozat jön létre. A jogfolytonossági viszonyban semmilyen változás nem történik.
3. *Szimmetrikus összeolvadás:* Két taxon összevonása eggyé, új néven. A művelet feltétele, hogy a két taxon szigorúan testvérpár legyen, vagyis közös

közvetlen őstől származzanak. Egyes tulajdonságok megegyezését illetően további feltételek is állíthatók. A két eredeti taxon megszűnik, és tovább nem használhatók, majd létrejön egy új taxon. Az új taxon egyes tulajdonságait kívülről kapja, más tulajdonságai közösek. Ezen túli tulajdonságaira egyesítési megoldást kell meghatározni. A nyitott feltételeket illető meghatározások az elemzési, ill. a tervezési modell részei. Az új taxon jogutódja a két összevont taxonnak.

4. *Aszimmetrikus összeolvadás:* Két taxon összevonása eggyé valamelyik nevén. Taxon beolvadása másik taxonba. A tulajdonságokat illetően az összeolvadásra további feltételek is kiköthetők (ld. feljebb). A megszűnő taxon jogelődje a másiknak.
5. *Szimmetrikus szétválás:* Egy taxon szétszedése két vagy több új taxonra. Több taxonpéldány létrehozása, ahol mindegyik új nevet (és új azonosítót) kap. Az eredeti taxon jogelődje mindegyik új taxonnak.
6. *Aszimmetrikus szétválás:* Egy taxon szétszedése két vagy több taxonra, miközben egy megtartja a korábbi nevét (és azonosítóját). A korábbihoz hasonló művelet. A megmaradó taxon jogelődje többinek.
7. *Szintemelkedés:* Egy alfaj faji szintre emelkedik. Az eredeti taxon megmarad, csupán az alfajt jelző mező lesz fajt jelző értékké átírva. Az alfajhoz vezető általánosítás megszűnik, helyette a saját fajához vezető általánosítással megegyező általánosítás jön létre.

## 4. Eredmények összegzése és köszönetnyilvánítás

A cikkben leírtak az ún. kétszintű vagy modellvezérelt szoftverek működésének és készítésének néhány alapkérdését, valamint a kidolgozott megoldások alkalmazását mutatták be természetvédelmi célú szoftveralkalmazásokban. A megoldások általánosak, vagyis nem csak a természetvédelemben alkalmazhatók. Bár előtanulmányokban egyes megoldások tesztelve lettek, a leírtak egészének konkrét működés közbeni vizsgálata még nem történt meg. A cikk megírását megelőzte egy természetvédelmi célú információs rendszer tervének rögzítése, de a konkrét megvalósítási tervek elkészítése és az implementációs munkák megkezdése csak a későbbiekben várható.

Köszönetemet szeretném kifejezni a SILK EU projektben résztvevő munkatársaimnak, mert a leírtak kikristályosításában a SILK felépítmény megismerése, és a projektben történő aktív részvételem meghatározó szerepet játszott. Úgyszintén köszönetemet szeretném kifejezni a Természet és Környezetvédelmi Minisztérium Természetvédelmi Főosztályán működtetett Természetvédelmi Informatikai Tanácsadó Testület tagjainak, akik révén a szakmai munka egyáltalán megtörténhetett, ill. a szükséges szakmai információk birtokába juthattam.

## Hivatkozások

[1] Joaquin Miller-Jishnu Mukerji: *Model Driven Architecture*, OMG Document, July 2001.

[2] Jon Siegel: *Developing in OMG's Model-Driven Architecture*, OMG White Paper, November 2001.

[3] Joaquin Miller-Jishnu Mukerji: *MDA Guide Version 1.0*, OMG Document, July 2003.

[4] R.G.G.Cattell-D.Barry és mások: *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann publishers San Francisco, USA 1999.

[5] *Human-Usable Textual Notation (HUTN) Specification Version 1.0*, OMG, August 2004.

[6] *SILAN – the SILK language*, IQSOFT, Hungary 2000

[7] *Rumbaugh-Jacobson-Booch*: *Unified Modelling Language Reference Manual*, Addison-Wesley-Longman Inc. 1999.

KILIÁN IMRE
PTE-TTK, Informatika tanszék
7624 Pécs, Ifjúság u. 6.
kilian@gamma.ttk.pte.hu

THE CONSTRUCTION OF MODEL DRIVEN SOFTWARE II.

Imre Kilián

The article demonstrates how and when the application of model driven architecture can be advantageous. In the first section the implementation is described: starting from the simplest, partially model-driven architecture, when only a dynamic property-list is aimed, up to the complete model-level.
The second section describes how the concept of two level software can be used for the dynamic behaviour of taxon hierarchies in biological and ecological software applications. This section also presents the concept and implementation of a 'time machine', i.e. a framework that enables the retrospective operation of a software.