PROGRAMTERVEZÉSI MINTÁK ÉRTELMEZÉSE NORMÁL FORMÁKKÉNT

Kusper Gábor, gkusper@aries.ektf.hu
Eszterházy Károly Főiskola

TEMATIKA:

A programtervezési minták, röviden PM-ek, egy receptet írnak elő a programozó számára, hogyan készítsen hatékony, újrafelhasználható és könnyen bővíthető programot, mint ahogy egy gulyás recept leírja, hogyan készítsünk ízletes gulyáslevest. A PM-ek a négyek bandájának könyvével (*Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz*) kerültek a központba a 90'-es években. Az eltel idő alatt kivívták az őket megillető helyet, mind a gyakorlatban, mind az oktatásban. Ugyanakkor probléma, hogy nehézen formalizálhatok. Ennek oka, hogy hiányzik egy széles körben elfogadott, az oktatásban is jól használható modell.
Az eddigi modellek az Objektum Orientált Programozás, röviden OOP, alapelveinek leírására koncentrálnak, hiszen ez a PM-ek nyelve. Ezen modellekben a PM-ek leírása bonyolult.
A cikkben javasolt új modell egy megfeleltetést add a Relációs Adatmodell és az OOP fogalmai közt, úgy mint: Tábla / Osztály, Egyed / Objektum, Külső kulcs / Referencia.
Ez azért szerencsés, mert a Relációs Adatmodell tudományosan megalapozottak és oktatásban is jól használható, mert fogalmai könnyen életszerűvé tehetők.
Például a Normálformák könnyen magyarázhatóak: Feladatuk a függőségek csökkentése, azért hogy például egy egyszerű bővítésnél ne kelljen sok helyen változtatni a modellt.
Hasonlóan könnyű megérteni a PM-ek ebből a szemszögből: Feladatukat a függőségek csökkentése, hogy például a program bővítéséhez ne kelljen sok helyen átírni a kódot.
Jól látható a PM-ek és a Normálformák közti párhuzam, ami a javasolt modellben formálisan is megmutatható például a Pehelysúlyú minta esetén.

Bevezetés

Ebben a cikkben a programtervezési mintákkal, röviden PM-ek, foglalkozunk. A PM-ek a négyek bandájának könyvével [GOF95] (*Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz*) kerültek a központba a 90'-es évek közepén. Az eltel idő alatt kivívták az őket megillető helyet, mind a gyakorlatban, mind az oktatásban. Ugyanakkor probléma, hogy nehézen formalizálhatok. Ennek oka, hogy hiányzik egy széles körben elfogadott, az oktatásban is jól használható modell.
Az eddigi modellek az Objektum Orientált Programozás, röviden OOP, alapelveinek leírására koncentrálnak, hiszen ez a PM-ek nyelve. Ezen modellekben a PM-ek leírása bonyolult [MK04].
A cikkben javasolt új modell egy megfeleltetést add a Relációs Adatmodell és az OOP fogalmai közt. A modell kiinduló alapja, hogy az OOP és a Relációs Adatmodell, röviden RA, fogalmi közt megtalálható egyfajta megfeleltetés [Amb01] alapján PM-ek fogalomrendszert a RA fogalmaival magyarázzuk meg.
A megfeleltetés a következő. RA esetén az adatokat táblákban (relációkban) tároljuk, egy táblában több oszlop (attribútum) lehet és több sor (egyed). Egy sor konkrét értékeket

tartalmaz minden attribútumhoz. Az attribútum névből és típusból áll. A tábla az attribútumok halmaza. A tábla minden sora egyértelműen azonosítható az elsődleges kulcs segítségével. A táblák külső kulcs segítségével összekapcsolhatok. Az összekapcsolt táblák azon sorai tartoznak egybe, ahol a külső kulcs és elsődleges kulcs értéke megegyezik a két táblában.
Egy osztály mezőkből és metódusokból áll. A mezőnek van neve és típusa, a metódusok ezeken a mezőkön végeznek műveleteket, azaz az osztály egységbe zárja az adatokat és a rajtuk végrehajtható műveleteket. Az osztály mezőit elrejti, azokat metódusai segítségével lehet lekérdezni, módosítani. Ezért az osztály metódusainak halmazának tekinthető. Az osztálynak lehetnek példányai, úgynevezett objektumok, amiknek konkrét belső állapotuk van, amit a mezőinek értékében tárol. Minden objektumban lehet hivatkozni a speciális this értékre, ami magára az objektumra ad referenciát. Az objektumok ismerhetik egymást, amit egy referenciával oldanak meg. Ha egy objektumnak van referenciája egy másik objektumra, akkor a referencia értéke megegyezik ezen másik objektum this értékével.

