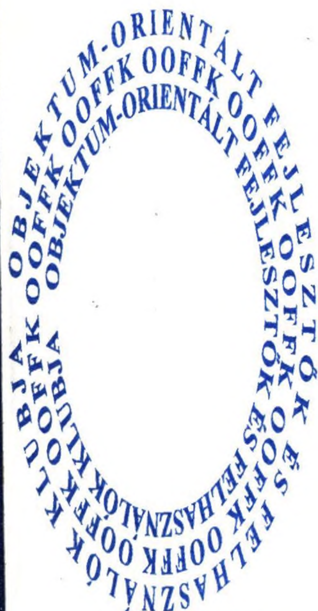


I. Országos Objektum-Orientált Konferencia

KECSKEMÉT

1996. OKTÓBER 10-11.

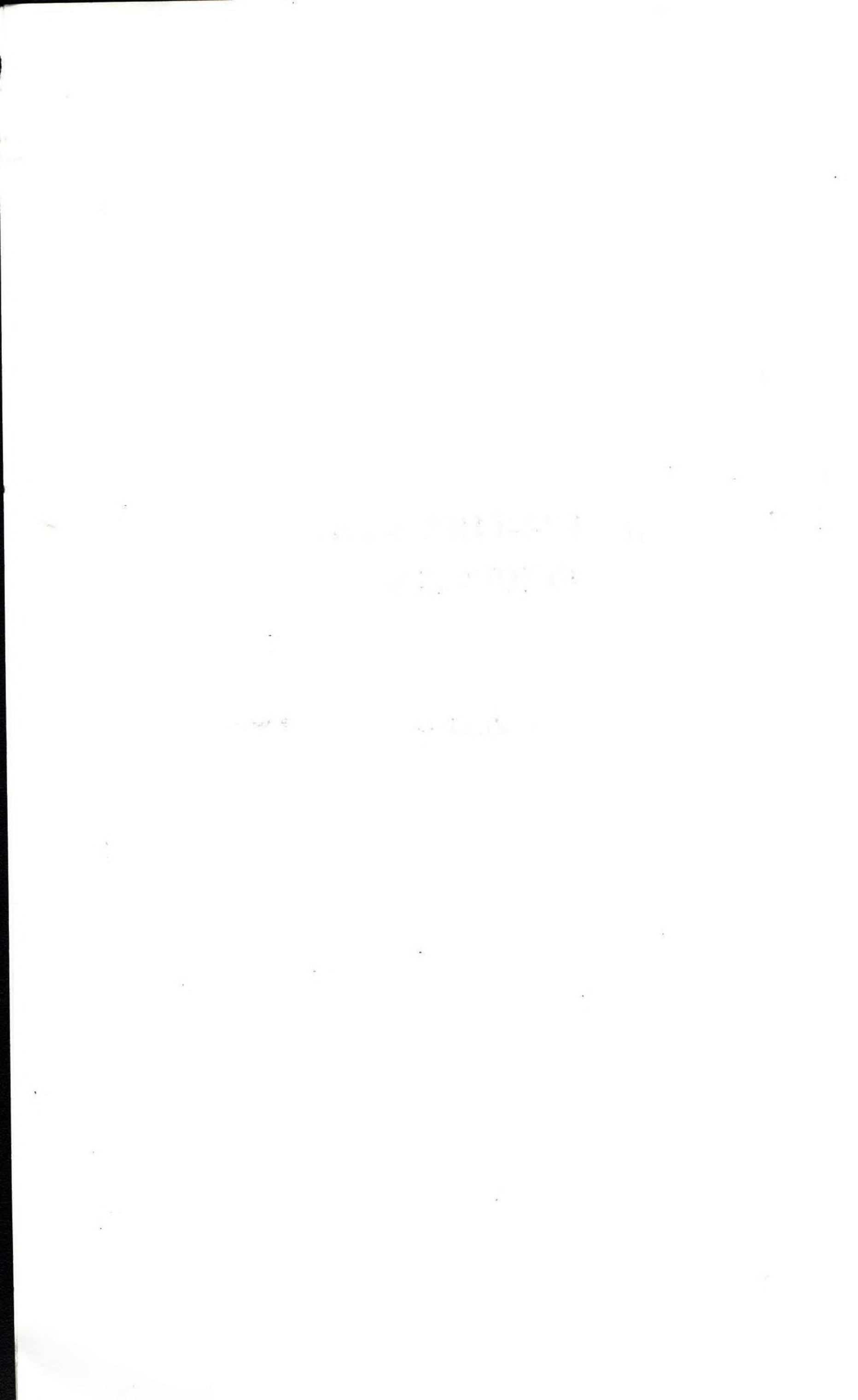
ITA/334

I. Objektum-Orientált Konferencia

Előadásvázlatok

Kecskemét

1996. október 10-11.



Öröklődés és konkurencia az objektum-orientált programozásban*

Blum László *Kozma László*

Veszprémi Egyetem
Matematikai és Számítástechnikai Tanszék
e-mail: bluml@almos.vein.hu

Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék
e-mail: kozma@ludens.elte.hu

1. Bevezetés

A konkurens problémamegoldás több fajtája ismeretes [1], amelyek különböző feladatosztályok megoldására nyújtanak hatékony eszközt. Így például a *pipeline konkurencia* jól alkalmazható olyan feladatok megoldására, ahol a megoldás több, egymás után végrehajtható részfeladatra bontható és a lépéseket ciklikusan ismételni kell. A konkurens *problémamegoldás együttműködéssel* módszere az objektum-orientált programozás.

Tanulmányunkban röviden áttekintjük a konkurens objektumok modelljeit. Megvizsgáljuk a konkurencia és az öröklődés kapcsolatát. Tárgyaljuk röviden az objektumon belüli párhuzamosság problémáit. Ebben az esetben az objektumon belül a vezérlés több szálon haladhat előre és ezáltal az objektum inkonzisztens állapotba kerülhet. A metódusok megfelelő ütemezésére mind a specifikációs szinten, mind a megvalósítás szintjén szinkronizációs eszközöket kell biztosítani. A szinkronizáció és az öröklődés azonban egymás ellen hatnak és ez további problémákat vet fel, amelyeket a szakirodalom öröklődési anomáliák néven ismer. Dolgozatunkban megvizsgáljuk az anomáliák fajtáit és egy újabb eszközt javaslunk az anomáliák kiküszöbölésére.

2. Objektummodellek

Az objektum-orientált programozás egyik központi fogalma az objektum, amely olyan entitás, ahol az adatok és a műveletek (metódusok) egységbe zárva alkotnak egyetlen kiszámítási egységet. A másik fontos fogalom a tudásmegosztás, amely jelentheti a kódmegosztást (újra felhasználást) és az osztályozást (hierarchikus strukturálást, típus - altípusképzést) is. Az objektummodellek abban különböznek egymástól, hogy az objektum belső viselkedését milyen módon specifikálhatjuk az adott modellben [1,4,5, stb.]. A konkurens objektumok kiszámítási modelljeiben [2-10] az objektumok együttműködésének módját is specifikálnunk kell. Különböző tervezési megközelítési módok az objektumok közötti kommunikáció különböző modelljeihez vezettek. Ezek között az egyik legáltalánosabb az *aktor modell* [2] és ennek kiterjesztése a *reflektív modell* [3]. A *folyamat modellek* [8] az aktor modell egyszerűsített változatai. Ezek a modellek az objektumon belüli párhuzamosságot egyáltalán nem támogatják, vagy csak igen korlátozott mértékű párhuzamosságot engednek meg. A *passzív entitásokon* alapuló objektum modellek [9] ezzel szemben lehetővé teszik, hogy az osztott memóriában elhelyezkedő passzív objektumokat egyidőben egyszerre több folyamat elérhesse. Az *aktív objektumok* egyes modelljei szintén megengedik az objektumon belüli párhuzamosságot [10].

Az ***aktor modell*** [2, 13] egy objektum alapú kiszámítási modell, amely még nem tekinthető objektum-orientált modellnek. A modell lényege röviden a következő: Egy objektum összetevői egy *viselkedés (behavior)* és egy *levélláda (mailbox)*. A viselkedés egy *leírásból* és az *ismerősök halmazából* áll. A leírás egy *kódtörzs*, amely a beérkező üzenetekre adandó választ definiálja. Az objektum egy üzenet feldolgozása közben üzenetet

* A dolgozat a T 017800 sz. OTKA és a 464/95 sz. AMFK támogatásával készült.

küldhet bármely ismerősének, bármely olyan objektumnak, amelyet az átvett üzenet megnevez és bármely olyan objektumnak, amelyet a kódtörzs végrehajtása közben az objektum hozott létre. Egy objektum levélládája azokat az üzeneteket tartalmazza, amelyeket az objektum már megkapott, de még nem dolgozott fel. Az objektumokhoz létrejöttük pillanatában egyedi levélcímek rendelődnek hozzá. A modellben sem a levélláda, sem a viselkedés nem kezelhető objektumként. Egy objektum blokkolttá válik egy üzenet feldolgozásának kezdetekor. Az üzenet feldolgozásának végén az objektum aktuális viselkedésének helyébe egy új viselkedés lép. Ezzel egyidőben az objektum blokkoltsága megszűnik. Ez azt jelenti, hogy az aktor modell nem engedi meg az objektumon belüli párhuzamosságot. Az objektumon belül korlátozott párhuzamosságot érhetünk el, ha a kódtörzs teljes végrehajtása előtt beállítjuk az új viselkedést.

Az ***objektumok reflektív modellje*** az aktor modell egy lehetséges kiterjesztése objektum-orientált kiszámítási modellé [3]. Egy objektum ebben a modellben további négy másik objektum esetleg rekurzív kompozíciójából áll:

- *Meta objektum:* a további három objektumot felügyeli, vezérli.
- *Konténer objektum:* a leírást és az ismerősök halmazát reprezentálja. A leírás metódusokra bomlik. A metódusok maguk is objektumok lehetnek, így egy objektum metódusai éppen olyan objektumok, mint más ismerősei.
- *Processzor objektum:* felel egy üzenet kezelését végző metódus tárolásáról, végrehajtásáról és koordinálja az állapotváltozásokat.
- *Levélláda objektum:* szerepe azonos az alap modellben betöltött szerepével, de a leveleket különböző politikák alapján választhatjuk ki feldolgozásra.

Az objektumok egy ***folyamat modelljét*** például az Eiffel nyelv [12] párhuzamos kiterjesztésében használják. A konkurens modell részletes leírása megtalálható [8]-ban.

A modell lényege röviden:

- egy processz egy olyan osztály egy példánya, amely a PROCESS osztály leszármazottja;
- szinkron request átvitel kivétel kezeléssel;
- szinkronizáció a jövő típusú üzenetátadáson alapul.[18]
- A modell az *osztály* és a *processz* fogalmát egyesíti.

3. Szinkronizáció és öröklődés problémái

A konkurens objektum-orientált programozás központi fogalma a konkurencia mellett a tudásmegosztás. A tudásmegosztás lényege az objektumok leírásainak újrafelhasználása. A tudásmegosztás előnye egyrészt az, hogy növekszik a modularitás, másrészt lehetőség nyílik a hierarchikus strukturálásra. A tudásmegosztás eszközei az altípusképzés és az öröklődés. A szekvenciális objektum-orientált nyelvekben ezek a fogalmak erősen keverednek. A két fogalom között a különbség az absztrakciós szintek közötti különbségben van. Az öröklődés a megvalósítás szintjének a fogalma és a kódmegosztást jelenti, míg az altípus a specifikációs, a viselkedés leírás szintjének a fogalma és az altípus hierarchia az objektum példányok viselkedésén alapul [5].

Az utóbbi időben számos újabb probléma merült fel az öröklődéssel kapcsolatban. Nézzünk meg ezek közül néhányat! Az alap aktor modellre épülő nyelvekben a kódmegosztás eszköze a *delegáció*. Az ***Act-1*** nyelv delegációs sémája például a következő [6]: Egy *a* objektum (aktor) további objektumokat ismer. Ezek neve *proxy* (helyettes). Az *a* objektum a hozzá küldött, de általa fel nem ismert üzeneteket elküldi a megfelelő helyetteséhez. Az üzenetet a helyettes (proxy) fogja lekezelni az *a* objektum helyett, de a választ váró szempontjából úgy, mintha az *a* objektum lenne. Ennek az a következménye, hogy a változók elérése *indirekt* a delegációs sémában. Ez azt jelenti, hogy a változókat a metódusokhoz hasonlóan üzeneteken keresztül érhetjük

el. Így a változók elérése függetlenné válik annak definiálási helyétől. A delegációs sémában a változók megvalósítására többféle stratégia ismert, például "változó ≡ metódus" vagy "változó ≡ objektum" stratégia.

A szinkronizáció és az öröklődés részben egymás ellen ható fogalmak. Ez számos problémát vet fel, amelyek az alap aktor modellben csak nehézkesen vagy egyáltalán nem oldhatók meg. Ezért először lecserélték a delegációs protokollt a *receptkérés* protokolljára. Ennek lényege az, hogy az objektum az általa fel nem ismert üzenetet nem küldi tovább a megfelelő helyettesnek, hanem elkéri a "receptet", metódust a helyettestől, hogy a választ ő maga adhassa meg. Önmagában a receptkérés protokollja azonban még kevés. A szinkronizációs protokollok megosztása az objektumok között továbbra is nehéz. A szinkronizációs megszorítások pedig interferálhatnak az öröklődéssel. Ennek kiküszöbölésére hozták létre például az *objektumok reflektív modelljét* [3].

Egy szinkronizációs mechanizmus *interferál* az öröklődéssel, ha az osztálystruktúrában egy lokális változás egy adott helyen a struktúra egy másik helyén is szükségessé teszi a szinkronizációs megszorítások újra definiálását.

Az objektumok reflektív modellje természetes módon támogatja az öröklődést. A reflektív modellben ugyanis a leírást kis egységekre bonthatjuk, amelyek könnyebben komponálhatók össze más egységekkel egy új részosztály létrehozásakor. Ezzel szemben a leírás (kód) monolitikus kezelése a delegáció protokolljához vezet, ami mint láttuk nem a leghatékonyabb kódmegosztást eredményezi.

A reflektív modellben a szinkronizációs protokollok megosztása az objektumok között továbbra is bonyolult, komoly bufferelési technikát igényel. Ennek megoldását jelentheti az, ha bevezetjük az üzenetkezelés következő két szintjét. Az objektumok szintjén az objektumok közötti üzenetekkel foglalkozunk, a reflektív szinten pedig a konténerek közötti üzeneteket kezeljük. Ez a szint felelős a megosztott protokollok megvalósításáért is. Egy objektum blokkolt, amíg a konténere valamelyik szülőjével kommunikál. Hogy egy objektum egy adott időpontban melyik üzenetet kezdheti el feldolgozni, a szinkronizációs megszorításoktól függ.

A párhuzamos objektum-orientált nyelvek a szinkronizációs megszorítások programozására nyelvi primitíveket és/vagy általános objektum szintű sémákat kínálnak. A konkurens objektum-orientált programozásban a leggyakrabban használt szinkronizációs sémák a következők:

- őrfeltételes utasítások (guarded commands) esetén minden metódushoz egy logikai kifejezés, ún. őrfeltétel tartozik és csak azok a metódusok választhatók ki végrehajtásra, amelyeknek az őrfeltétele igaz.
- szinkronizáció megengedett műveletek halmazaival (synchronization with enabled-sets). Egy metódus végrehajtása után az objektum mindig meghatározza azon üzenetek halmazát, amelyekre a válasz a következőkben generálható.

A különböző szinkronizációs sémák alkalmazása esetén a kód újrafelhasználása során különböző nehézségek léphetnek fel, amelyek öröklési anomáliák néven kerültek be a szakirodalomba.

Általában *öröklési anomáliá*ról beszélünk akkor, amikor valamilyen nehézség támad a szinkronizációs kód újrafelhasználásában. Az anomáliák különböző fajtáit különböztethetjük meg:

Az *állapotfelosztási anomália (state partitioning anomaly)* akkor léphet fel, ha az alkalmazott szinkronizációs séma a megengedett műveletek halmaza. Egy objektum absztrakt állapotait részhalmazokba sorolhatjuk úgy, hogy egy részhalmazba kerüljenek azok az állapotok, amelyekhez ugyanaz a megengedett halmaz tartozik. Így az objektum szinkronizációs megszorításának megfelelő diszjunkt részhalmazokat kapunk az állapottérre vonatkozóan. Ha egy alosztályban új metódust definiálunk, akkor ez a diszjunkt felosztás tovább finomodhat az új metódus szinkronizációs megszorításának megfelelően, a finomítás pedig oda vezethet, hogy az összes többi metódus bizonyos részeit újra kell definiálni. Ezt hívjuk állapotfelosztási anomáliának.

A *történetérzékenységi anomália (history-only sensitiveness of acceptable states anomaly)* az őrfeltételes metódusok alkalmazásánál léphet fel, akkor, ha új, történetérzékeny metódust definiálunk egy alosztályban. Ekkor az új metódus engedélyezését eldöntő őrfeltételhez egy új jelzőváltozót kell bevezetni, ami a többi metódus kódjának megváltoztatásához vezet.

Az *állapotmódosítási anomália* szintén az őrfeltételes metódusok alkalmazásánál léphet fel. Hozzunk létre például két osztályból többszörös öröklődéssel egy harmadikat! Ha a két osztály metódusai egymásra ortogonálisak, akkor a szülő osztályok állapotainak nem szabadna megváltozni az alosztályban. Ennek ellenére az egyik osztály metódusainak végrehajtása módosíthatja a másik osztálytól örökölt metódusok elfogadásához szükséges állapotokat. A szinkronizációs megszorítás megváltozik, és ennek megfelelően az őrfeltételek módosítása válik szükségessé.

[14]-ben többféle megoldást találhatunk az öröklési anomáliák kiküszöbölésére. A megoldások azon a tényen alapulnak, hogy az öröklési anomáliák jelentkezése az alkalmazott szinkronizációs sémától függ. Egyetlen szinkronizációs sémát használva az anomáliák könnyen felléphetnek, míg a sémák váltogatásával elkerülhetők. Erre lehetőséget ad a szinkronizációs kód és séma lokalizálása az adott objektumra. Így egy alosztályban teljesen más szinkronizációs sémát alkalmazhatunk, mint a szülő osztályban. A szinkronizációs kód megosztása az objektumok között hasonló módon történhet a metódusok örökléséhez. A fenti cél elérhető például az ún. szinkronizálók és az átmenet specifikációk, mint szinkronizációs sémák használatával.

A *szinkronizálók* az őrfeltételek és a megengedett műveletek halmazainak kombinációjából jöttek létre. Így ezek rugalmasabb eszközt jelentenek, mint az egyszerű őrfeltételes metódusok, mivel itt egy feltételhez egy metódushalmaz is rendelhető.

Egy *átmenet specikikáció* a szinkronizálók helyett (vagy velük együtt) alkalmazható alternatív séma. Egy metódushoz tartozó *átmenet* rögtön a metódustörzs után kerül végrehajtásra és a szinkronizációs megszorításnak megfelelően meghatározza a megengedett műveletek új halmazát. Egy átmenet specifikáció több átmenetből áll. Mindegyik átmenetnek van típusa, továbbá tartozik hozzá egy metódushalmaz és egy elhagyható őrfeltétel. A sémák részletes leírása [14]-ben megtalálható, de ezen sémák alkalmazásával sem tudunk bizonyos anomáliákat kikerülni. Ezért a következő fejezetben egy újabb sémát definiálunk és mutatunk be példákon keresztül.

4. Egy szinkronizációs séma

Az általunk javasolt szinkronizációs séma a múltra vonatkozó operátorokkal kiterjesztett temporális logika formuláit használja fel. Az ítélet alapú temporális logika az ítéletkalkulus egy olyan kiterjesztése, amelyben az atomi ítéletek igazságértéke időben változik [15, 16, stb.]. Minden időponthoz egy világot (állapotot) rendelünk hozzá. Az atomi formulákból a szokásos módon, de már temporális operátorokat is használva, építhetjük fel a formulákat. Többféle temporális logika ismeretes. Mi a továbbiakban a múltra vonatkozó operátorokkal kiterjesztett, MTL temporális logikát használjuk[15].

A továbbiakban az objektumok reflektív modelljét feltételezzük. Az objektumok metódus nevei egyúttal atomi formulákat is jelölnek [15]. Egy metódus név által jelölt atomi formula akkor igaz, ha az adott metódus éppen végrehajtás alatt áll, így a reflektív modellből következik, hogy minden időpillanatban csak egy ilyen atomi formula lehet igaz.

Szinkronizációs halmazok

A továbbiakban egy erősebb eszközt adunk a történetérzékenységi anomáliák leküzdésére. Őrfeltételek alkalmazásánál az egyik probléma az volt, hogy a feltétel kódja hozzátartozott metódusához, így:

- Még mindig nem különül el a szinkronizációs rész az implementáció egyéb részeitől eléggé.

- Nem tudunk őrfeltételeket "örököltetni", nem lehet az ős őrfeltételén a leszármazottak tulajdonságainak figyelembe vételével újabb megszorításokat tenni. Ezért lép fel például az állapotmódosítási anomália [14]-ben.

Elgondolásunk lényege, hogy az egyes metódusokhoz tartozó őrfeltételeket halmazba gyűjtjük. Nevezzük ezt a halmazt szinkronizációs halmaznak (SZH). A SZH elemei az alábbi párok:

[metódus, {log.kif.1, log.kif.2, ..., log.kif.n}]

Nevezzük ezt szinkronizációs elemnek(SZE). Egy SZE tehát egy metódusból és a hozzá tartozó megszorítások (logikai kifejezések) halmazából áll. Logikai kifejezések az MTL formulái lehetnek.

Öröklődéskor a meglévő őrfeltételekhez újabb, a leszármazott objektum tulajdonságaihoz igazodó megszorításokat tehetünk. Kétféleképpen is előírhatunk újabb megszorításokat:

1. Egy meglévő szinkronizációs elem kifejezéshalmazának bővítésével.
2. Új szinkronizációs elem felvételével.

Felvételnél szerepelhet egy, már meglévő metódus a szinkronizációs elemben. Legyenek SZH1 és SZH2 szinkronizációs halmazok, MH pedig metódusok egy halmaza. A származtatott objektumban az alábbi műveleteket használhatjuk:

a., SZH1 + SZH2 jelöli az SZH1 és SZH2 unióját.

b., SZH1 ++ [MH, {log.kif.1, log.kif.2, ..., log.kif.n}] jelöli az SZH1-beli azon metódusok kifejezéshalmazának bővítését a {log.kif.1, log.kif.2, ..., log.kif.n} kifejezések halmazával, amelyek MH-ban szerepelnek. Ha MH az SZH1 összes metódusát tartalmazza, akkor a következő rövid jelölést használhatjuk: SZH1 *++ {log.kif.1, log.kif.2, ..., log.kif.n}

Egy SZE igaz egy adott időpontban, ha minden logikai kifejezése igaz.

Egy adott S szinkronizációs halmazzal rendelkező objektumhoz beérkező metódushívási kérelem engedélyezett, ha:

- Az S-ben van olyan szinkronizációs elem, melynek metódusa azonos a kért metódussal és a kifejezés halmaza igaz.
- S-ben nincs olyan szinkronizációs elem , melyben szerepel a kért metódus.

Így a szinkronizációs halmazok segítségével egy adott eljárás szinkronizációs részéhez:

a., újabb megszorításokat vehetünk hozzá a ++ művelet segítségével,

b., felvehetünk egy, az örököltektől független megszorítást a + művelettel.

Megjegyezzük, hogy az a. és b. pontok kombinációjával akár az összes, ősöktől kapott szinkronizációs részt felülbírálhatjuk azzal, hogy egy {False} kifejezéssel bővítjük az ős SZH-t és ezután a + művelettel hozzávesszük az új megszorítást.

Példák az egyes öröklődési anomáliák megoldására *szinkronizációs halmazokkal.*

Az öröklődési anomáliák lényege mint láttuk az, hogy öröklődéskor pusztán a szinkronizáció miatt újra kell definiálnunk az ős objektum egy vagy több metódusát. Az alábbiakban megadunk néhány klasszikus példát és megoldását az egyes öröklődési anomália típusokra.

Történet érzékenységi anomália léphet fel például az őrfeltételek alkalmazása esetén.

Ez a javasolt séma alkalmazásával elkerülhető. Ennek bemutatására tekintsük a következő példákat!

a.) Legyen egy korlátos buffert leíró objektum a következő:

```
Class b-buf: ACTOR {
        int in, out, buf[SIZE]
```

```
public:
        void b-buf() { in=out=0;}
        void put( int item ) {in++; }
        void get () {out++;}
synchset:
        b-bufS={[put, {in<out+SIZE}], [get, {in>=out+1}]}
}
```

Ezek után szeretnénk származtatni egy olyan osztályt, amelyet egy *gget* metódussal egészítünk ki, melynek ugyanaz a feladata, mint a *get*-nek, de csak akkor hajtható végre, ha előtte egy *get* műveletet hajtottunk végre. Ennek megvalósítása a következő:

```
Class gb-buf: b-buf {
public:
        int gget() {out++;//item torles}
synchset:
        gb-bufS=b-bufS+{[gget, {in>=out+1, ●get}]}
}
```

Példánkban csak a *gget* kifejezés halmazát kell megadnunk a többi feltétel az őstől öröklődik.

b.) Egy másik történet érzékenységi anomáliát okozó eset, mikor olyan buffert szeretnénk, ahol pl. 3 *get* után *put*-nak kell következnie. Erre a megoldás:

```
Class sb-buf: b-buf {
public:

synchset:
        sb-bufS=b-bufS+{[get, {(●get and ●◗put) or (●get and ●●get and ●●◗put)    or (◗put and ◗●
put)}], [put, {(●put and ●◗get) or (◗get and ◗●get and ◗●● get)}]}
}
```

A példa jól mutatja a temporális formulák alkalmazásának előnyeit, mivel a feladat tudomásunk szerint a [14]-ben közölt sémák egyikével sem oldható meg.

c.) Legyen adott egy régi típusú autóösszeszerelő robot. A robot különféle műveleteket tud végezni a karjával. Egy vezérlő központ utasításokat adhat a robotnak, mely azokat végrehajtja. A vezérlő csak az utasítás megfelelő időben történő kiadásáért felelős, a robotkar feladata, hogy ezen utasítás helyességét elbírálja. Legyen pl. három utasításunk a következő: *tengely_be, kerek_fel és auto_le*. Egy autó tehát akkor helyezhető le a földre, ha már van kereke. A régi típusú robotok egy szigorú sorrendet állítanak fel az egyes utasításokra. Pl. a tengely behelyezés után közvetlenül a kerék berakásnak kell következnie, utána az autó leengedésének. A robotkart az alábbi objektum reprezentálja:

```
Class robot: ACTOR {
```

```
void Robot {//egy robot működési feltételeinek beállítása, vizsgálata}
...
void tengely_be {...}
void kerek_fel {...}
void auto_le {...}
synchset:
robotS={...[kerek_fel, {●tengely_be}], [auto_le, {●kerek_fel}], ...}
}
```

A szinkronizációs halmazunkat nem töltöttük ki teljesen, csak szemléltettük a kitöltés lényegét. A szekvencia leírására kiválóan alkalmas a ● operátor.

Az újabb fajta robotoknál azonban nem szükséges követnünk ezt a fajta sorrendet. A vezérlő a gyártás során az autó tulajdonságához igazodva másféle sorrendet is megadhat. Ekkor a karnak a végrehajtás érvényességét kell ellenőriznie:

```
Class n-robot: robot {
...
synchset:
n-robotS=robotS+[kerek_fel, {◆tengely_be}]+[auto_le, {◆kerek_fel}]
}
```

Tehát csak akkor lehet feltenni az autóra a kereket, ha előtte valamikor már berakták a tengelyt és csak akkor lehet letenni, ha már a kerekét is rárakták.

Az állapotmódosítási anomália a javasolt szinkronizációs sémával szintén elkerülhető.

Klasszikus példa, mikor definiálunk egy lezárásra és felnyitásra alkalmas objektumot. Ennek örököseire jellemző, hogy metódusait blokkolni és feloldani lehet metódusaival. Példánkban az ős objektumok a b-buf és az alábbi zár objektum:

```
Class Lock: ACTOR {
        bool locked
public:
        void Lock() { locked=0;}
        void lock() {locked=1;}
        void unlock() {locked=0;}
synchset:
        LockS={[lock, {¬locked}], [unlock, {locked}]}
}
```

Legyen a leszármazott a következő korlátos buffer:

```
Class lb-buf: b-buf, Lock {
public:
        void lb-buf();
synchset:
```

lb-bufS=b-bufS*++{¬locked}+LockS

}

Tehát a *b-buf*-tól örökölt metódusok csak akkor engedélyezettek, ha a *locked* változó hamis. A fenti megoldás előnye, hogy nem kellett az ősöktől örökölt műveleteket újradefiniálni a szinkronizációs rész megváltozása miatt.

5. Összefoglalás

A javasolt szinkronizációs séma absztrakciós szintje nagyon magas. Éppen ezért a nyelv implementációs szint helyett a viselkedésleírás szintjén jobban használható szinkronizáció specifikáció eszköznek tűnik. A viselkedésleírás (specifikáció) szintje azonban még további kutatásokat igényel.

6. Irodalom

[1] G. Agha: Concurrent object-oriented programming, Comm. of the ACM Vol. 33 No. 9, pp. 125-141, 1990.

[2] G. Agha: Actors: A model of concurrent computation in distributed systems, MIT Press, Cambridge, 1986.

[3] C. Tomlison, V. Singh:Inheritance and Synchronization with Enabled-Sets, In: Proc. of the OOPSLA'89 pp. 103-112, 1989.

[4] P. America: Issues in the Design of a Parallel Object-oriented Language, In: Formal Aspect of Computing, pp. 366-411, 1989,.

[5] P. America: Inheritance and Subtyping in a Parallel Object-oriented Language, LNCS Vol. 276, pp. 234-242, 1987.

[6] J-P. Briot, A. Yonezawa: Inheritance and Synchronization in Concurrent OOP, LNCS Vol. 276, pp. 32-40, 1987.

[7] C. Baquero, F. Moura: Concurrency Annotations in C++, ACM SIGPLAN Notices Vol. 29, No. 7, pp.61-67 1994.

[8] D. Caromel: Toward a Method of Object-oriented Concurrent Programming, Comm. of the ACM Vol. 36, No. 9, pp. 90-102, 1993.

[9] D Decouchant, P. le Dot, M. Riveill A Synchronisation Mechanism for an Object Oriented Distributed System, Bull IMAG, Z.I. de Mayencin - 2, rue Vignate 38610 Gieres -France,1990.

[10] C. Neusius: Synchronising Actions, In: Proc. of the ECOOP'91, pp. 118-132, 1991

[11] C. McHale, B. Walsh, S. Baker, A. Donnelly: Scheduling Predicates, LNCS Vol. 612, pp. 177-193, 1991.

[12] B. Meyer: Eiffel, the Language, Prentice Hall, Englewood Cliffs. N. J.,1992.

[13] C. Hewitt: Viewing control structures as patterns of passing messages, J. Artifi. Intell. Vol. 8, No. 3 , pp. 323-364, 1977.

[14] S. Matsuoka, K. Taura, A. Yonezawa: Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, OOPSLA'93, pp. 109-126.

[15] C. Arapis: A Temporal Perspective of Composite Objects, In: Object-Oriented Software Composition Ed. by O. Nierstrasz & D. Tsichritzis, pp. 123-152, Prentice Hall 1995

[16] F. Kröger: Temporal Logic of Programs, Springer-Verlag, Berlin, Heidelberg, 1987.

[17] S. Crespi Reghizzi, G. Galli de Paratesi, S. Genolini: Definition of reusable concurrent software components, LNCS Vol. 512, pp. 148-166, 1991.

[18] A. Yonezawa, J-P. Briot, E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1 Association for Computing Machinery, pp. 258-268, 1986.

Common Object Request Broker Architecture
és Interfészdefiníciós nyelve

Molnár István, Moskovits Péter

(molnar@fsz.bme.hu, mosko@ttt-202.ttt.bme.hu)

Budapesti Műszaki Egyetem

Villamosmérnöki és Informatikai Kar

Folyamatszabályozási Tanszék, Távközlés és Telematikai Tanszék

A cikk az objektumok elosztottságával foglalkozó defacto szabvánnyal, a *CORBA* felépítésével foglalkozik. Az alapfogalmak összefoglalása után a kommunikáló felek és a kommunikációhoz nélkülözhetetlen közvetítő *Object Request Broker* kerül bemutatásra. A legfontosabb gondolat az objektum külső interfészének - rögzített szabályoknak megfelelő- leírása.

Bevezetés

Az 1989-ben megalakult *Object Management Group* (OMG-*http://www.omg.com*) elsődleges célja elosztott objektumorientált alkalmazások számára történő együttműködést és hordozhatóságot biztosító szabvány létrehozása volt. Az OMG-nek - mely deklaráltan specifikációkat és nem szoftvert készít - többszáz informatikai és szoftvercég a tagja (SunSoft, HP, IBM, DEC, stb.).

Az OMG által rögzített specifikáció alapja az *Object Management Architecture* (OMA), mely magas absztrackiós szinten definiálja az elosztott objektumorientált rendszerekkel szemben támasztott követelményeket. Az OMA - melynek magja az *Object Request Broker* (ORB) - a magasabb szintek felé eltakarja az objektum helyét, aktivitását, kommunikációját. Egy specifikációnak megfelelő ORB bizonyos tulajdonságoknak meg kell feleljen. A lehetséges ORB-k egy konkrét specifikációja a *Common Object Request Broker Architecture* (CORBA), mely az interfész és a szolgáltatások egy konkrét leírása. A Rev. 1.1 1991-ben, a 2.0-s változat 1993-ban(?) jelent meg. Míg az 1.1-t azonnal, az elkészítés pillanatában, a 2.0-t csak hosszabb kivárás - a tapasztalatok összegyűjtése - után tették nyilvánossá.

A CORBA fő komponensei:

- ORB mag,
- Interface Definition Language (IDL),
- Dynamic Invocation Interface (DII),
- Interface Repository (IR),
- Object Adapter-ek (OA).

Alapfogalmak

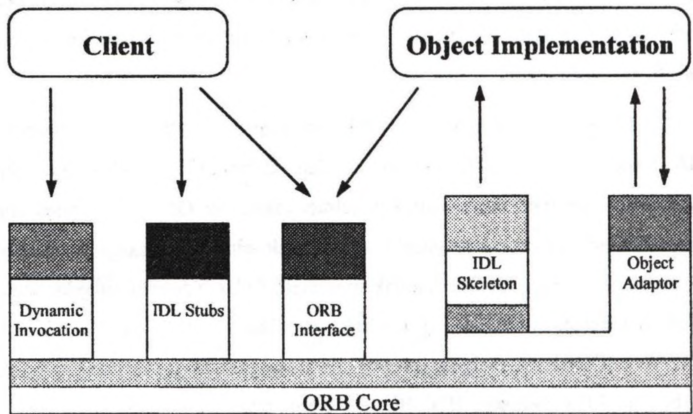
Objektum: azonosítható, egységbezárt entitás, mely egy vagy több szolgáltatást nyújt kliensnek.

Interfész: Azoknak a lehetséges kéréseknek a leírása, melyekkel a kliens objektumhoz fordulhat. Az interfész definíciós nyelve az IDL, melyet olyank használnak, amikor statikusan - fordítási időben - ismert az objektumok interfész (Dinamikus hozzáférésről a DII és az IR gondoskodik.)

Objektumreferencia: az objektumok egyedi azonosítására, azokra tört hivatkozáshoz használt speciális - mutató jellegű - adattípus.

Kérés (Request): olyan esemény, melyet a kliens küld a kiszolgáló objektumn (CORBA terminológiában: objektumimplementációnak). A kéréshez tart információ tartalmaz egy műveletet, egy célobjektumot, esetleg paraméterek valamint egy opcionális kérés-környezetet a kérés további részleteiről.

Művelet (Operation): olyan szolgáltatás, melyet a kliens az objektumimplementációt kérhet. A műveletet meghatározza azonosítója, valamint aláírása, melynek rész képezik az adattípusok, a művelet eredményei valamint a kivételkezelés.



1. ábra: A Common Object Request Broker Architecture felépítése

Az ORB Core

Az ORB feladata az objektum reprezentáció biztosítása (objektumok megtalálása), az objektum-implementációk felkészítése kérések fogadására, valamint a kérésben foglalt adatok továbbítása. A kérés feldolgozása során az ORB megkeresi a megfelelő objektum-implementációt, átadja a paramétereket és a vezérlést az objektumnak, és végül visszaadja a kimenetet és a vezérlést a kliensnek. A kérés kiszolgálása alatt az objektum-implementáció igénybe vehet ORB szolgáltatásokat az OA-n keresztül.

A kliens által látott objektum teljesen független attól, hogy az objektum fizikailag hol helyezkedik el, milyen programozási nyelven írták, és minden egyébtől, ami az objektum interfészében nincs megadva.

Az ORB az objektumreferencia segítségével tartja nyilván, azonosítja az objektumokat. Új objektum létrehozása esetén ORB Core-t értesítik, biztosítva ezzel, mindig minden objektum elérhető legyen a hálózaton.

Az objektum implementáció

Az objektum implementáció tartalmazza az objektum szemantikus definícióját, azaz az adatszerkezetet és a metódusokat. Az objektum implementáció a kérés kiszolgálásához természetesen fodulhat más kérdéssel további objektumokhoz. Ha ezt teszi, egyszerre játssza el a kliens és az objektum implementáció szerepét.

Kapcsolattartás az objektumok és az ORB Core között

Az *IDL stub* (IDL csonk) a kliens és az ORB között teremti meg a kapcsolatot. Az ORB Core az objektum implementációt az *IDL skeletonon* (IDL csontváz) keresztül érheti el. Az ORB-vel folytatott nagyon alapvető információcserét az *ORB Interface*-en keresztül bonyolítja a kliens és az objektum implementáció. Fontos jellemzője, hogy közvetlenül az ORB-hez vezet, különböző ORB-k esetén is megegyezik, valamint hogy független az objektumok interfészétől, vagy az objektumadaptertől. Lehetőség szerint nem ezt a kapcsolatot, hanem az objektum adaptert, az IDL csontvázat, az IDL csonkot, stb. használják kommunikációra. Az objektumimplementáció az ORB szolgáltatását az *objektum adapteren* keresztül érheti el. Az ORB objektum adapteren keresztüli szolgáltatásai a következők:

- objektumreferenciák generálása és értelmezése
- metódusok hívása
- beavatkozások biztonsági kérdései

- objektum és implementáció aktiválás és deaktiválás
- objektum referenciák megfelelő objektum implementációkhoz való rendelése
- implementációk nyilvántartása (registration)

Ha ezeket a szolgáltatásokat az ORB nem nyújtja, az objektum adapter maga kell h gondoskodjon róluk. Például ha az objektumreferencián kívül még egyéb adatot is el akar tárolni az objektum implementációról, akkor azért, hogy ne kelljen minden egyes alkalom ezért az adatért az objektum implementációhoz kéréssel fordulni, az ORB (általában objektum adapter) ezt eltárolhatja. Ha ezt az ORB nem támogatja, minden olyan eset amikor erre a speciális adatra van szükség, kérést kell az objektum implementációhoz intéz

A CORBA specifikáció több objektum adaptert definiál. Ezek közül n különböztetett szerepet játszik a *Basic Object Adapter* (BOA). A BOA általános (speciális) objektum implementációkkal tud együttműködni. Előfordul azonban, hogy mégsem kielégítő. Éppen ezért lehetőség van speciális objektum adapter elkészítés használatára is. Ilyen objektumadapterek lehetnek például a *Library Object Adapter*, n "pehelysúlyú", könyvtári implementációként létező objektumokat kezel, az objektumorien adatbázis adapter, mely egyrészt biztosítja a kapcsolatot az adatbáziskezelőhöz, másr gondoskodik az objektumok perzisztens (hosszútávú) tárolásáról.

Interface Repository

Az *Interface Repository* (IR) ORB része, mely állandó (perzisztens) tárolási h biztosít az objektum interfész definíciók számára és nyilvántartja az objektumimplement helyét. Elsődleges feladata a *Dynamic Invocation Interface* (DII) támogatása.

Dynamic Invocation Interface

Előfordulhat, hogy nem ismerjük fordítási időben az objektumok interfészeit, f időben dől el, hogy melyik objektummal is akarunk foglalkozni az IR-ben tárolt objektu közül.

Egy grafikus felhasználói interfész (Graphical User Interface - GUI) tervező prog listát kínál a felhasználható elemekből, ezek közül tallózunk. Megnézzük, hogy melyik viselkedik, hogy néz ki. Ezek fordítási időben nem (feltétlenül) álltak rendelkezésre, utól készülhettek felhasználható elemek. Egy ilyen GUI készítő programnak csak az len feladata, hogy tudja, hogy lehet az IR-ból az egyes elemek megjelenítéséhez szük adatokat lekérni.

A DII tehát olyan dinamikus problémák esetében segít, amikor az IDL csonk már tehetetlen. Ugyanakkor, ha csak egy mód van rá, érdemes elkerülni a használatát, mert bonyolult, kommunikáció igényes, költséges megoldás.

Összefoglalás

Az informatikai világban a 90-es években két vonalon is robbanásszerű fejlődés következett be. Az egyik a szoftverfejlesztésben alkalmazott objektum-orientált koncepciók elterjedése, a másik a nagy hálózatok általánossá válása. Felmerül tehát az igény, hogy lehet a két terület által nyújtott előnyöket egyszerre kihasználni. Mi értelme például olyan objektumokat elkészíteni, melyeket már valaki korábban elkészített? A választ az OMG-be tömörült résztvevők látszanak megadni: elosztott objektumok a hálózaton. Ahhoz, hogy a megoldás kellően általános lehessen, pontosan specifikált, rugalmas építőkockákból kell álljon.

A kliens-kiszolgáló rendszerek új generációja: CORBA alapú elosztott rendszerek

Kovács András
HiCare Kft.

Bevezetés

A kliens-kiszolgáló rendszerek korábbi, mérföldkövet jelentõ generációit - *file-szerver* - és *adatbázis szerver alapú architektúrák* - valamint kisebb jelentõségû vonulatait - *a csoport-munka rendszerek (groupware), és tranzakció feldolgozó monitor (TPR) bázisú alkalmazá* - követõen egy új korszak köszönt ránk, a CORBA alapú elosztott rendszerek (*distributed object computing*) világa.