Ezek alapján a megfeleltetés a következő:

Tábla	< = >	Osztály
Attribútum	< = >	Metódus
Egyed	< = >	Objektum
Elsődleges kulcs	< = >	this
Külső kulcs	< = >	Referencia

Ezt a fajta leképzést nem mindenki tekinti relevánsnak. Sok szerző úgy gondolja, hogy az adatok redundancia mentességére, állandóságára koncentráló RA szemlélet és az objektumok dinamikus változását szem előtt tartó OOP nézőpont nem egyeztethető össze. Ezt a nézetmódbeli eltérést az irodalom „The Object-Rational Impedance Mismatch”-nek nevezi. Részletes kifejtése itt található: http://www.agiledata.org/essays/impedanceMismatch.html .

Normálformák a Relációs Adatmodellben

RA területén nagyon fontos szerepet töltenek be a normálformák. Több normálformát, röviden NF, ismerünk, de ezek közül a gyakorlatban az első három használata terjed el. Az NF-ek definiálásához szükségünk van a funkcionális függőség és a kulcs fogalmára.

Definíció: Legyen R a $D_1, \ldots, D_n$ attribútumokon értelmezett reláció. Legyen továbbá A és B két részhalmaza a $\{ D_1, \ldots, D_n \}$ attribútum halmaznak. Azt mondjuk, hogy B **funkcionálisan függ** A-tól az R relációban, ha bármely két r_1, r_2 eleme R-re $r_1(D_i) = r_2(D_i)$, minden D_i eleme A esetén teljesül, akkor ebből következik, hogy $r_1(D_j) = r_2(D_j)$, minden D_j eleme B-re.

Definíció: Legyen $A = \{ D_1, \ldots, D_n \}$ az R reláció attribútum halmaza. Legyen K az A valódi részhalmaza. K-t az R reláció **kulcsának** nevezzük, ha A funkcionálisan függ K-tól és K-nak nincs olyan valódi részhalmaza amelytől szintén funkcionálisan függne A. Egy relációnak több elsődleges kulcsai is lehet. **Elsődleges attribútumoknak** nevezzük azokat az attribútumokat, amelyek valamely kulcshoz tartoznak, **másodlagosaknak** azokat, amelyekre ez nem teljesül.

1NF: Egy reláció első normálformában van, ha a reláció minden értéke elemi.
2NF: Egy reláció második normálformában van, ha első normálformájú, továbbá egyetlen másodlagos attribútuma sem függ funkcionálisan egyetlen kulcsának valódi részhalmazától sem.

3NF: Egy reláció második normálformában van, ha első normálformájú, továbbá a másodlagos attribútumai közt nincs funkcionális függőség.

A normálformák használata csökkenti a redundanciát az adatbázisban és megszünteti a **törlési, módosítási és beszúrási anomáliát**. Ezeket egy példával szemléltetjük: A Dolgozó táblában tároljuk el a dolgozó törzsszámát, nevét, lakhelyét, születési helyét, idejét, a projekt azonosítóját, amin dolgozik és a projektvezető törzsszámát. Mivel egy dolgozó több projekten is dolgozhat, ezért a kulcs tartalmazza a projekt azonosítót is. De ekkor új dolgozót csak úgy tudunk felvenni (beszúrás), ha rögtön van projektje is, ami például próbaidő esetén nem igaz (beszúrási anomália). Ha egy dolgozó, aki több projekten is dolgozik, megváltoztatja a lakhelyét (módosítás), akkor annyi sorban kell ezt átírni, ahány projekten dolgozik. A probléma akkor adódik, ha emiatt több százezer sort kell módosítani (módosítási anomália). Tegyük fel, hogy egy projekten csak egy dolgozó dolgozott, azt sikeresen befejezte, ami után elment a cégtől (törlés). Ekkor a hozzá tartozó sorok törlésével elvész az általa befejezett projektre vonatkozó minden információ is (törlési anomália).

Programtervezési Minták

A négyek bandája által írt Programtervezési minták [GOF95] című könyv a fő forrása ennek a cikknek. A szerzők így határozzák meg a PM-ek két alapelvét:

Programozzunk a felületre a megvalósítás helyett.

Használjunk objektum-összetételt osztályöröklés helyett, amikor csak lehet.

Ez a két alapelv azért nagyon fontos, mert használatuk esetén csökken az osztályok függősége és csatolása, ami növeli a kód rugalmasságát, bővíthetőségét és megérthetőségét.
A könyv ezekről a függőségekről és csatolásokról beszél:
Függőségek (dependence):

- Fordítási függőség (compilation dependence): Ha megváltózik a kód (egy alrendszer kódja), a függő részeket is újra kell fordítani. Elkerülhető például a Homlokzat (Facade), Híd (Bridge) tervezési mintával.
- Megvalósítási függőség (dependence on implementation): He egy osztály megvalósítása felhasználja, hogy hogyan van megvalósítva egy másik osztály, akkor annak megvalósításától függ. Elkerülhető az első alapelvvel.
- Környezeti függőség (dependence on environment): Ha egy osztály megvalósítása felhasználja, hogy milyen környezetben (például milyen operációs rendszeren, hardveren) fog futni, akkor attól a környezettől függ. Elkerülhető a létrehozási minták segítségével.

Csatolások (coupling):

- Szoros csatolás (tight coupling): Az objektumok közvetlenül hivatkoznak egymásra, a hatékonyság érdekében közvetlenül férnek hozzá megvalósítástól függő részekhez is. A szoros csatolás oszthatatlan rendszerekhez vezet, ahol nem változtathatunk vagy törölhetünk egy osztályt a rendszerből számos más osztály megértése és átírása nélkül.
- Laza csatolás (loose coupling): Közvetett ismeretség. Az objektumok nem ismerik egymást, így nem is hivatkozhatnak egymásra. Az ismeretséget egy vagy több közbeiktatott objektum valósítja meg. Megvalósítható például a Közvetítő (Mediator) és a Felelősséglánc (Chain of Resposibility) tervezési mintával.

- Elvont csatolás (abstract coupling) [WGM88]: Egyfajta laza csatolás. A csatolást megvalósító objektum elvont ős típusú. Nemcsak a közvetlen kapcsolatot rejti el, hanem a megvalósítást is. Megvalósítható például a Homlokzat (Facade) és a Megfigyelő (Observer) tervezési mintával.
- Rétegezés (layering): Egyfajta laza csatolás. Az objektumok rétegekbe vannak szervezve. A rétegek egy jól definiált interfészen keresztül kommunikálnak. A rétegek akár más-más számítógépen is futhatnak. Megvalósítható például a Homlokzat (Facade) és a Híd (Bridge) tervezési mintával.

Vegyük észre, hogy a szoros csatolásnál leírt problémák mennyire hasonlítanak a RA törlési és módosítási anomáliáira. A beszúrási anomáliának is találunk könnyen megfelelőt: Egy szorosan csatolt rendszert nehéz bővíteni, mert a bővítéshez meg kell érteni és általában nehezen érthető.
Ezen anomáliák megszüntetéséhez a csatoltság lazítását javasolja a könyv. Ez felel meg annak, amikor egy táblát a normalizálás folyamán több táblára bontunk, hogy csökkentsük a funkcionális függőségek számát. Már csak az a kérdés, hogy a funkcionális függőségnek milyen fogalom felel meg.

Csatolás

Az előző fejezetben láttuk, mennyire fontos szerepet játszik a csatolás a tervezési mintákban. Ugyanakkor eddig még nem definiáltuk magát a csatolást. Erről a Wikipedia-ban a következő cikket találjuk: http://en.wikipedia.org/wiki/Coupling_(computer_science) .