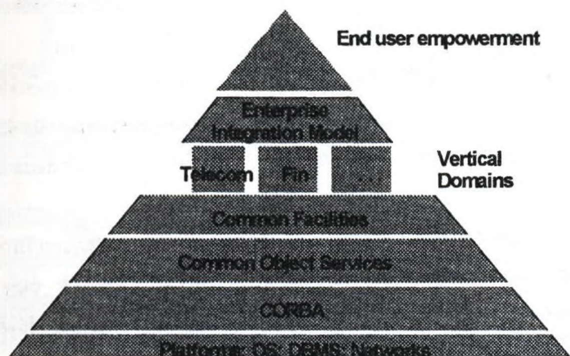
Az elosztott objektum technológia igéri a legrugalmasabb cliens-kiszolgáló arhitektúrát. Az objektumok magukba ágyazzák az adatokat és az üzleti logikát, menedzselik önmagukat és kezelik erõforrásaikat. Különbözõ platformokon, különbözõ operációs rendszerek alatt, földrajzilag egymástól távol esõ helyeken futhatnak, továbbá kommunikálhatnak a hagyományos technológiával épített alkalmazásokkal object wrapperek segítségével.

Az üzleti vagy más néven probléma orientált objektumokból (business objects) - amelyek különbözõ nyelveken készíthetõek, lásd a 2. ábrát - épülnek az alkalmazások, az alkalmazás jellegétõl függõen statikusan, vagy dinamikusan. Az alkalmazás a probléma specifikus komponensek segítségével - a megvalósítás részleteit elrejtve - képezi le az üzleteti folyamatokat. Ennek az új ún. "intergalaktikus kliens-kiszolgáló" korszaknak a technológi infrastruktúráját rakja le az Object Management Group vezérletével a világ számítástechnik közössége, eddig soha nem tapasztalt közös erõfeszítéssel és konszenzussal, a CORBA szabványok formájában.

Az elõadás ismerteti a CORBA komponens alapú arcitektúra felépítését, ennek fõ elemeit, majd a röviden ismeretjük a MediNet projektet, mely egy nagyméretû elosztott egészségügy projekt, ahol a HiCare Kft a piacvezetõ object request brokert (ORB), az IONA Techologi Orbix nevû termékcsaládját alkalmazza.

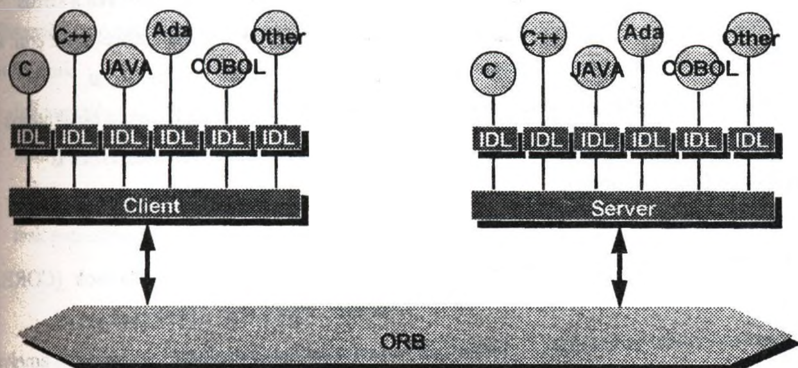
A CORBA komponens alapú architectúra

Az Object Management Group (OMG) - amely már több mint 700 tagot számlál - egy elosztott, komponens alapú arhitektúrához szükséges teljes infrastuktura kialakítását és szabványosítását tûzte ki célul maga elé. A komponens alapú architektúra felépítését az 1. ábrán láthatjuk.



1. ábra CORBA komponens alapú architektúra. Referencia modell

A komponens architektúrában a CORBA-val jelölt szemantikus réteg a szoftver busz (2.ábra), amelyen keresztül a különbözõ gyártók által különbözõ nyelveken megírt objektum komponensek együtt tudnak mûködni a hálózatokon, operációs rendszereken keresztül. Az egyes komponensek interfészei CORBA interfész definícós nyelven - IDL-ben -vannak specifikálva. Az IDL programozási nyelv független.



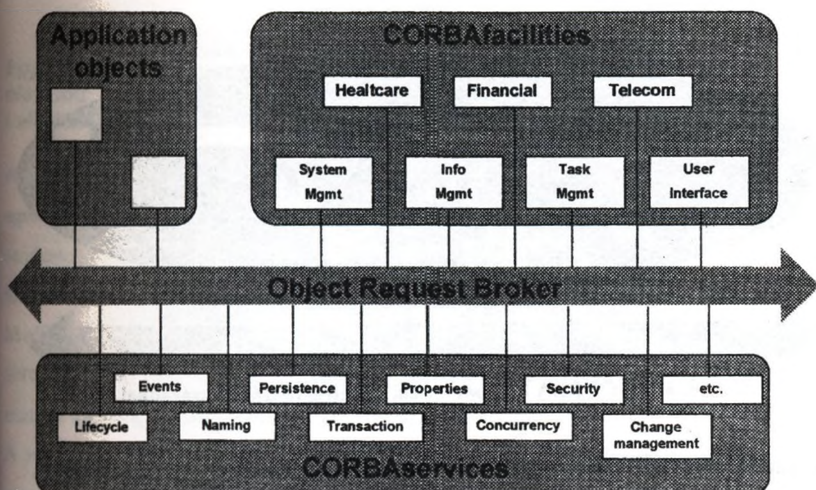
2. ábra CORBA Object bus

A CORBA szabvány [1] szerint az interfész és az implementáció szigorúan szét van válasz
A komponesek interfésze definiált, ennek segítségével kapcsolódik a szoftver buszra, de az
interfész mögötti implementáció tetszõleges, és a külvilág elõl el van rejtve: a szállítónál
alkalmazott technológiától függ.
Az IDL a kapcsolat, szerzõdés, amely összeköti a szervereket és a klienseket. A szerverek
valamilyen szolgáltatást biztosító objektum komponensek, a kliensek ezeket felhasználó
objektumok. Egy szerver természetesen más szerverek kliense is lehet.
Az objektum komponensek IDL-ben definiálják a szolgáltatásaikat - metódusok, attribútu
hibakezelõ kivételek, öröklési hierarhia formájában - amelyeket exportálnak, azaz elérhetõ
tesznek más komponensek számára. A CORBA a metódus hívásokat disztribútálja: az
attribútumok is metódusokon keresztül érhetõk el.
A szoftver vagy objektum buszt az egyes nódokon futó Objekt Request Broker (ORB) ne
rendszer komponsensek biztosítják. Az ORB-k segítségével a kliensek vagy statikusan
(fordítási idõben) vagy dinamikusan (futásidõben) adhatnak ki metódus hívásokat. Interfé
tároló (interface repository) segít a futásidõben történõ interfész keresésben, ill. hívás
öszeállításban.

A következõ szemantikus réteg azon közös szolgáltatásokat megvalósító komponensek -
CORBA rendszerszintû szolgáltatások vagy CORBAServices [2], - amely szolgáltatásokra
funkciókra az elosztott komponens alapú rendszerek építése során általában szükség van.
fontosabb szolgáltatások a 3. ábrán láthatóak.
A komponensek szállítói úgy tudják a komponenseket elkészíteni, hogy nem kell a közös
szolgáltatásokat kifejleszteniük, azokat standard formában minden alkalmazás el tudja érni
szállítók komponensei a közös szolgáltatásokat igénybe véve látják el feladataikat,
integrálódnak a rendszerbe. Nincs szükség a felhasznált szolgáltásokat biztosító objektumo
forrás kódjára, nem kell linkelni õket. Ezt a réteget rendszerszintû keretrendszernek is
tekinthetjük (lásd a 4. ábrát).

A következõ lépés a szemantikus hierarchiában az alkalmazás szintû szolgáltatások (CO
terminológiával CORBAfacilities [3]) rétege, amelynek objektumai aalkalmazásszintû
keretrendszer funkciókat látnak el, biztosítják azokat a szolgáltatásokat és szabályokat ame
az alkalmazási objektumok (business objects) rendszerbe integrálásához, menedzsésükhöz

szükségesek. Ezeket a szolgáltatatásokat közvetlenül a domain alkalmazási objektumok használják.



3. ábra CORBA Object Management Arcitecture

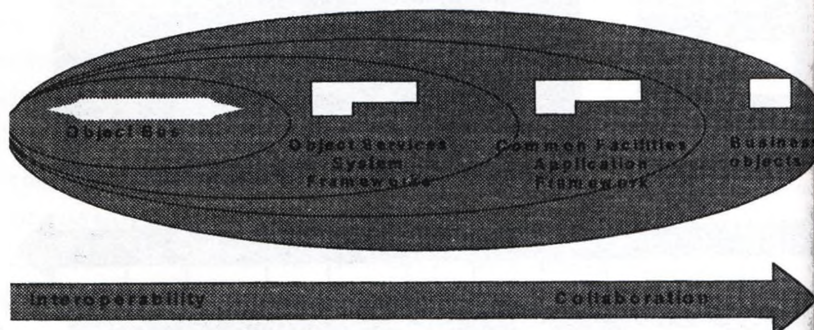
A CORBAfacilities szolgáltatásait alapvetõen két kategóriába sorolják:

- horizontális, és
- vertikális

A horizontális komponensek minden domain alkalmazás számára szükségesek lehetnek, míg a vertikális elemek egy alkalmazási terület alapvetõ, közös, a fejlesztõk által felhasználható/specializálható komponensei. A horizontális elemeket négy kategóriába sorolhatjuk:

- felhasználói interfész
- információ menedzsment
- rendszer menedzsment
- task menedzsment (workflow, scripting, hosszú tranzakciók, agent technológia, scripting)

A vertikális alkalmazási keretrendszerek közös szematikus, infrastruktúrális alapokon nyugszanak, ez közös alkalmazási architektúra (business application



4. ábra. CORBA komponensek, az infrastuktúra határokkal

arhitecture), amely az 5. ábrán látható. Az OMG-n belül a következõ területeken folyik a vertikális keretrendszerek szabványosítása:

- egészségügy
- tévközlés
- gyártéstechnológia
- pénzügy
- elektrónikus kereskedelem

A hierarchia csúcsán a vállalat-specifikus alkalmazási objektumok foglalnak helyet. A COR alkalmazási architectura megteremti a szemantikus és infrastuktúrális alapokat a cliens-kiszolgáló rendszerek új generációjához. Az elosztott objektum komponensek segítségével nagy monolitikus alkalmazások kisebb, jobban kézben tartható részekre bonthatóak, amely az objektum buszon együtt mûködnek egymással. Az alkalmazás szintû objektumok a problémaspecifikus, a valós élet objektumait valósítják meg, amelyekbõl új alkalmazások építhetõek akár dinamikusan is. Az elosztott objektum orientált technológia az az eszköz, amely képes lesz a globlis hálózaton létezõ millónyi szoftver elem együttes mûködtetésére, menedzselésére.

5. ábra alkalmazási CORBA arhitektúra

Enterprise Specific Business Objects

Financial Business Objects | Manufacturing Business Objects | Other Business Objects

Common Business Objects

Business Object Facility

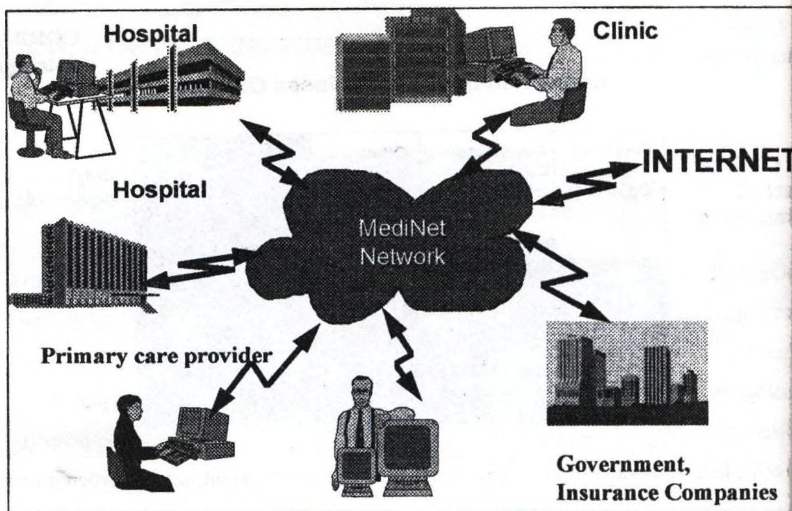
CORBA, CORBAservices, CORBAfacilities

Egy CORBA alapú egészségügy elosztott i alkalmazás

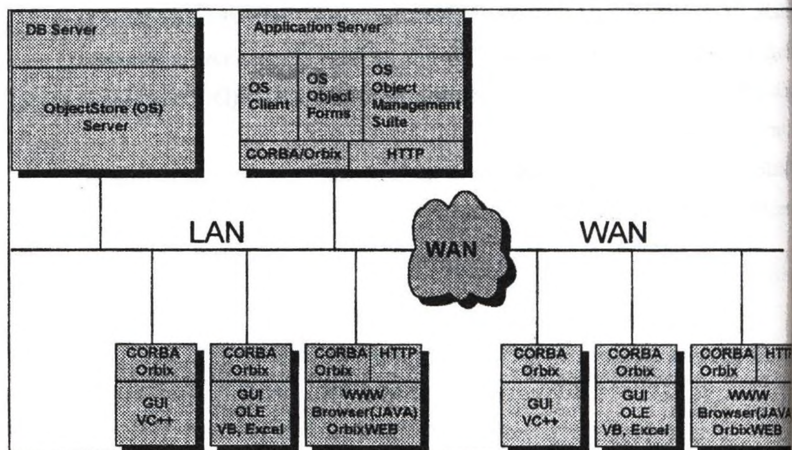
A HiCare Kft új, elosztott egészségügyi információs rendszerét a CORBA technológiára építi. A MediNet rendszer (6. ábra) célja, hogy öszekapcsolja az egészségügy különbözõ résztvevõit, számottevõen javítva ezzel az egészsügyi ellátás szakmai és gazdasági hatékonyságát [4].

A projektben az IONA Technologies Orbix ORB technológiáját alkalmazzuk. Az Orbix piacvezetõ termék a CORBA technológiában. Jelenleg támogatott nyelvek: C, C++, Ada, Smalltalk, JAVA. A CORBAServices szolgáltatások közül a már rendedelkezésre állók: Naming, Events és rövidesen kaphatóak lesznek az Object Transaction és Security szolgáltatások. Az Orbix hatékonyan megoldja a CORBA és OLE(DCOM) komponensek együttmûködését.

Az alkalmazott technológiát jól szemléltetõ kórházi-rendelõintézeti modul rendszer architektúrája a 7. ábrán látható.



6. ábra. A MediNet projekt



7. ábra. A MediNetben alkalmazott arhitektúra és technológiai elemek

Összefoglalás

Az elosztott objektum technológia igéri a legrugalmasabb cliens-kiszolgáló arhitektúrát. Az objektumok magukba ágyazzák az adatokat és az üzleti logikát, menedzselik önmagukat és kezelik erõforrásaikat. Különbözõ platformokon, különbözõ operációs rendszerek alatt, földrajzilag egymástól távol esõ helyeken futhatnak, továbbá kommunikálhatnak a hagyományos technológiával épített alklamazásokkal object wrapperek segítségével.

A CORBA architectura megteremti a szemantikus és infrastuktúrális alapokat a cliens-kiszolgáló rendszerek új generációjához. Az elosztott objektum komponensek segítségével a nagy monolitkus alkalmazások kisebb, jobban kézben tartható részekre bonthatóak, amelyek az objektum buszon keresztül együtt mûködnek egymással. Az alkalmazás szintû objektumok (business objects) a problémaspecikus, a valódi objektumokat valósítják meg, amelyekbõl új alkalmamazások építhetõek akár dinamikusan is.

Az elosztott objektum orientált technológia az az eszköz, amely képes lesz a globális hálózaton dolgozó millónyi szoftver elem együttes mûködtetésére, menedzselésére.

IRODALOM

[1] Object Management Group: The Common Object Request Broker: Arhitecture and Specification

[2] Object Management Group: CORBAservices

[3] Object Management Group: CORBAfacilities

[4] Kovács A, Nick J,: MediNet: Elosztott, integrált egészségügyi információs rendszer, I. Országos Objektumorientált Konferencia, 1996

Az OO és strukturált módszertanok összehasonlítása

(SSADM és UML)

Frigó József, Hontvári József (TRIAD Kft.)

Kivonat

Bevezető

A cikkben a módszertanok összehasonlításához használható szempontok meghatározásával, és az objektum-orientált és a strukturált módszertanok összehasonlításával foglalkozunk. A strukturáltakból az SSADM-et, az objektum orientáltak közül az UML-t vesszük alapul.

Először általában szólunk a módszertanokról. A módszertan fogalmát szoftverfejlesztési technológiaként definiáljuk. Leírjuk, hogy általában milyen célokat tartanak szem előtt a módszertanok fejlesztésénél: az elkészült rendsze feleljen meg a megrendelő valós igényeinek, legyen karbantartható, a projekt legyen menedzselhető, a módszertan adjon szabványt a projekttagok munkavégzésére, növelje a hatékonyságot, segítse elő a hibák korai felismerés csökkentse az alapszoftvertől való függést.

A következő részben leírjuk, hogy a fenti célok eléréséhez a módszertanok mily eszközöket használhatnak. Külön kitérünk ezek közül minden tervezési munka alapjára, a modellezésre. Végül megadjuk hogy milyen elemekből is épülnek fel módszertanok (dokumentációs szabvány, tevékenység terv, technikák).

Röviden ismertetjük (célszerűen több önálló keretes részben), az oo és a strukt módszertanok elvi alapjait, az SSADM, UML módszertant, valamint hogy miért az előbbi módszertanok választottuk az összehasonlításhoz.

Megadjuk ás indokoljuk az összehasonlítási szempontokat. Külön kiemeljük, ho függnek össze a szempontok egymással és a módszertanok elé kitűzött célokka

A leginkább hangsúlyos részben a fenti szempontok alapján összehasonlítjuk az és a strukturált módszertanokat. Az összehasonlítást döntően az információs rendszerek szemszögéből végezzük.

Az utolsó részben a módszertanok közötti választásról írunk. Egy döntési helyzetben lévő szervezetnek nem önmagában kell értékelnie egy módszertant, hanem azt is figyelembe kell vennie, hogy milyen környezetbe kerül be a módszertan. A szempontok súlyát mindenképp az alkalmazási környezet függvényében kell meghatározni.

A módszertanok céljai

A módszertanok összehasonlításához elsőként meghatározzuk, milyen célja va módszertanok alkalmazásának. Később majd megvizsgáljuk, hogy az egyes módszertanok milyen eszközöket adnak a fenti célok eléréséhez, és hogy ezek eszközök egymáshoz viszonyítva mennyiben alkalmasak a cél elérésére.

A módszertanok elé három alapvető célt szoktak kitűzni: biztosítsa az elkészült rendszer minőségét, a projekt kézbentarthatóságát és a rendszer karbantarthatóságát. Módszertanonként változó, hogy melyikre mekkora hangsú fektetnek.

A készülő szoftvernek meg kell felelnie a felhasználók igényeinek. Ez egy alapvetően fontos és nem triviális feladat. Számtalan esetben fordult már elő, hogy a felhasználó másra gondolt, amikor a szoftvert megrendelte, mint amit a fejlesztők termékként előállítottak. A megrendelőnek tisztában kell lennie azzal, hogy mi a célja az informatikai beruházással, de csak ritkán van határozott — és később is használhatónak bizonyuló — elképzelése a tervezett rendszerről, ezért a fejlesztőnek kell felmérnie, hogy mit tudnának a felhasználók munkájuk során hatékonyan használni. Az igények helytelen megállapításának következménye, hogy a megrendelő a tervezett rendszert egyáltalán nem, vagy csak részlegesen, alacsony hatékonysággal tudja használni. A felelősséget természetesen át lehet hárítani a megrendelőre, de a szoftveripari tendenciák azt mutatják, hogy megfelelő technológia használatával a fejlesztők ki tudnák küszöbölni az ilyen jellegű problémákat.

Hibák minél korábban felismerhetők legyenek. A specifikációs és a tervezési hiba kijavítása annál költségesebb, minél későbbi fejlesztési fázisban derül ki. Az elemzés során nem jelenthet túl nagy problémát, ha kiderül, hogy két követelmény ellentmond egymásnak, a fizikai tervezés vagy az implementáció során viszont számtalan módosítást kell a terven eszközölni.

A projektnek tervezhetőnek kell lennie. A szoftverfejlesztés egy hagyományosan kockázatos iparág. Ez többek között azzal magyarázható, hogy még nincsenek általánosan használt, kiforrott menedzsment technikák.

Teljesen eltérő szereppel rendelkező emberek munkáját kell támogatnia. A szoftverkészítés csapatmunka, amelyben részt vesznek vezetők, tervezők, programozók, tesztelők és még sokan mások. A szervezeten belüli kommunikációhoz szükség van egy közös nyelvre, amelyet mindenki megért. Hosszú idő alatt a közös nyelv magától is kialakul, de ha nincs írott szabvány, egy új ember csak hosszú idő alatt tud beilleszkedni.

Hatékony fejlesztési munkát és működést kell lehetővé tennie. Jó ha a módszertan segíti például a párhuzamos munkavégzést. Az is fontos, hogy jól integrálható legyen a kor szellemének megfelelő fejlesztői eszközökkel.

A szoftvernek könnyen karbantarthatónak kell lennie. A szoftver életciklusa nem ér véget azzal, hogy át lett adva a felhasználónak. A rendszert felhasználó szervezet igényei és külső körülményei (például az üzleti célok, jogszabályok) állandóan változnak, és ezt az informatikai háttérnek követni kell. Nem ritka eset, hogy a szoftver már akkor elavult, amikor még el sem készült. Költség szempontjából is fontos a karbantarthatóság. A karbantartási munkák általában jóval drágábbak a fejlesztésnél — 2 és 4 közötti szorzó sem ritka. Olyan módszertanra van szükség tehát, amellyel utólag is könnyen megérthető, hatékonyan módosítható és bővíthető rendszert kapunk.

Alapszoftvertől való függés elkerülése. A hardver, az operációs rendszer, adatbázis-kezelő stb. gyorsabban avul el, mint az információs rendszer alapjául szolgáló üzleti tevékenység. A rendszer több megrendelő esetén több platformon is létezhet. Fontos lehet ezért, hogy egy új rendszerre történő átállás során a régi munkabefektetés minél nagyobb részét meg lehessen őrizni.

Szempontok

A módszertanok elé kitűzött célok és a módszertanok által biztosított eszközökből kiindulva megadunk egy szempontrendszert, mely véleményünk szerint a részletes összehasonlítás alapja lehet.

Meg kell jegyezni, hogy a módszertanokat értékelni aszerint kell elsődlegesen, hogy milyen mértékben elégítik ki a szoftverfejlesztéssel szemben támasztott követelményeket, amelyek azonban a környezettől függenek. Módszertanokat tehát önmagukban értékelni értelmetlen, egy választásnál figyelembe kell venni a külső körülményeket is. Így például meg kell vizsgálni, hogy a módszertan képes-e

egyáltalán egy adott rendszer modellezésére, és ha igen, akkor milyen korlátoz
mellett.

Az elkészült rendszer minőségével összefüggő szempontok

- Milyen mértékben kényszeríti ki a módszertan a **keresztellenőrzéseket**? (A keresztellenőrzések elősegítik a hibák korai, megvalósítást megelőző felismerését.)
- Mennyiben segíti elő az elkészült rendszerterv és a módszertan a **tesztelés**
- Mennyire **felhasználóbarát** a módszertan **jelölésrendszere**? (Segíti a felhasználóval való kommunikációt.)
- Milyen mértékben segíti a módszertan a **prototípus** készítést? (A felhaszná igények felmérését, a hibák korai felismerését segíti, ha a módszertan rész prototípus készítés. A prototípus a vizuális ábrázolás egy módja.)
- Milyen mértékben segíti a módszertan kész elemek **újrafelhasználását**? (Újrafelhasználásra példák a tervezési minták, az eljárás- és objektumkönyvtárak. Ezek felhasználása csökkenti a fejlesztési időt és növ megbízhatóságot. Az újrafelhasználásánál nagy megtakarítást jelenthet pél az, hogy elmaradhat a tesztelés.)

Az elkészült rendszer könnyű karbantarthatóságával összefüggő szempontok

- Mennyire lesz érthető a módszertan szerint készített rendszerterv és a forráskód? (A karbantartáshoz a fejlesztőnek meg kell értenie a jelenlegi rendszer felépítését, a rendszerrel szembeni követelményeket stb. Lényege lehet például, hogy mennyire közvetlen a kapcsolat a rendszerterv és a forráskód között.)
- Mennyire lesz az elkészült rendszer bővíthető? (A karbantartás egyik típusa amikor új szolgáltatásokat építenek a rendszerbe. A bővíthetőség szempont mind a rendszertervre, mind a rendszerre felmerül.)
- Mennyire lesz az elkészült rendszer módosítható? (A karbantartás másik típ amikor a rendszer egy meglévő típusát módosítják. A módosíthatóság szempontja szintén mind a rendszertervre, mind a rendszerre felmerül.)

A szoftverfejlesztési folyamattal, a projektvezetéssel kapcsolatos szempontok

- Mennyire egységes a módszertan által javasolt jelölésrendszer? (Az egység egyszerű jelölésrendszer a különböző részterületeken dolgozó fejlesztők kommunikációját segíti.)
- Mennyire érthető a jelölésrendszer? (A könnyen érthető jelölésrendszer csökkenti a fejlesztők betanulási idejét.)
- Mennyire bontható állomásokra a módszertant használó projekt? (A projekt tervezhetőségét segíti, ha minél több, jól megfogható lépésre bontható egy projekt.)
- Mennyire lehet a projekt során párhuzamoson dolgozó, független csoportok kialakítani? (Elősegíti a párhuzamos munkavégzés, az ideális csoportméret kialakítását.)
- Tartalmaz-e, és ha igen milyen minőségben ellenőrzési útmutatót a módsze (Megadja a minimális követelményeket az egyes fejlesztési lépéseket lezár ellenőrzés számára.)

Módszertanok környezetére vonatkozó szempontok

- Milyen rendszertípus esetén használható a módszertan? (Egy módszertan c bizonyos típusú rendszerekre lehet alkalmas attól függően, hogy milyen nézőpontú modelleket használ.)

- Milyen a módszertan CASE támogatottsága? (Előbb-utóbb minden módszertanhoz elkészül a CASE támogatás, mivel még egy ember is nehezen tud egy papíros rendszertervet kezelni. Önmagában nem sokat jelent, hogy van-e a módszertant támogató CASE eszköz. A lényegesebb kérdés az, hogy az eszköz mennyire teljes: csak rajzolóprogram, vagy a módszertant „értő", az implementációs eszközzel kapcsolatot tartó komplex eszköz.)
- Melyek a szóba jöhető adatbázis-kezelők? (Az elkészülő információs rendszer része lesz egy adatbázis-kezelő. Milyen egyszerű a tervezett rendszerbe integrálni az adatbázis-kezelőt?)
- Milyen implementációs technikák, fejlesztőeszközök állnak rendelkezésre? (A módszertannal elkészült rendszerterv hatékony implementálásához a módszertanhoz illeszkedő fejlesztőeszközt célszerű választani.)
- A régi rendszerekkel összefüggésben mekkora igény van a reverse engineeringre? (A reverse engineering a meglévő kódok visszafejtését támogató eszköz, illetve technika: A korábban módszertan használata nélkül elkészült szoftverekre is érdemes lehet a karbantartás során az új technológiát alkalmazni. Rendelkezésre áll-e a módszertanhoz és a korábbi fejlesztő eszközökhöz illeszkedő reverse engineering eszköz?

Összehasonlítás

A kivonat utolsó részeként címszavakban leírjuk az összehasonlítás vázlatát:

A két módszertan

SSADM:

Strukturált módszertan (DFD-alapú)

Nagyon részletesen kidolgozott

Bürokratikus rendszerek felmérésére és tervezésére

UML:

Objektum-orientált módszertan

Általános célú

Modellek (nézetek)

UML/SSADM

Használati esetek (Use Case)/DFD+FD

Objektummodell/LDM+ELH

Dinamikus modell/ELH

Életciklus

UML

elemzés (Use Case, alapmodellek)

rendszerterv (gépek, programok, párhuzamos feldolgozás)

részletes terv (alapmodellek kiegészítése implementációs részletekkel)

SSADM

megvalósíthatósági tanulmány (RCAT, DFD, LDM alapszinten)

követelményelemzés (RCAT, DFD, LDM)

követelményspecifikáció (EEM, FD)

logikai tervezés (LDPD)

fizikai tervezés

Transzformációk

SSADM: a tervmodellek jelölésrendszere különbözik az elemzési modellekétől

UML: a tervmodell az elemzési modellel azonos jelölésrendszert használ, a te[...] elemzési modellek kiegészítése implementációs részletekkel.

Karbantarthatóság

SSADM:

- Célja (többek közt) a karbantartható rendszerek készítése.
- Eszköz: alapos, szigorú és terjedelmes dokumentáció
- Probléma: a dokumentációt is karban kell tartani
- Funkció beépítése vagy átalakítása egyszerű
- Adatváltozás túl sok funkciót és túl mélyen érint (az SQL nézet nem elég e[...] eszköz)

UML:

- Célja (többek közt) a karbantartható rendszerek készítése.
- Eszköz: a dokumentáció és a rendszer azonos modellekre épül
- Eredmény: a dokumentáció és a program párhuzamos karbantartása egysz[...]
- Funkció beépítése sok objektumot érint, de nem igényel mély változásokat
- Adatváltozás hatásai a rendszer többi része elől elrejthetők (pl. Demeter-e[...]

Menedzselhetőség és kockázat

SSADM:

- Kockázat: Hosszú időt igénylő nagy lépésekben halad
- Ellensúly: Aprólékosan kidolgozott termékstruktúra és tevékenységi terv

UML:

- Kockázat: A termékstruktúrát és a tevékenységi tervet az alkalmazási terü[...] megfelelően kell kialakítani
- Ellensúly: Iteratív fejlesztés

Minőségbiztosítás

SSADM:

- Látszólag erős
- Minden termékre definiálták a minőségi kritériumok
- Szövevényesen összefüggő modellek
- Bonyolult keresztellenőrzések szükségesek

UML:

- A lényegre koncentrálhat

- A minőségi kritériumok definiáltak
- A modellek közötti kapcsolatok egyszerűbbek
- A keresztellenőrzések egyszerűbbek

Fejlesztőeszközök jellemzői

SSADM:

- Tipikus: RDBMS + 4GL
- Elosztott rendszer az RDBMS vagy 4GL speciális szolgáltatásaként
- A rendszer logikája „szétkenődik" az alkalmazások között

UML:

- Tipikus: RDBMS vagy ODBMS és OO nyelv
- Elosztott rendszer a DBMS szolgáltatásain túl természetesen illeszkedő szabványokkal is (CORBA)
- A rendszer logikája egységbezárva az üzleti objektumokban

A fejlesztési munka hatékonysága

SSADM:

- 4GL a primitív megoldások és a „buta prototípusok" fejlesztése látványosan gyors
- Az alkalmazási logika beépítése nehézkes
- Speciális megoldások különösen sok munkával vagy egyáltalán nem készíthetők

UML:

- OO nyelv: nincs látványosan gyors munkafázis
- Az alkalmazási logika beépítése egyszerű
- A speciális megoldások az alapnyelven készíthetők

Objektum-orientált módszertanok kapcsolódása

Frigó József, Hontvári József (TRIAD Kft.)

Kivonat

Az általánosan használt objektum-orientált módszertanok nagy projektek esetén önmag
már nem alkalmasak a teljes életciklus megfelelő minőségű lefedésére. Az objektum-ori
elemzés és tervezés elveinek általános és univerzális jellege viszont lehetővé teszi, hogy a
szertanokat ötvözve a teljes életciklusban megfelelő minőségű és részletességű mode
dolgozhassunk.

A cikkben esettanulmányként a Business Process Reengineering keretében a ren
elemzésben kiemelkedő Objectory, az információs rendszerek területén és az objektum
zésben erős OMT, és a rendszertervezésben és az implementációs megoldások kidolgozá
hasznos Booch módszertanok együttes használatát, integrációját és CASE támogatot
valamint a hagyományos módszerekkel való összeegyeztethetőségét vizsgáljuk, különös
tettel arra, hogy a módszertanok szerzői jelenleg is a módszertanok összehangolásán d
nak. Az integráció fontossága és lehetőségei az alkalmazási területtől és az impleme
eszköztől függően is értékelhetők.

Objektum-orientált módszertanok

Más módszertanokhoz hasonlóan az objektum-orientált módszertanok is - bár tagadják
cifikusak az alkalmazási területre és az életciklusnak csak egy részét fedik le. Azt is be k
lanunk, hogy a strukturált módszertanok között olyat is találunk, melynek kidolgozo
sokkal részletesebb, mint az objektum-orientáltaké, és ez az SSADM. Van viszont az
tum-orientált módszertanoknak is egy igen előnyös vonásuk: a modellezési elveik minde
ben az objektumok, osztályok, attribútumok, metódusok és az alrendszerek köré épülnek
viszonylag könnyű összekapcsolni a különböző módszertanok által használt modelleket.
elvet Rumbaugh és Booch is megfogalmazták az egységes módszertanuk kidolgozásána
idejében, ugyanis az első lépés a módszertanaik egyesítésében a közös metamodell (
modellt leíró modell) kidolgozása volt.

A modellezési elvek hasonlósága és a modellek egyszerű illeszthetősége azt ere
nyezi, hogy a OO módszertanok halmazából össze lehet rakni a konkrét rendszer kifejle
hez leginkább alkalmas egyedi módszert. Egy ilyen lehetőség például az információs re
rek fejlesztésére a következő:

A megvalósíthatósági tanulmány készítésének fázisában, egészen a rendszer hatá
és szolgáltatási szintjeinek kialakításáig a *Business Process Reengineering* módszereit h
hatjuk, a konkrét modellezési technikákat az OO módszertanokból választva, ilyenek pé
használati esetek (use cases), az eseménymodellezés (event trace) és az üzleti szabályok
foglalása.

Következő szinten lépnek be a rendszert különböző szempontból ábrázoló mod
statikus, dinamikus, funkcionális (adatáramlási), lásd OMT, Booch.

A meglévő rendszerekhez történő illeszkedés, egy cég közösen használt adatmo
nek rögzítésére, illetve a relációs adatbázis-kezelőkkeli kapcsolat miatt az OO techniká
lehet egészíteni a strukturált gyakorlatból átvett adatmodellezéssel. Az újabb adatmode
technikák az OO fogalmakhoz hasonlóakat használnak, így nem jelent gondot az együtt
kalmazásuk.

lit hoz a jövő?

. két legsikeresebb OO tervezési módszertan szerzője, Grady Booch és James Rumbaugh '94 sze óta együttműködik egy egységes és a szerzők meghatározó szerepe miatt várhatóan szabányértékű közös módszertan kidolgozásában. Az integráció eleinte nehezen haladt, mindkét erző ragaszkodott az általa kidolgozott jelölésrendszerhez, viszont meg tudtak egyezni aban, hogy mi legyen a modelljeik tényleges jelentése. Először tehát egy közös metamodellt olgoztak ki, amely a rendszerelemzés és tervezés során készülő modellek adattartalmát defiiálja. Kijelentették, hogy a közös metamodell alapján minden fejlesztő nyugodtan használhatja vább a már megszokott OMT vagy Booch jelölésrendszert.

Ezután jött a meglepetés: előzetes értékelésre közzétették a „Unified Method" gyesített módszer) 0.8-as változatszámmal jelzett dokumentációját, melyben nagyon jó érkkel összegyűjtötték mindkét módszertanból a legkifejezőbb jelölésmódot. Úgy tűnik, hogy a ódszertanok közös részében az OMT jelölésrendszere dominál, kisebb változásokkal, de a ódszer kiegészült számos részletes tervezési modellel is, melyek korábban csak a Booch ódszerben szerepeltek. A jelölésrendszer használati módját, a modellek készítési sorrendjét író módszertanban viszont továbbra sem egységesek, ami talán a kiinduló módszertanok eltéfókuszára vezethető vissza: az OMT határozottan adatközpontú, adatmodell jellegű, ezért a ódszertani leírás is az információs rendszerek készítéséhez szükséges elemeket hangsúlyozza. Booch módszer inkább technikai jellegű, a beágyazott rendszerek tervezésének részleteire ncentrál.

A módszertan dokumentációjából legutóbb a 0.9-es változatot tették közzé, amely tudonképpen csak egy kiegészítés az előzőhöz. Fontos változás, hogy a csapathoz csatlakozott harmadik nagy név", Ivar Jacobson is. A cím is megváltozott: a visszajelzéseknek és talán új határidőnek köszönhetően belátták, hogy most még korai volna egységes metódus kilgozása, az új cím tehát „The Unified Modelling Language for Object-Oriented velopment" (Az objektum-orientált fejlesztés egységes modellezési nyelve". Az új táridőt" pedig az OMG (Object Management Group) felkérése adja, amely egy szabványjalat készítésére szól.

ASE támogatottság

A módszertanok keveréséhez segítséget kell kapnunk a CASE eszköztől is. Elvileg den metacase típusú eszköz, amely ismeri a használni kívánt módszertanokat, lehetőséget atna arra, hogy keverjük a módszertanokat, de ez gyakran csak annyit jelent, hogy felváltvaználhatjuk a módszertanokat, tehát egy projektnél eldönthetjük (és el is kell döntenünk), y melyiket akarjuk használni. Ahhoz, hogy ténylegesen keverjük a módszertanokat, a SE eszköznek speciális szemantikus transzformációkat és kapcsolatokat kell biztosítania.

Az UML-re vonatkozó előzetes hírekre építve a Software through Pictures CASE esz(StP) jelenleg a „régi" Booch, OMT és Objectory módszertanok együttes használatát tátja közös metamodell és tárház (repository) biztosításával. Az StP a fenti módszertanokat észíti egyéb technikákkal is, például a követelménykatalógust követelménytáblázat formáismeri és lehetőséget nyújt a követelmények és az elemzési/tervezési modellek elemeinek kapcsolására, ezáltal a követelmények sorsa és a terv magyarázata is visszakereshető.

Az előadás során ismertetjük a Software through Pictures CASE eszközhöz javasolt nológiát, amely a következő módszertanokkal és technikákkal, valamint ezek összekapcsol foglalkozik: BPR, OMT, Booch, Jacobson, Követelmény katalógus, Információs mozés.

Objektum-orientált tervezés alkalmazása rugalmas gyártórendszerekné

Sándorné Kmecs Ildikó

MTA SzTAKI

1111 Budapest, Kende u. 13-17

Tel: 1 665 644 / 268, Fax: 1 667 503

e-mail: kmecs@lutra.sztaki.hu

1 Bevezetés

Napjainkban az objektum-orientált programozás mellett az objektum-orientált ter módszerek alkalmazása is egyre jobban elterjedt. Ez elsősorban annak köszönhető, hog objektum-orientált tervezési módszer alkalmazásával készült szoftver átlátható, kö módosítható, a fejlesztés életciklusa rugalmas, a kész szoftver könnyen alkalma megváltozott körülmények között is.

Az objektum-orientált tervezésnek számos módszere-módszertana létezik. Ezek kö cikkben ismertetendő feladatok megoldására a Booch [Booch 1991] által kifejl objektum-orientált tervezési módszert (Object Oriented Design - Objektum-orientált Ter OOD) használtuk, kiegészítve Jacobson [Jacobson 1992] „use-case" tervezési elvével (Oriented Software Engineering - Objektum-orientált Szoftver Fejlesztés - OOSE).

Feladatunk alapvetően műszaki jellegű: a számítógéppel segített (irányított), automatizált gyártás rendszereinek minél egyszerűbb, hatékonyabb, megbízhatóbb model és szimulációja a tervezés elősegítése érdekében.

A számítógéppel irányított termelés egyik legfontosabb gyártóeleme a rug gyártórendszer (Flexible Manufacturing System - FMS), amely különböző funkciójú és gyártórendszer-modulokból áll, és a gyártás számítógépes numerikus vezérlésen (Co Numerical Control - CNC) alapul.

A rugalmas gyártórendszerek objektum-orientált modellje ezeknek a rendsze csaknem természetes leírása, hiszen az egyes gyártórendszer-modulok egy-egy jól körülh osztályon belül főleg paramétereikben térnek el egymástól.

A gyártórendszerek szimulációja nagyon fontos segédeszköz egyrészt tényleges, m rendszerek vizsgálataira, másrészt az adott gyártási feladatok végrehajtására ter létesítendő rendszerek vizsgálatára, a legjobb tervvariáns kiválasztására.

2 Objektum-orientált tervezési módszerek

2.1 Az Objektum-orientált Tervezés (OOD)

Az OOD olyan módszer, amely a rendszer objektum-orientált szétbontásához vezet. Az OD definiálja komplex szoftver-rendszerek létrehozásának jelölésrendszerét és folyamatát, és gikai illetve fizikai modellek gazdag választékát kínálja, amelyek segítségével a fejlesztés alatt lló rendszert különböző megvilágításokban átgondolhatjuk.

Az alapabsztrakciót azok az osztályok és objektumok jelentik, amelyek a problémakör zótárát alkotják és amelyeket az analízis, valamint a tervezés fázisában definiálunk. Az 1. brán az OOD szempontjából fontosnak tartott, a tervezésnél használatos modellek láthatók. zek a modellek kifejezésteljesek, azaz lehetővé teszik, hogy a fejlesztő megragadja az összes nyeges tervezési döntést, és eléggé teljesek ahhoz, hogy rendszertervként szolgáljanak szinte rmely objektum-alapú, vagy objektum-orientált nyelvben történő megvalósításhoz. A szletes jelölésrendszer a rendszer tervezésének dokumentálására szolgál.

A gyakorlatban lényeges elválasztani a különböző tervezési döntéseket. Többek között laszolni kell az OOD alapkérdéseire:

- Mely osztályok létezzenek és ezek milyen kapcsolatban álljanak egymással?
- Milyen mechanizmusok szabályozzák az objektumok együttműködését?
- Hol legyenek az osztályok és objektumok deklarálva?
- A folyamat melyik processzorhoz legyen beosztva, és az adott processzor hogyan ütemezze a folyamatait?