Csatolás (coupling) vagy függés (dependency) alatt annak fokát értjük, hogy egy program modul milyen mértékben alapszik a többi modulon. A függést szokás a kohézió (cohesion) ellentéteként értelmezni. Alacsony fokú csatolás általában magas fokú kohéziót eredményez, és oda-vissza. A csatolás mértékét Larry Constantine csoportjának munkája [CHG96] alapján a következő módon számoljuk.
Definíció: OOP-ben a **csatolás** annak mértéke milyen erős kapcsolatban áll egy osztály a többi osztállyal. A csatolás mértéke két osztály *A* és *B* között növekszik, ha:

- *A*-nak van *B* típusú mezője.
- *A* meghívja *B* valamelyik metódusát.
- *A*-nak van olyan metódusa, amelynek visszatérési típusa *B*.
- *A B*-nek leszármazottja, vagy implementálja *B*-t.

Ha egy osztály lazán csatolt, akkor ha változik az implementációja, akkor az nem vonja maga után a többi osztály megváltoztatását.

Javasolt OOP Normálformák

A bevezetőben megadott megfeleltetés és az RA normálformák alapján a következő OOP normálformákat javasoljuk. Elsőként megismételjük a megfeleltetést:

Tábla	< = >	Osztály
Attribútum	< = >	Metódus
Egyed	< = >	Objektum
Elsődleges kulcs	< = >	this
Külső kulcs	< = >	Referencia

Kényelmes lenne ezen megfeleltetés alapján a normálformákat egyszerűen átírni, de akkor már rögtön az első normálformával lemondanánk az összetett típusok használatáról, habár minden OOP nyelvben azok nagy támogatást élveznek, hiszen maga az osztály is összetett típus. Másrészt a normálformák az egyedek belső szerkezetére tesznek megkötést, míg az objektumok belső szerkezete rejtve van. Ezért az OOP normálformáknak az objektumok rendszeréről kell beszélnie, arról, hogy egy objektum hogyan viszonyul a környezetéhez. Az OOP normálformáknak azt kell biztosítaniuk, hogy az objektumnak könnyen lecserélhetőek legyenek és könnyen más környezetbe helyezhetők.

Ezek alapján a következő OOP normálformákat javasoljuk:
1NF: Az osztály első normálformában van, ha környezete elől elrejti a mezőit.
2NF: Az osztály második normálformában van, ha első normálformájú, továbbá a referencia típusú mezői elvont osztály típusúak.
3NF: Az osztály második normálformában van, ha második normálformájú, továbbá minden metódus hívás által visszaadott érték és belső állapot átmenet környezetétől független. Azaz a metódusainak visszatérési érték és a példány belső állapotban bekövetkező változás csak a példány belső állapotától és a metódus paraméteritől függ.

Az 1NF minden OOP nyelvben természetes. Az egységbezárás elvének következménye, hogy a mezőket csak az objektum saját metódusai írhatják, olvashatják. De ennek betartása nem kötelező, egy mező akár publikus is lehet. Ezt zárja ki az első normálforma.
A 2NF lényegében a PM-ek első alapelvét fogalmazza meg, miszerint *„Programozzunk a felületre a megvalósítás helyett.”*, hiszen ha csak elvont osztályokat ismerünk, akkor csak a felületet ismerhetjük. A második normálforma még nem zárja ki, hogy egy metódus a környezetében lévő objektumok mindenféle metódusát meghívva (és azok visszatérési értékét felhasználva) lényegében megjósolhatatlan belső állapotba vigyen. Ezt a harmadik normálforma zárja ki.
A 3NF megértéséhez meg kell értenünk, hogy ha meghívunk (az osztály egy példányának, azaz) egy objektumának egy metódusát akkor az egyrészt visszaad egy értéket, másrészt megváltoztathatja az objektum belső állapotát. Ha a visszatérési érték és az állapot átmenet is csak az objektum belső állapotától és a metódus paramétereitől függ, akkor harmadik normálformában van az osztály. Ez biztosítja azt, hogy az osztály könnyen lecserélhető, megérthető legyen.
Az a tény, hogy a PM-ek második elvét (*„Használjunk objektum-összetételt osztályöröklés helyett, amikor csak lehet.”*) még nem használtuk fel, azt mutatja, hogy további normálformák felfedezése valószínűsíthető.
Jó lenne megtalálni a függőségi fogalmat is, amire a normálformákat lehetne építeni, mint ahogy RA esetén a normálformák a funkcionális függőségen alapszanak. Erre alkalmasnak tűnik a környezeti függőség, amit az alábbiakban definiálunk.

Definíció: Azt mondjuk, hogy az objektum **függ a környezetétől**, ha megvalósítsa felhasználja a környezetében lévő objektumok által okozott mellékhatásokat. **Mellékhatásnak** nevezzük, ha egy objektum megváltozatja környezetét. **Mező környezeti függőségről** beszélünk, ha az objektumnak van kívülről elérhető mezője és annak egy másik objektum értéket ad.

Mellékhatás például, ha egy objektum a képernyőre-, állományba-, külső portra ír, ha értéket ad egy másik objektum kívülről elérhető mezőjének, vagy ha a rejtett mezőt egy kívülről

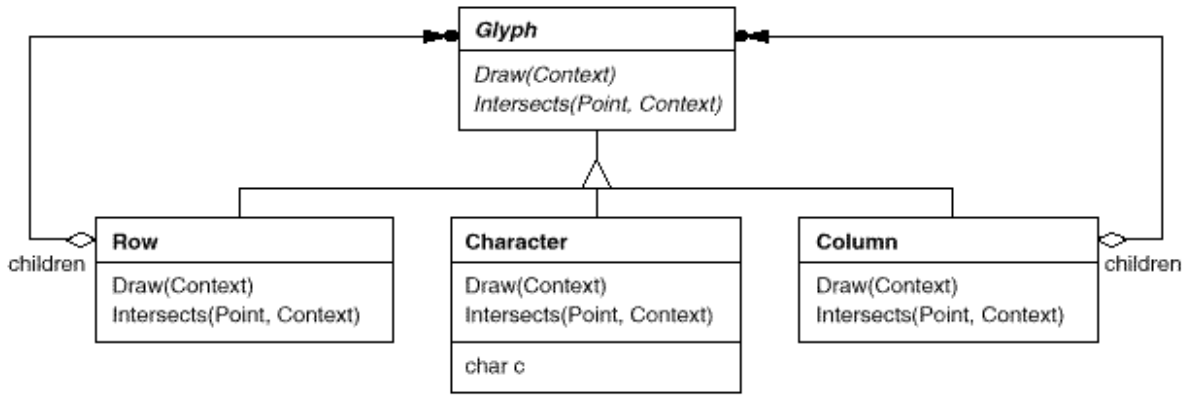
elérhető metódussal állítja be. Ez az utóbbi elkerülhetetlen, hiszen az objektumok maguktól nem csinálnak semmit, amíg a környezetükből meg nem hívják valamelyik metódusukat.
A környezeti függőség azt jelenti, hogy a programozó felhasználja az osztály megvalósításánál, hogy a többi osztály hogyan van megvalósítva, lényegében olyan tudást használva fel, ami az osztály előtt rejtve van. Csakhogy így az osztály és objektumai nem cserélhetők le egy másik programozó által a többi osztály kódjának megértése nélkül.
Egy osztály egy másik osztályból csak annyit ismer, ami annak a publikus interfészében található, tehát a publikus mezőket és a publikus metódusok fej részét (név, visszatérési típus, formális paraméterlista, kiváltott kivételek).
E környezeti függőség egy lépés a jó irányban, de segítségével nehézkes a fent javasolt normálformák definiálása.

Programtervezési Minták mint Normálformák

Ebben a részben rávilágítunk, hogy a PM-ek és az előző részben javasolt normálformák közt szoros összefüggés van. Ezen kívül megmutatjuk a Pehelysúlyú (Flyweight) minta mind az OOP, mind a RA szemléletmódban 3NF-ban van.
A szoros összefüggés abból adódik, hogy a javasolt normálformák közül a 2NF megfelel a PM-ek első alapelvének. 2NF alakban van például az Építő (Builder) minta. Építő mintát érdemes alkalmazni például akkor, ha egy RTF állományt sok más különböző formátumra akarunk konvertálni, mint például DOC, TXT, TeX. Ekkor érdemes minden formátumhoz egy konkrét építőt rendelni, amik mind-mind az elvont építő leszármazottjai. Az RTF olvasónak paraméterben átadhatjuk bármely konkrét építőt, amit eltárol. Ahogy olvassa a RTF állományt meghívja a konkrét építőt, ami elkészíti a kimeneti állományt. Mivel minden konkrét építőre működnie kell, az a mező, ami a konkrét építőre mutató referenciát tartalmazza, szükségszerűen ezek közös őse, azaz építő típusú, azaz elvont osztályra mutat. Tehát az Építő minta 2NF-ban van.
Hasonlóan könnyen megmutatható, hogy szinte az összes PM 2NF-ban van.