Az első két kérdés a tervezett rendszer absztrakt felépítését, míg a második két kérdés a nkrét fizikai megvalósítást hivatott elősegíteni. A kérdésekre a választ a következő grammokban ábrázoljuk:

- osztály diagramm
- objektum diagramm
- modul diagramm
- folyamat diagramm

Ez a négy diagramm alkotja az OOD jelölésrendszerének alapját. Az első kettő a rendszer kai jellemzése, mivel a tervezés alapabsztrakcióját írják le. A fennmaradó két diagramm a dszer fizikai szerkezetét adja, az implementálás konkrét szoftver és hardver nponenseinek leírását tartalmazza.

- Az osztály diagramm a létező osztályokat és azok kapcsolatait ábrázolja, és a rend osztályszerkezetének egészét, vagy annak egy részét tartalmazza.
- Az objektum diagramm a létező objektumokat és azok kapcsolatait ábrázolja. Elsőso az objektumok közötti mechanizmusok illusztrálására szolgál. Az objektum diagr egy pillanatképet jelent az időben az egyébként ideiglenes objektum események közö
- A modul diagramm az osztályok és objektumok modulokba rendezését tartalmazza.
- A folyamat diagramm a folyamatok processzorokhoz rendelését írja le.

Ez a négy diagramm statikus. Mivel az események dinamikusan következnek be, két to diagramm is része az OOD-nek, amelyeken keresztül a tervezés dinamikus jellegét fejezzük

Ezek:

- állapot diagramm
- időzítési diagramm

- Az állapot diagramm egy konkrét osztály egyedeinek állapotát ábrázolja, eseményeket, amelyek bekövetkezése állapotváltozást okoz, és az akciókat, amely állapotváltozásból erednek.
- Az időzítési diagrammot az objektum diagrammokhoz kapcsolva használjuk, ho különböző objektumok közötti dinamikus együttműködést, az események id sorrendjét, elküldésüket és feldolgozásukat megmutassa.

A rendszertervezés lépései:

- Az osztályok és objektumok meghatározása az absztrakció adott fokán.
- Ezen osztályok és objektumok szemantikájának meghatározása.
- Az osztályok és objektumok közötti kapcsolatok meghatározása.
- Ezen osztályok és objektumok implementálása.

Ezek a lépések spirális folyamatot alkotnak. A folyamat azon osztályok és objekt meghatározásával kezdődik, melyek a problémakör szótárát alkotják, és akkor fejeződ amikor úgy találjuk, hogy már nincs újabb alapabsztrakció, vagy mechanizmus.

2 Az Objektum-orientált Szoftver Fejlesztés (OOSE)

Az objektum orientált szoftver fejlesztés alternatív módszertanát kínálja a Jacobson által fejlesztett OOSE. Az OOSE módszer egyedi vonása a "use-case" elv bevezetésében rejlik. Az OSE-ben a szoftverfejlesztés életciklusát 5 modellben lehet végigkövetni, ezek a vetelmény-, analízis-, tervezés-, megvalósítás-, valamint tesztelés-modell. Elsőként a use-se elv alapján létrehozzuk a use-case modellt, majd az összes többi modellt ennek a lhasználásával készítjük el.

A use-case modell kezdeményezőkből (actor) és eseménysorokból (use-case) áll. A zdeményező és a felhasználó közti különbség abban nyilvánul meg, hogy a felhasználó az tuális személy, aki használja a rendszert, míg a kezdeményező bizonyos szerepet jelképez, elyet a felhasználó eljátszhat. A kezdeményező több műveletet válthat ki a rendszerből, ezek ozata jelenti a use-case-t. A use-case-t részletes szöveges leírás adja meg.

Az objektum-orientált tervezési módszerek felhasználása

Ha az OOD módszert kibővítjük az OOSE módszer use-case elvével, akkor a dszertervezés folyamata a use-case modell megalkotásával kezdődik, és ennek a modellnek elhasználásával építjük fel a tervezett rendszer teljes modelljét.

Ezt a módszert alkalmazva jelenleg egy olyan általános FMS modellen dolgozunk, amely a ódi (tervezett, vagy tényleges) FMS-ek egy széles körének szimulációjára alkalmas, ítségével a tervezés folyamán kiépíthető egy ideális, az adott problémára illeszkedő FMS. dellünk tartalmaz majd egy általános gyártó elem könyvtárat, és az abban levő gyártó elem lonok segítségével interaktív módon lehet egyedi FMS-eket tervezni és konfigurálni. A rtórendszer-modulok az FMS-ek építőkövei. Közéjük tartozik az összes megmunkálógép, a önböző kiszolgáló robotok, a szállítóeszközök, stb. Ezek mindegyikéhez létrehozunk egy lános osztályt, és ezek egyedei lesznek a konkrét gyártórendszer-modulok. Az FMS lános leírását az osztály hierarchia alkotja, melynek felső részén elhelyezkedő osztályok a rtórendszer-modulok absztrakt leírását adják, az alsó részén elhelyezkedő osztályok a krét gyártórendszer-modulokat jelképezik.

A tervezett rendszer használata közben a felhasználónak lehetősége van a már létező FMS lellekkel dolgozni, vagy újabb modellt hozhat létre a könyvtárban elhelyezkedő sablon-ályok klónozható egyedeinek segítségével és ezek megfelelő adatainak kitöltésével. etőség van továbbá arra is, hogy a felhasználó, módosítva az osztályhierarchiát, a felsőbb en elhelyezkedő osztályok felhasználásával újabb sablon-osztályokat, majd ezekből újabb tórendszer-modulokat hozzon létre.

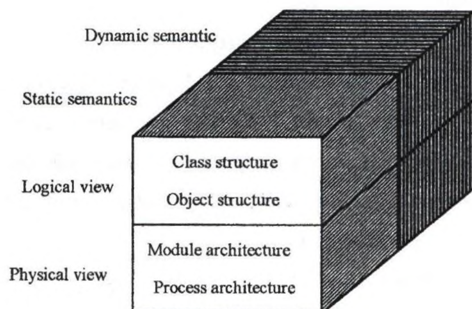
a munkadarab rögzítése még előtte kell megtörténjen. Ez a leírás feltételezi, hogy a pal munkagéphez tartozik, valamint egyszerűsítettük az esetet azáltal, hogy a meghibáso lehetőségeket nem vettük figyelembe.

4 Konklúzió

A rugalmas gyártórendszer a számítógéppel irányított termelés egyik legfontosabb el Egy gyártórendszer helyes összeállítása és adott kritériumok szerinti optimális működt nem egyszerű feladat. Maguk a gyártóelem-modulok drágák, és a gyártórendszerek műk költsége sem alacsony. A gyártórendszer meghibásodása termeléskieséshez vezet Mindezek miatt nagyon fontos a modellezés és a megbízható szimuláció szerepe a vizsgál során anélkül, hogy a tényleges rendszert üzemeltetni kellene. Egyrészt a meglévő, mű rendszereket vizsgáljuk megváltozott feladatok esetén, másrészt a tervezett, új rendszerek azok működését modellezzük és szimuláljuk, az optimális összeállítást keresve.

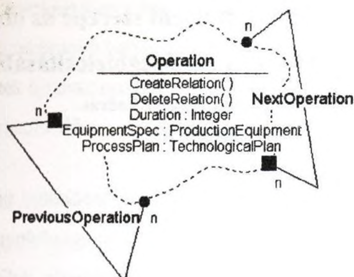
A szimulációs modell elkészítéséhez az objektum-orientált tervezési módszerek segítséget nyújtanak az objektum-orientált tervezés előnyeinél fogva.

5 Ábrák



Ábra 1

Az OOD modelljei



Ábra 2

A megmunkáló művelet ábrázolása

Ábra 3

A megmunkáló-központ ábrázolása

5 *Irodalom*

[Booch 1991] G. Booch: Object-Oriented Design with Application, Benjamin/Cummings, 1991.

[Jacobson 1992] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: Object-Oriented Software Engineering, Addison-Wesley, 1992.

A szoftver konfiguráció menedzsment szerepe az objektum orientált szoftverfejlesztés minőségbiztosításában

Nick János, Kovács András
HiCare Kft.

Bevezetés

Napjaink robbanásszerűen fejlődő világában az élet minden területén az érvényesüléshez, a sikerhez, sokszor a fennmaradáshoz elengedhetetlen a gyorsan változó körülmények változásaira való gyors reagálás képessége. Különösen igaz ez a számítástechnikával, szoftverfejlesztéssel foglalkozó vállalkozásokra.

Az üzleti élet egyre nagyobb mértékben igényli a napi működéséhez a számítástechnikai támogatást. Az alkalmazott szoftverek nagy szerepet játszanak a piaci igényekre való gyors megfelelő reagálásban, a termelékenység növelésében, a költségek csökkentésében. Ennek megfelelően a felhasználók jó minőségű és megbízható rendszereket, magas színvonalú támogatást, azonnali hibajavítást és a felmerülő új igények gyors megvalósítását igénylik a fejlesztőktől.

A fejlesztési projektekkel szemben általános elvárás, hogy a projekt a vállalt határidőre, az előre definiált költségkereteken belül jó minőségű, a megrendelő igényeinek megfelelő te előállításásval befejeződjön. Ennek megfelelően a minőségbiztosítás is egyre nagyobb szerepet kap a szoftverfejlesztések esetén is.

Napjaink fejlesztéseinek jellemzői

A ma készülő alkalmazásokra jellemző a viszonylag rövid fejlesztési életciklus. A gyorsan változó igények miatt a hagyományos vízesés modell már nem megfelelő sokkal inkább a rendszert kisebb részletekben megvalósító, inkrementális fejlesztési életciklus modell a követendő. Ennek megfelelően az alkalmazásból több, akár párhuzamosan is továbbélő v keletkezik. A vállalkozások méretének igényeinek változatossága miatt skálázható, és ennek következményeként sokszor több platformos, igen összetett alkalmazásokat kell fejleszteni

A bevezetőben felsoroltak következtében a bonyolult, nagy alkalmazások esetén nagyszámú fejlesztőt kell alkalmaznunk, akár több helyszínen is folyó fejlesztés elvégzésére. Továbbá megnő a már elkészült elemek újrahasznosításának szükségessége, ami szintén a gyorsan változó igények kielégítését segíti elő.

A fejlesztés sajátosságai miatt különböző problémák léphetnek fel az alkalmazások előállítása során. Az alkalmazások bonyolultsága és összetettsége miatt egy adott verzióhoz szükséges elemek (osztályok, forrás fájlok, alrendszerek) nagy száma és különböző változatai miatt a programok előállítása a rendelkezésre álló időhöz képest sokáig tart. Igen nagy annak a veszélye is, hogy nem a megfelelő forrásokat, osztályokat, modulokat építjük össze. Ugyanazon komponensnek párhuzamos (több platformos) fejlesztését a komponensek nagy száma miatt nehéz kézben tartani. Előfordulhat az is, hogy az egyik verzióban kijavított hiba egy újabb verzióban ismételten megjelenik vagy egy igényelt és elvégzett módosítás mégsem kerül bele az új verzióba. Még gyakoribb, hogy egy program adott verzióját nem tudjuk előállítani, mert az akkori állapotot nem tároltuk el.

A tapasztalatok azt mutatják, hogy az alkalmazások fejlesztéséhez hagyományosan felhasznált - CASE, GUI fejlesztő, Teszt, Projekttervező - eszközökön túlmenően szükség van a keletkezett termékek - terv dokumentumok, forrás fájlok, kódok - nyilvántartásának támogatására. Ezt a feladatot látják el a szoftver konfiguráció menedzsment eszközök.

zoftver konfiguráció menedzsment

konfiguráció menedzsmentnek a fejlesztési folyamatban kiemelkedően fontos szerepe van. árolnia kell a fejlesztési eredmények különböző verzióit és lehetőséget ad egy-egy elem árhuzamos fejlesztésére(verzió kezelés). Biztosítja a rendszer moduljainak előállításához a egfelelő forrás fájlokat (konfiguráció kezelés). Támogatja a fejlesztő csoportokat akár több lephelyes fejlesztés esetén is (elosztott fejlesztés támogatása). Lehetőséget ad arra, hogy az kalmazott életciklusnak megfelelő fejlesztési lépéseket követni tudjuk (folyamat vezérlés). sszegyűjti az adott alkalmazás különböző verzióihoz tartozó hibajelzéseket, észrevételeket, ódosítási és változtatási igényeket, valamint összerendeli ezeket a végrehajtott ltoztatásokkal (változtatás kezelés).

következőkben a konfiguráció menedzsment jellemzőit ismertetjük részletesebben.

Verzió kezelés

A verzió kezelés a konfiguráció menedzsment egyik alapeleme. A verzió kezelés biztosítja a idők folyamán létrejövő komponensek (fájlok) különböző változatainak tárolását és visszakereshetőségét. A párhuzamos fejlesztések miatt az egyszerű tároláson túlmenően gondoskodni kell az egyes komponensek esetén különböző verzió ágak (branching) kezeléséről és a verziók beszédes elnevezési lehetőségéről. Mindezen lehetőségeknek azonban a fejlesztők számára adott esetben átlátszónak is kell lenniük, hogy ne gátoljak vagy nehezít a munkát. Eszközt kell kapnunk továbbá egy forrás fájl különböző verzióinak minél egyszerűbb összemásolásához, hiszen a termék fejlesztése és a már telepített verziók hibajavítása egyidejűleg történhet.

A fejlesztés során keletkező forrás fájlokon túlmenően keletkező bináris állományokat, terv egyéb dokumentációkat, projekt terveket, táblázatokat, stb. is verzió kezelés alá kell vonnu

A változtatások későbbi követhetősége szempontjából fontos, hogy az elvégzett módosításokról megjegyzéseket fűzhessünk az egyes verziókhoz.

Az egyes komponensek verzióinak nyilvántartása az ismertetett összes problémát még nem oldja meg. A konfiguráció menedzsmentnek további szolgáltatásokat kell nyújtania a haték fejlesztési támogatáshoz

Konfiguráció kezelés

Egy termék konfigurációja mindazon komponensek összessége, amelyek a termék elõállításához szükségesek. A komponensek nem csak a program elõállításához közvetlenü szükséges forrás fájlok, hanem a fejlesztés egyéb résztermékei (dokumentációk, tervek) is.

Egy termék konfigurációján belül lehetõség van a komponensek tetszõleges csoportosítás a komponensek kívánt verzióinak használatára. Az esetek többségében egy komponens legutolsó verzióját használjuk, de természetesen bármely verziót használatát biztosítania ke konfiguráció kezelésnek, lehetõleg automatikusan, elõre definiált szabályok alapján. A megjelölt termék konfigurációkat (baseline) bármikor elõ kell tudnunk állítani. Biztosítani továbbá, hogy a konfigurációnak különbözõ jellemzõit (operációs rendszer, fordító, verzió stb.) rögzíteni lehessen.

Mivel nagy rendszerek fejlesztése esetén sok fejlesztõ vagy több fejlesztõ csoport dolgozik egy projekten biztosítani kell az egyes fejlesztõi munkaterületek izolációját. Ez azt jelenti,

egy komponens csak akkor elérhetõ mások számára, ha a fejlesztõ (felelõs) ezt jóváhagyta és a változtatásokat a komponens felhasználója tudomásul vette.

Az egyes komponensek többszöri vagy több alkalmazásban való felhasználása azt igényli, hogy egy ilyen komponens módosítása elõtt fel tudjuk mérni, hogy a változtatás milyen más komponenseket vagy termékeket érint (impact analysis).

A nagy rendszerek bonyolultsága és összetettsége miatt a konfiguráció mendzsmentnek támogatnia kell a teljes rendszer alapkomponensekbõl történõ automatikus újraépítését (fordítás) akár elosztott módon, a hálozatt több számítógépét felhasználva (distributed and remote build).

'áltoztatás kezelés

A gyakorlati tapasztalat azt mutatja, hogy egy termék fejlesztése során igen sok változtatási, módosítási igény merül fel. Bizonyos változtatások lokális hatásúak, mások akár a követelmények vagy a tervek megváltoztatásával is járhatnak. A változtatások ellenõrizetlen, koordinálatlan végrehajtása kaotikus viszonyokat teremthet.

A változtatás kezelés a problémák vagy módosítási igények tárolását, szelektálását, csoportosítását biztosítja. Fontos , hogy a termékekrõl összegyûjtött információkat (hibajelzések, további fejlesztési ötletek, észrevételek) tároljuk és a végrehajtott modosításokkal összerendeljük és idõszakosan értékeljük.

A problémákhoz, módosítási igényekhez feladatokat rendelhetünk, amelyeket személyre szabottan a fejlesztõkhöz rendelhetünk. A fejlesztõ a megadott komponensek módosításához hozzárendeli a feladatot ami alapján a munkáját végezte. Így a késõbbiekben is képet kaphatunk arról, hogy egy-egy módosítási igény mekkora munkával járt.

olyamat vezérlés

A minőségbiztosítás szempontjából a konfiguráció menedzsment igen fontos eleme. A folyamat vezérlés biztosítja azon fejlesztési fázisok definiálását, amelyeken a komponenseknek végig kell haladniuk. Definiálhatjuk azokat a feltételeket is, amelyek teljesülése esetén a komponensek egyik fázisból a másikba átléphetnek. Meghatározhatjuk továbbá, hogy a változtatásokat és a fázisátmeneteket mely fejlesztők hajthatják végre és így elkerülhetjük a véletlenszerű

módosításokat. A komponens a termékbe csak abban az esetben építhető be, ha az éltcikl utolsó fázisát is elérte. Az 1. ábrán láthatunk egy példát a forrás fájlok fázisaira.

1. ábra Komponensek életciklusa

A különböző típusú komponenseknek más és más lehet a fejlesztési életciklusa. A 2. ábrá láthatjuk a telepítendő termékekre vonatkozó fázis diagramot.

2. ábra Termék életciklus

A komponenseken túlmenően a változtatási igényeket, problémákat szintén különböző szinteken értékelhetjük és dönthetünk a megvalósításáról vagy elvetéséről. A 3. ábra egy tipikus probléma életciklust mutat.

3. ábra Probléma életciklus

A 4. ábra a fejlesztőkhöz rendelt feladatok végrehajtásának különböző állomásait mutatja.

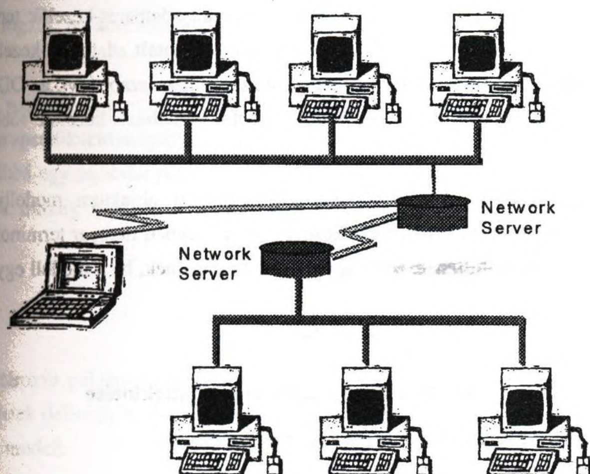
4. ábra Feladat életciklus

A különböző életciklus definíciókon túlmenően biztosítani kell azt is, hogy megfelelő jogosultság hierarchia épüljön fel az állapot átmenetek ellenőrzött végrehajtására.

A fejlesztők hatékony együttműködése szempontjából igen fontos, hogy a komponensek, problémák és feladatok különböző fázisátmeneteiről jelzést, üzenetet kapjanak. Így a szám fontos komponensek megfelelő verzióit fel tudják használni.

Elosztott fejlesztés támogatása

Napjainkban mind gyakoribb és a jövőben egyre terjedő tendencia, hogy a fejlesztések a megfelelő erőforrás kihasználás miatt földrajzilag is elkülönülő helyszineken történik. Ez nem csak egy cég különböző telephelyeit, hanem több cég vagy otthon dolgozók fejlesztők együttműködését jelenti (5. ábra). A hatékony konfiguráció menedzsmentnek támogatnia kell az ilyen elosztott fejlesztést is. Eszközöket kell biztosítani arra, hogy a különböző helyszíneken történő változások minden érintet félnél megjelenjenek on-line vagy valamilyen adathordozó eszköz segítségével. Az otthon dolgozók részére biztosítani kell, hogy a megfelelő komponensek frissítése egyszerűen és gyorsan megtörténjen.



5. ábra Elosztott fejlesztési környezet

Összefoglalás

Napjaink szoftverfejlesztéseinél a minőségbiztosítás, és ezen belül a szoftver konfiguráció menedzsment egyre nagyobb szerepet játszik abban, hogy a termékek határidőre, megfelelő minőségben elkészüljenek.

A konfiguráció menedzsment biztosítja a szoftverfejlesztés teljes életciklusa során keletkezett résztermékek (terv dokumentumok, leírások, stb) és az alkalmazás komponensek (források, modulok, alrendszerek) különböző verzióinak tárolását és a komponensek fejlesztésének ellenőrzött végrehajtását. Hatékonyan támogatja sok fejlesztő és több fejlesztő csoport együttes munkáját párhuzamosan fejlesztett alkalmazás verziók esetén is.

AZ ODMG-93 SZABVÁNY

Juhász István
KLTE Információ Technológia Tanszék
pici@math.klte.hu

Bevezetés

Az elmúlt évtizedben számos kísérlet történt olyan modellek megkonstruálására, am
objektumok szemantikáját írják le. Az objektum orientált adatbázis-kezelők területén a
Database Management Group (ODMG), egy objektum orientált adatbázis-kezelőket fej
forgalmazó cégekből álló konzorcium is megalkotott egy ipari szabványt, az ODMG-93
szabványt a jelenleg piacon levő objektum orientált adatbázis-kezelők többsége t
valamilyen szinten.

A következőkben az ODMG-93 szabvány egyszerűsített objektum modelljét isme
részletek és a kezelő nyelvek teljes mellőzésével. A szövegben magyar terminológiát h
A típusneveket is magyarítottam, ezek nagybetűvel szerepelnek, ha a modell egy alkotó
van szó.

Az ODMG objektum modell áttekintése

Az objektum modell röviden a következőképpen jellemezhető:

- Az alapvető modellezési eszköz az *objektum*.
- Az objektumok *típusokhoz* tartoznak. Minden objektum a saját típusának viselkedés
és állapotaival rendelkezik.
- Az objektumok viselkedését *műveletek* egy adott csoportjával definiáljuk. Ezek a m
az adott típus bármely objektumán végrehajthatók.
- Az objektumok állapotát *tulajdonságok* segítségével adjuk meg. Ezek a tulaj
lehetnek *attribútumok*, amelyek egy-egy objektumhoz kötődnek és *kapcsolatok*, am
vagy több objektum közötti viszonyra vonatkoznak.

Egy típusnak egy *interfész* része és egy vagy több *implementációja* létezik. Az interfész
azokat a jellemzőket, amely alapján az adott típus példányait használni tudjuk. Az imple

finiálja egyrészt az *adatok struktúráját*, amelyen az adott típus példányainak fizikai prezentációja alapul, másrészt a *módszereket*, melyek az adott struktúrán értelmezettek, ezáltal ztosítják a kívülről látható viselkedésmódot és az állapotok kezelését.

típusok maguk is objektumok, ezáltal rendelkeznek tulajdonságokkal. Egy típus interfész finíciója ezekhez a típus tulajdonságokhoz *értékeket* specifikál.

y adott típus példányainak műveleteit *műveleti specifikáció* írja le. Minden specifikáció gadja a művelet nevét, az argumentumok nevét és típusát, a visszatérési érték nevét és típusát *kivétel* neveket, amely kivételek a művelet végrehajtása közben bekövetkezhetnek.

y adott típus példányainak attribútumait *attribútum specifikációk* adják meg. Minden ecifikáció tartalmazza az attribútum nevét és típusát, mely típus az attribútumok ktartományát rögzíti. Az attribútumok értékként *literálokat* (számok, sztringek, stb.) vehetnek

kapcsolatokat, melyekben egy adott típus objektumai részt vesznek, *kapcsolat specifikáció* iniálja. Minden specifikáció megadja azon objektum(ok) típusát, melyekhez az adott objektum csolódik, továbbá egy *bejárási függvény* nevét, amely a kapcsolódó objektum(ok) elérésére . A kapcsolat mindig bináris. A kapcsolat számossága egy-egy, egy-több és több-több ránt lehet.

Típusok és példányok

típus meghatározza példányainak állapotát és viselkedését. Az állapotot tulajdonságok, a lkedést műveletek definiálják. Az állapotot és a viselkedést összefoglaló néven *jellemzőknek* az objektum modell.

usok (mint minden OO rendszerben) egy hierarchiába szerveződnek. A hierarchia megadja a rtípus-altípus összefüggéseket.

bjektum modell fő beépített típusainak hierarchiája a következő:

- Egyedi objektum
  - Objektum
  - Literál
- Jellemző
  - Művelet
  - Tulajdonság
    - Attribútum
    - Kapcsolat

A beépített típusok fa hierarchiát alkotnak. Az ODMG-93 a felhasználó által definiált
megengedi a többszörös öröklődést, ezért azoknak a hierarchiája egy aciklikus gráfot
többszörös öröklődés esetén esetleg előforduló névütközések feloldására az örökölt
átnevezhetősége szolgál.

Az objektum modell ismeri az *absztrakt típus* fogalmát. Ezek azok a típusok,
példányosíthatók, vagyis nincs implementációjuk. Az absztrakt típusok csak
definiálnak, amelyeket majd (a már példányosítható) altípusok örökölnek.

Az objektum modellben a beépített típusok közül csak az objektum típusból szárm
felhasználó altípusokat.

Egy adott típus összes példányának együttesét a típus *kiterjedésének* hívjuk. Termész
egy A típus altípusa a B típusnak, akkor A kiterjedése részhalmaza B kiterjedésének.

Egy típusnak egy vagy több *implementációja* lehet. A különböző implementációk saj
névvel rendelkeznek.

Egy implementáció egy *reprezentációt* és *módszereket* tartalmaz. A reprezentáció nem
adatstruktúrák együttese, a módszerek pedig alprogram törzsek. A specifikációban
összes művelethez tartozik egy-egy módszer.

A típus interfész specifikáció és az implementációk valamelyike együtt egy *osztályt* alk

Objektumok

Az objektum típusok hierarchiájának tetején az Egyedi objektum típus áll. Az objekt
osztályozása két ortogonális szempont szerint történik: (1) változók és állandók,
atomiak és strukturáltak. Ennek megfelelően a részletesebb hierarchia:

- Egyedi objektum
  - Objektum
    - Atomi objektum
    - Strukturált objektum
  - Literál
    - Atomi literál
    - Strukturált literál

Minden egyedi objektum *azonossággal* bír. A literálok azonosságának reprezentálásár
az értéküket kódoló bitsorozat szolgál. Az objektumok viszont egyedi, saját
azonosítóval (OID) rendelkeznek. Az objektum azonosító szerkezetét a modell nem de
reprezentációs probléma.

Objektum típus példányai változó objektumok, ez azt jelenti, hogy megváltozhat bármelyik ibútumuk értéke, illetve a kapcsolat, amelyben részt vesznek. De az objektum azonosító ezen tozásokra nézve invariáns.

objektum modellben az objektum azonosító egyediségét adatbázis szinten kell biztosítani.

kényelmesebb használat miatt az objektumok nevesíthetők. Egy objektum több névvel is delkezhet. Egy perzisztens objektum nevét (neveit) futási időben lehet definiálni, mészetesen egy név hatáskörén belül csak egyetlen objektumot nevezhet meg.

y objektum élettartama független a típusától. Az objektum modell háromféle élettartamot logat. Azon objektumok élettartama, melyeket egy alprogram fejében deklarálunk, az rogram működésének idejére terjed ki. Definiálható olyan objektum, amelynek élettartama a gram (processz, taszk) működési ideje. Végül a perzisztens objektumok adatbázis elemekhez hetően élnek (lap, klaszter, stb.).

Strukturált objektum típusnak két altípusa van: a Struktúra és a Gyűjtemény.

truktúrák fix számú saját névvel rendelkező komponensből állnak, amelyek mindegyike egy ektumot vagy literált tartalmaz. A komponensek további, különböző típusú elemből állnak. Az nekre a komponensek nevével történő minősítéssel hivatkozhatunk.

yűjteménynek ezzel szemben tetszőleges számú, azonos típusú elemből állnak és nincsenek nponenseik. A gyűjtemények bővítése vagy az elejüktől/végüktől számított abszolút pozíción, y egy kurzor által mutatott elem előtt/után történhet. Az elemek elérése abszolút pozíciójuk adásával, kurzor segítségével, vagy az elem valamely attribútuma értékének megadásával ténhet.

gyűjtemények lehetnek rendezetlenek és rendezettek. A rendezettséget adhatja az az orrend, ahogy az elemek a gyűjteménybe bekerültek vagy pedig a gyűjtemény objektumainak mely értéke szerint lehet rendezni.

trukturált objektum altípusai:

Gyűjtemény
- Halmaz
- Multihalmaz
- Lista
- Tömb

Struktúra

rukturált objektum és altípusai ortogonálisak, vagyis tetszés szerint kombinálhatók (létezik túrák halmaza, halmazok struktúrája, listát tartalmazó tömbök gyűjteménye, stb.).

Literálok

A literálok példányai állandók.

Az atomi literál típusok implicit módon eleve léteznek. Az atomi literál típusok példány azonossággal rendelkeznek, de nincs OID-jük. A modell a következő atomi literál a támogatja:

- Egész
- Valós
- Logikai
- Karakter

Ezen típusok implementációja természetesen programozási nyelv függő.

A strukturált literál teljesen a strukturált objektumot valósítja meg literál szinten. A literálok egyszer létehozhatók és aztán nem módosíthatók. OID-jük nincs. Altípusai:

- Állandó gyűjtemény
  - Állandó halmaz
  - Állandó multihalmaz
  - Állandó lista
  - Állandó tömb
  - Felsorolás
- Állandó struktúra
  - Dátum
  - Idő
  - Időbélyeg
  - Intervallum

A legutolsó négy típus értelmezése megfelel az ANSI SQL specifikációnak. A Felso típus generátor. Minden felsorolásos deklaráció egy név nélküli típust definiál, amely név szerint felsorolva a deklaráció tartalmazza. Ezek a példányok csak névvel ren tulajdonságaik és műveleteik nincsenek.

A programozó származtathat altípusokat a literál típustól, de a műveleteit nem definiálh

Az állapotok modellezése - tulajdonságok

Mint már korábban említettem a tulajdonságokat az attribútumok és a kapcsolatok al attribútumok egy típushoz kapcsolódnak és értékül literálokat vehetnek fel. A kapcs változó típus között értelmezhetők (az ODMG-93 csak bináris kapcsolatokat értelmez).

z attribútumok nem rendelkeznek OID-vel. Egyediségüket az határozza meg, hogy egy egyedi bjektumra vonatkoznak.

z alapmodellben az attribútumok nem teljesítik az egységesség követelményét, vagyis nem het attribútumokon attribútumokat értelmezni, attribútumok között nincs kapcsolat és az tribútumokhoz nem lehet altípus műveleteket értelmezni.

kapcsolatnak nincs neve és nem rendelkezik OID-vel. Egyediségét az összekapcsolt bjektumok adják meg. A kapcsolat esetén viszont mindkét irányban létezik egy bejárási útvonal egnevezés. Ezeket a neveket a kapcsolatban résztvevő típusok interfész részében kell megadni.

kapcsolatok kezelik a hivatkozási integritást. Ha egy kapcsolatban résztvevő objektum törlődik, kor a kapcsolatra történő hivatkozás egy kivételt vált ki.

z egy-több kapcsolat „több" oldalán értelmezhető rendezettség, a több-több kapcsolatnál pedig indkét oldal rendezhető. A több-több kapcsolat visszavezethető egy-több kapcsolatok lmazára, ezek mindegyike pedig egy-egy kapcsolatok halmazára.

A viselkedés modellezése - műveletek

műveleteket mindig egyetlen típuson értelmezzük, a művelet nevének ezen típus definíción ül kell csak egyedinek lenni. A műveletek tehát *túlterhelhetőek.*

argumentumok és a visszatérési érték egyedi objektum vagy azok halmaza lehet. Bármely veletnek lehet mellékhatása. Egy olyan művelet, amelynek csak mellékhatása van a *nil* ékkel tér vissza.

objektum modell a műveletek szekvenciális végrehajtását tételezi fel, tehát nem követeli meg árhuzamosságot, viszont nem is zárja azt ki.

objektum modell dinamikusan egymásba ágyazható kivételkezelők meglétét támogatja. A ételek maguk is objektumok és hierarchiába szervezhetők. Az őstípus rendelkezik egy olyan velettel, amely kiír egy üzenetet, ha nem kezelt kivétel következett be és félbeszakítja a amatot.

A teljes típus rendszer

objektum modell minden típusa a Típus típus egy példánya. Maga a Típus típus altípusa és ben példánya az Atomi objektum típusnak.

objektum modell szigorúan típusos. Két objektum típusa akkor és csak akkor azonos, ha nazon típus példányaként állnak elő. Implicit objektum konverziót a modell nem támogat.

Az értékadás kompatibilitás követi a típus hierarchia szupertípus/altípus viszony altípusa T-nek, akkor S bármely objektumának értéke átadható T objektumának, fordít nem.

Két atomi literál azonos típusú, ha ugyanahhoz az atomi literál altípushoz (egész tartoznak. A befogadó programnyelvben ezen típusok között lehet implicit konverzió.

A strukturált literálok típusa azonos, ha szerkezetük minden szinten megegyezik és ato típusa azonos. Egy T típushoz tartozó strukturált literál értékül adható az S típusú akkor és csak akkor, ha T és S szerkezete minden szinten megegyezik és komponensének típusa azonos T megfelelő komponensének típusával, vagy altípusa a

Egy adatbázis adott típus halmaztól származó perzisztens objektumokat tárol. Minde rendelkezik egy *sémával*. A séma maga típus definíciókból áll. Minden adatbázis a típus egy példányaként áll elő.

Az adatbázis sémájának típus nevei és a hozzájuk tartozó kiterjedések globálisak az nézve és minden olyan program látja őket, amelyik megnyitotta az adatbázist. A tartalmazhat nevesített objektumokat (az ún. *gyökér objektumok*), amelyekre hivatko felsorolt nevek azok, amelyek segítségével az adatbázis elérhető, feldolgozható.

Tranzakciók

Az objektum modell egy dinamikus, beágyazott tranzakció modellt támogat.

Az zárolás objektum szinten történik. Olvasás előtt egy olvasási zárat, írás előtt eg kell elhelyezni az objektumra. Az objektumot olvasók nem kerülnek konfli olvasókkal, de aki írja az objektumot az kizárólagosan foglalja le.

Irodalom

The Object Database Standard: ODMG-93 (Edited by R.G.G. Cattel). Morgan Publishers, San Francisco, 1994.

Objektumorientált adatbáziskezelő rendszerek

Moskovits Péter

(mosko@ttt-202.ttt.bme.hu)

Budapesti Műszaki Egyetem

Villamosmérnöki és Informatikai Kar

Távközlési és Telematikai Tanszék

A cikk az objektumorientált adatbáziskezelő rendszerek elméleti alapjaival foglalkozik. Néhány példán keresztül bemutatja a relációs adatmodell gyengeségeit. Az objektumorientált adatbáziskezelők egy lehetséges származtatásának összefoglalása után az objektumorientált koncepciók rugalmasságát mutatja be a típuskonstruktorok, az asszociációk, majd a verziókezelés segítségével. A lekérdező nyelvek terén foglalkozik a hagyományos adatbáziskezelő nyelvek objektumorientált továbbfejlesztésével, valamint az objektumorientált nyelvek adatbáziskezelő nyelvként való használatával. Végül szót ejt az objektumorientált adatbázis koncepcióról uralkodó más véleményekről is.

Előzmények

Napjaink legszélesebb körben használt adatbáziskezelői a relációs alapokon álló rendszerek. gtaláljuk őket az élet csaknem minden számítástechnikát alkalmazó területén a pénzvilágtól a közlésig. A relációs adatbáziskezelők (Relational Database Management System - RDBMS) rtói a világ legnagyobb forgalmú szoftvercégei között találhatók. Mégis, miért merült fel az igény, y egy a programozási nyelvek, módszertanok területén jól bevált elméletet próbáljanak meg az tbáziskezelés területeire is átültetni? A kérdés megválaszolásához többféle irányban is dulhatunk.

- Az objektumorientált tervezési, fejlesztési módszerek összes előnye elvárható: alaposan tervezett, végiggondolt struktúrák, osztályszerkezetek kialakításával robosztus, könnyen ábbfejleszhető rendszereket, könnyedén újrafelhasználható részegységeket kapunk, a fejlesztés duktívabb, a létrejövő szoftver minősége javul. Mindezek a tulajdonságok - mint minden nagy ftverprojektben - az adatbáziskezelésben is nagyon fontosak.

- Az objektumorientált szemlélet szakítva az évtizedeken át uralkodó ritmusközpontúsággal, az absztrakciós szint finomításának gyakorlatával az adatokat és a rajtuk zhető műveleteket állította a középpontba. Az ilyen típusú struktúrák eltárolására a relációs ell rugalmatlannak bizonyult, ezért az objektumorientált szemlélet bevezetése új *adatmodell* ségességét vetette fel.

Mivel az elsőként említett jellemzők minden objektumorientált metodológiáva projekttel szemben elvárhatók, cikkünkben csak az utóbbi, adatbázis specifikus ké foglalkozunk.

A relációs adatmodell gyengeségei

Az adatmodell az adatok és a rajtuk végezhető műveletek összessége. Az adatbá fejlődésében legfontosabb adatmodellek a *hierarchikus*, a *hálós* és a *relációs adatmodellek* objektumorientált koncepciók megjelenése az adatmodellek területén elsősorban a relációs konkurenciát, mivel a többi rendszert ma már csak elvétve találjuk meg a piacon.

Már régóta érett az igény egy a relációstól alapjaiban különböző, rugalmasabb, u komplexebb adatmodell kialakítására. Ennek oka, hogy vannak olyan problémák, nehézkesen lehet a relációs adatmodellre leképezni. Erre néhány egyszerű példát mutatunk be

Példa 1.: Hogyan írhatjuk le objektumorientált módon, hogy az autót garázsban tárol

Objektumok: autó, garázs
Műveletek: tárolás

Ezek után ha bármikor az autóra hivatkozunk, annak összes alkatrészét (is) ért(het) az objektumok összetettsége könnyen leírható. Relációs adatmodell használata esetén az au információt szét kell bontani relációkba (táblákba), normalizálásra van szükség; az autó al esetenként akár külön táblákba is szét kell osztanunk: kerekek, gyertyák, motor, stb. E bármikor, amikor az autóra, mint egységre hivatkozunk, a táblákba szétosztott ad adatbáziskezelőnek kell összeválogatnia. Ez nemcsak időigényes, de logikailag összeill szétszabdalására is kényszeríti a rendszer tervezőjét.

Újabb nehézség merül fel, ha menet közben kiderül, hogy az adatstruktúrán válto Objektumorientált esetben ehhez - ha a tervezés kellően alapos volt - csak egy, vagy néhány belső szerkezetét kell átalakítani. Relációs esetben komolyabb változtatásokra lehet szükség.

Példa 2.: A relációs lekérdező nyelvek nem támogatják a rekurzív lekérdezéseket. példának egy rekurzív relációt. Célunk egy - fa analógiájú - egyszerűsített vállalati struktú ahol főnökök és bosztottak vannak. A beosztottaknak legfeljebb egy főnökük van, kapcsolattal állunk szemben. Ez azt jelenti, hogy a vállatnál mindenki dolgozó, és más dol való kapcsolata alapján lehet főnök és/vagy beosztott. A relációs modellben ezt egy olyan írhatjuk le, ahol nyilvántartjuk a dolgozó adatait, majd egy olyan speciális külső kulccsal hiv a főnökére, amelyik ugyanannak a relációnak egy sorára mutat. Tehát főnök és beosztott ug a relációban foglal helyet.

A gondok akkor kezdődnek, amikor meg akarjuk tudni, kik egy adott főnök akárhányadik ntű beosztottjai. A relációs lekérdező nyelvek (SQL) nem támogatják az ilyen típusú - rekurzív - rdések megválaszolását.

Példa 3.: Grafikák, folyamatábrák, CAD tervek eltárolásakor gyakran merül fel az igény szőleges számú töréspontot tartalmazó vonalak eltárolására. Ennek a struktúrának relációban téně eltárolásának egy lehetséges módja:

```
VonalPontok={VonalID, PosX, PosY, Pozicio}
```

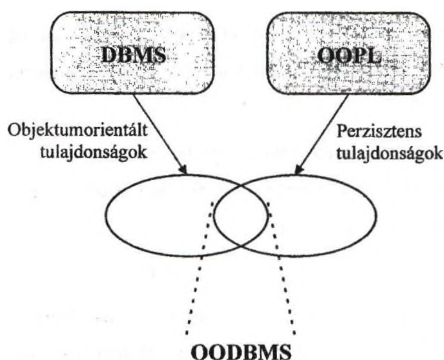
ahol `VonalID` azonosítja, hogy a `PosX` és `PosY` koordinátákkal megadott pontok melyik nalhoz is tartoznak. A `Pozicio` attribútum pedig azt adja meg, hogy a vonalon az adott pont nyadik helyen található; nem mindegy ugyanis, hogy milyen sorrendben kötjük össze a pontokat. ig ha az eltárolás meg is oldható, kényelmesnek, kézenfekvőnek nem nevezhető. Megállapíthatjuk át, hogy a relációs adatmodell nem támogatja a lista jellegű információ tárolását.

Objektumorientált adatbáziskezelő rendszerek

Az objektumorientált adatbáziskezelő rendszerek történelmileg két irányból fejlődtek. Az ik az objektumorientált programozási nyelvek (Object Oriented Programming Languages - PL), a másik az adatbáziskezelők iránya. Ahhoz, hogy adattárolásra képesek legyenek, az ktumorientált programozási nyelveket a perzisztencia tulajdonságával kell felvérteznünk. Ez azt nti, hogy az objektumok ne csak a program futása alatt létezzenek, hanem a futás befejeződése is. A hagyományos adatbáziskezelőket pedig objektumorientált (osztályhierarchiák, öröklődés, morfizmus, egységbezárás, stb.) tulajdonságokkal kell ellátni (1. ábra).

Az objektumorientált adatbáziskezelők területén nem létezik egységes, szabványosított, de akár csak hallgatólagosan elfogadott adatmodell sem. Ennek ellenére természetesen vannak olyan vető jellemzők, melyek minden, vagy legtöbb adatbáziskezelőben megtalálhatók. Cikkünkben sorban ezekre összpontosítunk.

Az objektumorientált adatbáziskezelőkben történő adatábrázolás és a relációs adatmodell tti különbségek ellenére a könnyebb megértés kedvéért bizonyos analógiákat nevezhetünk meg. az objektumorientált rendszerek objektuma egy rekordnak (tupelnek, a reláció egy sorának), az ly magának a relációnak, ill. annak attribútumainak feleltethető meg. Az objektumorientált llben *sémadefiníció* alatt az osztályok és a köztük fennálló kapcsolatok leírását értjük. A böző osztályokhoz tartozó objektumok és a köztük lévő kapcsolatok (asszociációk) összessége zi magát az objektumorientált adatbázist.



DBMS: Data Base Management System
OOPL: Object Oriented Programming Language
OODBMS: Object Oriented Database Management System

1. ábra: Az OODBMS-ek származtatása

Típuskonstruktorok

Minden programozási nyelv és adatbáziskezelő rendszer ismer bizonyos ala (Integer, Real Character, stb.). Ezeknek létezhetnek bizonyos előre definiált kiterjeszté Time, stb.) is. Azonban a felhasználó által definiált tetszőlegesen bonyolult struktúrákat modellben nem tudunk leírni. Erre a problémára megoldásként az objektumorientált adatb egy része közvetlenül vagy közvetve a *típuskonstruktorokat* nyújtja. Példaként lássunk h típuskonstruktort:

Halmazkonstruktor:

T_{set}=**SET** OF (A:T), ahol A attribútum, T pedig az attribútum típusa. A hal legfontosabb jellemzője, hogy elemei rendezetlenek.

Listakonstruktor:

T_{list}=**LIST** OF (A:T), ahol A attribútum, T pedig az attribútum típusa. A lista típus hogy elemei szekvenciálisan rendezettek. A programozási nyelvek láncolt listájával analóg kialakítására ad lehetőséget.

Tupelkonstruktor:

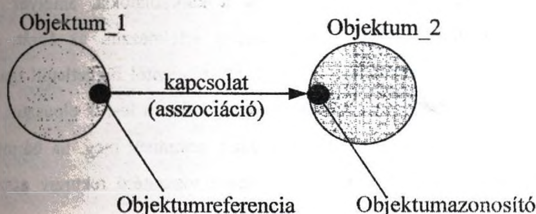
T_{tuple}=**TUPLE** OF (A_1:T_1, ..., A_n:T_n), ahol A_i attribútum, T_i pedig az A_i attribútum tupel típus a relációs modellben a reláció egy sorának felel meg.

A típuskonstruktorok tetszőlegesen bonyolult - akár rekurzív módon is történő - felhasználásával kapott típusok komplexitásukban hasonlítanak a Pascal nyelv *record* illetve a C nyelv *struct* fogalmához.

Kapcsolatok - asszociációk

Az objektumorientált adatbáziskezelők nagy újítása, hogy tetszőlegesen bonyolult kapcsolatokat is átláthatóan tud kezelni. Ezeket a kapcsolatokat a szakirodalom - a relációs adatmodell, illetve az entitás-relációs modell relációjával való fogalmi keveredés elkerülése végett - gyakran asszociációnak nevezi (2. ábra). Az asszociációkat osztályokra definiáljuk, így az asszociációk az osztályok példányai, az objektumok között teremtenek kapcsolatot. Ha egy A objektum kapcsolatban áll egy B objektummal, ez azt jelenti, hogy A-ból B elérhető. Ha B-ből A is elérhető, az asszociáció *kétirányú*. Asszociációk segítségével könnyedén leírhatjuk a *több-több* típusú (Pl.: tanár-diák) *kapcsolatokat* is. A kapcsolatok mentén történő objektumról objektumra lépkedést *navigációnak* nevezik. A navigációnak lekérdezésekkor van elsősorban jelentősége. Egy objektumból kifelé irányított asszociáció mentén elérhetünk egy újabb objektumot, melynek nyilvános adatmezőihez így hozzáférhetünk. Az asszociáció kiindulási oldalán *objektumreferencia*, azaz a célobjektum *objektumazonosítója* található. Az objektumazonosító szolgál arra, hogy - még ha az objektumok tartalma teljesen azonos is - az objektumokat meg tudjuk különböztetni (2. ábra).

Az asszociációk mentén *terjedési tulajdonságokat* is meghatározhatunk. Asszociációk mentén

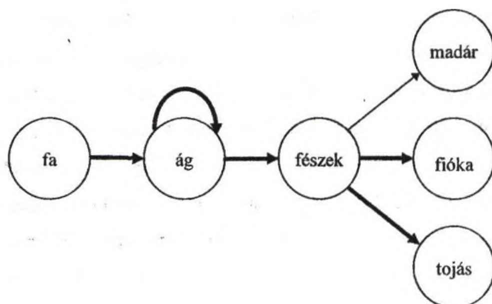


2. ábra: Az objektumazonosító és az objektumreferencia

jedő tulajdonságok lehetnek például az objektum törlése, zárolása (lock) és zár alól történő oldása (unlock). Ez azt jelenti, hogy ha egy olyan A objektumot törlünk le, melynek egy másik B jektum felé törléssel terjedő asszociácója volt, akkor ez a bizonyos másik B objektum is letörlődik. B objektumnak is van ilyen asszociációja C objektum felé, akkor ugyanez történik tovább látlan mélységben. Mivel az esetek túlnyomó részében olyankor alkalmazzuk ezt a tulajdonságot, ikor egy objektum részekből épül fel (ld. Példa 1.: autója), azt az objektumot, amelyből terjedő zociáció indul ki *tartalmazó objektumnak* (composite object) nevezik.

Példa 4.: Erdész megbízónk arra kért, hogy egy speciális erdőrészletet nyilvántartásba. Külön kérése volt, hogy mivel az erdő igen értékes és egyedülálló, a fák fő és az azokon fészkelő madarakról is vezessünk nyilvántartást. A fáknak vannak ágaik; az ág ágakra bomolhatnak és így tovább. Az ágakon - az egyszerűség kedvéért az elágazásokat kívül hagyjuk - helyenként fészkek találhatók. A fészkekben lehetnek madarak, fiókák, Mivel a fák nagyon öregek, előfordul, hogy villám csap beléjük, és kidőlnek. Ilyenkor a to fiókák odavesznek, a szülőmadarak pedig - szívük szakad ugyan, de - elrepülnek.

Hogyan ábrázoljuk ezt a struktúrát az asszociációk és a terjedési tula felhasználásával? (3. ábra)



3. ábra: Terjedő asszociációk

Az ábrán vastaggal jelöltük azokat a kapcsolatokat, amelyek mentén terjed tulajdonság. Visszafelé, jobbról balra haladva értelmezzük az ábrát. A tojások és megsemmisülnek ha a fészek megsemmisül. Ez mindentől függetlenül igaz. A szülők to attól még, hogy a fészek elpusztul. Tovább lépve balra: a fészek elpusztul, ha az ág, ami letörik, elég, stb. Az ág rekurzív módon akkor semmisül meg, ha bármelyik olyan ág amelyik neki "őse". Ezt fejezi ki az önmagába visszatérő rekurzív asszociáció. Ha p elpusztul, az összes ága is elpusztul.

Verziókezelés

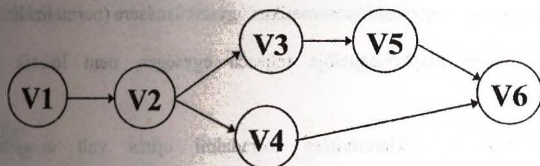
Különösen nagy rendszerek esetén jól jöhet, ha az objektumok állapotát nem csak pillanatban ismerjük, hanem korábbi állapotokat is előhívhatunk. Erre szolgálnak a verziók kezelését a legtöbb objektumorientált adatbáziskezelő rendszer támogatja.

---

[1] Didaktikailag talán helyesebb lenne ebben a kontextusban fa helyett fatörzsről beszélni.

A verziók fejlődése lehet lineáris és elágazó. Nem kizárt különböző verziók újbóli "egymásra lálása" sem.

Példa 5.: Tekintsük egy autómodell fejlesztését (4. ábra). Kezdetben gyártottak egy modellt V1), amit évről évre fejlesztgettek (V2). Amikor jól kezdett menni a cégnek, kínálatszélesítésként bb almodellt is elkezdtek gyártani (V3, V4) (nyitható tető, kombi, stb.), melyek a maguk útján jlődtek (V5). Később költségkímélés miat beszüntették a széles skála gyártását, ismét csak az lapmodellel foglalkoztak (V6), amiben viszont megtalálható volt az összes eddig külön fejlesztett ág néhány tulajdonsága.



4. ábra: Verziók

Nyelvek

Hasonlóan az objektumorientált adatbáziskezelőkhöz, a lekérdező nyelvek is két irányból lődtek.

Az egyik út, melyet elsősorban a relációs adatbáziskezelők gyártói járnak az SQL jektumorientált kiterjesztésének megalkotása. A legjelentősebb ezek közül az SQL3, melynek bványosítása folyamatban van, a bizottsági tervezet 1996-ban elkészült. A nemzetközi szabvány vezete 1997. közepére, maga a nemzetközi szabvány 1998-ra várható. Az SQL3 a hagyományos ációs lekérdezéseken felül ismeri és kezeli az absztrakt adattípust annak minden tulajdonságával etódusok, öröklés, objektumazonosság, polimorfizmus, stb.). Az SQL3 előreláthatóan könnyebben z integrálható más programozási nyelvekkel, mint elődei voltak.

A másik út, ami előttünk áll, a C++, mint legelterjedtebb objektumorientált nyelv perzisztens ajdonságokkal való ellátása. A legtöbb napjainkban működő objektumorientált adatbáziskezelő ezt utat választotta.

Míg a relációs adatbáziskezelőknél az esetek többségében külön kellett foglalkoznunk tdefiníciós, adatmanipulációs és host (gazda) nyelvekkel, objektumorientált esetben ezek egységes szként jelennek meg. Az adatdefiníció például ugyanúgy C++-ban készülhet, mint a lekérdezések. adatbáziskezelő nyelvének más nyelvbe történő beágyazására pedig értelemszerűen nincs szükség, zen például az SQL-t legtöbbször éppen C-be ágyazzák be. Azzal, hogy nem csak az általános

szoftverfejlesztés, hanem az adatbáziskezelés is egy objektumorientált programozási nyelven tö
a távolság a két terület között nagyon lecsökkent, a hatékonyság nőtt.

Összegzés

Az objektumorientált adatbázis koncepció megítélése nem egyértelmű. Bizonyos tekint
még visszalépés is történt a relációs adatbáziskezelőkhöz képest.

- Mivel az objektumorientált adatbáziskezelők mögött nem áll olyan erős matematika, mint
  relációs rendszereket támogatja, nincs, vagy korlátozott lehetőségek vannak csak a lekérde
  optimalizációjára, a séma - nagyrészt - automatikus egyszerűsítésére (normalizálás).
- Míg a relációs modell alapkoncepciója teljesen egységes, nem létezik ilyen egy
  objektumorientált modell.
- A relációs lekérdezések deklarativitása forradalmi újítás volt megjelenésükkor.
  objektumorientált lekérdezések pedig jellemzően mégiscsak navigáción alapulnak.

Másrészről az objektumorientált adatbáziskoncepciók újítása a hagyományos - elsős
relációs - adatbáziskezelőkhöz képest az, hogy az emberi gondolatok számítógép számára tö
leképzésénél alkalmazott absztrakciós lépcső kisebb, azaz elképzeléseinket sokkal természete
vetíthetjük le ilyen rendszerekre. Említésre méltó még, hogy az objektumorientált rendsze
bizonyos esetekben jóval - gyorsabbak lehetnek relációs társaiknál.

Felmerül tehát a kérdés: Mikor érdemes objektumorientált adatbáziskezelőt használ
Általában olyan esetekben, amikor az objektumok közötti kapcsolatok és az objektumok struk
komplex. Tipikusan ilyen alkalmazások lehetnek a telekommunikáció, a CAD, a CASE, a térkép
geográfia, a multimédia, stb. területének problémái. Ezeken a területeken az objektumori
adatbáziskezelő rendszerek előretörése egyre intenzívebb.

A cikkel kapcsolatos WWW linkek a ***http://gedeon.ttt.bme.hu/~mosko/interest.html*** *címen érhetők el.*

Objektum-orientált adatbázisok logika-alapú megközelítései

Hajas Csilla

KLTE, Matematikai és Informatikai Intézet,

Információ Technológia Tanszék

E-mail: hajas@math.klte.hu

1. Áttekintés

Az utóbbi években az adatbázisok területén is egyre népszerűbbé vált az objektum-orientált filozófia, és megjelentek az első objektum-orientált adatbázis-kezelő rendszerek. Több javaslat is született az objektum-orientált modellre és az adatbázis nyelvekre, de még nincs olyan teljeskörű javaslat, amely tökéletesen eleget tenne egy objektum-orientált adatbázis-kezelő rendszertől elvárt összes követelménynek. Az objektum-orientált modellekben az alapfogalmak (például komplex objektumok, objektum azonosság, bezárás, típusok és osztályok, típus és osztály hierarchia, öröklődés) adatbázis technológiákba való átültetése az egyik fő törekvés, miközben sajnos fontos DBMS funkciók (például az objektumok módosítása, az update) kezelése nem kielégítő, még az objektum-orientált adatbázis-kezelés szabványaként javasolt ODMG-93 adatmodellben sem, [6,ODS] [13,NOD].

Ezzel párhuzamosan a programozási nyelvekben és az adatbázisok világában is egyre népszerűbbé vált a deduktív megközelítés, és egy érdekes kutatási terület alakult ki: hogyan lehet összekapcsolni a lényegében "érték-orientált" dedukív megközelítést az objektum-orientáltság fent felsorolt fogalmaival. A következő oldalon szereplő 1. Tábla segítségével összehasonlíthatunk néhány különböző logika-alapú megközelítéset. Természetesen az irodalomban ennél jóvabb bővebb a választék, itt ebben az áttekintésben öt szempontot veszünk tekintetbe, mennyire támogatja a nyelv az objektum-orientált-fogalmak közül hármat, a komplex objektumokat, az osztály/alosztályt és az objektum azonosságot, valamint a dedukció korlátozva van-e a logika egy részére (általában a Horn-szabályokra) és a rekurzió megengedett-e.

Nyelv	Szerzők	Komplex objekt.-ok	Osztály / alosztály	Objektum azonosság	Dedukció
[17,NF2]	Roth-Korth-Silberschatz	Nemciklikus	Nem	Nem	Igen
[4,DEN]	Benczúr-Hajas-Kovács	Nemciklikus	Nem	Nem	Igen
[3,CCO]	Bancilhon-Khoshafian	Nemciklikus	Lehet	Nem	Korláttal
[1,COL]	Abiteboul-Beeri	Nemciklikus	Nem	Nem	Korláttal
[14,LDM]	Kuper-Vardi	Igen	Nem	Igen	Igen
[5,ψ-terms]	Beeri-Nasr-Tsur	Igen	Igen	Nem	Korláttal
[18,FOD]	Rubinski	Igen	Igen	Igen	Igen
[2,IQL]	Abiteboul-Kanellakis	Igen	Igen	Igen	Igen
[7,C-Logic]	Chen-Warren	Csak halmaz	Igen	Igen	Korláttal
[15,O-Logic]	Maier	Igen	Igen	Igen	Igen
[10,F-Logic]	Kifer-Lansen-Wu	Igen	Igen	Igen	Igen

1. Tábla: Egy áttekintés deduktív objektum-orientált nyelvekről

2. Útközben: Nested Datalog

A korszerű adatbázis-kezelés egyik jellemzője az összetett típusú értékek tárolá és lekérdezésének a lehetősége. Ezt az objektum-orientált adatbázis-kezelők valósítják. Az objektum-orientált modellek matematikailag nem jól definiáltak, ezért is érd foglalkozni az egyszerűbb a beágyazott (nested) relációs adatmodellel, amely egy s átmenetet jelent a komplex objektumok precíz kezeléséhez. A beágyazott relációs adatm a relációs adatmodell nem-első-normálformájú relációk kezelésére alkalmas kiterjes amelyben a táblák soraiban egymás alatt azonos szerkezetű összetett értékek (tábla-érték szerepelhetnek. Íly módon téve lehetővé az összetett típusok használatát és jelentős mért csökkenve a redundanciát. A beágyazott relációs adatmodellhez bevezetett leké rendszerek a jól ismert klasszikus relációs modellhez definiált lekérdező nyelvek (rel algebra, biztonságos relációs kalkulusok) beágyazott modellre történő kiterjesztései.

Osztályok:

```
intézet [ főnök ⇒ (intézet, igazgató);
          kor ⇒ középkorú;
          legmagasabbVégzettség ⇒ végzettség;
          cikkek ⇒⇒ publikáció;
          legmagasabbFokozat •→ phd;
          átlagFizetés → 50000 ]

személy [ név ⇒ string;
          barátok ⇒⇒ személy;
          gyerekek ⇒⇒ gyerek(személy)]

alkalm [ mhely ⇒ tanszék;
         főnök ⇒⇒ alkalm;
         kapcsMunkák@alkalm ⇒⇒ jelentés]

tanszék [ mtársak ⇒ (hallgató, alkalm);
          tszvez ⇒⇒ alkalm]
```

Deduktív szabályok:

```
E [főnök → M ] ← E : alkalm ∧ D : tanszék
                 ∧ E [ mhely → D [ tszvez → M : alkalm ] ]

X [kapcsMunkák@Y →→ Z ] ← Y : intézet ∧ X : intézet
                          ∧ Y [cikk →→ Z] ∧ X [cikk →→ Z]]
```

Lekérdezések:

```
? – X : alkalm ∧ X [ főnök → Y;
                     kor → Z : középkorú;
                     mhely → D [ tsznév → "InfTechn"] ]

? – mari [ kapcsMunkák@Y →→ jacm90 ]

? – mari [ kapcsMunkák@zoli →→ Z ]
```

3. Az egyik út: F-logic példákon keresztül

Az 1.részben több különböző logika-alapú nyelvet hasonlítottunk össze, ame[...] többé-kevésbé támogatták az objektum-orientáltság jól-ismert fogalmait. Az előadásban [...] közül egyet részletesebben is áttekintünk. Az "F-logic" (ahol "F" a "Frame-logic" rövid[...] nyelvet Michael Kifer és James Wu vezette be ([10,F-logic], [11,LCO]), alapvetően [...] "O-logic" nyelve alapján [15,O-logic].

Bizonyos értelemben az F-logic hasonló kapcsolatban áll az objektum-orie[...] paradigmákkal, mint ahogyan a klasszikus predikátum kalkulus kapcsolódott a relá[...] megközelítéshez. Az objektum-orientáltság fontosabb alapfogalmait az F-logic közvetl[...] megvalósítja, ugyanakkor ebben más jellemzők is könnyen modellezhetők, például a [...] tartalmazza az objektum azonosságot, komplex objektumokat, öröklődést, polimorfizm[...] bezárást, stb. valamint a lekérdezése módokat, stb. és a dedukciót is.

A jól-érthetőség és az egyszerűség érdekében az előadásban példákon keres[...] tekintjük át az F-logic főbb jellemzőit, most ebben az összefoglalóban csak a minta-adatb[...] adjuk meg, amelyen keresztül áttekinthetjük az F-logic sajátosságait.

Példa: A minta-adatbázis.

- Adatbázis tények:

laci [*név* → *"Laci"*;
kor → 36;
mhely → $inftechn_1$ [*tsznév*→*"InfTechn"*;
tszvez → *laci*;
mtársak →→ {*jános, kati*}]]

mari [*név* → *"Mari"*;
végzettség → *egyetem*;
barátok →→ {*laci, kati*};
mhely → $inftechn_2$ [*tsznév*→*"InfTechn"*]

A doktori értekezésem [9,BRA] egyik témaköre, hogyan képzelhető el a beágyazott és a logikai adatmodell kombinációja, ez az ún. nested Datalog. A szabályok hierarchikus szerkezetűek, azaz egy szabály alprogramokat is tartalmazhat, ahol az alprogramok szabályai is hasonló módon épülhetnek fel. Így egy nested Datalog szabály egy fával ábrázolható, és ezen a fa-reprezentáción alapul a biztonságosság, a helyesség, a konzisztencia, a megelőzési gráf és a rekurzó fogalmak definiálása, és a szabályok kiértékelése.

Példa: Egy absztrakt példa. Legyen *r* szabály a következő:

$q(A, B, C, Y)$:- $p(A, X)$ & $pred.X(B)$ & $p_1(C, D)$ &
$p_1(C, D)$:- $p_3(C, D, c)$;
$p_1(C, D)$:- $p_4(A, C, V)$ & $pred.V(B, D)$; &
$pred.Y(E, U)$:- $p_3(A, B, E)$ & $p_2(F)$ &
$pred.U(G)$:- $p_1(E, G)$;
$p_2(F)$:- $q(A, B, F, W)$; ;

Az *r* főszabálya: $q(A, B, C, Y)$:- $p(A, X)$ & $pred.X(B)$ & $p_1(C, D)$;

ahol a pontosvesző a szabály végét jelzi. Az *r* szabálynak két alprogramja van. Az első lprogram két alszabályból áll, és mindkét alszabálynak csak főszabálya van, ez az alprogram p_1-hez tartozó relációt határozza meg. A második alprogram pedig egy alszabályból áll, melynek van egy két alszabályból álló alprogramja is. A második alprogram az *Y* változóhoz rtozó összetett tábla-értéket számolja ki, valamint a p_2-höz tartozó relációt.

A táblaértékek kezelésére bevezetünk egy új változó predikátum szimbólumot (*pred.X*) a a szabályok szemantikájának megadásához megkülönböztetünk lokális illetve globális B predikátumokat Változó predikátum szimbólumok nemcsak a szabályok törzsében, nem a belső alszabályok fejében is előfordulhatnak. A szabály törzsben előforduló változó edikátumok (pl. *pred.X*, *pred.V*) EDB predikátumok, ezek segítségével tudunk hivatkozni y belső reláció saját soraira. Az alszabályok fejében előforduló változó predikátumok (pl. *ed.Y*, *pred.U*) lokális IDB predikátumok, ezzel tudunk tábla-értékeket hozzárendelni az zetett attribútumokhoz. A nested Datalog részletes leírása [4,DEN] cikkben is gtalálható. A nested Datalog szemantikája a szabályok bizonyos sorrendben történő rtékelése, ahol a kiértékelés eredménye olyan beágyazott relációs algebrai kifejezés gadása, amellyel kiszámolhatók a nested Datalog szabályok által definiált relációk.

Irodalomjegyzék

[1,COL] ABITEBOUL, S., BEERI, C.: On the power of languages for manipulation of complex o

ABITEBOUL, S., GRUMBACH, S.: COL: A logic-based language for complex objects. *Adva Database Technology EDBT'88. LNCS* No 303. pp. 271-293., 1988.

[2,IQL] ABITEBOUL, S., KANNELAKIS, P.C.: Object identitz as querz language primitive. *Proce Int.Conf. on Management of Data,* pp. 159-173., 1989.

[3,CCO] BANCILHON, F., KOSHAFIAN, S.: A calculus for complex objects. *Journal of Computer and Sciences,* No. 38, pp. 326-340, 1989.

[4,DEN] BENCZÚR, A., HAJAS, Cs., KOVÁCS, Gy.: Datalog extension for nested relations. *Compute Mathematics with Applications,* Vol.30, No.12, pp.51-79, 1995.

[5,ψ-terms] BEERI, C., NASR, R., TSUR, S.: Embedding ψ-terms in a Horn-cluse logic language. *MCC Te Report,* ACA-T-050-88, 1988.

[6,ODS] CATELL, R.G. ed.: *The object database standard: ODMG-93,* Morgan Kaufmann Publisher, 19

[7,C-logic] CHEN, W., WARREN, D.S.: C-logic for complex objects. *Proceedings, Symp. on Princ Database Systems,* pp. 369-378., 1989.

[8,DLP] ELIENS, A.: Principles of object-oriented software development. *Addison-Wesley Publ.Comp.,* 1

[9,BRA] HAJAS Cs.: Beágyazott relációs adatmodell és lekérdező nyelvei. *Egyetemi doktori értekezés, Matematikai és Informatikai Intézet,* 1995.

[10,F-logic] KIFER, M., LAUSEN, G., WU, J.: Logical foundations for object-oriented and frame-based lan (accepted in *Journal of* ACM), *Technical Report, SUNY,* pp. 1-100., 1995.

[11,LCO] KIFER, M., WU, J.: A logic for programming with complex objects. *Journal of Computer and Sciences,* No. 47, pp. 77-120, 1993.

[12,QLN] KOVÁCS, Gy., HAJAS, Cs., QUILIO, I.: Representations and query languages of nested re *Procedeedings of the Fourth Symposium on Programming Languages and Software Tools,* pp.360-373, 1995, and accepted in *Annales Univ. Sci, Budapest, Sectio Computatorica.*

[13,NOD] KOVÁCS, Gy., BENCZÚR, A., CSERGES, E.: Nested relations and ODMG collections. *Co Large Parallel Database Project No.93:6638 Deliverable Report,* Dec. 1995.

[14,LDM] KUPER, G. M., VARDI M. Y.: The logical data model. *ACM Transactions on Database Syste* 18, No. 3. pp. 379-413., 1993.

[15,O-logic] MAIER, D.: A logic for objects. *Proceedings, Workshop on Foundation of Deductive Databa Logic Programming,* pp. 6-26., 1986.

[16,NSQL] ROTH, M.A., KORTH, H.F., BATORY, D.S.: SQL/NF A query language for ¬1NF re databases. *Information Systems,* Vol. 12, No. 1, pp. 99-114, 1987.

[17,NF2] ROTH, M.A., KORTH, H.F., SILBERSCHATZ, A.: Extended algebra and calculus for relational databases. *ACM Transactions on Database Systems,* Vol. 13, No. 4, pp. 389-417, 1988

[18,FOD] RUBINSKI, H.: On first-order databases. *ACM Transactions on Database Systems,* Vol. 12, N 325-349, 1987.

A világ vezető objektum-orientált adatbáziskezelője: az ObjectStore

Szerző: Ertner Péter

Intelligens Software Rt.

1142 Budapest Teleki Blanka u. 15-17.
Tel: 251-5449 Fax: 220-5598
E-mail: iqsoft@iqsoft.hu
WWW: http://www.iqsoft.hu

Az előadás az object design Inc. termékével az objectStor-ral foglalkozik. Ez a termék jelen pillanatban az objektum-orientált adatbázis-kezelők piacán a vezető terméknek számít.

Manapság az OO technológiák a tervezésben és a megvalósításban egyre nagyobb szerepet játszanak. A hagyományos adatbázis-kezelők használata azonban nem ideális az OO szemléletű alkalmazások számára.

Bemutatásra kerülnek azok az alapvető technológia megoldások, amelyek segítségével az objektum- orientált tervező- fejlesztőeszközök segítségével készült alkalmazások számára igen kedvező környezetet teremt az objectStore:

A kliens oldali fejlesztőeszköz oldaláról nézve nincs jelentős különbség a csak a memóriában létező (tranziens) és az adatbázisban tárolt (perzisztens) objektumok között, ebből a szempontból ez a technológia teljesen átlátszó. A relációs technikával ellentétben nincs szükség az adatstruktúrák megfeleltetésére (mapping), az OO technológiával tervezett modellek egyértelműen leképezhetőek az adatbázisbeli struktúrára.

Az OO adatbázisokban az objektumok összekapcsolása mutatókon keresztül történik, ami az objektumok összekapcsolását, illetve az objektum láncokon való haladást igen gyorssá teszi.

Ezekben az adatbázisokban az elosztott adatbázis-kezelés természetesnek számít, az elérhető adatbázisok között meg van valósítva a kétfázisú commit. Ennek és a haté-

kony cache-elési technikának köszönhetően az interaktív alkalmazások használatáho nagyon jól igazodik ez a típusú adatbázis-kezelés.

Egyre terjed azoknak az alkalmazásoknak a száma, amelyek multimédia adatokat kezelnek komplex struktúrában a hagyományos szöveges adatokkal, az objectStor megfelelő komponensei ehhez is aktív támogatást nyújtanak.

Az OO adatbázis-kezelők könnyen integrálhatóak különböző gyártók ORB (Object Request Broker) kiszolgálóihoz. Ez a technológia - az elosztott objektumok (OLE, CORBA, stb.) támogatása - az egyik legdinamikusabban fejlődő területe a szoftver iparnak.

Az ObjectStore ODBMS szerepe a vállati információs rendszerekben

Kovács András, Nick János, Újfalussyné Nagy Gyönyvér
HiCare Kft

Bevezetés

Az objektum orientált adatbáziskezelõk elérték azt a fejlettségi szintet - és ez különösen igaz a piacvezetõ ODBMS-esre, az Object Design Inc. ObjectStore (OS) nevû OO adatbáziskezelõjére -, hogy alkalmassá váltak vállati információs rendszerekben nagyméretû, kritikus alkalmazások megvalósítására.. Az ObjectStore rendelkezik mindazokkal a képességekkel, beleértve a megbízható mûködést biztosító jellemzõket, amelyek a vállati IT rendszerekben történõ alkalmazásokhoz szükségesek.
Az elõadás azt vizsgálja, hogy egy ODBMS-nek milyen követelményeknek kell megfelelni a vállati IT rendszerekben, és ezeknek az ObjectStore mennyire tesz eleget.

Napjaink vállati IT rendszerének jellemzése

Ha napjaink vállati IT rendszerében keressük az ObjectStore ODBMS helyét és szerepét, akkor elõször azt kell vizsgálnunk milyen ez a környezet, mik a legfontosabb elemei és mit nyújt ebben a környezetben az ObjecStore, a világ vezetõ objektum orientált adatbáziskezelõje.
Milyen is ez az IT környezet? Mindenekelõtt erõs a törekvés az OO technológia bevezetése. Az OO technológia megszünteti a tradícionális struktúrált fejlesztésben a életciklus különbözõ fázisai közötti koncepcionális, módszertani réseket. Az életciklus egyes fázisai közötti átmenet sima, gördülékeny, így inkrementális, iteratív fejlesztési életciklus alkalmazható. A termékek piacradobásanak ideje, a módosításokhoz, javításokhoz szükséges idõ lényegesen csökkenthetõ.
Az OO technológia a híd szerepét betöltve áthidalja a felhasználók és fejlesztõk közötti távolságot: a két fél közös nyelvet használ, a domain osztályok a valóságot modellezik, képezik le.
Milyen feladatokat kell megoldania a vállati IT rendszernek? A legfontosabbak a következõek:

1. Az üzleti folyamatok újratervezése, korrekciója (Business Process Reengineering vagy BPR). A BPR módszertanok erõteljesen az OO irányba tartanak.
2. Analízis és tervezés CASE eszközök alkalmazásával. A CASE világában az OO technológia módszertanok térhódítása még erõteljesebb mint a BPR területen, sõt itt már az OO módszertanok konvergenciája is megfigyelhetõ.
3. Implementáció, lehetõleg OO technikával. Manapság minden RAD, 4GL és egyéb alkalmazásfejlesztõ eszköz többé-kevébé OO megközelítéssel dolgozik. Sok a 4GL, de jelenleg a C++ a legáltalánosabban használt OO programozási nyelv. A JAVA nyelv gyors fejlõdését és térhódítását mindnyájan érzékeljük.
4. Növekvõ igény van nagy teljesítményû, nagy méretû, elosztott adatbázisokat kezelni képes adatbáziskezelõkre, ugyanis az adatmennyiség drasztikusan nõ, az információhoz való hozzáférés sebessége pedig a felgyorsult üzlet életben sok esetben kritikus lehet.
5. Erõs igény fogalmazódik meg a fokozott megbízhatóságra; terrorcselekmények, természeti katasztrófák esetén sem szabad a fontos információkat elveszíteni.
6. Hatékony Internet/Intranet fejlesztést biztosító technológia.
7. A komponens alapú elosztott rendszerek (CORBA, DCOM) implementálásának támogatása. Ez a pont elviekben az õt megelõzõhöz tartozik, de fontossága miatt kiemeltük.
8. Utoljára, de nem utolsó sorabn a meglevõ alkalmazások integrálása, a relációs adatbázisokban tárolt adatokhoz való egyszerû és gyors hozzáférés, az új objektum orientált és a meglévõ hagyományos alkalmazások együttélése, együttes mûködtetése.

ObjectStore szerepe vállati IT rendszerekben

Ebben a pontban az elõekben megfogalmazott kérdésekre keressük a választ, hogy a felsorolt igények alapján mit az ObjecStore mit tud biztosítani a vállati IT rendszerek számára. Csak azokkal a rendszerekkel foglalkozunk, amelyek már áttértek vagy át akarnak térni az OO technológiára.

BPR és OO analízis/tervezés

A BPR módszertana az OO technológia felé tart. Az analízis és tervezési módszerek között az OO módszertanok dominálnak, sõt ezen módszertanok határozott

konvergenciája figelhetõ meg. Az ObjectStore támogat minden OO paradigmát: öröklés, többszörös öröklés, polimorfizmus stb., azaz, korlátozások nélkül modellezhetünk.

OO implementáció

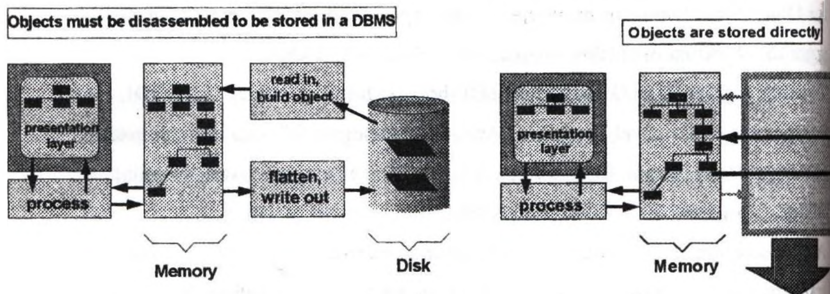
Az ObjectStore támogatja az elterjedt szabványos illetve a de facto szabványnak számító objektum orientált nyelveket: C++, Smalltalk, JAVA.
Továbbá elérhetjük az OS adatbázist SQL-bõl, az OpenAccess kiegészítõ ODI komponens segítségével, továbbá az ActiveX-et támogató fejlesztõ környezetekbõl, pl.:Visual Basic, Dephi, Excel,stb., az ActiveX SDK alkalmazásásával, valamint HTML környezetbõl az ObjectForms fejlesztõ/futtató eszköz használatával.
Az ObjectStore az OO programozási nyelvekkel szorosan össze van integrálva, azok transzparens kiterjesztése. Csak az objektumok létrehozásakor kell azt eldöntenünk, hogy a létrehozandó objektum tranziens vagy perzisztens. Az objektumokat , a relációs adatbázisban történõ tárolással ellentétben, nem kell táblákra leképezni, nem lép fel ún. impedancia illesztetlenség.

Nagyméretû, elosztott adatbázisokat hatákony kezelése

Az ObjectStore speciális tervezése - “leadership by design” a cég egyik szlogenje - lehetõvé teszi nagy méretû, elosztott adatbázisok hatékony, gyors kezelését. Az objectStore gyakorlatilag nem korlátozza az adatbázisok méretét (az eméleti határ 2^{89} byte). Az ObjectStore adatbázisban elhelyezett pointerek hivatkozhatnak a 2^{89} méretû címtérben tetszõleges objektekre, amelyek kükönbözö támogatott platformokon, különbözõ logikai/fizikai tárolókon és adatbázisokban helyezkedhetnek el. Az ObjectStore megengedi a közvetlen mutatók használatát, tekintet nélkül arra, hogy az adatok azonos vagy különbözõ adatbázisokban továbbá az adatbázisok azonos vagy különbözõ szervereken helyezkednek el.

A pointer hivatkozások feloldásának sebessége kulcstényezõ az objektum kezelõ alkalmazásokban. A maximális teljesítmény érdekében a perzistens hivatkozások feloldása ugyanúgy egy utasításban történik mint a tranzisens hivatkozásoké az ObjecStore Virtual Memory Mapping (VMMA) arhitektúrájának következtében. Az OS a számítógépek virtuális memóriakezelõ hardverének és az operációs rendszerek virtuális memória kezelõ rendszerének a speciális kiterjesztése.

Az ODBMS használat esetén nem lép fel az ún. impedancia illesztetlenség esete, amely objektumok relációs adatbázisban történõ tárolásának a kísérõje. Ekkor minden perzistens objektumot le kell képezni RDBMS táblákra, és futásidõben minden beolvasáskor ill. kitöltéskor a memória kép és a táblák közötti átalakítást el kell végezni. Az RDBMS és ODBMS-sel történõ objektum tárolást az 1. ábra szemlélteti.



1.ábra objektumok tárolás RDBMS és ODBMS adatbáziskezelõkkel

Fokozott megbízhatósági/ rendelkezésre állási igények

A vállati szinten a fokozott megbízhatóságnak, rendelkezésre állásnak kiemelt szerepe van. Ezen követelmények biztosítására az ObjecStore hatékony ezközöket biztosít az alkalmazó számára:

- Transaction processing using two phase commit
- On-line backup and restore
- Distributed backup
- Database replication
- Failover

A komponesn alapú OO technológia támogatása

Az ObjectStore-t, a vezetõ Object Request Broker termékekkek már integrálták, így pl. az IONA Technologies piacvezetõ Orbix ORB-jével. Az ODBMS-ek és ORB-k egymást természetes módon kiegészítõ eszközök, integrálásukkal a két terület

leghasznosabb jellemzõit együttesen alkalmazhatjuk. A CORBA [1]objektumokat az ODBMS-ben tárolhatjuk, azaz perzisztenciájukat az objektum orientált adatbázis kezelõ biztosítja és természetes az ODBMS nyújtotta egyéb szolgáltatásokat (query, relationships, transactions, stb.) is használhatják a CORBA objektumok. Az Integrálással a CORBA-ObjectStore objekumok metódusai a hálózaton keresztül más platformokról, operációs rendszerekbõl, programozási nyelvekbõl is elérhetõvé válnak. Az objektumokat így nemcsak az ODBMS által támogatott nyelvekbõl érhetjük el, hanem mindazokból a nyelvekbõl amelyeket a CORBA támogat (C, C++, Smalltalk, ADA, COBOL, JAVA, stb) és alkalmazhatjuk a szükséges CORBA szolgáltatásokat, a CORBAservices-ket (pl . Security).
A CORBA összekapcsolja a két versengõ komponens arhitektúrát, a CORBA-t és az OLE/DCOM-ot. Pl. az *Orbix for desktop* segítségével az OLE és a CORBA komponensek kliensként és kiszolgálóként is kölcsönösen elérhetõek egymás számára. Meg kell továbbá említenünk, hogy az egy új ObjectStore-hoz kapcsolódó termék az ActiveX SDK segítségével az ObjectStore-ben tárolt objektumok az ActiveX-et támogató termékekbõl is elérhetõek (pl VB, Delphi, stb).

A meglévõ adatbázisokkal és alkalmazásokkal történõ integrálás/együttmûködés

Az ObjectStore a DBConnect nevû termék segítségével képes a relaciós adatbázisok elérésére, míg az OpenAccess alkalmazásával az ObjectStore-ban tárolt objektumok válnak elérhetõvé a meglévõ SQL és ODBC alapú fejlesztõeszközök számára. Mindkét esetben az ObjectStore SchemaMapper biztosítja a szükséges transzformációk hatékony definiálását és végrehajtását.
A DBConnect alkalmazásával a meglévõ adatbázisok és az új objektum orientált alkalmazás hatékonyan együtt tud mûködni. Különbözõ jellegû alkalmazások készíthetõek, pl:

- Objektum orientált frontend a relációs adatbázisokhoz
- Nagy teljesítményû adatelérést biztosító ún. *data replication* alkalmazások
- *Object warehause* jellegû alkalmazások, ahol a különbözõ relációs és az ObjectStore-ban tárolt informóciók egységes objektum formában állnak az OO Internet/Intranet alkalmazások rendelkezésére WEB, CORBA ORB, vagy ObjectStore saját kommunikációs mechnizmusán keresztül.

Hatékony Internet/Intranet fejlesztési technológia

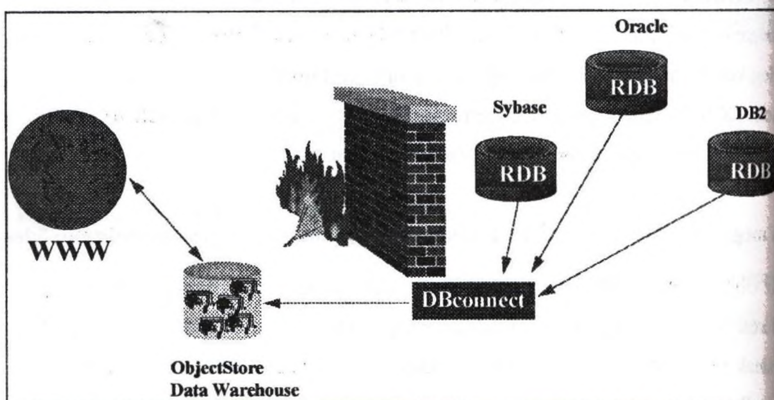
Az ObjectStore kiegészítõ termékei segítségével

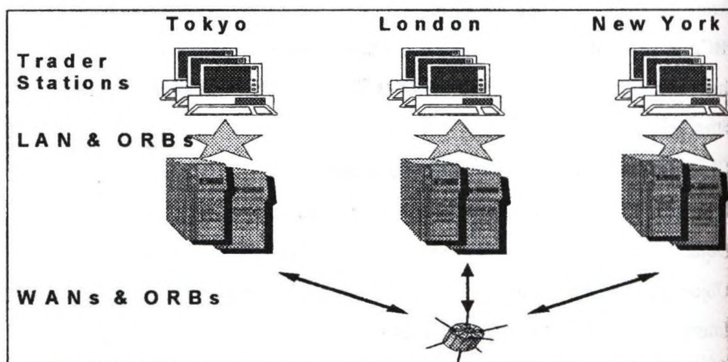
- DBConnect: relációs adatbázisok elérése
- OpenAccess: ObjectStore elérése SQL-bõl és ODBC-n keresztül
- ObjectForms: WEB interfész az ObjectStore-hoz [2]

valamint a

- CORBA Object Request Broker technológiával integrálva

az Internet/Intranet fejlesztések számára a korszerû és hatékony megoldások széles tárházát kínálja.