Vizsgáljuk meg a Pehelysúlyú (Flyweight) mintát. Ez a minta akkor használandó, ha az alkalmazás nagy számú objektumot használ és a legtöbb objektum-tulajdonság külsővé tehető. A „*Programtervezési minták*” című könyv [GOF95] a következő példát hozza:
Az objektumközpontú szövegszerkesztők jellemzően objektumokkal ábrázolják az olyan beágyazott elemeket, mint a táblázatok és az ábrák. A dokumentum egyes karaktereit azonban már nem önálló objektumok képviselik, még ha ez a program legfinomabb szintjén is jelentene rugalmasságot. Ha így lenne, a karaktereket és a beágyazott elemeket egységesen kezelhetnénk, kirajzolási és formázási módjuktól függetlenül. A program a nélkül lenne bővíthető új karakterkészletek támogatásával, hogy a többi szolgáltatásra ez hatással lenne. Az alkalmazás objektumszerkezete pontosan tükrözhetné a dokumentum „fizikai” szerkezetét.
A módszer hátulütője a magas költség. Még a szerényebb méretű dokumentumok is a karakter objektumok százezreit igényelnék, ami rengeteg memóriát emésztene fel, a futási sebesség pedig elfogadhatatlan szintre csökkenne.
Erre a problémára nyújt megoldást a Pehelysúlyú minta, úgy hogy megmutatja, hogyan oszthatunk meg objektumokat finomabb szinten való használatra anélkül, hogy a költségek az egekbe szöknének.
A pehelysúlyú objektum olyan megosztott objektum, amely egyidejűleg több környezetben használható. Mindegyik környezetben önálló objektumként viselkedik, vagyis nem különböztethető meg egy nem megosztott objektum példánytól, és nem élhet feltételezésekkel működési környezetéről. A minta kulcsa a belső és külső állapot megkülönböztetése. A belső

állapot a pehelysúlyú objektumban tárolódik, és olyan információkból áll, amelyek függetlenek a pehelysúlyú objektum környezetétől, így megoszthatók. A külső állapot ezzel szemben a környezettől függően változik, ezért megosztása nem lehetséges. A hívó feladata, hogy külső állapotot adjanak át a pehelysúlyú objektumnak, amikor az igényli.
Egy szövegszerkesztőben például az ábécé minden betűjéhez létrehozhatunk egy-egy pehelysúlyú objektumot. Ezek csak egy karakterkódot tárolnak; a dokumentumban elfoglalt helyét és a betűstílust a szöveg-elrendező algoritmusok és a karakter megjelenési helyén érvényben lévő formázási parancsok alapján határozzuk meg. A karakterkód belső állapotinformáció, míg a többi külső.
A következő OMT osztálydiagram [Rum94] a fent leírtaknak megfelelő osztályszerkezetet mutatjuk be.