2. ábra ObjectStore WEB alkalmazás

3. ábra ObjectStore-CORBA/Orbix trader alkalmazás

Összefoglalás

Az ObjectStore rendelkezik mindazokkal a képességekkel, beleértve a nagy megbízhatóságot biztosító jellemzõket is, amelyek a korszerû objektum orientált vállati IT rendszerekben történõ alkalmazásokhoz szükségesek. A termékcsalád többi tagjával és a CORBA/ORB termék-integrálásokkal a korszerû és hatékony megoldások széles spektrumát kínálja a CAD/CAM alkalmazásoktól az Internet/Intranet alkalmazásokig.

Irodalom

[1] Kovács A.: A kliens-kiszolgáló rendszerek új generációja: CORBA alapú elosztott rendszerek, I. Országos Objektumorientált Konferencia, 1996

[2] Nyilas I., Újfalussy G.:WWW objektumok kezelése: Web fejlesztés ObjectForms fejlesztõ eszközzel, I. Országos Objektumorientált Konferencia, 1996

RELÁCIÓS ADATBÁZISOK ELÉRÉSE OO MÓDON: AZ OBJECTCONNECT TERMÉKCSALÁD

Takáts Tamás

AXIS Számítástechnikai Kft.

A megvalósítandó feladatok komplexitásának növekedése miatt fokozatosan az obje orientált technológiák felé fordul a világ. Egy 1996. júniusában végzett amerikai felm szerint a fejlesztők közel 80 százaléka már jelenleg is használja, vagy tervezi az átállá objektum-technológiára. Ez olyan kihívást jelent, amely elől nem térhetnek ki a nagy adatb kezelő gyártók sem. A különféle SQL nyelvjárások kezdettől fogva tartalmaztak kiterjesztéseket, amelyek az objektumorientált technológia irányába mutatnak. Köztudott, elsőként a Sybase tett lépéseket ezirányba, pl. a tárolt eljárások és triggerek segítsé lehetőséget adott absztrakt adattípusok és a hozzájuk kapcsolódó műveletek definiálására

A Sybase objektum-orientált technológia fejlesztésének mai három fő iránya a háromré architektúrának megfelelően: szerver – middleware – kliens. Az alábbiakban a szerver middleware lehetőségeket tekintjük át, a kliens eszközökkel külön előadás foglalkozik.

1. Objektum-orientált adatbáziskezelés

A fejlesztés alatt álló **Sybase Adaptive Server** fő újdonságai:

- Object Storage
- Összetett adattípusok (vektor, mátrix, egymásba ágyazott típusok)
- Különleges adattípusok (például audio, video, koordináta, idősor) kezelése
- Felhasználói adattípusok definiálása
- Felhasználói metódusok definiálása Transact-SQL-ben, vagy Java-ban
- Speciális műveletek (pl. a polygon adattípus esetén az *inside*, *outside* relációk)
- Az egyes típusokhoz speciális indexelési technikák alkalmazása
- Az SQL3 teljeskörű megvalósítása

Az objektumok és a hagyományos táblák kezelése egyaránt lehetséges, ehhez (a adatbázis-kezelő gyártóhoz hasonlóan) kompatibilitási okok miatt ragaszkodik a S Emiatt a Sybase Adaptive Server az ORDBMS kategóriába tartozik.

2. Objektum-orientált middleware

Az objektum-orientált kliens oldali fejlesztőeszközök és a relációs adatbáziskezelési technika közötti űr áthidalásának már ma is zökkenőmentes módszere a middleware technológia alkalmazása: a kliens és a szerver között megjelenő új réteg lehetővé teszi, hogy a tervező és a fejlesztő objektum-orientált adatbázis-fogalmakkal dolgozhasson, és ugyanakkor megőrizze a relációs technika biztonságát és hatékonyságát.

A middleware-rel szemben támasztott követelmények:

- Tegye lehetővé, hogy az applikáció fejlesztői a relációs adatbázis táblái, sorai stb. helyett az OO fogalmai szerinti objektumokat kezelhessenek (öröklődés, beágyazódás, polimorfizmus)
- Automatikusan biztosítsa az előző pont szerinti objektumok konzisztenciáját (objektumok lockolása a többszörös módosítás megakadályozására, tranzakció kezelés, adatbázis-módosítások időbeli ütemezése stb.)
- Tegye lehetővé az osztott (és rendszerint heterogén) adatbázisok egységes elérését.

A Sybase által a közelmúltban bejelentett **ObjectCONNECT** middleware-család teljes mértékben megfelel a fenti követelményeknek. A sorozat első két tagja az **ObjectCONNECT for C++** és az **ObjectCONNECT for OLE**. Az előbbi a C++, az utóbbi az OLE kezelést támogató fejlesztőeszközök használói számára teszi lehetővé az adatbázis-objektumok használatát. Mindkét termék az alábbi komponensekből áll:

- ObjectBuilder: grafikus fejlesztőeszköz, mellyel elkészíthetők és módosíthatók az adatbázis-objektumok.
- ObjectService: az adatbázis-objektumok futtatója.

Az **ObjectCONNECT** termékcsalád jelentősen növeli az alkalmazásfejlesztés hatékonyságát, mivel

- a fejlesztőknek nem kell megismerkedniük a relációs adatbáziskezelés sajátosságaival (azt elrejti előlük a transzparens *object-relational mapping* technika);
- a fejlesztőknek nem kell foglalkozniuk a perzisztens objektumok adatintegritási problémáival (az **ObjectCONNECT** maga szervezi a tranzakciókat és az object constraint-ek futtatását);
- a kliens-szerver kommunikációt optimalizálja az **ObjectCONNECT**.

Az ObjectCONNECT termékcsalád a nyílt eszközök kategóriájába tartozik. Kép
együttműködni a Sybase, Oracle, Informix és Ingres relációs adatbázis-kezelő rendszerek
Az ObjectCONNECT for C++ támogatja a legtöbb Windows és Unix alatt futó C
implementációt; az ObjectCONNECT for OLE pedig együttműködik minden eszközzel, ame
OLE objektumok befogadására képes (ilyen pl. a PowerBuilder, az Optima++, a Visual Bas
vagy az Excel).

Az ObjectCONNECT termékcsalád rendelkezik modellépítő (reverse engineering) funkció
is: az Object Builder a létező relációs adatbázisból automatikusan előállítja az objektu
osztályokat. Az így létrejövő definíciókat a felhasználó kiegészítheti az egyéb attribútum
osztályok közötti kapcsolatok, speciális metódusok megadásával. Ehhez az Object Buil
hatékony támogatást ad a Class Editor, Attribute Editor, Relationship Editor és Foreign K
Editor komponensek segítségével.

Az Object Builder segítségével készített objektumok a program egyéb részeiből meghívhat
használhatók. A program számára nem létezik RDBMS, azt az objektum teljesen elfedi
tranzakciók szervezéséről, az object-constraint-ek futtatásáról, az objektumok egyérte
azonosításáról és az adatbáziskezelőhöz fordulások optimalizálásáról az Object Service
ObjectCONNECT futtató rendszere) automatikusan gondoskodik.

Ha összehasonlítjuk az ObjectCONNECT technológiát a korábban alkalmazott hagyomány
megoldásokkal, az alábbiakat állapíthatjuk meg:

- „Kézi" kódolású objektumok: az applikáció egyedi igényeire készített objektumok
megfelelő metódusokban kialakított SQL utasításokkal. Fő hátrányai a n
munkaigény és a nehéz utólagos módosítás.
- Általános objektumosztály könyvtárak: készen kapható objektumoszt
gyűjtemények (pl. Dbtools), melyeket használat előtt testre kell szabni. A könyvtá
rendszerint nem tartalmazzák az integritás-ellenőrző funkciókat, és a transzpar
heterogén adatbáziselérést. A testreszabás és az utólagos módosítás gyakran túl
időt vesz igénybe.

Az ObjectCONNECT technológia használatával hagyományos módszerekhez képes
programkészítés/módosítás hatékonysága megnő, az adatbiztonság fokozódik, és a progra
futási sebessége is növekszik.

Az ObjectCONNECT technológia egyetlen hiányossága jelenleg, hogy mindig a kliens gépeken fut. Az osztott objektumok technikáját alkalmazva leegyszerűsödhetne a feldolgozás. Az adatbázis-objektumok adminisztrálása ugyanis egy központi helyen (az Alkalmazás-szerverben) történhetne, feleslegessé téve a jelenleg adatbáziskezelő alatt futó objektum összehangolást. A közeli jövőben ez is megoldódik:

Az ObjectCONNECT termékcsalád következő tagja a 4. negyedévben megjelenő **ObjectCONNECT Server**, mely az osztott objektumok kezelésének két legfontosabb standard módszerét (OLE/DCOM illetve CORBA) támogatja. Segítségével teljessé válik az osztott heterogén relációs adatbázisok objektum-orientált (transzparens) elérése.

Alkalmazás fejlesztői eszköz a nagyobb hatékonyságért

Michaletzky Géza

Next Software Kft.

Az előadás célja

Egyrészt bemutatni egy olyan objektum orientált programozási környezetet, amelyben nyelvi eszközöket és a fejlett adatbázis technológiát teljesen integrálták. Ez az eszköz az 1981 óta fejlődő DataFlex adatbázis kezelő és fejlesztői környezet. Ennek többféle változatát használják már ma is hazánkban. Ezek közül a legújabbat, ma még béta-teszt szintjén lévőt fogom bemutatni.

Másrészt, hogy az objektum orientált fejlesztési szemlélet hogyan hat vissza a prog-mozást megelőző munkafázisokra, ott milyen kritériumokat támaszt.

1. A DataFlex 4 nagy vonalakban

A DataFlex 4 egy client/server alapú vizuális fejlesztő eszköz nagyteljesítményű, kulcs-fontosságú adatbázis alkalmazások készítéséhez. Ez az új 32-bites verzió egysze- komponens összeállítást biztosít egy eredeti objektum-orientált fejlesztői környezetben és nyelven.

Vizuális eszközök

A DataFlex 4 az AppBuilder-nek nevezett, erősen intuitív alkalmazás fejlesztő eszközt kínálja.

Amint a név is elárulja, az AppBuilder segítségével kifejlett, kulcsfontosságú adatbázis alkalmazásokat lehet készíteni. Az adatbázis alkalmazás készítése közben soha nem kell kilépni az AppBuilder-ből. Minden elérhető ebből az eszközből, ami csak egy ilyen fejlesztéshez kellhet.

A DataFlex 4-ben az alkalmazások fejlesztése közben az AppBuilder-rel lehet az alkotóelemeket létrehozni és beilleszteni. A látvány alkotóelemek széles skálájából lehet választani. Sőt, a létező osztályok (class) működésének kiterjesztésével felhasználói osz

lyokat is meg lehet alkotni az AppBuilder használatával. A legtöbb esetben soha nem lesz szükség az előre felépített osztályok kibővítésére, mert a DataFlex 4 class könyvtára eleget tesz az alkalmazás fejlesztés közben fellépő minden kívánalomnak.

Az Application Wizard képek sorozatán visz keresztül, amikben ki lehet választani az elemeket, amik felépítik az alkalmazást (pl.: action bar, button bar, adatbázis file-ok, view). Az alkotóelemek elhelyezése után az alkalmazás kész a telepítésre.

Az Application Wizard néhány további tulajdonságai között van a file-kapcsolatok automatikus detektálása, az adatbázis szűrők automatikus létrehozása, és az automatikus, adattípuson alapuló objektum osztály választás. Az adatbázis szűrőkkel lehet felügyelni az adatok megtalálását és a végső megjelenítését. Egy master/detail szerkezetben, például, korlátozni akarjuk a gyerek file-t (detail) és a szülő file-t (master) azért, hogy így csak a két file-ban megegyező record-okat lehessen megtalálni. A legtöbb termékben ezt a működést külön kell programozni.

Az Application Wizard-hoz ráadásként az AppBuilder-be van integrálva a vizuális tervező eszközök széles készlete. A fordító számos opciót szolgáltat, amikkel teljes ellenőrzés alatt lehet tartani a programok fordítását.

Az adatok karbantartására segítségként lehet használni az AppBuilder Database Administration Utility-t. Ezzel nemcsak kezelni lehet a DataFlex adatbázisokat, hanem azonnali hozzáférést ad az ODBC kiszolgáló szerverekhez, mint a Sybase, a Microsoft SQL Server, az Informix, az Oracle és az Interbase.

A DataFlex 4 tartalmaz egy mindennel felszerelt segédprogramot, amivel a fejlesztők nyomon tudják követni az objektumok között küldött üzeneteket, figyelni tudják a változókat és kifejezéseket, és elemezhetik az objektum- és focus fákat. A üzenet követés (message tracing) fontos tulajdonság az objektum-orientált programokban, mert egy átlagos alkalmazás az üzenetek százait generálhatja. Az üzenetek figyelésének lehetősége nélkül a fejlesztő nem tudja teljesen megérteni az alkalmazás viselkedését.

A valós világban nem lehet nagyméretű, kulcsfontosságú alkalmazásokat programozás nélkül létrehozni. Akkor válik nyilvánvalóvá a DataFlex 4GL valódi ereje és előnye,

amikor a programírás kezdődik. Az App. Builder-rel generált forráskód a fejlesztő rendelkezésére áll, azt tetszés szerint tovább bővítheti, módosíthatja.

Nyelvi erősség

A DataFlex 4 valódi objektum-orientált 4GL, melynek előnyei a pontos objektum modellezés, osztály alkotás, öröklődés, zártság és polimorfizmus. A jól bevált eljárás gyűjteményt (DataFlex Application Framework) integrálva a DataFlex 4-nek meg van az ereje, hogy a legtöbbet követelő feladatokkal is megbirkózzon.

Az objektum modellezés területén a DataFlex lehetőséget ad az objektum struktúrák modellezésére az alkalmazás tervének ábrázolása érdekében. Az objektumok beágyazottsága (nesting) az objektum modellezés legfontosabb része, mivel ezzel lehet üzeneteket küldeni (delegate) az objektum hierarchiába. Ez a tulajdonság azért fontos, mert a fejlesztők anélkül tudnak összetett objektum struktúrákat létrehozni, hogy az üzenet küldéssel foglalkozni kellene. Az üzenet küldés automatikus.

Adatbázis Technológia

Mint adatbázis fejlesztő nyelv, a DataFlex 4 egyik legnagyobb ereje a nyelv és adatkezelés nagyfokú integrálása. Továbbá, a DataFlex DBMS record orientáltsága miatt még többet megért az adatfile-ok és indexek szerkezetéről, mint a hagyományos SQL adatbázisok. A record orientációval összekapcsolva az integráció sok jó tulajdonsággal rendelkezik.

Különleges, összefüggés érzékeny adatbázis kezelői felület

Előre és hátra lehet kapcsolgatni az indexelt mezők között, meg lehet nyomni a Keresés gombot, és a DataFlex automatikusan megtalálja a record-okat, a definiált indexnek megfelelően. Például, ha kattint egy indexelt mezőn, beírja a név első néhány karakterét és megnyomja a find-ot (pl.: részlet kulcsos keresés), a DataFlex automatikusan visszakeresi a várt record-ot. Most kapcsoljunk egy más indexű mezőre (vagy egy másik filera) és nyomjuk meg a Previous Record gombot. A DataFlex megjeleníti az előző record-ot azzal az index-szel.

A DataFlex pontosan tudja, hogy milyen indexet kell használni, és hogyan kell rendezni a record-okat.

E tulajdonság erejére elsődleges példa a választási lista. A DataFlex automatikusan újrarendezi az adatot a választási listán, amikor csak a felhasználó hasábról hasábra vált.

Jól működő adatkezelő (data-aware) ellenőrzés

Az adatbázis műveletek, mint a keresés, törlés vagy mentés beépültek az adatbáziskezelő ellenőrzésbe. Még egy részlegesen adott kulcsú keresést is végre lehet hajtani egy indexelt mezőn, hogy végignézzük a file-t és megtaláljuk a megadott record-ot. Ez mind tökéletesen működik, mert a nyelv szinkronizálva van az adatbázissal.

Beépített billentyű interface az adatkezelő ellenőrzéshez

A billentyűk, a DataFlex-ben "Flex Key"-ként hívva, közvetlenül az OOP rendszerbe vannak kötve. Amikor egy billentyű lenyomódik, egy speciális üzenetet küld. Mint a többi üzenet a DataFlex-ben, ez is megsemmisülhet, vagy megsokszorozódhat. Más szavakkal, a billentyűkre saját működést lehet programozni. Ezt téve, a fejlesztő a Flex Keys-el teljes ellenőrzés alatt tarthatja az alkalmazást. Mivel ezek a billentyűk természetesen logikaiak, különböző funkciókat (vagy task-ot) lehet hozzárendelni a billentyűk mindenféle fizikai kombinációjához, helyzetről helyzetre. Más szavakkal, fel lehet készíteni a helyzet különleges követelményeire.

Könnyen belátható a Flex Key használatának előnye egy nagyszámú adatbeviteli környezetben. Az egér használata ezekben a környezetekben nagyon hatástalan. Ez azért van, mert a felhasználónak kattintania kell az egérrel, gépelni kell a billentyűzeten, olvasni kell a forrásdokumentumból és egyszerre figyelnie is kell a képernyőt.

A gyorsító billentyűk hozzárendelése az olyan feladatokhoz, amelyekben kattintani kell az egéren, szükségtelenné teszi az egér használatát a gépelés alatt. Ez egyértelműen sokkal hatásosabb módszer az adatbevitelre.

A fent említett ellenőrzések mind használhatók az AppBuilder-ben. Sőt mi több, az ellenőrzések objektum orientáltsága miatt teljesen kiterjeszthetők és alkalmassá tehetők arra, hogy a legtöbbet követelő alkalmazásokban használják fel őket. Röviden, a fent említett működéssel teljesen irányítható vagy ellenőrizhető az adat megjelenése.

Driver technológia

A mi driver technológiánk egy darabban egyesíti a DataFlex 4 adatbázisának viselkedését minden támogatott adatbázisba. Eredményként, a fent említett helyi adatbázisokhoz való Data Management tulajdonságok (pl.: szokatlan, helyzet érzékeny kezelői felület) elérhetők a DataFlex Server-en, Btrieve-en és minden ODBC-t kiszolgáló adatbázison.

2. Egyéb, fontos sajátosságok

Application Framework

A DAF módszert integrálták a DataFlex 4 software-ébe és dokumentációjába. A DAF a Data Access Corporation által javasolt és dokumentált megközelítése a fejlesztői termelékenység maximalizálásának, objektum-orientált alkalmazások készítésére. Ez a bevált módszer jelentősen lerövidíti a fejlesztők tanulási időszakát, amíg az objektum-orientált programozást (OOP) elsajátítják.

A DataFlex Application Framework-el a fejlesztők gond nélkül integrálhatják az alkalmazásaikat a többi felhasználókéval, ezzel téve kezelhetővé és célszerűvé még a nagyszabású csapat fejlesztő-munkákat is.

Platformok közötti fejlesztés/telepítés

A DataFlex nagyfokú hordozhatóságot nyújt az platformok között az alkalmazások számára. Ez azt jelenti, hogy telepíthető egy karakteres (DOS, UNIX vagy OS/2) DataFlex 3 alkalmazás Windows környezetben az interface (pl.: View-k) egyszerű kicserélésével, tükrözve a GUI környezetet. Az AppBuilder-rel ez a feladat könnyen teljesíthető.

Nem kell aggódni az üzleti szabályok miatt, mert a DataFlex robusztus adat könyvtáraival a Windows, DOS, UNIX és OS/2 alkalmazások osztoznak az üzleti szabályokon, tekintet nélkül a használt fizikai adatbázisra (DataFlex DBMS, Sybase, Oracle, dBase, stb.) Ez teszi teljesen hordozhatóvá az üzleti alkalmazásokat. A client/server környezetben ez a magas szintű hordozhatóság biztosítja egy AIX alatt futó karakteres- , egy Windows NT alatt futó karakter módú- és egy Windows 95 alatt futó DataFlex 4 (GUI) alkalmazásnak, hogy elérjék ugyanazt a servert. Sőt, ez teszi lehetővé mindezen alkalmazásoknak, hogy osztozzanak az üzleti szabályokon (business rules).

Adatbázis nyelv

Mint adatbázis fejlesztő nyelv, a DataFlex 4 fényesen vezet a versenyben a nyelv és adat közti nagyfokú integrációval. Továbbá, a DataFlex DBMS record orientáltsága teszi lehetővé az adatfile-ok és index-ek szerkezetének, a hagyományos SQL adatbázisokénál jobb megértését. A record orientációval összekapcsolt integráció sok tulajdonságot nyújt.

Egyszerű telepítés a Client/Server környezetekbe

Az adatkezelőhöz kapcsolódó interface-t úgy építették föl, hogy ugyanazt az alkalmazást módosítások nélkül lehet telepíteni client/server környezetben is.

Programírás nélküli master-detail adatbázis alkalmazások készítése

A DataFlex 4 Application Wizard vizuális eszközeiben teljesen fel lehet építeni egy kész master-detail alkalmazást. Ez a lehetőség azért lényeges, mert a legtöbb üzleti alkalmazás főleg master-detail viszonyokat használ.

Valódi példaprogramok mutatják be, hogy milyen könnyen lehet alkalmazásokat készíteni a DataFlex 4-ben

A DataFlex 4 változatos valódi minta alkalmazásokkal kapható, amik illusztrálják, hogy milyen gyorsan és könnyen lehet robusztus-, vizuálisan vonzó alkalmazásokat készíteni a DataFlex 4 használatával. A minta programok valódi, a fejlesztők által használt, üzleti szabályokat tartalmaznak.

Adatbázis-független Data Dictionary

A Data Dictionary az alkalmazások definiált üzleti- és érvényességi szabályainak központi raktára. A data dictionary használatának sok előnye van: (1) A Data Dictionary-ben központilag vannak a tárolva a mező érvényesítés és az üzleti szabályok. (2) Teljes mértékben támogatja a DataFlex OOP nyelvet, mert ez is egy class. Ez maximális hajlékonyságot biztosít az üzleti szabályok létrehozására. Megengedi azt is, hogy a Windows, DOS, UNIX és OS/2 alkalmazások osztozzanak az üzleti szabályokon. (3) A DataFlex a data dictionary-ben tárolt üzleti szabályokat képes teljesen lefordítani a DataFlex adatbázis szintről a cél adatbázisra (vagy adatforrásra). Ezért lesznek az üzleti

szabályok adatbázis függetlenek. Más szavakkal, a Data Dictionary-ben definiált ü
szabályok minden támogatott adatbázisra alkalmazhatók.

Az alkalmazás viselkedésének jobb megértése az üzenetfigyelésen keresztül

A DataFlex 4 ad egy mindenre felkészített "kém" segédprogramot, amivel a fejlesz
követhetik az objektumok között küldött üzeneteket, figyelhetik a változókat és ki
zéséket, és elemezhetik az objektum- és focus fákat. Az üzenetkövetés azért fontos
lajdonság az objektum-orientált programokban, mert egy átlagos alkalmazás több
üzenetet képes generálni. Az üzenetek figyelése nélkül a fejlesztő nem tudja tel
megérteni az alkalmazás működését.

Hajlékony jelentéskészítő megoldások

A DataFlex 4 két megoldása a jelentések készítésére egy alkalmazásban: a WinQL
WinPrint. A "client" által készített, vagy a könnyen- és gyakran beállítandó jelenté
a legjobb megoldás a WinQL jelentésíró. A "turn-key" alkalmazásokhoz jobb az al
mazásba kódolni a jelentés készítést, mert gyorsabban, kevesebb költséggel fognak
ködni, kevesebb nehézség lesz az alkalmazás beállításában. Erre a célra a legme
lőbb a WinPrint.

A WinPrint fő tulajdonságai:

A DataFlex Data Dictionary-k teljes támogatása

Minden jelentésben vagy batch process-ban a DataFlex objektum-orientált nyel
teljes ereje. Teljes font ellenőrzés. Preview mód Grafikonok, képek és bitmap-ek
nálata.

A DataFlex 4 ad egy help fordítót is, a Windows-alapú alkalmazások Help file-j
készítéséhez.

3. Programozás és szervezés

A DataFlex alapvetően objektum-osztályból építkezik. Alaposztály készletet ad
lesztőknek, melyet tetszés szerint lehet bővíteni, módosítani, testre szabni. Ezek
osztályokból lehet újabb osztályokat készíteni, vagy konkrét objektumokat. Újab
tályt akkor van értelme készíteni, ha azt a programrendszeren belül többször is fel

juk használni. Az adatbázist módosító algoritmusok pl. olyanok, amelyeket a rendszeren belül számtalan helyen használják. Ebből egyértelműen következik, hogy az adatbázist módosító eljárásokat, függvényeket osztályokba kell helyezni. Ezeket egyszer kell definiálni, dokumentálni. Ha valahol speciális esetek adódnak, pl. különleges jogosultság kezelés, akkor az adott konkrét objektumban azt a módosító eljárást ki lehet bővíteni. Lehetőség van a lecserélésre is, de akkor az objektum-osztály szemléletet már megsértem.

Ebből következik, hogy azokat a funkciókat, amiket a procedurális programozási környezetben a programtervezés szintjén kezeltünk általában (a fent említett adatbázis-kezelői szabályok, képernyő formák, stb.), azokat most már a szervezői szintre kell hozni, hiszen az ezekben a kérdésekben hozott döntések mindig rendszer szinten jelentkeznek. A korábbi módszerben az elvárás az volt, vagy inkább megfelelt az a módszer is, hogy programonként határozták meg pl. adatbázis kezelés szabályait, funkciós billentyűk szerepét, stb. Ha most is így járunk el akkor az osztály-alkotás által adott előnyöket abszolút nem tudjuk kihasználni. Ebben az esetben nem csinálunk mást csak egy kicsit másabb programozási eszközt használunk. A munkánk hatékonyságán lényegesen nem változtatunk. Ha viszont még a konkrét programoktól függetlenül meg tudjuk határozni a kritériumokat, akkor ezeket már osztály szinten tudjuk kezelni, és programozás "nem lesz más" (nagyon határozottan idézőjelben persze), mint az osztályok helyes használata.

Ez a jelenlegi gyakorlathoz képest más munkamegosztást, a projectek átfutási idejének belső struktúrájának megváltozását eredményezi. Újra nagyon-nagy hangsúlyt kell hogy kapjon a rendszer-szemlélet.

OBJEKTUM-ORIENTÁLT ALKALMAZÁSFEJLESZTÉS A POWERSOFT LEGÚJABB ESZKÖZEIVEL

Pokó István

AXIS Számítástechnikai Kft.

Objektum-orientált "kliens" oldali eszközök

A kliens-szerver rendszerek hagyományosnak mondható felépítésében a szer relációs adatbáziskezelő fut, és csak a kliens oldalon használhatunk objektum-ori fejlesztő eszközöket. Ezzel a hagyománnyal szakított a Powersoft (a Sybase kliens- fejlesztő diviziója), amikor népszerű tervező- és fejlesztő eszközeinek új ver megalkotta. Egyrészt a relációs adatmodell felépítésénél használhatunk olyan obje orientált fogalmakat, mint például az öröklődés; másrészt az osztott alkalmaz készítésének lehetőségével az objektum-orientált programozás a szerver oldal lehetővé válik.

Az S-Designor 5.1 CASE programcsomag

Az S-Designor programcsomag új verziója jelentősen megnövelte a szoftver-élet támogatását. Az elemzés az adatfolyam-modellezéssel kezdődhet, ezt köv hagyományosnak tekinthető logikai- és fizikai adatbázismodell kialakítás, ma adatbázis-generálás. Ezután kerülhet sor az applikáció-prototípus generálásán, jelenleg PowerBuilder és Visual Basic alá történhet (fejlesztés alatt van a Centu Optima++ és a Delphi változat). A szolgáltatásokat verziókövetés és reverse-engi funkciók egészítik ki.

Az S-Designor négy független, együttműködni képes modulból áll:

- **ProcessAnalist**: Adatfolyam modellezés

  (Négyféle módszertan támogatása: OMT, Yourdon/DeMacro, Gane & S SSADM. Többszintű adatfolyam-ábrák, kétirányú kapcsolat a DataA logikai modelljével)

- **DataArchitect**: Az adatbázis-tervező fő eszköze
  (Logikai- és fizikai adatmodell kialakítása, adatbázis generálás, reverse engineering, kiterjesztett attributumok átadása a kliens oldali fejlesztőknek. Példaként említhető meg - az eszköz objektum orientáltságát bizonyítva - , hogy a táblák definiálásakor lehetőségünk van egy ős definíciójából utódokat létrehozni. Az örököltetés folyamán különböző módszerek között választhatunk a fizikai modell legenerálását illetően úgymint: csak az ős, csak az utódok, mindkettő, az utódok közül csak bizonyos feltételeknek megfelelő stb. táblák létrehozása.)
- **AppModeler**: A programfejlesztő által használt eszköz
  (Fizikai adatmodell készítés, adatbázis generálás, reverse engineering, kiterjesztett attributumok átadása a kliens oldali fejlesztőknek. Az eszközzel lehetőség van alkalmazás prototipus generálására is minta objektumkönytárak használatával.)
- **MetaWorks**: A teammunkát támogató modul
  (Adatmodellek tárolása adatbázisban, hozzáférési jogok hozzárendelése almodellekhez, módosítási igények összehangolása, konszolidálás, verzió-követés)

A PowerBuilder 5.0 fejlesztő eszköz

A kliens oldali alkalmazások fejlesztésének kimagasló eszköze a PowerBuilder 5.0, mely grafikus felületeken (Windows 3.x, Win '95, Windows NT, Macintosh, UNIX/Motif, OS/2 Win) fut, és az objektum-orientált programfejlesztési kritériumoknak mindenben eleget tesz. Az új verzió lényeges újdonságai közül objektum-orientált szempontból az alábbiakat érdemes kiemelni:

- Függvények polimorfizmusa: azonos függvénynevek esetén az eltérő argumentumlista alapján történik a hívott függvény kiválasztása
- Paraméterezhető események
- Függvények hívása az eseményeknél megszokott Trigger illetve Post módon
- Windows alkalmazások integrációja: OLE/OCX/ActiveX, DDE és DLL

- Osztott objektumok alkalmazás-szerverek készítéséhez. E technikával megoldhatjuk a központi szabály-nyílvántartást és karbantartást, valamint szétválaszthatjuk a felhasználói felületet az üzleti logikától. Valójában egy szintű modellt építünk ki, ahol az 1. szint: kliens oldali felhasználói felület, Szint: üzleti szabályok az alkalmazás-szerveren, 3. Szint: adatok az adatbá szerveren.
- PowerBuilder Foundation Class Library: előre elkészítettobjektum osztály és minta alkalmazások a fejlesztés meggyorsítása érdekében.
- Beépített, adatbázis alapú verzió ellenörző eszköz az ObjectCycle: lehetőséget nyújt arra, hogy követni tudjuk az egyes objektumok különböz verzióinak fejlődését (elágazások, összevonások).
- Internet és Intranet alkalmazások készítése:
  - DataWindow előállítása HTML formátumban: igényes, kifinomult adatmegjelenítés
  - PowerBuilder Child-window objektum beágyazása HTML lapokba, melyeket a web kliensek letölthetnek és gépeiken futtathatnak. Lehetőség a különböző alkalmazás verziók dinamikus letöltésére.
  - PowerBuilder alkalmazás-szerver objektumok kapcsolódása a Web serverhez (CGI, NSAPI, ISAPI): dinamikus HTML generálás

Az Optima++ fejlesztő eszköz

A PowerSoft új objektum-orientált fejlesztő eszköze az **Optima++**, melynek célja, ho egyesítse a C++ programozási nyelvek hatékonyságát a grafikus fejlesztőeszközök termelékenységével.

Az eszköz az alábbi főbb tulajdonságokkal rendelkezik:

- Programozás C++ nyelven
- 'Drag and drop' programozási technika
- Java applet-ek készítése
- Komponens-centrikus vizuális környezet, a fejlesztés meggyorsítására
- DataWindow, DataPipeLine objektumok az adatbázis eléréshez
- Integrált, nagy teljesítményű debugger
- Beépített Sybase SQL Anywhere adatbáziskezelő

- Verzióellenőrzési lehetőségek (ObjectCycle, 3. gyártó termékei)
- Előregyártott OCX (ActiveX) kontrolok, OLE szerverek
- Optima++ objektumok beillesztése PowerBuilder alkalmazásokba
- SUN/Netscape, MicroSoft Internet szabványok támogatása

A fentiek is illusztrálják a Sybase/PowerSoft eltökélt szándékát, hogy a rendszerfejlesztés minden fázisában hatékony objektum-orientált eszközöket adjon a fejlesztők és felhasználók kezébe.

Integrált vállalatirányítási szoftvercsomag testreszabása CORBA alapú ObjectBroker segítségével

László István, Digital Equipment Magyarország Kft.

Az ObjectBroker a Digital CORBA alapú elosztott objektum-kezelője. Ez a termék n csak a kereskedelmi forgalomban aratott sikert, hanem a Digital saját belső informá rendszerének a fejlesztésében is hasznos eszköznek bizonyult. Alábbiakban az SAP R vállalatirányítási rendszer ObjectBroker segítségével történő integrálása során szerze tapasztalatokat, valamint magát az ObjectBroker terméket ismertetjük.

Applikációk integrálása a SAP R/3 vállalatirányítási rendszerrel az ObjectBroker segítségével

Saját testreszabott információs rendszer kifejlesztése drága, nehéz karbantartani, továbbfejleszteni. Ennek felismerése vezetett arra, hogy több vállalat - így a Digital i integrált szoftvercsomagot vásárolt a saját fejlesztés helyett.

A vezető integrált vállalatirányítási szoftvercsomagok, így az SAP R/3 is, oly széles területet fednek le, hogy gyakorlatilag kielégítik egy tipikus vállalat igényeit minden üzleti területen. Az R/3 lehetővé tesz bizonyos testreszabást un. konfigurációs táblák segítségével, azonban az R/3 ilyenkor is megőrzi a specifikus applikáció- és adatstruktúráját. Ugyanakkor sok vállalatnál, mint például a Digitalnál is, ezen testreszabás nem elegendő, szükség van ezen applikációk integrálására más szoftvercsomagokkal, meglévő és a továbbiakban fejlesztendő speciális applikációk Néhány példa az ilyen applikációk közötti integráció szükségletre:

- Meglévő (nem R/3) applikációk számára lehetővé tenni, hogy használjon, ill. updateljen adatot az R/3 adatbázisában
- Third-party applikációk integrálása olyan területre, amit az R/3 nem fed le

- Remote grafikus interfészű applikációk fejlesztése amelyek elérhetik az R/3 funkcióit
- A meglévő megszokott képernyőformátumok megőrzése olymódon, hogy valójában ezek közvetlenül integrálódnak az R/3-ba
- R/3 funkció programozása olymódon, hogy az kapcsolatba lépjen egy külső applikációval (onnan információt nyerjen ill. küldjön)

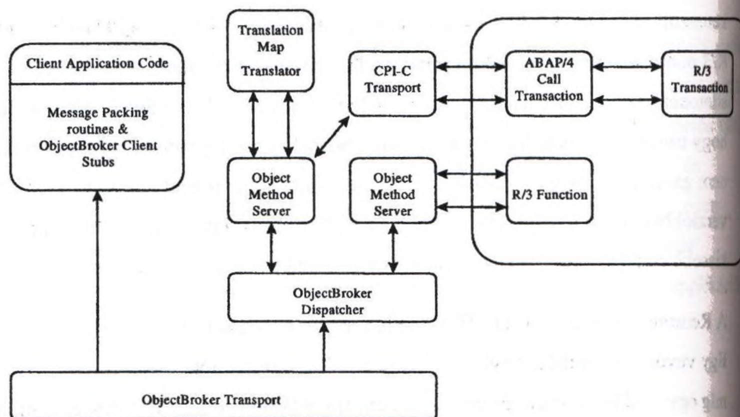
Külső applikáció integrálásához az R/3 két fontos programozható interfészt biztosít: a Batch Data Inputot és a Remote Function Callt. Ezeknek a korlátja az, hogy megkövetelik, hogy olyan környezetben fussanak, ahol az R/3 elérhető. Ha egy olyan applikációt akarunk integrálni amely igényli az elosztott hozzáférést az R/3 adatokhoz és funkciókhoz különféle gyártók rendszereinek heterogén hálózatában, akkor szükségünk van egy további szoftver elemre, amely Digital esetében az ObjectBroker middleware.

A Batch Data Input CALL TRANSACTION metódusa az egyik interfész alapja. Ennél emuláljuk az R/3 felé mintha a mintegy 20000 féle R/3 tranzakció egyike a SAP GUI-n keresztül lett volna bevíve. Ennek előnye a kiszámíthatóság és megbízhatóság, mivel az R/3 pontosan úgy fog működni mintha a felhasználó közvetlenül vitte volna be az adatokat, ugyanazok a security- és adatellenőrzések hajtódnak végre. További előny, hogy ezekre a tranzakciókra támaszkodva biztosított az upgrade-elhetőség. Az SAP nem garantálja, hogy az adatbázis szerkezete változatlan marad az elkövetkezendő verziókban, ezért a magasabb szintű CALL TRANSACTION-on keresztüli hozzáférés kissebb rizikót hordoz mint az alacsonyabb szintű adatbázis hozzáférés.

A Remote Function Call-lal (RFC) az alacsonyabb szintű R/3 funkciókat érhetjük el. Egy vevői megrendelés bevitele például egy R/3 tranzakcióként kerül végrehajtásra, míg egy eladási ár számításának művelete egy R/3 funkció. Az R/3 funkciók mind olvasási, mind update műveleteket is végrehajtanak. Előnyük, hogy gyorsak, egy hívással nagy mennyiségű adat kezelhető. Ugyanakkor ezek veszélyesebbek is lehetnek, ha nem megfelelően hívják meg, mivel ezek kevesebb ellenőrzést hajtanak végre az

inputon és ezért inkonzisztenciát okozhatnak. Ezért sok fejlesztő csak olvasási műveletre használja funkció hívást, az update műveletekre a tranzakció hívást használják.

Az R/3 10000 adatbázis táblával és 20000 tranzakcióval jellemezhető széleskörűsége nem tette lehetővé, hogy statikus "business object"-ek halmazát helyezzük az R/3 szolgáltatások fölé. Ehelyett egy rugalmasabb dinamikus mechanizmusra volt szükség. A Digital implementációjában definiálásra került egy új szolgáltatás halmaz, amely lehetővé teszi, hogy kulcsszavakkal és hozzátartozó értékekkel definiálják a végrehajtandó R/3 szolgáltatást. A kliens oldali applikáció összeállít egy requestet kulcsszó/érték formában, amelyet elküld az object method szerverek egyikének, ahogy az az ábrán látható. A megfelelő szerver ezután leképezi a requestet vagy egy generic tranzakció hívásra, vagy egy R/3 funkció hívásra. A kulcsszó/érték pároknak megfelelő tranzakcióra vagy funkcióra történő leképezés történhet vagy magában az object szerverben, vagy egy külön fordító komponens (Translation Map) hajtja végre. Ezáltal az R/3 egy objectummá válik, amelyen értelmezhető a "do transaction" és a "do function" method.



A megoldott problémák közül két példát ragadunk ki. Az egyikben a Digital információt szolgáltat a felhasználóknak és a partnereknek az Interneten ill. a World Wide Weben keresztül. Az egyik legfontosabb ilyen információ az up-to-date ár és

rendelési státusz. A WWW interface dinamikusan változott, ennek ellenére az ObjectBrokerrel sikerült a megfelelően rugalmas integrációt megvalósítani.

Egy másik példában az R/3 a kliense egy másik applikációnak. A Digital különféle országokba szállít termékeket, és ehhez un. "transzfer árat" kell kalkulálni vám és adó célokra. A TPAS (Transfer Price Application Server) applikáció hajtja végre ezeket a számításokat egy OpenVMS rendszerben, míg az R/3 Digital UNIX alatt fut. Ezért egy multiplatformos megoldásra volt szükség. Az ObjectBrokert használván itt az object lett a "TPAS" és egy method pl. a "GetPrice". A TPAS ezáltal egy multiplatformos árazó szerverré vált anélkül, hogy további portolásra lenne szükség.

ObjectBroker

Az ObjectBroker volt az első kereskedelmi forgalomban megjelenő termék amely az Object Management Group (OMG) által lefektetett szabványon, a Common Object Request Broker Architecture-n (CORBA) alapult. Az ObjectBroker 1991-ben jelent meg a piacon, ekkor még Application Control Architecture Services (ACAS) néven, tükrözve, hogy a gyökerei applikáció-integrációból és -vezérlő mechanizmusból származtathatók. Az ACAS korábbi verzióját ugyanis egyes szoftver termékek fejlesztésében már 1989-ben felhasználták.

A Digital az OMG rendelkezésére bocsájtotta a CORBA specifikációhoz az ACAS néhány kulcsfontosságú komponensét, így pl. a Dynamic Invocation API-t, a context object és a distributed class repository használatát. Mindezen CORBA jellemzők a Digital által szolgáltatott technológián alapulnak, és a Digital továbbra is részt vesz a OMG nyitott szabványainak specifikálásában.

A megfelelés a nyitott szabványoknak rendkívül fontos a Digital számára, de ugyanilyen fontos, hogy többletszolgáltatásaival segítse sikerre a felhasználót. Így az ObjectBroker kiemelkedik a sok felhasználónál bizonyított használhatóságával, skálázhatóságával, PC illeszthetőségével és a legszélesebb hardver és operációs rendszer platform lefedésével.

Használhatóság

Az ObjectBroker rendelekzik olyan a CORBA specifikáción túli többletszolgáltatá
mint a vezérelhető szerver kiválasztással, amikor több szerver áll rendelkezésre, vagy
pl. eszközöket szolgáltat amelyek lehetővé teszik meglévő hagyományos rendszerei
beintegrálását.