A képjel (Glyph) a grafikus objektumok elvont őse. Leszármazottjai közül a karakter (Character) pehelysúlyú. A dokumentum hasábokból (Column) és sorokból (Row) áll. A képjelet ki lehet rajzolni (Draw). A sor többek közt tudja a helyét, így amikor karakterenként kirajzolja magát, akkor a karakterrajzolónak át tudja adni, hová kell kirajzolni a karaktert.
Láthatjuk, hogy ez a minta az általunk javasolt 3NF-ban van, hiszen a műveletének eredménye csak attól függ, hogy mi a belső állapot (a kirajzolandó karakter kódja) és milyen paramétereket kapott (a környezet (Context) tartalmazza hová kell kirajzolni).
Mivel a különböző karakter objektumok száma jóval kevesebb, mint a dokumentumban lévő karaktereké, lényegesen kevesebb objektum lesz, mint egy naiv megvalósításban.
Figyeljük meg, mi történik, ha a fenti problémát RA szemszögéből közelítjük meg. A naiv megoldás az az, ha minden karakternek egy sor felel meg egy hatalmas táblában, ahol az attribútumok tartalmazzák az összes lehetséges szövegformázási lehetőséget és többek közt a karakter helyét. Ez a tábla nyilván nincs 3NF-ben. A normalizálás folyamán felfedezzük, hogy kell egy Hasáb tábla, egy Sor tábla, egy Karakter-kód tábla és egy Formázási-lehetőségek tábla. Ezek után a dokumentum egy nagy kapcsolótábla lesz ezek között a táblák között. Ezzel a megoldással sokkal kevesebb helyet foglalunk.
A két módszer ebben az esetben nagyon hasonló eredményre vezet! Ha belegondolunk ez természetes. Az RA szemléletmóddal a redundanciát csökkentettük, az OOP nézőpontból pedig a környezettől való függőséget. Ha az objektumom redundás adatokat tartalmaz, akkor megváltozása más objektumok megváltoztatását teszi szükségessé, tehát környezet függőséget okoz.
Ha a két szemléletmód ugyanoda visz, akkor miért áll fent a két világ közt The Object-Rational Impedance Mismatch néven ismert ellentét. Erre talán az a válasz, hogy mind a két világ vállal redundanciát illetve környezeti függőséget a jobb válaszidő elérése érdekében, de ez az OOP világában sokkal jobban elfogadott, mint az RA berkeiben.

Más OOP Normálformák

Az OOP területén már történtek kísérletek a normálformák bevezetésére. Ezek közül talán a legismertebb Ambler és McGibbon *Building Object Applications That Work* [AM97] című munkája. Ők a következő úgynevezett osztály normálformákat, röviden ONF, javasolják (az egyértelműség kedvéért nem fordítjuk le a normálformákat csak a bevezető idézetet):
„Osztály normalizálás alatt azt az eljárás értjük, amivel felismerjük egy objektum séma struktúráját úgy, hogy közben növeljük annak koherenciáját és minimalizáljuk a köztük lévő csatolásokat."
1ONF: A class is in first object normal form when specific behavior required by an attribute that is actually a collection of similar attributes is encapsulated within its own class.
2ONF: A class is in second object normal form when it is in 1ONF and when "shared" behavior that is needed by more than one instance of the class is encapsulated within its own class(es).
3ONF: A class is in third object normal form when it is in 2ONF and when it encapsulates only one set of cohesive behaviors.

Az 1ONF lényege, hogy az osztály mezőihez tartozó (azokat karbantartó, azokon műveleteket végző) metódusoknak szerepelniük kell az osztályba. A 2ONF lényege, hogy azokat az osztályokat, amik több dologként is képesek viselkedni (pl.: vizsgaként, és tanteremként, ahol a vizsga van) szét kell szedni. A 3ONF-ben leírt „set of cohesive behaviors" sokféleképpen értelmezhető. Épp ez ennek a kezdeményezésnek a gyengéje. Nincs mögötte egy jól definiált fogalom, mint az RA normálformái esetén a funkcionális függőség.

Irodalomjegyzék

[Amb01] S. W. Ambler: *Mapping Objects to a Relational Database*. http://www.ambysoft.com/essays/mappingObjects.html

[AM97] S. W. Ambler és B. McGibbon: *Building Object Applications That Work.* Cambridge University Press/SIGS Books, 1997, ISBN: 0521-64826-2.

[CHG96] L. Constantine, B. Hendersons-Sellers és I. M. Graham: *Coupling and Cohesion: Towards a Valid metrics Suite for Object-oriented Analysis and Design.* Object Oriented Systems, 3:143-158, 1996.

[GOF95] The Gang of Four: E. Gamma, R. Helm, R. Johnson és J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education Inc., Addison Wesley Professional, 1995, ISBN: 0201-63361-2.

[MK04] Szabolcs M. és Kusper G.: *Understanding Design Patterns as Constructive Proofs.* Proceedings of ICAI 2004, Volume II., 173-182, 2004.

[Rum94] J. Rumbaugh: *The life of an object model: How the object model changes during development.* J. of Object-Oriented Programming, 7(1):24-32, 1994.

[WGM88] A. Weinand, E. Gamma és R. Marty: *ET++ - An object-oriented application framework in C++.* Object-Oriented Programming Systems, Languages and Applications Conference Proceedings, 46-57, 1988.