Így például lehetőség van mind a kliens (azaz a felhasználó) mind a szerver
jellemzőinek regisztrálására. Ha pl. egy applikáció pénzügyi átutalásokat végez, al
lehetőség van arra, hogy egy nagyértékű átutalás egy speciális biztonsággal ellátott
szerveren hajtódjon végre, vagy pl. bizonyos napszakban vagy bizonyos kliensről
műveletek le legyenek tiltva.

Természetesen minden ORB lehetővé teszi új elosztott O-O rendszer kifejlesztését
nagyon kevés szervezet engedheti meg magának a "zöldmezős" megközelítést, ami
a meglévő applikációit és hardver eszközeit kidobhatja. Az ObjectBroker rendelke
olyan tool-okkal amelyek lehetővé teszik olyan hagyományos applikációk integrál
amelyeket soha nem terveztek integrálásra, sőt még hívható API-val sem rendelke
Az ObjecBroker évek óta sikerrel valósítja meg az "object wrapper" technikát, ahol
hagyományos applikációt beborítjuk egy objectum burkolattal.

Az ObjecBroker használhatóságát a piac honorálta:

"Standish Group International Inc. tanulmánya alapján a Digital Equipment Corp.
vezeti 33%-os részesedéssel a CORBA alapú object middleware termékek tavaly
millió dollárosra becsült piacát. Iona Technologies a következő 25%-kal, őt követi
IBM (17%), Expersoft Corp. (14%), PostModern Computing Technologies (6%),
valamint a Hewlett-Packard és az ICL Ltd. (mindkettő 3%)." (Object Magazine 19
Augusztus 12. old.)

Skálázhatóság

A skálázhatóság azt jelenti, hogy a szoftver jól működik akkor is, ha az elosztott
rendszer mérete növekszik. Sok szoftver jól működik kevés számú felhasználóval
használhatatlanná válik nagy számú node-nál.

Az ObjectBroker üzemel sok olyan felhasználónál, ahol több száz node működik együtt. Néhány felhasználónál ezernél több node van, és egy felhasználónál a node-ok száma meghaladja a 10000-et.

Hardver és operációs rendszer platformok támogatása

Az ObjectBroker támogatja a legtöbb (több mint 20) platformot az összes ORB közül. Fut az összes népszerű UNIX platformon (pl. AIX, Digital UNIX, HP-UX, Solaris, SunOS), csakúgy mint mainframe-en (MVS), mini gépeken (OS/400, OpenVMS), Intel gépeken (MS Windows, Windows NT, OS/2) stb.

PC illeszthetőség

A teljes vállalati integrációt megcélozva, és felhasználva a Digital/Microsoft együttműküdés biztosította előnyöket az ObjectBroker magaszintű PC illeszthetőséget nyújt az OLE Network Portal-lal, amely lehetővé teszi, hogy PC applikációk módosítás nélkül elérjék a CORBA szervereket.

Most, hogy az OMG specifikálta a szabványos OLE-to-CORBA interfészt, egy második generációs illesztő termék kerül kifejlesztésre, a Desktop Connection™. A Desktop Connection azon túl, hogy az OMG szabványnak teljesen meg fog felelni, további olyan szolgáltatást fog nyújtani, ami lehetővé teszi, hogy módosítás nélküli távoli CORBA objecteket használjunk ugyanúgy, ahogy ma használjuk a lokális OLE objeckteket.

A Desktop Connection része a Digital AllConnect programjának, amelynek célja, hogy fejlessze az együttműködést a Windows NT, OpenVMS és a Digital UNIX rendszerek között. Az ObjectBroker és az AllConnect biztosítja, hogy tetszőlegesen kevert platformok esetén is az applikációk, adatok transzparensen integrálhatók.

Kódgenerálás az *ObjectTeam* CASE eszközzel

Balogh Kálmán

Informix Technology Center Hungary
1063 Budapest
Bajnok u. 13.
Tel: (06-1)-302-33-88/117
Fax: (06-1)-302-33-95

E-mail(Internet): kbalogh@informix.hu

Az *ObjectTeam* a legnagyobb CASE technológiára szakosodott cég, a *Cayenne* objekt orientált, a szoftverek teljes életciklusát támogató terméke. A cég és termékeinek bemutatása után az ObjectTeam kódgenerálási képességeit tekintem át.

A Cayenne CASE eszközei

Februárban jött létre a Cayenne cég az amerikai *Cadre* és a *Bachmann* egyesülésével; koncentrálódás a második lépcsője annak a folyamatnak, amelynek során egy évvel kor a Cadre és a holland *Westmount* egyesült. A burlingtoni központú Cayenne tagvállalat együttes éves árbevétele 70 milló dollár. Ezzel az eredménnyel a világ 50 legnagy szoftveres vállalata közé tartozik, egyben a legnagyobb, amely a CASE technoló szakosodott. A Cayenne-t létrehozó két cég összesen 50 ezer licencet adott el. A fejle Amerikában, Hollandiában és Indiában folyik. A Westmountot az *InTeC* elődje, az Op képviselte Magyarországon, de a többi terméknek nem volt magyaroszági "helytartója"; ezután az InTeC forgalmazza a teljes Cayenne termékskálát.
A Cayenne termékei közül a hagyományos, strukturált módszertanon alapuló *VantageTeam* az objektumorientált *ObjectTeam* mind technológiai, mind Informix-támogatás szempont testvérek. A teljes életciklust átfogó, ügyfél-kiszolgáló architektúrájú CASE eszköz adatbázis alapú információs rendszerek tervezhetők és hozhatók létre. A szerver platform vagy valamilyen UNIX, vagy WindowsNT lehet; az előtéri eszköz pedig UNIX Windows NT-n vagy Windowson futhat.
Bár a Cayenne-nek vannak más CASE eszközei is - így a TeamWork hagyományos elsősorban valós idejű termelésirányítási rendszerek készítésére való; az Ensemble visszafejti a kódot, a GroundWorks adatbázis-modellezésre, a Terrain pedig adat

tervezésre, -generálására és visszafejtésre használható -, az InTeC kínálatában a VantageTeam és az ObjectTeam áll a fókuszban.

Referenciaként említethetjük a Hungarocamiont és az APEH-et, utóbbinál elsősorban a VantageTeamet használják. Az oktatási intézmények közül például a Pénzügyi Főiskolán és a Műszaki Egyetemen tanítják a Cayenne CASE eszközeit, de maguk az InTeC is használja a Cayenne-t néhány projektjénél. A Windows NT-s MS SQL Serverrel működő változat támogatására a közeljövőben új partnert von be az InTeC. Jelenleg két-három munkatársa foglalkozik - nem teljes munkaidőben - a CASE eszközök támogatásával és értékesítésével, hosszú távon azonban perspektivikus területnek tartják a CASE technológiát, amelyre tapasztalataik szerint egyre inkább megérik a magyar piac.

A VantageTeam és az ObjectTeam kódgenerálási képességei

Nyitott kódgenerátoraik nemcsak Informix, hanem Oracle, CA/Ingres, Sybase serverre is generálhatnak SQL kódot, beleértve tárolt eljárásokat. Ami a front-end oldalt illeti: az ObjectTeam a NewErára, az Informix alkalmazás particionálást lehetővé tevő objektumorientált, grafikus előtéri eszközére is előállíthat kódot, beleértve beágyazott SQL utasításokat, és összekapcsolódhat a NewEra fejlesztőrendszerrel. Hasonló képességekkel rendelkezik a strukturált módszertant támogató VantageTeam az INFORMIX-4GL-re vonatkozólag. A harmadik generációs programozási nyelvek közül a C-re, C++-ra, Adára, Javára, Corba IDL-re, SmallTalkra és Visual Basicra vonatkozóan képes SQL utasításokat generálni a két CASE eszköz. A kódgenerátorok nyitottak, úgynevezett TCL nyelven készülnek, és forrásszinten, dokumentációval együtt rendelkezésre állnak. A TCL felület révén a kódgenerátorok és a repository (modelladatbázis) is módosíthatók. Ez kiegészül a CASE eszköz felhasználói felületének testreszabhatóságával, és ezek együtt teszik lehetővé az integrálást más fejlesztőkörnyezetekkel, döntéskövető (DOORS) és dokumentációkészítő eszközökkel (FrameMaker, Interleaf, Word) is.

Generálás szempontjából a következő képességek érdekesek leginkább:

- a generálás eredménye: front-end és back-end kódok, célnyelvek
- a generálás lépései, eszközei (a CAD diagram, a TCL nyelv, template-ek)
- a generálás befolyásolása, testre szabása
- az újrafelhasználás lehetőségei
    - az újragenerálás módja
    - verzió és konfigurációkezelés
    - osztálykönyvtárak használata.

WWW objektumok kezelése:

Web fejlesztés ObjectForms fejlesztõ eszközzel

Nyilas István, Ujfalussyné Nagy Gyöngyvér

HiCare Kft.

Napjainkban, a WEB alkalmazások rohamos terjeszkedésével egyre nagyobb igény mutatkoz[ik az] adatbáziskezelõk alkalmazására ezen a területen is. A jelenlegi WEB alkalmazások túllépnek [a] statikus, egyirányú hypertext határain. A WWW lapokat sokszor dinamikusan állítjuk elõ és [a] számítógép elõtt ülve nem csak nézegethetünk egy távoli adatbázist, hanem akár adatokat [is] módosíthatunk benne. Például hirdetést adhatunk fel az állásközvetítõ adatbázisába, vagy rendelhetünk egy "elktronikus áruházból". Ezek mögött a szolgáltatások mögött általában valamilyen adatbáziskezelõ és adatbázis alkalmazás húzódik meg. Azonban az adatbázis WEB-hez való illesztésével sem a lapok megjelenését meghatározó HTML nyelv, sem [a] kommunikációt leíró HTTP protokoll nem foglalkozik érdemben. Ezzel szemben több feltétel[t] is támaszt az alkalmazással szemben a megjelenítés, az adatbevitel, a paraméter átadás, és [az] eljárások indítása terén.

Bizonyos szabványok hiánya, és az adatbáziskezelõk sokszínûsége miatt rengeteg különböz[õ] megoldás született az adatbázisok WEB-hez történõ illesztésére, de ezek általában csak egy egy adott WEB-szerverhez, adatbáziskezelõhöz - jobb esetben adatbázis típushoz - tartozna[k]. Adatbázis elhelyezése egy WEB szolgáltatás mögé nem is olyan egyszerû. Röviden tekintsük [át] a fõbb követelményeket, amiknek egy jól átgondolt WEB-es adatbáziskezelõ alkalmazásnak meg kell felelnie:

(1) A WWW egyik legnagyob lehetõsége a multimédiás alkalmazásokban rejlik: kép, hangátvitel, video, különbözõ formájú statikus és dinamikus egyedi anyagok, programok, Java alkalmazások, stb. folyamatosan bõvülõ serege nyüzsög a WEB oldalakon. Az ad[atok] és elõállításuk sokfélesége a szerver oldalon meghaladja egy operációsrendszer [...] adattároló és rendszerezõ képességét. A hagyományos RDBMS-ekkel is igen erõltetett[nek] tûnik a rengeteg különbözõ típusú adat menedzselése. Természetesebbnek tûn[ik] valamilyen objektum-szemléletû megközelítés.

(2) Az adatok tárolásán és előhívásán kívül komoly feleadat azoknak WWW lapokká töténõ alakítása és megjelentetése a WEB szerveren segítségével. A felhasználói felület lényegében adott: a HTML szabavány a lapok megjelenését, a HTTP pedig a felhasználó - WEB szerver kommunikáció formáját határozza meg. A felhasználó böngészése során kiadott üzeneteket valahogyan tolmácsolni kell az adatbázis-alkalmazás felé, a válaszul kapott adatokat pedig össze kell gyúrni a WWW lapok megjelenését meghatározó HTML nyelvvel, hogy valami élvezhetõt lássunk a túloldalon. Ennek a feladatnak az egyszerû adatbázis lekérdezõ nyelvek (pl. SQL) legtöbbször nem felenek meg. Általában valamilyen procedurális nyelvi bõvítésre bízzák az input/output feldolgozásának feladatát (Oracle-OCI,PL/SQL), és/vagy megpróbálják "eldugni" az adatbázis kezelõ WEB szerverhez való integrálásával (Navi Server-Illustra). A HTML nyelv és a HTTP protokoll folytonos fejlõdésével azonban nehéz lépést tartani. A legújabb slágerek (most pl. a realtime hangátvitel, Java) támogatására a termékfüggõ megoldásokban sokat kell várni. Ezért a WEB illesztés többletmunkáját leggyakrabban még mindig a jó öreg C, Perl vagy valamilyen más, sokkal általánosabb és szabadabb nyelv végzi. Ezek mellet szól az is, hogy a WEB-es alkalmazások nem kizárólag adatbázis-alkalmazások. Az adatbáziskezelõ nagyon jól használható sok feladatnál, de alapvetõen csak *egy* a sok integrálandó alkalmazás közül. Ezek után miért pont egy adatbáziskezelõ nyelvre bíznánk a feladat legnagyobb részét?

(3) Bár logikailag az elõzõ problémakörhöz tartozik, és az adatbáziskezelõket csak ugyanannyira annyira érinti, mint a WEB alkalmazás bármely más résztvevõjét, mégis érdemes néhány szót vesztegetni az úgynevezett "session-less" problémára. Ez ugyanis egy fontos probléma a komplex WEB alkalmazások írásánál, és elsõ megközelítésre igen gyakran elsiklunk felette. A WEB szerver lényegében nem csinál mást, mint folyamatosan válaszolgat a hozzá érkezõ lap-kérésekre. Hiába nézegetjük azonban csak *egy* adott szerver lapjait, az újabb és újabb lapok lekérdezésekor mindig új vendégként kezel minket a WEB szerver. Az elõzõ kérdéseimre - és fõleg az elõzõleg lekért lapok tartalmára - már nem emlékszik, és lehet, hogy a két utolsó kérésem között még huszonkét másikat is kiszolgált. A WEB lapok nézegetése a (WEB-ezés) tehát nem "bejelentkezés" (session) jellegû. Pedig a felhasználó oldaláró nézve igencsak annak kellene lennie. Egy adatbázis mûvelet elvégzésekor lehet, hogy szükség van egy elõzõ mûvelet eredményére, vagy akár gondoljunk csak egy WEB-es bevásárlás során a virtuális bevásárló kosarunkra. Ki tárolja

annak adatait? És honnan derül ki, hogy az az adathalmaz kié? Bár erre a problémára folyamatosan mindenféle megoldás születik (a HTTP protokoll session jellegû bõvítésétõl kezdve a Netscape-cookie-ig), nyilvánvaló, hogy a feladatot SQL-lel és egy különálló adatbáziskezelõvel elég nehéz lenne megoldani. Ez a probléma is azt mutatja, hogy az adatbázis-kezelést úgy kell integrálni a WEB alkalmazásba, hogy megfelelõ szabad tér maradjon a legkülönbözõbb feladatok végrehajtására.

Foglaljuk röviden össze:

A WEB-es alkalmazások az adatbáziskezelõkkel szemben speciális követelményeket támasztanak. Ezek közül két nagyon fontos a következõ:

- A WEB-en használt adatok sokfélesége miatt célszerû olyan adatbáziskezelõt használni, amely a legkevesebb plusz munkával tudja a különbözõ típusokat tároni, típusonként más és más módon menedzselni (keresni, megjeleníteni). Erre a feladatra legjobbnak valamilyen objektum alapú adatbáziskezelõ típus tûnik: ORDBMS-t, vagy OODBMS-t.

- Egy komplex WEB alkalmazásnál az adatbáziskezelõ és adatbázis-alkalmazás legtöbbször csak egy a sok integrálandó alkalmazás közül. Az adatbáziskezelõnek megfelelõ teret kell nyújtani a nem adatbázis jellegû feladatok végrehajtására is (ide értve a HTML és HTTP alapú megjelenítést). Bár elképzelhetõ, hogy a teljes WEB alkalmazás valamilyen jól megválasztott adatbáziskezelõ nyelven van írva; azonban célszerûbbnek látszik, hogy inkább valamilyen általános nyelvbõl érjük el az adatbázist. Ez az a pont, ahol a WEB alkalmazások számára az OODBMS elõnybe kerül az ORDBMS-sel szemben. Az OODBMS-ek ugyanis valamilyen általános objektumorientált nyelvhez szervesen integrálhatók és az adatok elõvarázslása nem külön feladattá, hanem az objektum orientált adattípusok használatának természetes velejárójává szelídül.

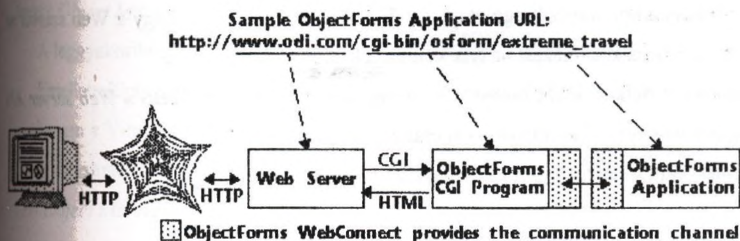
Ezért a Web alkalmazások adatainak tárolására sokkal természetesebb megoldásnak tûnik egy objektum orientált, mint egy relációs adatbáziskezelõ. Az *ObjectStore*, mint a világ vezetõ objektum orientált adatbáziskezelõje, alkalmasnak látszik erre a célra.

Az Object Design, az *ObjectStore* kifejlesztője, egy komplett megoldást kínál dinamikus, interaktív Web alkalmazások készítésére. Az *ObjectForms* fejlesztőeszköz az *ObjectStore* adatbázisban tárolt adatok megjelenítését és kezelését teszi lehetővé tetszőleges Web browseren keresztül, míg az *ObjectStore Extended Object Management Suite* az új, multimédiás adattípusokat támogatja.

Az ***Object Forms***

- egyrészt kommunikációs csatornát biztosít a Web szerver és az *ObjectStore* alkalmazás között, *(WebConnect)*
- másrészt az *ObjectStore* adatok elérését és HTML formázását biztosítja. *(ObjectForms Engine)*

Az *ObjectForms* fő alkotórészeinek helye és szerepe a Web alkalmazásban egy példa URL-en keresztül az 1. ábrán látható.

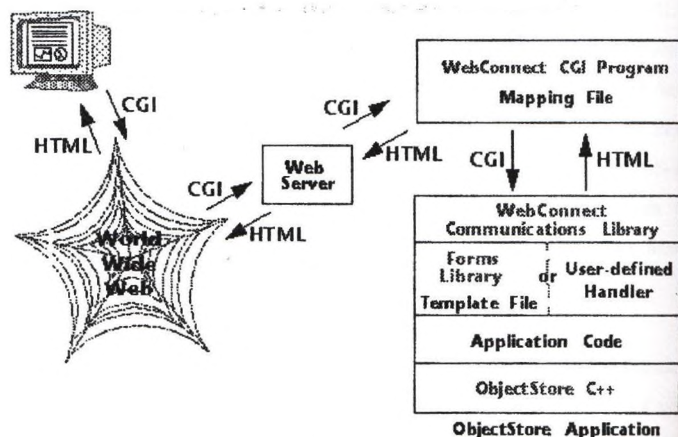


1. ábra

Egy *ObjectForms* alkalmazás a következõ lépésekbõl áll össze:

1. Az adatbázis elérést, kollekciókat és lekérdezéseket megvalósító *ObjectStore* (C++) alkalmazás megírása.
2. Az *ObjectForms* service, azaz a tényleges Web alkalmazás elkészítése, amely a fenti *ObjectStore* alkalmazás adatait a *WebConnect* CGI program számára elérhetõvé teszi. Lényegében ez is egy *ObjectStore* alkalmazás *ObjectForms API* kiegészítésekkel.
3. A megjelenítendõ adatok HTML formázására két lehetõség van. Az elsõ, és igen kényelmes megoldás a templétek definiálása. Az ObjectForms templétek lényegében HTML file-ok néhány fix formátumú taggal és lekérdezésekkel kiegészítve. Ezeket a tagokat a templét processzor behelyettesíti az adatbázisból a lekérdezés eredményeként kapott Web objektumokkal. Ha a templétek nyújtotta keret nem elég, lehetõség van saját "User-defined Handler" írására is.
4. Az alkalmazásokhoz tartozik egy mapping file, ahol definiálni kell, hogy a Web szerver az adott ObjectForms alkalmazást melyik kommunikációs csatornán éri el.
5. Opcionálisan a default *WebConnect* CGI progam (*osform*) lecserélhetõ a *Web server API* segítségével írt saját, a kommunikációs csatornát biztosító CGI-re.

A fenti elemek kapcsolatát a 2. ábra szemlélteti.



2. ábra

Egy Web alkalmazás fejlesztése során tehát két lényeges - akár párhuzamosan is végezhetõ - feladat van.

Az egyik az adattartalom kinyerése az adatbázisból. Ez az ObjectForms API-jával leírt ún. "callback" függvényekkel történik, többnyire kollekciókon keresztül.

A másik feladat az adatok HTML formájú megjelelenítésének tervezése. Ez végezhetõ el a templétekkel, ahol a lekérdezések valójában a "callback" függvények meghívását, a helyettesíthetõ tagok pedig ezek kimenõ attribútumaira történõ hivatkozást jelentik.

A Web alkalmazások készítését segítõ másik eszköz az *ObjectStore Extended Management Suite*. Ez egy könyvtárkészlet, amely lehetõséget biztosít Web objektumok definiálására, tárolására és kezelésére ObjectStore adatbázisban. A könyvtárkészlet jelenleg az alábbi modulokat tartalmazza:

- *Image Object Manager és Image Analysis Manager*

  A leggyakoribb formátumú képek (jpg, gif, tiff, stb.) adatbázisbeli tárolását biztosítja. Emellett konverziós rutinokat is tartalmaz, és a képek nemcsak attribútum szerint, hanem a Virage Inc. képfeldolgozó eszközének integrálásával minta és tartalom szerint is kereshetõk.
- *Text Object Manager*

  Hosszú szöveges objektumok tárolására szolgál. A tárolt objektumokon a Verity Inc. Topics eszköze integrálásának köszönhetõen hatékony szabad szöveges keresést tesz lehetõvé.
- *HTML Object Manager*

  Elõre megformázott HTML oldalak tárolására szolgál.
- *Video Object Manager*

  Video klippek tárolására használható. A klippek mellé eltehetõk metaadatok, mint pl. a szerzõ, vagy a készítés dátuma.
- *Audio Object Manager*

  Hasonló célt szolgál, mint a Video Manager.
- *Java Object Manager*

  HTML oldalakba integrálható Java appletek tárolhatók metaadataikkal együtt.
- *Spatial Object Manager*

Térbeli objektumok (pl. pont, kör, poligon, felszín, stb.) kezelését, tárolását visszakereshetõségét valósítja meg. Az alpvetõ osztályokból tetszõleges bonyolultságú térbeli objektumok építhetõk. Fõleg GIS ill. CAD jellegû alkalmazások készítését segítheti.

Összegezve, az ObjectForms fejlesztõ eszköznek két nagy elõnye van:

- az adatkezelés és a megjelenítés élesen elválasztódik a templétek használatával;
- az újabb, bonyolult objektumok integrálása a teljesen objektum orientált megközelítés miatt könnyedén megtehetõ.

A Software through Pictures OO CASE eszköz

Frigó József, Hontvári József, Kelen András

Bevezető

Az informatikai fejlesztés köztudomásúlag a magas kockázatú tevékenységek közé tartozik. Az önálló szoftverfejlesztő cégeknek éles versenyben kell helyt állniuk, de a más típusú szervezetek részeként dolgozó informatikusoknak is folyamatosan kell az újabb és újabb kihívásokkal szembenézniük.

A kihívásokra adott lehetséges válasz a hatékonyság növelése és a minőség fejlesztése. A minőséget mind a termékre, mind a szoftverfejlesztési projektre vonatkoztathatjuk.

A hatékonyság növelése és a minőség fejlesztése, mint célok összefüggenek egymással. A szoftverfejlesztés mindenképpen munkaigényes folyamat. A minőségbiztosítás ugyanakkor ezen felül is sok adminisztrációval jár, és elvonja az időt az effektív munkától. Nem szabad figyelmen kívül hagynunk az emberi tényezőket sem, a megnövekedett adminisztráció kedvezőtlenül hat a munkakedvre.

Mindkét cél eléréséhez hozzásegíthet a CASE eszközök alkalmazása. Ezek az eszközök kiváltják a kézimunkát és hozzásegítenek egy jól definiált technológia kialakításához és alkalmazásához. Mindezeken túl hozzásegítenek ahhoz is, hogy a fejlesztők kiélhessék kreativitásukat, élvezzék a munkájukat.

Előadásunk a CASE eszközök alkalmazásáról szól egy objektum-orientált technológiát alkalmazó projektben, különös tekintettel a Software through Pictures CASE eszközre.

CASE eszközök alkalmazása a szoftver életciklus egyes lépéseiben

Elemzés

Jacobson use case (használati eset), forgatókönyv (scenario)

A Use case Ivar Jacobson által kifejlesztett modellezési technika, amely az általa kifejlesztett módszertan egy eleme (az Objectory módszertan az Object-Oriented Software Engineering című könyvében van leírva). A technika lehetőséget teremt rendszerrel történő magas szintű interakciók leírására. Tipikusan azokat a tranzakciókat (és a hozzájuk kapcsolódó eseményeket) tartalmazza, amelyeket a rendszer egy felhasználója kezdeményez a rendszernek egy meghatározott célú használata során. A use case eszközt biztosít az új vagy régi rendszerekkel szemben támasztott követelmények megértésére, és különösen előnyös abból a szempontból, hogy a leírásra használt jelölésrendszer egyszerű, könnyen megérthető. Ez azt is jelenti, hogy a *nem* fejlesztők és a végfelhasználók is közvetlenül részt vehetnek a rendszer elemzésében, továbbá, hogy a use case felhasználható a BPR korai fázisaiban.

Az StP lehetőséget teremt egy use case dekomponálására, amelynek során a use case-hez egy forgatókönyv halmazt rendelünk. A forgatókönyv az OMT Event trace jelölésével írható le. Például az, egy ATM rendszerben alkalmazott „Pénzfelvétel" use case-hez a „Sikeres tranzakció", „Nincs elég fedezet" és a „Napi limit túllépése" forgatókönyveket rendelhetjük hozzá. A forgatókönyvek segítségével előzetesen azonosíthatjuk a rendszer implementálásához szükséges osztályokat, továbbá segít azonosítani az eseményhalmazt, amelyre az osztályoknak válaszolniuk kell. Az így azonosított osztályok felhasználhatók az Osztály diagramok elkészítéséhez, az események pedig a Dinamikus modell diagramokhoz.

A Use case technika olyan eszköz, amelyet felhasználhatunk egy előzetes rendszer elemzés során. Ezen felül azonban másra is szükségünk van. Egy előzetes elemzés (megvalósíthatósági tanulmány) után, amikor az elemzés során azonosítjuk és felsoroljuk a rendszerrel szemben támasztott követelményeket, szükségünk van valamilyen módszerre, hogy követhessük, finomítsuk őket, továbbá hogy hozzárendeljük rendszerünk azon komponenseihez, amelyek kielégítik a megfelelő követelményt.

Követelmény követés

Bár sem az OMT, sem Booch nem ad semmilyen mechanizmust, mindkét módszertan szerzője felhívja a figyelmet, hogy milyen fontos a követelmények — vagy ahogy a Businnes Process Reengineering (BPR) gyakorlatban mondják „businnes rules" — helyes azonosítása és annak biztosítása, hogy az elkészülő rendszer kielégítse az azonosított követelményeket. Ezen felül, mivel a rendszerkövetelmények finomodnak ahogy az életciklusban az elemzéstől a tervezés felé haladunk, és a felhasználótól érkező visszacsatolás miatt, fontos a követelmények fejlődésének követése, hogy megértsük egy követelmény okát.

A követelmények kulcsszerepet játszanak bármely szoftver rendszer kifejlesztésében.

Annak érdekében, hogy biztosítsa a követelmények követését, az StP egy speciális követelmény tábla szerkesztőt tartalmaz. Ezen felül ismertetjük az RTM (Requirements Traceability Management) eszköz által nyújtott technikákat, amelyek különösen bonyolult követelményhalmazok kezelésére is alkalmasak.

Logikai, fizikai tervezés

Az elemzés után továbbhaladhatunk például az OMT objektum modell technikájával, amely egyszerű, tiszta, a figyelem középpontjában a domain modellezéssel és a kapcsolatokkal.

A következő lépés lehet a Dinamikus modell elkészítése.

Booch Objektum és interakció diagramjai segítségével leírhatjuk az alkalmazás-specifikus / kritikus forgatókönyveket.

Ha a rendszer komplexitása miatt szükséges, a Booch modul és Processzor diagramok lehetőséget teremtenek a rendszer fizikai vetületének modellezésére.

Az előadás során bemutatjuk azokat a szolgáltatásokat, amelyeket az StP a fenti lépések elvégzéséhez ad. Az StP nem írja elő, hogy a fenti technikákat kell alkalmaznunk, ezek a lépések célszerűnek látszanak, de a projekttől függően szükséges lehet más technikák alkalmazása is, ezért megmutatjuk azt is, hogyan teszi lehetővé az StP az OMT, Booch, Jacobson és az StP specifikus technikák keverését.

Tesztelés

Az előadásnak ebben a részében az automatizált tesztelésről fogunk beszélni.

Az automatizált tesztelést a következőképpen definiáljuk:

„Az automatizált szoftver tesztelés egy definiált, megismételhető, mérhető folyamat, amelyet eszközökkel hajtunk végre, hogy kvantitatívan kimutassuk: a szoftver a specifikációnak megfelelően, vagy nem annak megfelelően működik."

Az automatizált szoftver tesztelés az StP/T-vel négy lépésből áll:

- Tesztelhető modellek előállítása
- Teszt-esetek generálása
- Teszt-esetek végrehajtása
- Tesz-esetek kiértékelése

A tesztelés automatizálása több szempontból segíti a szoftverfejlesztést. Egyrészt javítja a szoftverfejlesztési folyamatot. Azáltal, hogy a teszt-eseteket az elemzési és tervezési modellekből generáljuk, biztosítjuk, hogy a rendszert pontosan az eredeti specifikációval szemben teszteljük. Ennek eredményeképpen azt a szoftvert fogjuk átadni, amelyet a végfelhasználók vártak (és nem mást). Az automatizált tesztelés növeli a hatékonyságot is, a tesz-esetek automatikus generálása rengeteg munkaidőt megspórol, így a tesztelők az unalmas munka helyett a lényeges dolgokra koncentrálhatnak, mint a tesztelés tervezése és a tesz eredmények kiértékelése. Miután a teszt-eseteket a rendszertervből generáljuk, a rendszer tesztelése korán elkezdődhet, és a terv változása esetén a teszt-eseteket automatikusan tudjuk újrageneráltatni. Az automatizált tesztelés fejleszti a minőséget is, az automatikusan generált teszt-esetek ugyanis garantáltan végrehajtatnak minden műveletet, különös tekintettel a legvalószínűbb hibákra.

Az előadásban bemutatjuk, mi a kapcsolat a rendszerterv és a tesz-esetek között, hogyan lehet a tesztelés fenti lépéseit automatizálni,

Implementáció

Bemutatjuk a CASE eszköz és a különböző fejlesztőeszközök kapcsolatát, továbbá egy objektum-orientált terv implementálási lehetőségeit objektum-orientált és relációs adatbázis-kezelővel.

További lényeges jellemzők

A CASE eszközöknek a szoftver életciklus előző, klasszikus lépéseinek segítésén túl más jellemzőkkel is bírniuk kell. Így beszélünk majd a szűk értelemben vett verzió-kezelésről, a konfiguráció menedzsmentről, az ületi rendszer elemzésről.

Összefoglalás

A CASE eszközök sokat lendíthetnek a hatékonyságon és a minőségen, de nem csodaszerek. Megvásárlásuk nem kis beruházás, amely csak akkor térül meg, ha megfelelő hangsúlyt fektetünk a bevezetésükre és az egyéb tényezőkre. Ha ezt sikeresen végezzük, a fejlesztés hatékonyabb lesz, biztosítjuk a folyamatos minőségfejlesztést, és nem utolsósorban érdekesebbek lesznek projektjeink.

Vezetői Információs Rendszerek fejlesztése OO módszerrel

(Fejér Gábor, SAS Institute)

Bevezetés

A kiszámíthatatlan gazdasági helyzetben a pontos és időben kapott információ sikerről vagy bukásról dönthet. A különböző vállalatoknál a döntéshozóknak egyre nagyobb az igényük a döntéshozatal alapjául szolgáló információkra. Egy adattárház olyan *tudást* biztosíthat számukra, mely a különböző, izolált forrásokból származó adattömegből kinyert releváns információn alapul.

Mivel az adattárház illetve általánosabban az információ szolgáltatás szempontjai még nem olyan elfogadott fogalmak, mint mondjuk az objektum orientáltság, és az ezen a területen tevékenykedő cégek sem olyan közismertek, az előadás az alábbi öt részből fog állni:

1. *SAS Institute bemutatása*
2. *Vállalati információ szolgáltatás (Különbség az üzemeltető és információ szolgáltató rendszerek között)*
3. *A SAS adattárház*
4. *OO vonatkozások*
5. *Összefoglalás*

Előadásunk nem egy OO alapkurzus, - az objektum orientált alaptudást feltételezzük, - a hangsúlyt inkább az adattárház építésének szükségességére, és feltételeire helyezzük, - természetesen OO eszközökkel.

SAS Institute

A magánkézben lévő, 1976-ban alapított SAS Institute Inc. egyike a világ tíz legnagyobb független szoftver vállalatának. A SAS Institute a SAS rendszer folyamatos fejlesztésére és támogatására kötelezte el magát, amely szuperszámítógépeken, nagyszámítógépeken, minigépeken, UNIX környezetekben, PC-s és Macintosh platformokon egyaránt fut. E célból a cég 175 millió dollárt fordított 1995-ben kutatás-fejlesztésre, miközben bevételei világszerte elérték a 562 millió dollárt. Ez a bevétel/fejlesztési ráfordítás arány duplája az iparági átlagnak.

A SAS Institute 1993. decemberében hozta létre a magyarországi irodáját a hazai felhasználók magasabb színvonalú kiszolgálása érdekében.

A SAS rendszer referencia helyei világszerte meghaladják a 30 ezret, hazánkban több, mint 100 helyen használják.

Vállalati Információ Szolgáltatás

(Különbség az üzemeltető és információ szolgáltató rendszerek között)

A Vállalati Információs Rendszer kialakítása a vállalat működtetési és döntéstámogatási környezetének fizikai szétválasztásával kezdődik. Sok vállalat "lelke" a nagy mennyiségű működtetési adat, amely általában online-tranzakció feldolgozó (OLTP) rendszerekből származik. A működtetési környezetben az adathozzáférési és az alkalmazáslogikai feladatok általában szorosan összekapcsolódnak, ami nem segíti elő a rugalmas adatlekérdezést.

Az alapvető technológiai és koncepcionális különbségeket foglalja össze az alábbi két táblázat:

Technológiai különbségek

Működtető (tranzakciós) rendszerek	**Információ szolgáltató rendszerek**
adat *BE*olvasás	információ *KI*nyerés
nagy mennyiségű *egyszerű* tranzakció	*kis* számú, de *komplex* lekérdezés
statikus alkalmazás	*dinamikus* alkalmazás
az *adat* folyamatosan változik	az *információ* tárolódik

Koncepcionális különbségek

Működtető (tranzakciós) rendszerek	**Információ szolgáltató rendszerek**
automatizálja a *rutinfeladatokat*	támogatja a *kreativitást*
napi ügymenet	*hosszú távú* stratégia
cél a *hatékonyság*	*versenyelőny*
technológia függő	*üzlet*-függő

Az adattárház tulajdonképpen a kapcsolat a két rendszer között. Ez egy olyan megoldás, amely kompakt, integrált, teljesen nyitott és lehetővé teszi, hogy a különböző szervezetek gyorsan üzleti információvá alakíthassák a nyers adatokat, és azt a leginkább megfelelő formátumban juttassák el a döntéshozókhoz.

SAS adattárház

Az adattárházat általában négy jellemző írja le:

témaorientált: az adatok nem alkalmazás, hanem téma szerint vannak szervezve

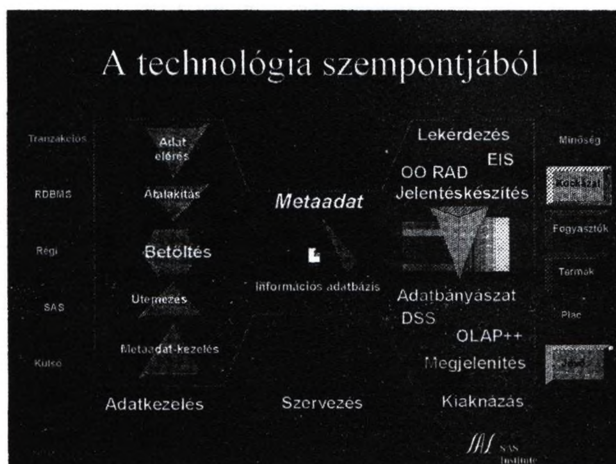
integrált: a különböző működtető rendszerekből és egyéb helyekről gyűjtött adatok globálisan konzisztens formában szerepelnek

idővariáns: az adattárház időcímkével ellátott, hosszú időn keresztül gyűjtött adatokat tartalmaz, amelyeket összehasonlítások, trendelemzés és előrejelzés céljára is használnak

nem változik: az adattárházba bekerült adatokat többé nem változtatják és nem frissítik, csak betöltik és olvassák őket

Az adattárházat döntéshozás támogatására és elemzés céljából hozzák létre, nem pedig tranzakciós feldolgozásra.

A következő ábra a SAS adattárház technológiai alapfunkcióit mutatja be sematikusan:



Objektum orientált vonatkozások

A SAS rendszer két objektum orientált alkalmazásfejlesztő (OOAD) eszközt is kínál, melyek közül a SAS/EIS-t fogjuk kicsit közelebbről szemügyre venni. A másik eszköz a SAS/AF, amelyben tulajdonképpen az EIS-t is fejlesztették. Ha az AF-et "lego" elemeknek tekintjük, akkor az EIS az olyan testreszabott "duplo" készlet, melyet az adattárház igényeinek megfelelően alakítottak ki az általános lego elemekből.

Mindkettőre igazak azonban a következő OO tulajdonságok, melyek kifejtésére az előadás csak részben tér ki:

- osztályok, objektumok, öröklődés
- beágyazódás
- testreszabható objektum könyvtárak
- delegálás és kompozit osztályok
- Meta Object Protokol (MOP)
  futás közbeni generálhatóság
- Model/Viewer/Controller szemlélet
  objektum tartalom: modell
  objektum megjelenés: viewer
  a kettő összehangolása: controller
- Per-Instance metódus
  metódus módosíthatóság egy objektumon belül
- esemény-kezelés
- drag & drop

A SAS objektumok kialakításánál is az elsődleges cél az olyan osztályok, objektumok definiálása volt, amelyek az adatmegjelenítést segítik a döntéshozók igényeinek megfelelően. (Közös jellemzőjük a fontok, színek, feliratok szabad állíthatósága és hogy beágyazhatók grafikus menükbe.)

Ilyen jellemző objektumosztályok például a következők:

- *többdimenzionális nézegető*
  Kényelmes grafikus felületet biztosít a felhasználónak dimenziók váltására, átrendezésére, stb. Az összesítési szintek egy egérkattintásra kibonthatók vagy összegezhetők. Az előre definiált hierarchiáknak megfelelően mélyre ásás végezhető.

- *szervezeti felépítés diagram*
  Adatvezérelt, dinamikus szervezeti felépítés gráf. A csomópontokban szöveg vagy kép egyaránt elhelyezhető, melyekre kattintva más alkalmazás vagy objektum indítható.

- *három dimenziós grafikus objektumok*
  Két és három dimenziós oszlop- és kördiagramok tetszőlegesen kombinálhatóak. Mindannyian adatvezéreltek és dinamikusan épülnek fel például egy-egy mélyre ásás során.

- *kritikus sikertényező*
  Adatvezérelt, dinamikusan keletkező összesített információ a kritikus tényezőkre. (pl. cash-flow vagy értékesített mennyiség)

- *megjelenítés kattintásérzékeny területekkel*
  Ezek az objektumok tipikusan térképek, melyekre kattintva az adott területre vonatkozó információk jelennek meg. A megjelenítés bármilyen korábban ismertetett eszközzel történhet.

- *további mintegy 20 objektumosztály*

Összefoglalás

A SAS rendszerben való fejlesztést olyan cégeknek ajánljuk, akiknek tapasztalataik vannak az objektum orientált fejlesztésben és kifejezetten valamilyen információ szolgáltató rendszert (pl. vezetői információs rendszer, döntéstámogató rendszer, elemzések, minőségbiztosítás, stb.) szeretnének létrehozni. A SAS Institute Quality Partner programja keretében partnereink széleskörű konzultációs szolgáltatásban részesülnek, annak érdekében, hogy megoldásainkat gyorsan és hatékonyan lehessen alkalmazni.

Fully Integrated Banktechnical Information System

megvalósítása ORACLE fejlesztő eszközön,

különös tekintettel a SERVER/REPREZENTÁNS/KLIENS

architektúra alkalmazására.

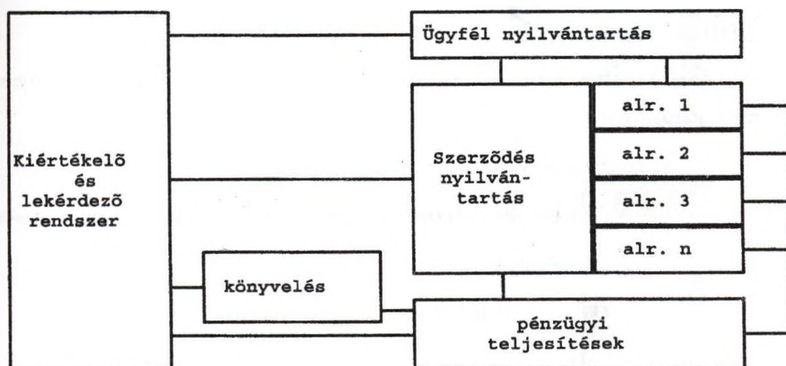
Róna György, Egervári Zoltán

R & E Systems

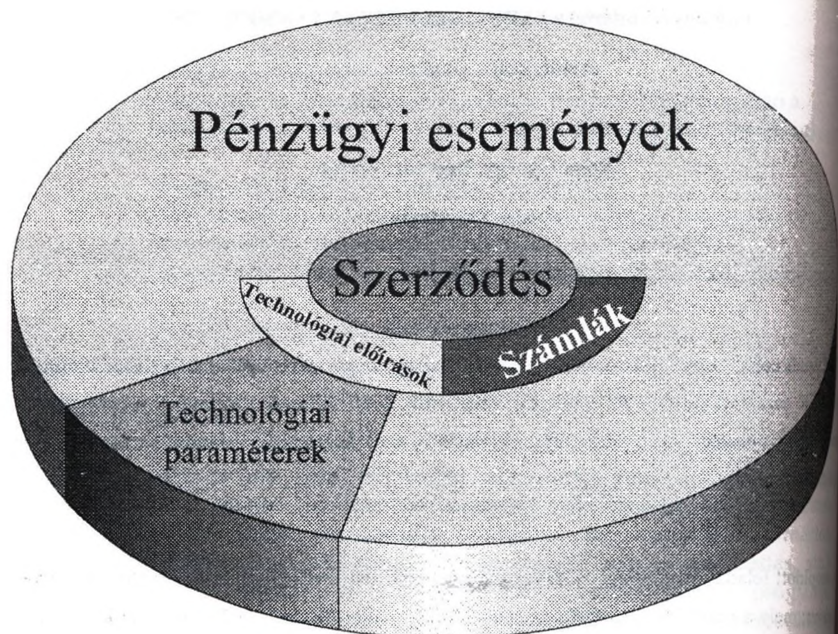
Előadásunk célja, hogy tájékoztassuk T. kollégákat egy nagy méretű feladat OO filozófiában ORACLE eszközökkel (7.x RDBMS, DEVELOPER 2000 fejl.eszk.) történő implementálás tervezése, kivitelezése során felmerült problémákról, azok kezeléséről valamint a megoldások tapasztalatairól.

A megoldott feladat, egy általános, teljes körű pénzintézeti tevékenységet támogató software rendszer, mely a Fully Integrated Banktechnical Information System - **FIBIS**- nevet kapta, új verziójának kialakítása.

A rendszer logikai áttekintése:



A rendszer logikai alapegységének tekintett objektum (szerződés) belső szerkezete:



Adatfeldolgozási filozófia

A tervezés fázisában két alapvető adatfeldolgozás filozófiai kérdés tisztázása játszott meghatározó szerepet.

- Az adatbázis tervezés során a

  - RES DB filozófia (feladat orientált, objektum orientált, több utas-hálós),

  - OO szemlélet ,

  - ORACLE RDBMS

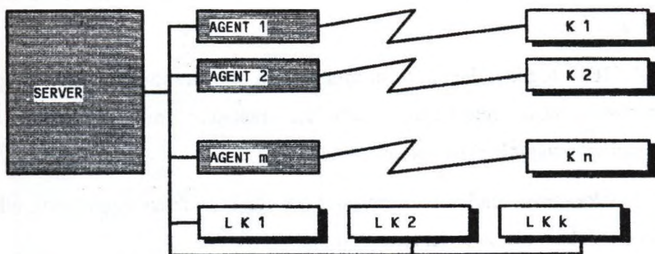
  összehangolása.

- A működési architektúra kialakításakor a

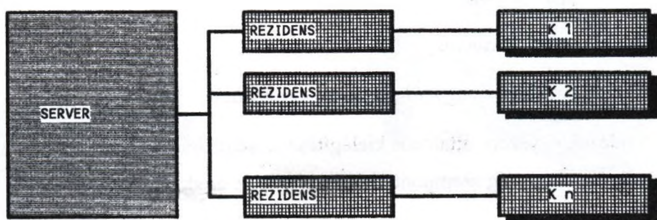
  - közbenső szint létrehozásának szükségessége,

- és annak megvalósítása

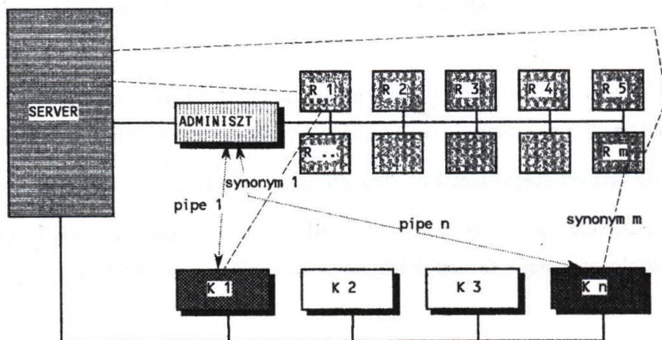
- agent,



- rezidens,



- reprezentáns



közötti választás.

A felmerült feladatok és problémák:

A tervezés során a feladat lefedéséhez szükséges adatbázis-struktúrával, adatfeldolgozó mechanizmusokkal szemben támasztott igények szükségessé tették a következő problémák megoldását is:

- A FIBIS logikai alapegysége, mint önálló objektum, önmagában is egy összetett, bonyolult kapcsolatokkal rendelkező rendszer, melynek mind logikai, mind pedig fizikai konzisztenciáját biztosítani kell.

- A feldolgozó rendszer szerteágazó feladatai gyakran egymásnak ellentmondó igényekkel lépnek fel az adatbázis tervezése során:

  - táblastruktúra variánsok,
  - egyedi táblák,
  - index variációk,
  - speciális és egyedi triggerek, procedúrák.

Mindezek egyszeri, általános kielégítése az adatbázisban mind volumen, mind működés, mind pedig hatékonyság szempontjából kedvezőtlen eredményeket hozna.

- Alapvető feladatként merült fel a rendszer csomópontjainak áthelyezhetősége mely a logikai működés mellett érinti, sőt meghatározza, a működtető erőforrások igényeit. Az erőforrások működtetésének stratégiája, a flexibilis megosztás lehetősége meghatározó hatékonysági tényező.

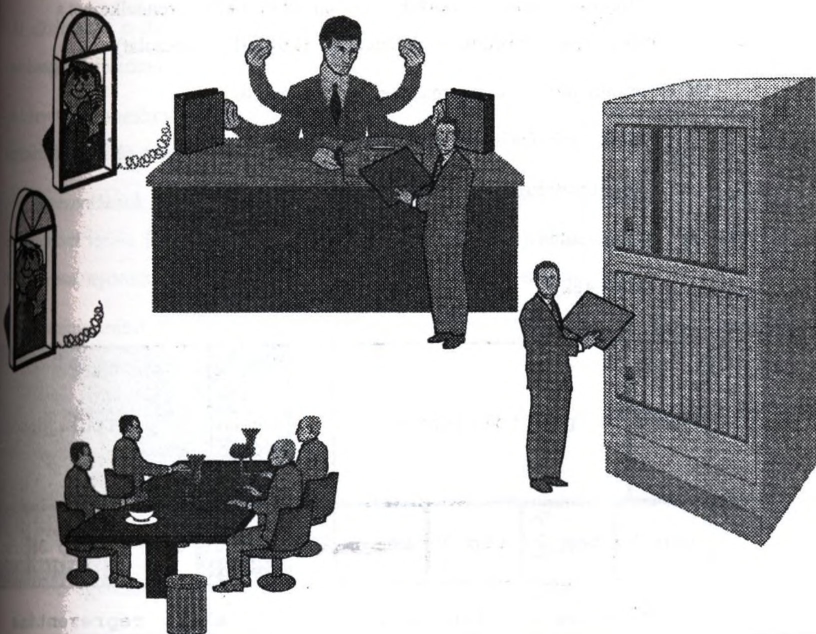
- Az általános adatbázis kezelők által biztosított LOCK mechanizmusok mellett szükségessé vált az un. feladat-specifikus LOCK stratégiák és mechanizmusok definiálása.

- Az igények meghaladták az általános tranzakció-kezelés lehetőségeit.

- Alapvető igény, hogy a felhasználó a nagy számítási igényű -így esetleg hosszú feldolgozási idejű- feladatok idejére „lekapcsolódjék" a feldolgozás folyamatáról.

Megoldás:

Adminisztrátor vezérelt reprezentánsok



A választott technológia objektumai:

- adminisztrátor
- reprezentáns(ok)
- kliens
- pipe

Működés:

- Az adminisztrátor feladata, hogy a reprezentánsok feladatait meghatározza, az azok végrehajtásához szükséges feltételrendszert biztosítsa, felügyelje (mint egy jó főnök).

- A kliens/adminisztrátor/reprezentáns kapcsolata:

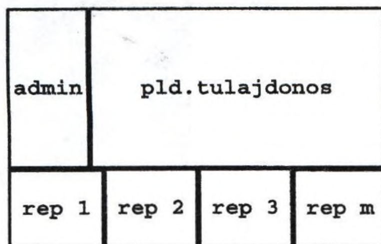
  - A kliens végrehajtandó feladatait az adminisztrátorral egyezteti. (Identifikálás, végrehajthatóság, visszaigazolás)

- A meghatározott feladatok ellátására az adminisztrátor reprezentánst jelöl

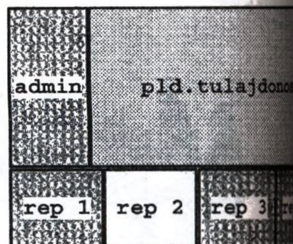
(kiválasztás, felhatalmazás, számonkérés)

- Bizonyos feladatcsoportok végrehajtáskor a kliens rendelkezhet a reprezentáns speciális lehetőségeivel. (közvetlen kapcsolat)

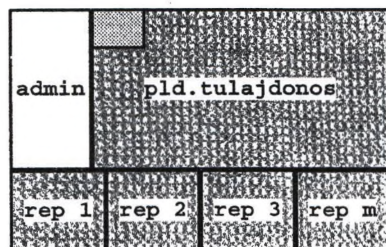
- Objektum tulajdonosok, egymáshoz való viszonyuk
  - alkalmazás (példány tulajdonos)
  - adminisztrátor
  - reprezentáns(ok)
  - kliens(ek)



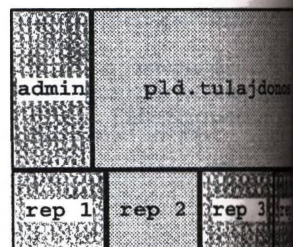
objektum tulajdonosok



aktiv reprezentán



admin jogai



user jogai-bekapcs
reprezentáns

- Jogok és synonymák.

- Az adminisztrátor megfelelő rálátással rendelkezik a teljes működési folyamat az un. szemafor rendszeren keresztül biztosítja annak hatékony működését.

Az alkalmazott megoldás „melléktermékei"

Az alkalmazott megoldás a kitűzött célok kielégítésén túlmenően „melléktermékként" további lehetőségeket biztosít:

- Az alkalmazott technika sokrétű segítséget nyújtott a relációs adatbázis / OO feldolgozás összehangolás feladatában.
- A közvetlen tranzakció kezelési problémákon túlmenően lehetőség nyílt a többszintű „super-tranzakció" kezelés megvalósítására.
- A reprezentánsok függetlensége komplett funkciók illetve elkülöníthető részfunkciók végrehajtását teszik lehetővé a kliens-oldal terhelése nélkül. (Pl: kiértékelési feltételrendszer feldolgozása, nyomtatások összeállítása, speciális perifériák kezelése, stb...)
- Az adminisztrátor felügyeletével a reprezentánsok valósítják meg a távoli kapcsolatok lekezelését. (Egységek közötti, GIRO, home banking, stb... kommunikációk).
- A reprezentáns-kliens ideiglenes feladatfüggő kapcsolata a jogosultsági rendszert biztonságosabbá és hatékonyabbá teszi.
- Az adminisztrátor folyamatosan felügyeli a rendszerben folyó feldolgozásokat, így lehetőséget biztosít a supervisor-i feladatok lefedésére is.
    - Az egymástól független feldolgozások több szinten nyomon követhetőek.
    - Az esetlegesen megszakadt feldolgozások
        - megismételhetőek,
        - folytathatóak,
        - „super tranzakciók" külső konzisztenciája biztosított.
    - Feldolgozási folyamat esemény vezérelhető (super tranzakciók felfüggeszthetők).
- Időzíthető és/vagy szituáció felismerő automatikus eljárások futtatásának lehetősége.

A procedurális paradigmától az eseményvezérelt szemléletmódig - *COBOL nyelven*

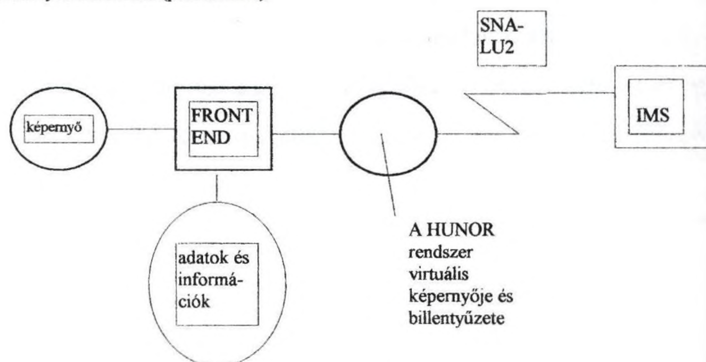
Huba Zoltán (HUNGÁRIA Számítástechnikai Kft.)

Kiindulási feltételek

A HUNGÁRIA Biztosító Rt. munkáját támogató számítógépes rendszer IBM nagyszámítógépen fut az IMS (Information Management System) vezérlete alatt. A rendszer szolgáltatásait az ország különböző helyein lévő kb 800 terminálról veszik igénybe. Ez a rendszer nem saját fejlesztésű, a HUNGÁRIA Biztosító egy osztrák biztosítótól vásárolta, ezért egyes új funkciók implementálása FRONT END módszerrel látszott legegyszerűbben és legbiztonságosabban megoldhatónak. A FRONT END módszer azt jelenti, hogy olyan programok készültek, amelyek egyrészt egy virtuális terminálon keresztül állnak kapcsolatban az alap rendszerrel, másrészt interaktív (párbeszédes) kapcsolatban vannak a felhasználóval. Ezeknek a programoknak ma már három generációja létezik, a legelső procedurális, a második korlátozottan eseményvezérelt (EVENT DRIVEN), a harmadik teljeskörűen eseményvezérelt. Ezek a programok COBOL nyelven készültek. Jelen előadás ezen programok jellemzőivel foglalkozik annak érdekében, hogy segítséget nyújtson azoknak, akik ma még hagyományos szerkezetű párbeszédes programokat írnak, de szeretnének objektum orientált eseményvezérelt programokat írni.

Első generáció

Az első feladat az volt, hogy a vásárolt számítógépes rendszert egészítsük ki egy segítségnyújtó (HELP) szolgáltatással. A megoldás lényege, hogy egy program, amely a **CICS/ESA** (Customer Information Control System/Enterprise Systems Architecture) vezérlete alatt az IBM nagygépen fut, egy virtuális terminálon keresztül kapcsolatban áll az alap rendszerrel, és egy valóságos terminálon keresztül pedig a felhasználóval. A program alapüzemmódban a virtuális terminál tartalmát átmásolja a valóságos terminálra és viszont, vagyis a felhasználó a számítógép csekély többletterhelése árán ugyanúgy használja a rendszert, mintha közvetlenül használná. Segítség-üzemmódban azonban a program a valóságos terminálon megjelenít egy menüt, amelynek segítségével különböző adatok és információk kaphatók meg, anélkül, hogy a virtuális terminál tartalma változna. Így az alapüzemmódba történő visszatéréskor a felhasználó ott folytatja a munkáját az alap rendszerrel, ahol azelőtt volt, vagyis az alap rendszer a segítség-üzemmódból semmit sem "vesz észre". A segítség üzemmód természetesen figyelembe veszi, hogy a felhasználó éppen mit csinált az alapüzemmódban (Context sensitive). A CICS/ESA alatt futó program hagyományos szerkezetű (procedurális).



A CICS BMS (Basic Mapping Support) lehetővé teszi, hogy az egyes képernyő elrendezéseket elő[...] elkészíthessük, vagyis megtervezzük, hogy hol lesznek a képernyőn konstans feliratok, fejlécek, oldallécek stb. [...]

hol és milyen méretű és tulajdonságú beviteli mezők. Ezen adatok megadása után készül egy úgynevezett Physical Map, amely a CICS számára tartalmazza a megjelenítés részletetit, és készül egy Symbolic Map, amely egy rekordkép, amit a CICS vezérlete alatt futó program használ arra, hogy beolvashassa ill módosíthassa a beviteli mezők tartalmát és tulajdonságait.

Tegyük fel, hogy megterveztük az alábbi képernyőt:

```
        OPERATOR INSTRUCTIONS

OPERATOR INSTR      - ENTER MENU
FILE INQUIRY        - ENTER INQY AND NUMBER
FILE BROWSE         - ENTER BRWS AND NUMBER
FILE ADD            - ENTER ADDS AND NUMBER
FILE UPDATE         - ENTER UPDT AND NUMBER


PRESS CLEAR TO EXIT
ENTER TRANSACTION:    NUMBER
```

A program számára készülő Symbolic Map a következő lesz:

```
01 MENUI.
   02 FILLER PIC X(12).
   02 MSGL   COMP PIC  S9(4).
   02 MSGF   PICTURE X.
   02 FILLER REDEFINES MSGF.
    03 MSGA   PICTURE X.
   02 MSGI PIC X(39).
   02 KEYL   COMP PIC  S9(4).
   02 KEYF   PICTURE X.
   02 FILLER REDEFINES KEYF.
    03 KEYA   PICTURE X.
   02 KEYI PIC X(6).
 01 MENUO REDEFINES MENUI.
   02 FILLER PIC X(12).
   02 FILLER PICTURE X(3).
   02 MSGO  PIC X(39).
   02 FILLER PICTURE X(3).
   02 KEYO  PIC X(6).
```

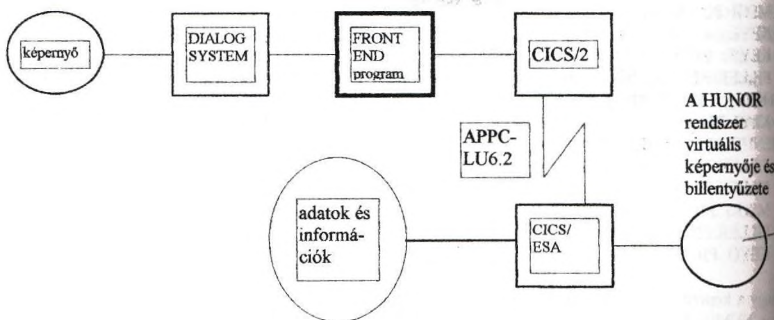
Látható, hogy a képernyő alján lévő 'ENTER TRANSACTION:' mezőoldalléc után következő mező az MSG nevet, míg a 'NUMBER' oldalléc utáni a KEY nevet kapta. Ha a CICS alatt futó program a képernyő tartalmát üres beviteli mezőkkel akarja megjeleníteni, akkor nem is kell foglalkoznia ezekkel a mezőkkel:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. FILECMNU.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 *
 PROCEDURE DIVISION.
   EXEC CICS SEND MAP('MENU') MAPSET('DFH$CGA')
         MAPONLY ERASE END-EXEC.
   EXEC CICS RETURN END-EXEC.
   GOBACK.
```

A SEND utasítással mondtuk meg a CICS-nek, hogy a 'DFH$CGA' nevű állományban keresse meg a MENU nevű képernyőleírást és jelenítse meg.
Ha a programban a
EXEC CICS RECEIVE MAP('MENU') MAPSET('DFH$CGA') END-EXEC.
utasítás is szerepel, akkor a program addig várakozik, amíg a képernyő előtt ülő felhasználó a beviteli mezőket nem tölti és ezt az ENTER gomb megnyomásával nem jelzi. Ekkor a beviteli mezőkbe beírt tartalmakat a program az MSGI ill. KEYI mezőkben találja, a tartalom karakterekben mért hosszát pedig az MSGL ill. KEYL mezőkben. Látható, hogy a rendszerben először a FILECMNU programot kell elindítani és ez a program indítja a képernyőtartalom megjelenítését majd vár a felhasználó válaszára, feldolgozza a kapott adatokat, majd dönt hogy ezután milyen képernyőtartalom jelenjen meg. Ez azt jelenti, hogy a képernyők sorrendjét vagyis a felhasználó munkáját a programban kódolt logika határozza meg. Az ilyen rendszert hívjuk procedurális rendszernek.

Második generáció

A második feladat az volt, hogy készüljön egy olyan személyi számítógépen futó program, amely a felhasználói grafikus felhasználói felületet (GUI) nyújt, és lehetővé teszi ezen a felületen a nagyszámítógépes rendszer funkcióinak és a segítő (HELP) információknak a használatát. A személyi számítógépre az **IBM OS/2** operációsrendszert telepítettük, mert a személyi számítógép és az IBM nagygép közötti **SNA LU6.2** protokoll szerinti kapcsolat így volt legegyszerűbben felépíthető. A személyi számítógépen futó program az IBM nagygépen a CICS/ESA vezérlete alatt futó egyik rutin segítségével kezeli a virtuális terminált, amelynek segítségével - az első generációnál kidolgozott FRONT END technológia alapján - az IMS alatt futó alaprendszert éri el, míg a másik rutin a segítségadáshoz szükséges nagygépes adatállomány elérését teszi lehetővé. Tekintettel arra, hogy a személyi számítógépen futó program funkcióinak jelentős része az első generációban elkészült, továbbra is a COBOL nyelvhez ragaszkodtunk, ezért a **MicroFocus** cég **COBOL WORKBENCH** programfejlesztő rendszerét választottuk. A programfejlesztő rendszerhez vásárolható MicroFocus **DIALOG SYSTEM** lehetővé teszi, hogy a COBOL nyelvű program által használt grafikus felhasználói felületet grafikus eszközökkel lehessen kialakítani. Az így elkészült személyi számítógépes rendszer korlátozottan eseményvezérelt.



A MICROFOCUS DIALOG SYSTEM lehetővé teszi, hogy a tervezett képernyőn elhelyezendő ablakok, nyomógombok, stb. kialakítását a személyi számítógép képernyőjén végezhessük el, egérrel megfogva elhelyezve az elemeket, billentyűzetről beírva a szövegeket stb.. Az így elkészített képernyők adatait a DIALOG SYSTEM tárolja egy adatállományban, és készít egy COBOL programba másolandó rekordleírást (COPY), amelyik a kommunikációs rekordot írja le, amelyben többek között az egyes képernyőkön megjelenítendő adatoknak van helye. Az alkalmazói program ebben a kommunikációs rekordban kitölti a mezőket, és meghívja a DIALOG SYSTEM-et, erre megjelenik a grafikus ablak, kitöltve a megfelelő tartalommal. A program marad várakozó állapotban, amíg a felhasználó el nem végzi a szükséges beírásokat, módosításokat az ablakon, nem jelzi - például egy nyomógomb megnyomásával, hogy befejezte. A program ilyenkor megkapja a módo

kommunikációs rekordot, amelynek elemzése után eldönti, hogy mi jelenjen meg a személyi számítógép képernyőjén.
Tervezzünk egy egyszerű ablakot a DIALOG SYSTEM segítségével, a rendszer által készített rekordleírás az alábbi lesz:

```
******************************************************************
* Data Block
******************************************************************

01 DATA-BLOCK-VERSION-NO PIC 9(8) COMP-5 VALUE 4.

01 VERSION-NO            PIC 9(2) COMP-5 VALUE 2.

01 SET-BUILD-NO          PIC 9(4) COMP-5 VALUE 17.

01 DATA-BLOCK.
   03 KERT-FUNKCIO        PIC X(8).
   03 FUNKCIO-PARAMETER   PIC X(60).
   03 BEKERO-DBOX-SZOVEG  PIC X(20).

******************************************************************
* End of Data Block
******************************************************************
```

Eddig a módszer kísértetiesen hasonlít a CICS alatt futó programnál látottakra. (Természetesen a MicroFocus cégnek ez is volt a szándéka, és CICS alatt futó rendszerek viszonylag könnyen tehetők át így személyi számítógépre.) Elindul egy COBOL program, és ez a program indítja el a képernyőtartalom megjelenítését majd vár a felhasználó válaszára, feldolgozza a kapott adatokat, majd dönt, hogy ezután milyen képernyőtartalom jelenjen meg. A CICS-nél megismert megoldáshoz hasonlóan a megjelenítendő ablak technikai részletei a programtól független helyen vannak tárolva, a program csak azt szabja meg, hogy milyen nevű ablak jelenjen meg. A program a megjelenő ablakon a felhasználó által végzett manipulációk utáni helyzetről (beadott adatokról) a DATA-BLOCK segítségével értesül. A DATA-BLOCK előtt látható mezők azt a problémát oldják meg, hogy ha módosítjuk az ablak grafikus leírását és így új DATA-BLOCK készült de nem fordítjuk újra a COBOL programot, akkor ez súlyos futási hibához vezetne. Ezek után egy DIALOG SYSTEM-et használó COBOL program az alábbi formájú lehet.
A program megértéséhez még azt kell tudni, hogy a program a vezérlő információkat a "ds-cntrl.mf" COPY struktúrában adja át a DIALOG SYSTEM-nek, a programban látható "ds-" kezdetű nevű elemek e struktúra részei.

```
******************************************************************
* HUNOR - A HUNOR GUI rendszer vezérlő programja
******************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. HUNOR.
AUTHOR. HUBA ZOLTAN.
DATE-WRITTEN. 1995 július 17.
DATE-COMPILED. MA.
***************************************************************
*EZ A PROGRAM - megjeleníti a kezdő képernyőt és a megadott
*funkciónak megfelelő alprogramot hívja
***************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
*KONSTANSOK:*****************************************************
 77 dsgerror                PIC X(8) VALUE "DSGERROR".
*KONSTANSOK VEGE*************************************************
 77 HELP-KOD                PIC X(6) VALUE "UN5001".
*
```

```
COPY "ds-cntrl.mf".
COPY "HUNOR.cpb".
PROCEDURE DIVISION.
VEZERLO SECTION.
   initialize data-block
   perform dialog-system-beallitas
   perform until FUNKCIO-PARAMETER equal to "EXIT"
     perform call-screen-manager
     evaluate kert-funkcio
     when "MAGYARAZ"
       call "MAGYARAZ" using HELP-KOD
       perform dialog-system-beallitas
     when "RENDSZAM"
       call "RENDSZAM" using FUNKCIO-PARAMETER
     when "SZERZODE"
       call "SZERZODE" using FUNKCIO-PARAMETER
     when other
       Stop run
     end-evaluate
     if return-code not = 0
     then
       inspect FUNKCIO-PARAMETER replacing all low-values
             by spaces
       move "hiba-kepernyo" to ds-procedure
     end-if
   end-perform.
dialog-system-beallitas section.
   initialize ds-input-fields
   move ds-new-set to ds-control
   move data-block-version-no
     to ds-data-block-version-no
   move version-no
     to ds-version-no
   move "HUNOR" to ds-set-name.

call-screen-manager section.
   perform test-dialog-error.

test-dialog-error section.
   if not ds-no-error
     call dsgerror using ds-control-block
   end-if.
```

A DIALOG SYSTEM a COBOL program által kért ablakot megjeleníti. Ezen ablakon sok különböző grafikus objektum (pl.: nyomógomb, szelekciós lista stb.) lehet, ezeken az elemeken különböző események lehetnek és az ablak tervezésénél kell az egyes esmények hatásáról rendelkezni. A DIALOG SYSTEM erre a célra rendelkezésre bocsájt egy meta nyelvet és az ezen a nyelven leírt eseménykezelést nevezik Dialog-nak. A például választott HUNOR nevű ablak tulajdonképpen egy olyan ablak, amely bizonyos fix szövegeken kívül csak néhány nyomógombot tartalmaz. Ennek az ablaknak a Dialog-ja a következő:

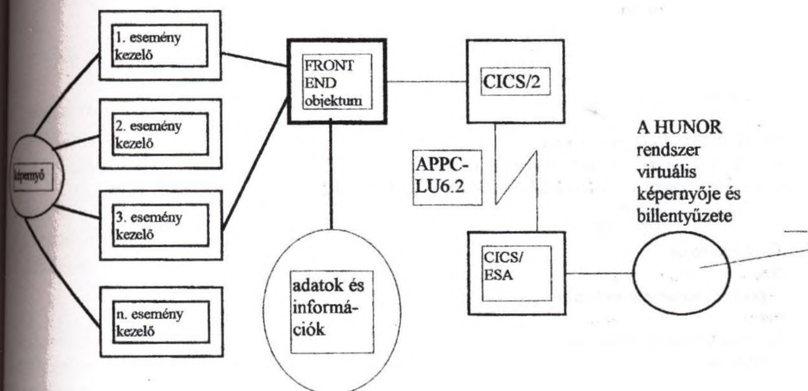
```
@KILEPES-MB
   MOVE "EXIT" KERT-FUNKCIO
   RETC
 @RENDSZAM-MB
   MOVE " " FUNKCIO-PARAMETER
   MOVE "RENDSZAM" KERT-FUNKCIO
   MOVE "a rendszámot!" BEKERO-DBOX-SZOVEG
   SET-OBJECT-LABEL BEKERO-DBOX "A rendszám bekérése"
   SET-FIELD-LIMIT BEKERO-DBOX-VALASZ-OBJ 12
   SET-FOCUS BEKERO-DBOX-VALASZ-OBJ
```

```
@MAGYARAZ-MB
MOVE "MAGYARAZ" KERT-FUNKCIO
RETC
@SZERZODES-MB
MOVE " " FUNKCIO-PARAMETER
MOVE "SZERZODE" KERT-FUNKCIO
MOVE "a szerződés-számot!"" BEKERO-DBOX-SZOVEG
SET-OBJECT-LABEL BEKERO-DBOX "A szerződés-szám bekérése"
SET-FIELD-LIMIT BEKERO-DBOX-VALASZ-OBJ 8
SET-FOCUS BEKERO-DBOX-VALASZ-OBJ
```

A Dialog-on látható @ jellel kezdődő sorok jelölik az egyes eseményeket. Tehát, ha a HUNOR nevű ablak megjelenése után a felhasználó a "Kilépés" feliratú nyomógombot nyomta meg, akkor elkezdődik a Dialog @KILEPES-MB kezdetű részének a végrehajtása. Ez egyrészt a DATA-BLOCK KERT-FUNKCIO azonosítójú elemének a kitöltését, másrészt a COBOL programba történő visszatérést írja elő (RETC). Idáig minden olyan mint a CICS alatt futó programnál, csak sokkal bonyolultabban kell kódolni. Ha azonban a felhasználó a "Rendszám bekérése" feliratú nyomógombot nyomta meg, akkor a Dialog @RENDSZAM-MB kezdetű részének a végrehajtása kezdődik el. Ez pedig egy újabb ablak megjelenítését írja elő, bizonyos változó szövegrészek beállítása után. Ezen ablak Dialog-jában kell rendelkezni az itt előforduló események kezeléséről és valahol valamikor vissza kell térni a COBOL programba (RETC). Az eddigiekből érzékelhető, hogy ez a rendszer lehet logikájában teljesen procedurális (pl.: a Kilépés esetén) vagy lehet több olyan esemény a képernyőn amelyről a COBOL program mit sem tud.

Harmadik generáció

Az 1995. év végén az IBM cég által piacra dobott **VisualAge for COBOL for OS/2** rendszer lehetővé tette, hogy a második generációs rendszerünket teljesen eseményvezérelt rendszerré dolgozzuk át, sőt COBOL nyelven írt objektumokat is használhassunk. A személyi számítógépen OS/2 WARP alatt futó grafikus felhasználói felület továbbra is a már bevált FRONT END technológiával csatlakozik a nagygépes alaprendszerhez és a segítségnyújtáshoz szükséges információkhoz. A teljes eseményvezéreltség lehetővé tette, hogy a személyi számítógépen egyes funkciók párhuzamosan - egy időben igénybevehetően - álljanak rendelkezésre.



A VisualAge for COBOL rendszerben is a képernyőn megjelenő ablakok megtervezésével kezdődik a munka. A különböző de logikailag összetartozó ablakokat csoportokba foglaljuk. Egy-egy ilyen csoport egy-egy *component*. Egy component ablakainak kezelésére szolgál egy COBOL program. Az ablakok kialakítását a személyi számítógép képernyőjén végezzük el, egérrel megfogva és elhelyezve az elemeket, billentyűzetről beírva a szövegeket stb.. Az így elkészített képernyők adatait a VisualAge for COBOL tárolja egy adatállományban, és

készít egy COBOL programba másolandó rekordleírást (COPY file), amely az egyes grafikus elemek hivatkozási pontjait (handle) tartalmazza. Ezen kívül generál egy COBOL program csontvázat. Ezen csontvázban egy-egy ENTRY minden esemény, amelynek kezelésére vállalkozunk. Például ilyen esemény egy nyomógomb megnyomása. A például választott HUNOR nevű ablak tulajdonképpen egy olyan ablak, amely bizonyos fix szövegeken kívül csak néhány nyomógombot tartalmaz. Ha a HUNOR nevű ablak megjelenése után a felhasználó a "Kilépés" feliratú nyomógombot nyomta meg, akkor elkezdődik a COBOL program végrehajtása a következő módon:

```
ENTRY
        "FOABLAK_MENU-LEK-KILEP_MENUSELECT"
              USING VDE-HUNOR.
        move vde-terminate-application to action-rc of VDE-HUNOR.
        GOBACK.
```

Az előbb említett component-ek egy application-t alkotnak. Itt a vde-terminate-application jelzéssel arra utasítjuk a VisualAge for COBOL rendszert, hogy az alkalmazás futtatását fejezze be, vagyis az alkalmazáshoz tartozó összes ablakot vegye le a képernyőről.

Ha azonban a felhasználó a "Rendszám bekérése" feliratú nyomógombot nyomta meg, akkor a COBOL program következő részlete aktivizálódik:

```
ENTRY
        "FOABLAK_MENU-LEK-REND_MENUSELECT"
              USING VDE-HUNOR.
        set RENDSZAM-FUNKCIO to TRUE.
        set NINCS-FUNKCIO-PARAMETER to TRUE.
        CALL "openWindow" USING BEKERO-ABLAK-HANDLE,
                        VDE-RC
        if not vde-rc-ok and not vde-object-already-opened
        then
           perform terminate-and-stop
        end-if.
        perform bekero-ablak-megnyitasa
        MOVE 19 TO Contents-Length
        MOVE "A rendszám bekérése" TO Contents-String
        CALL "setLabel" USING BEKERO-ABLAK-HANDLE,
                        Contents,
                        VDE-RC
        if not vde-rc-ok
        then
           perform terminate-and-stop
        end-if.
        MOVE 13 TO Contents-Length
        MOVE "a rendszámot!" TO Contents-String
        CALL "setContents" USING MIT-KER-SZOVEG-HANDLE,
                        Contents,
                        VDE-RC
        if not vde-rc-ok
        then
           perform terminate-and-stop
        end-if.
        move 12 to max-beviteli-hossz.
        GOBACK.
```

Amint látható, ennél az eseménynél a program egy újabb ablak megnyitását kéri. Az ilyen tevékenységek (action) standard eljárások felhívásával hajthatók végre. A különböző standard eljárások különböző hívási paramétereket igényelnek. Ezek között mindig az első annak a grafikus elemnek a hivatkozási pontja (handle), amelyen a tevékenységet végre akarjuk hajtani. A standard eljárások utolsó paramétere mindig a végrehajtás eredményességét mutató numerikus kód (VDE-RC). A most tárgyalt eseménykezelő első tevékenysége (openWindow) nem is igényel több paramétert, míg az új ablak címkéjét beállító tevékenység (setLabel) viszont a megjelenítendő szöveg hosszát és tartalmát is. Nagy előnye a VisualAge for COBOL rendszernek, hogy

különböző tevékenységek összeépíthetőek egy COBOL eljárássá, amely azután tetszőleges pontokról hívható (lásd: perform bekero-ablak-megnyitasa). A VisualAge for COBOL kiaknázza, hogy a COBOL program többszörösen hívható (reentrant) és az egyes ENTRY eljárások párhuzamosan futhatnak.

A COBOL nyelv szabványosításával foglalkozó ANSI munkacsoport már évek óta foglalkozik az objektum orientált programozási eszközök COBOL nyelvbe történő beépítésének vizsgálatával. Bár az új COBOL szabvány megjelenése 1997-re várható, az IBM és a MicroFocus már lehetővé tette ezen lehetőség használatát COBOL fordítóprogramjaik újabb változatában. Amint az ábrán látható, ilyen objektum az a rendszer elem is, amelyik a nagygépen futó IMS tranzakciók virtuális képernyőjét kezeli. Az objektumhívás szintakszisa hasonlit egy externális rutin hívásának szintakszisához. Az előbbi formája:

```
INVOKE HunorkapPtr "Titkosit" using user-params
                   returning Titkosit-return-code.
```

A HunorkapPtr nevű változó deklarációja "using object reference" és az objektum címét hordozza. A következő nyelvi elem az objektumon belüli metódust nevezi meg, a továbbiak pedig a metódus operandusai. Az objektum címét az objektum létesítésekor kapjuk meg, ennek szintakszisa:

```
INVOKE Hunorkap "somNew" returning HunorkapPtr.
```

Ha a hívandó program nem objektum, hanem externális rutin, akkor a hívása:

```
CALL "Titkosit" using titkosit-params
                returning Titkosit-return-code.
```

Minthogy egy externális rutinnak is lehet több ENTRY pontja, és a COBOL rutinok emlékeznek, (tehát változóik tartalma a második híváskor ugyanaz, mint az első hívás utáni visszatéréskor) a COBOL programozó aszerint dönti el, hogy objektumot vagy externális rutint készít, hogy több különböző állapot fenntartására van-e szükség vagy sem.

Objektumorientált szimuláció Java-ban

Simon Géza
Budapesti Műszaki Egyetem
Híradástechnika Tanszék
simon@inf.bme.hu

Összefoglaló

Ebben a cikkben az objektumorientált processzalapú szimulációt mutatom b egy konkrét eszközön, a Java Simulation Toolkiten (JUST) keresztül, különös figyelmet szentelve a terhelés szimulációjának (workload analysis). A JUST-o tanszékünkön fejlesztjük. A fejlesztés jelenlegi állásába is bepillantást kívánok adn a specifikáció szakmailag érdekesebb részeit kiemelve. Előzőleg összefoglalom processzalapú szimulációval, illetve a Java nyelvvel kapcsolatos fontosab alapfogalmakat.

Kulcsszavak: szimuláció, multiprocesszor, objektumorientált, Java, processzalapú

1 Bevezetés

1.1 Szimuláció

A nagymértékben párhuzamos multiprocesszoros számítógépe teljesítményének értékelése legbiztosabban mérésekkel végezhető el. Ehhe azonban először meg kell építeni a számítógépet, ami költséges, és az esetleg módosításokat szinte teljesen lehetetlenné teszi. Partikuláris hibajelenség vizsgálata szintén nehézkes, mert a kész rendszeren általában nincs mód bizonyo hibákat (például egyetlen összeköttetés meghibásodását) generálni.

A fentiek alapján a megbízhatósági és teljesítményanalízist a tervezé szakaszban, lehetőleg minél korábban kell elvégezni. Ehhez analitikus é szimulációs módszerek állnak rendelkezésre.

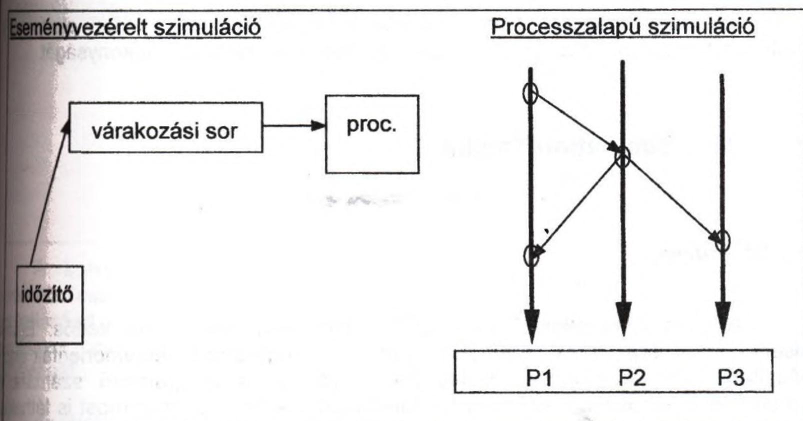
Az analitikus eljárásokat, mint a Markov-láncok, a sorbanállási rendsze vagy a különféle Petri-hálók, gyakorta alkalmazzák a tervezett rendszer modelljéne megalkotására. Ezeket vagy analitikus vagy numerikus módszerek segítségé értékelik ki. A teljesítmény-megbízhatóság analízis (performability analízi gyakorlatában az egyik leggyakrabban alkalmazott eszköz az általános sztochasztikus Petri-háló (GSPN).

A szimulációs eljárások az előbbiekhez képest előnyös tulajdonságok rendelkeznek. Például, míg a fent említett GSPN csak exponenciális idő

eloszlással operál, szimulációval tetszőleges eloszlást előállíthatunk. Hierarchikus modellezés is lehetséges, az analitikus módszerekkel ellentétben, melyek inkább az alacsonyszintű vizsgálatot támogatják. Itt a különböző szintek egymástól függetlenül modellezhetőek, így nem ütközünk a kezelhetetlenül nagy állapottér problémájába sem. Az előnyökön kívül a szimuláció természetesen magában hordja azt a veszélyt, hogy a szolgáltatott eredmények egyáltalán nem általános érvényűek.

A gyakorlatban két fő típusú szimulációt használnak. Az eseményalapú (event based) szimuláció szintén inkább az alacsony szinteken hatékony, hiszen egy komplex számítógéprendszer teljes működését (esetleg a rajta futó programokkal együtt) szinte lehetetlen elemi eseményekre fejben lebontani. Ezzel szemben a processzalapú (process based) szimuláció során párhuzamos (vagy látszólag párhuzamos) processzként fut minden egyes alkatrész szimulációja, és ezek kommunikálnak egymással, mint a valódi rendszerben a részegységek.

További információ, és több konkrét módszer összehasonlítása található a



[HEIN 94] irodalomban.

Mivel célom a nagymértékben párhuzamos számítógépek magas szintű vizsgálata fókuszában a terhelés analízisével, a fenti szempontok alapján processzalapú szimulációt választottam eszköznek.

1.2 A *Java nyelv*

A Sun által kifejlesztett új objektumorientált nyelv, a Java, sok vonása miatt nagyon alkalmasnak látszik a fentebb vázolt szimulációs rendszer megvalósítására.

Ez egy nagyon robosztus nyelv a megbízható memóriakezelés miatt. A C, illetve C++ nyelv pointeraritmetikája teljesen megszűnt, így a rossz címzések és az ezzel járó bosszantó hibák elkerülhetőek. Helyette a nyelv alapkönyvtára rengeteg adattípust tartalmaz készen, mint például hashtábla, tömb (határellenőrzéssel), sőt,

többek között létezik egy kész stringtokenizer osztály. A szabványos könyvtárakon kívül az interneten hozzáférhető még számos egyéb bővítés, melyek a programozó válláról leveszik a tényleges feladat szempontjából lényegtelen programozást.

A nyelv hordozhatósága szintén igen fontos, a legtöbb lényeges operációs rendszeren fut Java implementáció. Mivel nyelvi szinten lehetséges a hálózat használata, elképzelhető, hogy nagyobb számításigényű problémák esetében több gép kooperáljon (létezik JavaPVM is már).

A Java nyelv biztosítja a több szálon futó (multi threaded) programok írását. A processzalapú szimuláció enélkül elképzelhetetlen, hiszen az egyes processzeknek párhuzamosan, illetve kvázi-párhuzamosan kell futniuk.

A nyelv további előnyös tulajdonsága, hogy egy osztály megváltoztatása nem vonja maga után a többi forrás újrafordítását. Ez megkönnyíti az objektumorientált elvek tényleges gyakorlati alkalmazását: bármikor lehetőség van egy osztály kicserélésére, továbbfejlesztésére.

Várhatóan nagyon rövid határidőn belül elkészülnek minden platformra a röptében fordítók a Java nyelvhez, így megnövelve az eszköz hatékonyságát.

2 Java Simulation Toolkit

2.1 Célkitűzés

A Java Simulation Toolkit (JUST) kifejlesztésének célja kettős. Egyfelől kísérlet, hogy lehet-e, és érdemes-e szimulátort szigorúan objektumorientált alapon készíteni. Mennyi előnnyel, illetve hátránnyal jár a programozó számára, és mennyivel a felhasználó számára? A fontosabb szempontok már most is láthatók, de végleges választ csak a kész program használata során kaphatunk.

Másfelől szeretnék hozzájutni egy rugalmas eszközhöz, amely lehetővé teszi a parallel rendszerek teljesítmény-megbízhatóság analízisét. Ehhez a terhelést jelentő programok modellezése szükséges, ezért fektetünk nagy hangsúlyt a JUST tervezésében a terhelés parametrizálására.

2.2 A JUST felépítése

A szimulációs rendszer egy magból áll, amely a processzeket koordinálja, és egy bővíthető osztálykönyvtárból, melyben a szimulálható elemek találhatóak. A felhasználói felület ennek megfelelően az osztályok public módszereiből áll össze.

Egy szimuláció az alábbi lépésekből áll:

a) vezérlőprogram megírása

b) fordítás

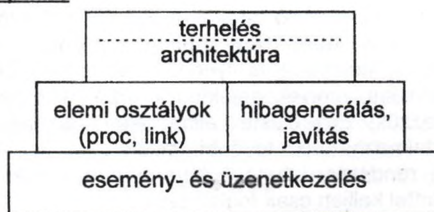
c) futtatás

d) kimeneti adatállományok kiértékelése

A vezérlőprogram maga is egy Java osztály, amely három - szorosan összefüggő - feladatot lát el. Létrehozza a szimulációban résztvevő objektumokat az előre elkészített vagy a felhasználó által definiált osztályokból, a megfelelő rendszereljárásokkal összeállítja ezekből a szimulálandó architektúrát és terhelést, végül egyéb paramétereket állít be, mint például hibavalószínűségek, javítási sémák, rendszerelemzők jellemzők stb.

A kimeneti adatállományokat ezután futás közben a mag (simulation engine) automatikusan generálja. Kiértékelésük legegyszerűbb módja például a GnuPlot programmal való megjelenítés.

A szimulátor felépítése



A belső mag (az ábrán a két alsó szint) felelős a processzek szinkronizálásáért, hibák injektálásáért, illetve a javításáért, és a statisztika generálásáért. Bizonyos alacsony szintű eszközöket (mint például a link osztály két processzor összekötéséhez) is itt definiálunk. Hasonló felépítést ismertet [GOSW 93].

Az ábrán látható, hogy ezek hogyan kapcsolódnak egymáshoz.

Az osztálykönyvtárak tartalmazzák az elemi osztályokból származtatott bonyolultabb elemeket. Ezek részben az architektúra, részben a rajta futó szoftver modellezését szolgálják.

2.3 *Fontosabb jellemzők*

Bővíthetőség. A rendszer rugalmasságát biztosítja, hogy ezek az osztályok kicserélhtőek tetszőleges más működés érdekében, illetve ezekből mind örökléssel, mind kompozícióval újak származtathatóak le. Az új osztályokból generált objektumok ugyanúgy felhasználhatóak a szimulációban, mint az előre definiáltak. Ehhez a tulajdonsághoz jól kihasználható a Java nyelv interface fogalma. Egy feladatot nem szükséges mindig ugyanannak az objektumnak elvégezni. Ha egy objektum megfelel a kívánt interface-nek, vagyis megvalósítja az ott leírt

módszereket, akkor helyettesítheti az eredeti objektumot. Az interface-használattal a hívó egyértelműen deklarálja, hogy neki milyen függvényekre van szüksége, a hívott pedig azt, hogy ő ezeket végre tudja hajtani. Így teljesen különböző belső felépítésű objektumok is végezhetik ugyanazt a feladatot. Itt megvalósul az egyik objektumorientált paradigma: az adatrejtés; az új objektum írójának nem kell ismernie a régi objektum működését.

Többszálúság. A processzalapú végrehajtás lényege, hogy több processz futhasson - látszólag - egyszerre. Mivel közös adatterületeket (is) használnak, egy program szálaiként kell végrehajtódniuk. A Java nyelv lehetővé teszi, hogy minden processz úgy érzékelje, ő a többiektől függetlenül, velük párhuzamosan fut. Valójában a végrehajtás a mai Java implementációkban csak kvázi-párhuzamos, az időzítő algoritmus a nyelvben nincs definiálva, így megvalósításonként különbözik. A különböző processzek Java-ban egy egy osztály objektumai, amelyek létrejöttükkor "életre kelnek", és üzenetekkel kommunikálnak egymással. Erre a célra a simulation engine-ben megfelelő osztályok és módszerek találhatóak, melyek a postaládákat képesek kezelni, és az üzeneteket a címzett objektumhoz juttatják.

Skálázhatóság. Mivel a szimuláció tárgyai, a parallel számítógépek tipikusan skálázható felépítésűek, vagyis létezik egy építőelem, amelyekből tetszőleges méretű gép rakható össze, a modellnek is ilyennek kell lennie. Ennek érdekében olyan osztályokat kell definiálni, melyek objektumai több részobjektumból állnak. Ezeket clustereknek nevezzük. Egy cluster állhat elemi alkatrészekből, például processzorokból, de tartalmazhatnak további clustereket is. Így tetszőleges mélységig bontható fel a rendszer, lehetővé téve a tervezőnek, hogy mindig a megfelelő absztrakciós szinttel kelljen csak foglalkoznia.

2.4 Parametrizált terhelés.

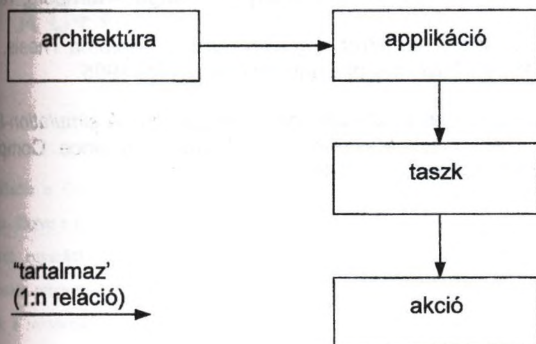
Eddig csak a hardver architektúra modellezéséről volt szó. A hardveren futó programok összessége, a terhelés (workload) jelentősen befolyásolja egy számítógép hibatűrését éppúgy, mint a teljesítményét. Bizonyos jellegű feladatokra írott szoftvert a struktúrájához legjobban alkalmazkodó hardveren lehet eredményesen futtatni. Hogy melyik programhoz melyik architektúra alkalmas, az csak néha látszik első pillantásra. A kezelhetőség érdekében a terhelést parametrizálni kell. Ez lehetővé teszi modellezését, így már a tervezés során kiderül az alkalmasság.

Első közelítésként a terhelést applikációkra bontjuk. Egy-egy applikáció nem más, mint egy felhasználói program. Ezekből több is futhat egyszerre a rendszeren. Ezek párhuzamos programok, tehát több taszkból állnak (Szokták ezeket is processznek, vagy workload processznek nevezni, de ne tévesszük össze őket a szimulátort alkotó processzekkel).

Minden ilyen taszk - nagyon durván - az ábrán látható ciklussal jellemezhető [SIMO 95]. A szimulációs rendszerünk egy ennél bonyolultabb sémát használ. A ciklusban több különféle akció definiálható, például üzenet küldése, fogadása, várakozás (számítási művelet szimulációja) stb. Egy taszk tehát jellemezhető a ciklusban található lépésekkel és az iterációk számával.

A teljes applikációt leírja a taszkjainak száma, azok leírása és a kommunikációs topológia, amely szintén kinyerhető az egyes taszkokból.

A terhelés osztályainak hierarchiája



A JUST megvalósításában az applikáció-osztályokat egy félkész applikáció-csontvázból a felhasználó definiálhatja a vezérlőprogramban, megadhatja a taszkok számát, és az egyes taszkok belső ciklusát. A ciklust alkotó akciók szintén Java objektumok, ezekből is lehetséges újakat definiálni, de ebből vannak készen használhatóak is.

A terhelés leképezése az architektúrára egy táblázattal történik, ez minden processzt egy processzorhoz rendel hozzá. Erre a célra szolgáló eljárások segítségével a hozzárendelés megváltoztatható, például processzorhiba után egy másik veszi át a hibás egység szerepét. A processz ekkor újraindulhat elölről, vagy a legutóbbi rögzített ellenőrzési ponttól.

3. A fejlesztés állása és további tervek

A JUST rendszer jelenleg még a fejlesztés kezdeti fázisában található, a specifikáció még nem végleges. Tanszékünkön informatikus hallgatók bevonásával folyik a munka. Az első működő verzió várhatóan 1997-re készen lesz. Reményeim szerint a szimulátort eredményesen alkalmazhatjuk majd a kutatáshoz.

Legfontosabb felhasználási iránya a terhelés minél általánosabb parametrizálásának kidolgozása lesz, de ehhez megbízhatóan kell működni az architektúra szimulációjának.

Irodalom

[HEIN 94] Hein, A. *Evaluation Report on Modelling Tools*. Internal Report 3/94 of the Dept. of Computer Science III. at the University of Erlangen-Nürnberg, 1994.

[SIMON 95] Simon G. *Workload Modeling with SimPar*. Diploma Thesis. Dept. of Computer Science III. at the University of Erlangen-Nürnberg, 1995.

[GOSW 93] Goswami, Kumar K. *Design for Dependability: A simulation-Based Approach*. PhD Thesis, Center for Reliable and High-Performance Computing University of Illinois at Urbana-Champaign, 1993.

A C++ nyelv lehetőségei és korlátai

Vég Csaba, vega@math.klte.hu

KLTE Információ Technológiai Tanszék

Az előadás a C++ nyelv objektum-orientált eszközkészletének néhány lehetséges felhasználási módját, illetve a nyelv koncepciójának egyes ellentmondásos elemeit ismerteti. A nyelv előnyeit és korlátait egyaránt a C++ gyakorlatias jellegéből származtathatjuk. A nyelvet programozók készítették programozók számára. Így előnye, hogy nem egy íróasztal mellett készült konstrukció, hanem a mindennapok valós és általában nagy méretű feladatainak megvalósítására alkalmas. Ugyanez a gyakorlatias szemlélet azonban néha hátrányt is jelent, mivel az eszközök általában nem egy átgondolt, elméletileg megalapozott szemléletet tükröznek. Más szavakkal: a nyelvbe általában nem egy koncepció teljes spektruma, a lehetőség összes változata kerül, hanem a nyelvet többnyire egyedi, a korábbiakhoz szervesen nem kapcsolódó elemekkel bővítik. Ugyancsak egyre terhesebb és egyre inkább felesleges koloncnak tűnő örökség, az eredeti C nyelv elemeihez való ragaszkodás, amely néha az áttekinthetőnek látszó programrészlet logikátlan (pontosabban C és nem C++ logikájú) működését eredményezi.

Bezárás

példánkban egy vermet kezelő hagyományos C programból indulunk ki. A verem mentációjára a lehető legegyszerűbb módszert, a tömbbel és egy egész típusú veremmutatóval ló ábrázolást választjuk. A verem használatát a példában egy egyszerű függvénnyel, a verem két elemén végrehajtott kétoperandusú művelettel mutatjuk be. A példa egy Postscript értelmező mben tárolt szám vagy szöveg operandusok, amely felső, adott számú elemén adott műveleteket tunk végre) erősen egyszerűsített változata.

```
#include <stdio.h>

#define     stLEN 40
double st[ stLEN ];  int sp = -1;

void Op(char c, double st[], int *sp) {
  double y = st[ (*sp)--],  x = st[ (*sp)-- ];
  switch( c ) {
   case '+': x += y; break;
   case '-': x -= y; break;
   case '*': x *= y; break;
   case '/': x /= y; break;     }
  st[ ++(*sp) ] = x;
}
```

```
void main() {
    st[ ++sp ] = 1; st[ ++sp ] = 2; st[ ++sp ] = 3;
    Op('*', st, &sp); Op('+', st, &sp);
    printf( "%g\n", st[ sp-- ] );
}
```

A hagyományos szemlélettel megírt programban szétválik az adatszerkezet és annak kezelése, működés. Sőt, mivel nincs olyan programozó, aki a verem push és pop műveleteire külön függvé írna (tovább tartana a függvény hívása, mint a tényleges művelet), ezért a vezérlés és implementáció sem válik el egymástól, azaz az absztrakt objektum (a verem) és annak absztra műveletei (push, pop) is csak megvalósításukban jelennek meg. Ez különösen akkor hátrányos, meg kell változtatnunk az objektum implementációját.

Az objektum-orientált szemlélet *egységbezárásnak* nevezett technikájával az adatot és az azt keze programrészleteket egyetlen egységben, az *objektumban* adhatjuk meg. Az átírt progr fordításának eredménye valószínűleg egyetlen gépi kódú utasításában sem fog eltérni C nyel megfelelőjétől. A szemlélet jelölése "mindössze" a logikailag összetartozó adat- és végreha elemek egyetlen egységben történő definiálását teszi lehetővé. Ezzel arra is lehetőségünk nyíl hogy az objektumot az implementációtól függetlenül, egy adott protokollon keresztül érjük el. objektum *bezárása* (encapsulation) az objektumot egy *felületre* (interface) és egy *megvalósítás* (implementation) bontja, valamint az objektum elérését csak a felületen keresztül engedélyezi. Ebb az objektum egyes implementációs elemei megváltoztathatók anélkül, hogy ez az objektum hatá túli módosításokat is eredményezne, azaz a kis változás nem fog továbbgyűrűzni.

```
#include <iostream.h>

    const  stLEN = 20;

struct Stack { double s[ stLEN ];      int    sp;
    Stack() { sp=-1; }
    void Push( double v ) { s[ ++sp ] = v; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
  };

struct RPN : Stack {
    RPN() :Stack() {}
    void Op( char c ) { double y = pop(),  x = pop();
        switch( c ) {
          case '+': x += y; break;
          case '-': x -= y; break;
          case '*': x *= y; break;
          case '/': x /= y; break;
        }
        Push( x );  return *this; }
  };
```

```
void main() { Stack s;
  s.Push( 1 ); s.Push( 2 ); s.Push( 3 ); s.Op( '*' ); s.Op( '+' );
  cout << s.pop() << 'n';
}
```

Objektum-csővezeték (object-pipe)

jektumokkal kapcsolatos műveletek alapvetően a környezetét módosító akció, illetve a belső t megváltozását eredményező átalakulás kettősére bonthatók. Az *akció* műveletei tikusan olyan módszerekkel implementálhatók, amelyek értéket utalnak vissza és ezek az k a hívó programban valamilyen változást indukálnak. A tisztán az objektum *átalakulását* ényező műveleteknek ezért nincs visszatérési értékük. Ezeknek a műveleteknek speciális lásával egy-egy műveletsorozat *objektum-csővezetékként* (object-pipe) rövidebben írható le. A y a következő: a tiszta transzformáció műveletei, amelyeknek nincs visszatérési értékük, k vissza magát az objektumot. Ezzel a transzformáció eredményén újabb transzformáció tó végre, azon újabb transzformáció hajtható végre..., azaz az objektumon végrehajtott formáció-sorozat egy csővezetékbe (pipe) rendezhető. Az objektum "végighalad" a (Unix pipe-hasonló) csővezetéken, miközben azon a megadott transzformációk sorban végrehajtódnak.

```
#include <iostream.h>

  const  stLEN = 20;

struct Stack {   double s[ stLEN ];    int    sp;
  Stack() { sp=-1; }
  Stack& Push( double v ) { s[ ++sp ] = v; return *this; }
  double  pop() { return s[ sp-- ]; }
  };

struct RPN : Stack {
  RPN() :Stack() {}
  RPN& Push( double v ) { Stack::Push( v ); return *this; }
  RPN& Op( char c ) { double y = pop(),  x = pop();
    switch( c ) {
      case '+': x += y; break;
      case '-': x -= y; break;
      case '*': x *= y; break;
      case '/': x /= y; break;
    }
    Push( x );  return *this; }
};

void main() {
  cout << RPN().Push( 1 ).Push( 2 ).Push( 3 ).Op('*').Op('+').pop();
}
```

Sajnos a csővezeték, pontosabban az egy adott objektumon végrehajtott transzformáció-
csővezeték-jellegű megadása még nem koncepcionális eleme a C++-nak, így ezt a jellegzetes
származtatott osztály esetén csak úgy tudjuk megőrizni, ha a transzformációt újra definiáljuk
műveletet delegáljuk az ősosztálynak.

Változó méretű objektumok

Az objektumok saját belső állapotuk mellett viselkedésüket is ismerik. A belső szerkezet a
segítségével elrejthető, így nem közvetlenül a külső vezérlés módosítja az adatokat, hanem
csak az objektum határát, érintkezési pontjait jelentő felületen keresztül érhetők el. A
felhasználásával intelligensebb, változó méretű és szerkezetű objektumok adhatók meg,
felhasználó szeme elől elrejtve szükség szerint módosíthatják belső méreteiket, esetleg felépítésü

A verem objektum esetén elképzelhető, hogy mélyen egymásbaágyazott kifejezések esetén nem
elegendő méretű a tömb. A verem implementációjának egyszerű megváltoztatásával
meghatározhatjuk a tömb méretét. Az alapértelmezett-operandusérték megadásával nem
módosítanunk az objektum eddigi hívásain. A C mutató-aritmetikájának köszönhetően az
felhasználó többi módszeren sem kell változtatnunk. Az implementáció további kis módosítá
elérhetjük, hogy a tömb mérete futás közben, az igényeknek megfelelően változzon.

```
struct Stack {
    int    len;
    double *s;
    int    sp;

    Stack& chk() { if( len<=sp+1 ) { double *z= s; int l= len; double
                                    s = new double [ len = sp+stLEN+1 ];
                                    for( ; --l >= 0; s[l] = z[l] ) ;
                                  }
                   return *this;  }
    Stack( int l = stLEN ) { sp = -1; s = new double [ len = l ]; }
   ~Stack() { delete s; }
    Stack& Push( double v ) { chk(); s[ ++sp ] = v; return *this; }
    double  pop() { return s[ sp-- ]; }
};
```

Az implementáció ugyanilyen könnyen láncolt listává alakítható át. Az implementáció
során sem a hívás-sorozaton, de még a származtatott osztályon sem kell változtatnunk.

Változó szerkezetű objektumok

objektum bezárása, a megvalósítás felület mögé rejtése elvezet az önállóan definiálható felületek, absztrakt osztályok technikájáig. Az absztrakt osztályok csak az elérés protokollját adják meg, melyet az abból származtatott konkrét osztályok töltenek fel konkrét implementációs tartalommal. Absztrakt osztályok és virtuális műveleteik segítségével a programban egyidejűleg létezhet egy absztrakt objektum több implementációja is, amelyek közül az objektum készítésekor választhatjuk az éppen szükségeset.

```
...
struct Stack {
    virtual  Stack& Push( double v ) = 0;
    virtual ~Stack()        = 0;
    virtual double top() = 0;
    virtual double pop() = 0;
  };

struct vaStack :Stack { ... };
struct lStack : Stack { ... }

struct RPN {   Stack *S;
    RPN(Stack *s) { S=s; }
    ~RPN() { delete S; }
    RPN& Push( double v ) { S->Push( v ); return *this; }
    ...
  };

void main() {
    cout << RPN((Stack *)new lStack).Push( 1 ).Push( 2 ).Push( 3 ).
                              Op('*').Op('+') .pop() << '\n';
  }
```

Bizonyos kiegészítő információk gyűjtésével és értelmezésével megoldható, hogy bizonyos feltételek teljesülésekor, *használat közben* maga az objektum váltsa implementációs szerkezetét. Nagyszámú adat esetén a változó méretű tömbként implementált verem egyre nagyobb memóriaterületet foglal, másol át, majd szabadít fel. Ezért célszerű, hogy adott elemszám elérésekor a verem automatikusan térjen át a kisebb memóriaterületek foglalását/felszabadítását jelentő láncolt lista-implementációra, illetve az elemszám adott határ alá történő csökkenése esetén térjen vissza a tömb-implementációra.

Változó típusú objektumok

Egy indexelést (indexhez érték rendelése) egyszerűsítsünk le arra a feladatra, hogy egy érték szerepel-e egy halomban, vagy sem. Első változatunk az értékeket egy tömbben tárolja.

```
    typedef enum {FALSE, TRUE} BOOL;
    const   int LEN = 20;

struct Arr {  int     n;      double a[LEN];

    int src(double v) {
        for(int i=n; i>=0; i--)
            if( a[i] == v ) return i;
        return -1; }
    Arr() { n = -1; }
    Arr& Add(double v) { a[++n] = v; return *this; }
    BOOL in(double v)  { return src(v)!=-1;  };
    };
```

A keresést jelentősen felgyorsíthatjuk, ha az elemeket a tömbben sorrendben helyezzük el. Minden egyes módosítást követő rendezés azonban a beírásokat lassítja le. A halommal kapcsolatos műveletek esetén általában nagy számú beírást nagy számú lekérdezés követ. Ezért optimális megoldáshoz vezet, ha csak az első kérés előtt rendezzük le a tömböt. Az objektumnak egy "piszkosság-bittel" számon kell tartani, hogy a tömb egy módosítás utáni, vagy már rendezett állapotban van.

```
struct sArr :Arr {  BOOL dirty;

    void srt() { /* ... rendezés ... */ }
    int src(double v) { /* ... (logaritmikus) keresés ... */ }
    sArr() :Arr(), dirty(TRUE) {}
    BOOL in(double v)  {
        if(dirty) srt(), dirty=FALSE;
        return src(v)!=-1;
        };
    };
```

A rendezett tömb a rendezetlen speciális esete, más szóval pontosítása. A végrehajtás során a módosításokat követő első lekérdezés előtt az objektumot egy származtatott, azaz egy szigorúbb feltételt teljesítő objektummá alakítjuk, hogy ezután a speciális változaton hatékonyabban működő módszereket használhassuk. A következő asszociatív tömbbe a beírás és a lekérdezés műveletét egyaránt a [] operátor valósítja meg. Ez a kényelmesebb és áttekinthetőbb vezérlési mód azonban nem kapcsolható össze a tömb automatikus rendezésével. Ugyanis a C++ nem engedi ugyanazon nevű, balértéket, illetve csak jobbértéket visszautaló két módszer definiálását.

```
const   int LEN = 20;
struct Arr {
   struct pair { int ndx; double v;
        void set(int N, double V = 0) { ndx = N, v = V; }
      };
   int    n;  pair a[LEN];
   Arr() { a[ n = 0 ].set(0); }
   int src(int ndx) {
        for(int i=n; i>=0; i--)  if( a[i].ndx == ndx ) return i;
        return -1; }
   double& operator [](int ndx) { int j;
        if( (j=src( ndx ))==-1 ) { j = a[n].v ? ++n : n; a[j].set(ndx); }
        return a[j].v; }
};
```

Konverzió

C++ lehetőséget ad objektumok más objektummá történő átalakítására, a *konverzióra*. A konverzióra alapvetően két lehetőség van. Maga a konstruktor is konverzióként működik, mivel operandusaiból elkészíti az adott osztály objektumát. A konverzió megadásának másik lehetséges módja a konverziós operátorok segítségével történik. Konstruktorral adott típusból az *objektumba*, konverziós operátorral *objektumból* az adott típussá történő transzformációt írható le. Mivel konverzió megadásakor a másik típusnak ismertnek kell lenni, ezért két osztály közötti átalakítások csak a később definiált osztály módszerei között adhatók meg. A két objektum elhelyezkedésének felcserélése így ahhoz vezet, hogy a konstruktort operátorrá, az operátort pedig konstruktorrá kell tenni. Az osztályok definíciós sorrendjének egyszerű felcserélése így a program jelentősebb átdolgozását is implikálhatja. A következetlenség valószínűleg arra vezethető vissza, hogy eredetileg csak az (előredefiniált, azaz a program minden pontján ismert) elemi típusokkal történő konverzió volt megengedett. Ezek a konverziós lehetőségek így csak egy részleges megoldás lehetőségeinek koncepció nélküli bővítése.

Bonyolultabb szerkezetek

C++-ban a könnyebb implementálhatóság miatt a dinamikusság összefonódik a dinamikus terület használatával. Ezért a bonyolultabb szerkezetek, például a többször hivatkozott struktúrák, illetve a változó, fordításkor még nem, csak futási időben megadott méretű adatok is csak mutatókon keresztül, áttételesen kezelhetők. A következő String osztály tényleges karakterei csak két mutatón keresztül érhetők el (egy a hivatkozás-számláló, egy a fordításkor még nem rögzített hosszúságú string miatt).

```
#include "string.h"

class String {
    struct str { int cnt; int len; char *str;
        str( int l ) { cnt = 1, len = l, str = new char [ l ]; }
        ~str()       { delete str; }
    };
    str *s;

    str *addRef() { if( s ) s->cnt++; return s; }
    void delRef() { if( s && !--s->cnt ) delete s; }
  public:
    String()            { s = 0; }
    String(char *sz){ s= new str(strlen(sz)), memcpy(s->str,sz,s->len);}
    String(String& S) { s = S.addRef(); }
   ~String()            { delRef(); }
    String& operator =(String& S) { S.addRef(), delRef(); s = S.s;
                                    return *this; }
  };

void main() { String s1 = "hello", s2 = "world";
    s2 = s1; s1 = "ok";
  }
```

Fix kezdőrészt követő változó méretű adatok kezelése csak a C++ szemléletéből kilépve, nehézkes, közvetett módon írhatók le. (Többször hivatkozott objektum közvetlen, nem egy mutatót tartalmazó objektum felhasználásával történő definíciója pedig gyakorlatilag lehetetlen).

```
class String {
    struct str { int cnt; int len; char str[1];
        void set( char *sz, int l ) { cnt = 1, len = l,
                                      memcpy(str,sz,len); }
        };
    str *s;
   ...
    String(char *sz){ int l= strlen(sz);
                      (void *)s=new char[sizeof(str)-1+l]; s->set(sz,l);}
   ...
  };
```

Irodalomjegyzék

Bjarne Stroustrup. The C++ programming language. (2nd ed.) Addison-Wesley, 1991.

J. Rumbaugh et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

Grady Booch. Object-Oriented Analysis and Design with Aplications. (2nd ed.) Benjamin/Cummings Publishing Company, Inc., 1994.

Rendszerelemzés és jogtudomány
avagy
Alkalmas-e a magyar jogrendszer arra, hogy ...
(Gáspár András, Harang BT, Budapest)

előadás a "Rendszerelemzéssel támogatott jogtudomány" témakörébe tartozik [17,18,19]. Azon a hipotézisen alapul, hogy a jogállam körülményei között megalkotható egy olyan rendszerleíró-nyelv, mely alkalmas az eljárásjog programszerű/matematikai leírására. A jogállamban ugyanis követelmény, hogy az eljárásjog szabályai világosak és egyértelműek legyenek, a jogi eljárások lefutása pedig kiszámítható legyen.

Tévedés lenne azt hinni, hogy nulláról kell indulnunk. Állításunk demonstrálására alkalmazzuk képzeletben először a szimulációs, majd később a teoretikus rendszerelemzés módszerét.

Tegyük fel először, hogy a jogállam eljárásjogát, SIMULA-bázisú szimulációs szándékkal közelítjük meg. Például vizsgálni akarjuk a jogalanyok viszontagságait az eljárásjog útvesztőiben. Ekkor a SIMULA-program számára egyrészt olyan folyamat-osztályokat kellene definiálnunk, mint "az ügyfél", "a közigazgatási ügyintéző", "a bíróság", "az Alkotmánybíróság", "a jogalkotó", stb., másrészt viszont, részben ezen folyamatok részeként, a képzeletbeli szimulátor számára meg kellene adni a hatályos eljárásjogot is:
- az államigazgatási eljárás általános szabályairól szóló 1957. IV. törvényt (a továbbiakban Áe.),
- a Polgári Perrendtartásról szóló 1952. évi III. törvényt (a továbbiakban Pp.),
- a büntetőeljárásról szóló 1973. I. törvényt (a továbbiakban Be.),
- az Alkotmánybíróságról 1989.évi XXXII. törvényt (a továbbiakban ABtv.),
- a jogszabályalkotásról szóló 1987. évi XI. törvényt (a továbbiakban Jat.),
- az emberi jogok és az alapvető szabadságok védelméről szóló EGYEZMÉNYt, azaz az 1993. évi XXXI. törvényt (a továbbiakban EGYEZMÉNY),
- az adózás rendjéről szóló 1990.évi XCI. törvényt (a továbbiakban Art.),
- a statisztikáról szóló 1993. évi XLVI. törvényt (a továbbiakban Stt.),
- stb.

SIMULÁn nevelkedett rendszermodellezőként azonnal hozzálátnánk a dekompozicióhoz és osztályozni kezdenénk. És csodák csodája, kiderülne, hogy az Art. és az Stt. számára az Áe. super-CLASS, hiszen:
- az Art. 4. §. (1) bek. szerint "Ha e törvény ... másként nem rendelkezik, az adóügyekben az Áe. rendelkezéseit kell alkalmazni"
- az Stt. 4. §. (1) bek. szerint pedig "A KSH ... országos hatáskörű közigazgatási szerv", aminek egyenes következménye az Áe. "szervi hatálya".

Kiderülne továbbá, hogy a Pp. egy olyan modul, amelynek
- I-III. és V. része az általános polgári peres eljárást szabályozza, így super-CLASS szerepet tölt be a IV. részben konkretizált "Különleges eljárások", mint részosztályok számára.

Felismernénk a jogszabályok diktálta folyamatokat:
- az Áe.-ben az alapeljárást, a (törvényességi-) felügyeleti intézkedést, a fellebbezést, a közigazgatási eljárás bírósági felülvizsgálatát, stb.

- a Pp.-ben a per megindítását, az I. fokú eljárást, a II. fokú eljárást, a felülvizsgálatot, a perújítást, stb.

Észlelnénk hogy a megkeresés master/slave módú hívás (CALL-jellegű), míg az ügyáttétel elhagyásos hívás (RESUME-jellegű). Látnánk, hogy az időmúlás a legtöbb folyamatban TIME-INTERRUPTot okoz. Már a szimulátorba beépített rendszermodell elkészítésekor felhasználhatnánk a matematika, a számítástudomány és rendszerelemzés felhalmozott ismereteit:
- az ügyek képzeletbeli sorokban várakoznának az ügyintézésre,
- kiszolgálásuk jog által definiált prioritási sorrendben történne,
- vizsgálhatnánk a jogszabályi esetek szétválasztásának teljességét,
- a jogi folyamatok végességét illetve átlagos időtartamát,
- a párhuzamos jogi folyamatok korrekt szinkronizálását,
- a végtelen ciklusok és a patt-helyzetek kizártságát.

Szemünk előtt lebeg tehát egy SIMULA-program, melyben mind a jogszabály-hierarchia leírására, mind pedig a jogalanyok tevékenységének megadására a SIMULA nyelv osztály-részosztály, objektum és folyamat fogalmát használnánk fel. Egy sor korábbi publikációban már kimutattam, hogy nincs ebben semmi különös, hiszen a SIMULA nyelv "közös alapnyelv" (Common Base Language) kívánt lenni a rendszermodellezés, a rendszerszimuláció számára, ezért rendszerleíróképességének biztosítására a nyelv tervezői különös hangsúlyt helyeztek [1,7,8].

2.2. A SIMULA nyelv a szimulációs célnyelvek világából a számítástudományba emelte a rendszerleírással programozás paradigmáját. Sajnos azonban a számítógépek valósága megnyomorította a rendszerleírást [18].

A nyelv tervezésekor használatos egyprocesszoros gépek miatt az igazi párhuzamosság helyett ki kellett alakítani a kvázi-párhuzamos végrehajtás modelljét. A számítógépek primitív utasításkészletéhez igazodva, elemi lépésekre kellett felbontani egy sor matematikailag jól definiált dolgot. Például egy szimultán differenciálegyenlet-rendszerrel könnyen definiálható folytonos függvény helyett, valamilyen numerikus közelítést (pl. a Runge-Kutta módszert) algoritmizálva, meg kellett adni az egyenletrendszer közelítő megoldását. Nem voltak kidolgozva a logikai programozás elvei sem, így programozni kellett, a rendszerre vonatkozó logikai állítások (egyenletek) megoldásának megkeresését is.

A SIMULA rendszerleíró képességének eltorzulását a számítógép okozta, kézenfekvő volt tehát a számítógépet száműzni és egy számítógéptől garantáltan mentes "papír-nyelvben" megfogalmazni az ideális SIMULÁt, a létező SIMULA kritikáját. Nem lehetett szó ugyanis a SIMULA módosításáról, hiszen a SIMULA Szabványosítási Bizottság kivont karddal őrködött a nagynehezen megszülető implementációk egyformasága felett. Másfelől viszont "papír-nyelvet" kívánt a megrendelésekben megjelenő szimulációs feladatok specifikációja

2.3. A DELTA rendszerleírónyelv tervezését 1973-ben kezdték meg [5]. A szigorú szintaxisú nyelv 1975-ös specifikációja új kontrukcióként tartalmazta a WHILE-ciklushoz hasonló szintaxisú, elemi folyamatleírást, amely megszakítható, felfüggeszthető és újraaktiválható. Az elemi folyamat mindaddig tart, amíg a WHILE alapszót követő feltétel, az úgynevezett felügyelt feltétel, hamissá nem lesz.

elemi folyamatelőírás egyik változatában megadható egy TULAJDON-
LEÍRÁS, ami tulajdonképpen egy relációsorozatot jelent:

{ reláció1, reláció2, ..., relációN }

Továbbá megadható a RENDSZERATTRIBUTUMOK egy listája is:

attr1, attr2, ..., attrK

elemi folyamatelőírás e változatának alakja:

```
WHILE felügyelt_feltétel
LET tulajdonságleírás
DEFINE rendszerattributumok
```

fenti utasítás jelentése: Az elemi folyamat lezajlása alatt a
jelölt rendszerattributumoknak úgy kell változni, hogy a tulajdon-
ságleírásban szereplő relációk mind igaz értéket adjanak. A folyamat
addig tart, amig a felügyelt feltétel teljesül. Például:

```
WHILE TIME < 10
  LET { X ** 3 - 3 * X ** 2 + 3 * X - 1 = 0 }
  DEFINE X;
```

"WHILE TRUE" rész el is hagyható.

elemi folyamatelőírás másik változatának alakja:

```
WHILE felügyelt_feltétel WAIT
```

utasítására a folyamat várakozni fog mindaddig, amíg a felügyelt
feltétel fennáll. Az utasítás elméleti jelentőségét egy sor
tanulmányban több szerző is leírta. A SIMULA irodalomban mindez a
"WAITUNTIL" eljárásról szóló, SIMULA Newsletter [4] cikkekben
jelent meg.

tulajdonságleírás segítségével olyan utasítást is be lehetett
vezetni a DELTA nyelvbe, amely szükségtelenné teszi egy számítás
algoritmikus részletezését. Az utasítás alakja:

```
DETERMINE attributumok_listája WHERE tulajdomságleírás
```

utasására a felsorolt attributumok úgy kapnak értéket, hogy a tulaj-
donságleírásbeli relációk mindegyike igaz értékű legyen. Például az
alábbi leírás-részlet

```
X,Y,Z,N : INTEGER;
...
DETERMINE X,Y,Z,N WHERE { X ** N + Y ** N = Z ** N, N > 2 }
```

meghatározza a nagy Fermat-sejtés "valamelyik" nem létező megoldását.

A DELTA-eljárásoknak 3 fajtája van
- a kiszámítást végző, rendszerállapotot nem változtató, mellékhatás nélküli függvényeljárás (FUNCTION),
- a 0-időtartam alatt bekövetkező eseményeket kiváltó, rendszerállapotváltoztató eljárás (PROCEDURE), valamint
- az időtrabló folyamatot leíró eljárás (TASK PROCEDURE)

A PROCEDURE és FUNCTION hívásakor hasonló dolog történik, mint a SIMULA nyelvben: A hívó várakozni fog a hívott befejeződésére, majd a befejeződéskor a vezérlés visszatér a hívóhoz (master/slave módú hívás).

A DELTA nyelvbeli NEW utasítás hatására új objektum születik, amely akárcsak egy SIMULA-objektum, "működésbe lép". A DELTA-objektum azonban valóban párhuzamosan működik a többi DELTA-objektummal. Életműködését nem egyszerűen egy utasítássorozatból álló forgatókönyv definiálja, mint a SIMULA nyelvben, hanem az osztály törzsében külön kell definiálni az úgynevezett PRIME TASK-ot. A deklaráció alakja:

```
PRIME TASK BEGIN
    ....
    END TASK
```

Minden TASK-nak saját vezérlési (hívási, visszatérési, megszakítási stb.) adminisztrációja van: a zajló tevékenységekről (a pillanatnyi elfoglaltságról), továbbá a későbbi tevékenységekről.

A későbbi teendők listáján (agenda) lévő TASK-ok majd érkezési, illetve különböző prioritások esetén az INTERRUPT prioritásának megfelelő sorrendben automatikusan kerülnek végrehajtására.

Az INTERRUPT küldő utasítás egyszerűsített formája a következő:

```
INTERRUPT mely komponensben okozzon megszakítást a task
BY        mely TASK-ot kell végrehajtani
POWER     a megszakítás ereje
```

Csakis TASK-ok, és azon belül is, csakis időben zajló folyamatok szakíthatók meg. Az időben zajló folyamatoknak a megszakításokkal szemben ellenállásuk (RESISTANCE) van. A pillanatnyi ellenállás értéke, az időben zajló folyamat adott pillanatban zajló elemi folyamatának részeként adható meg.

Itt az ideje annak, hogy bemutassuk a DELTA nyelvbeli elemi folyamat (WHILE) általánosabb alakjait. Első változat:

```
WHILE felügyelt_feltétel
  LET tulajdonságleírás
  DEFINE attributumok listája
  RESISTANCE ellenállás a megszakításokkal szemben
```

A második változat:

```
WHILE felügyelt_feltétel
  WAIT
  RESISTANCE ellenállás a megszakításokkal szemben
```

Mivel adható meg az INTERRUPT ereje és az elemi folyamat ellenállása? Mindenki természetes számokat várna, azonban a DELTA nyelvben a prioritások értékét azonosítókkal nevezik meg és az azonosítók egymáshoz való viszonyát prioritás deklaráció írja le. Hatására egy mátrix keletkezik, melynek elemei azt definiálják, hogy milyen erejű INTERRUPT, milyen ellenállású TASK-ot képes felfüggeszteni.

elemi folyamatelőírásnak opcionális része a felfüggesztés vezetés (EXIT) és a felfüggesztés befejezés (REENTRY) adása. Sőt a felfüggesztés bevezetése és befejezése az ADVANCE asítás segítségével megszakítható. A felfüggesztendő TASK-nak tát lehetősége van felülbírálni a "rosszkor jött" nagy prioritású egszakítást.

rdemes még két hasonló utasítást bemutatni:

CONCLUDE Hatására befejeződik a pillanatnyilag zajló TASK.

TERMINATE Terminálja a komponenst. Hatása egyenértékű egy PRIME TASK-ra vonatkozó CONCLUDE utasítással.

z INTERRUPT utasítás segítségével eddig bemutatott TASK-hívást ÜLSŐ TASK-HÍVÁSNAK nevezzük. Egy komponens azonban külső INTERRUPT élkül is, azaz saját elhatározásából is belekezdhet egy nagyobb rioritású tevékenységbe, vagy elhelyezhet az éppen zajló tevékeny- égénél kisebb prioritású tevékenységet a későbbi teendők listájára agenda). Erre ugyancsak az INTERRUPT utasítás szolgál. Egyszerű- ített alakja a már ismert INTERRUPT utasítás továbbegyszerűsítése:

INTERRUPT
BY mely task-ot kell végrehajtani
POWER a megszakítás ereje

z INTERRUPT utasítás segítségével történő TASK-hívás az úgynevezett lhagyásos TASK-hívás. A hívó TASK továbbzajlik és nem kér vissza- elentést a hívott TASK lefutásáról. A TASK-hívás másik esetében hívó bevárja a hívott lefutását és csak azután folytatja futását. z ilyen TASK-hívás egyszerűsített alakja a DELTA rendszerleíró- yelvben:

EXECUTE mely task-ot kell végrehajtani

Hosszan sorolhatnánk még a TASK-adminisztrációt kezelő különböző tasításokat. Annyi azonban már az eddig elmondottakból is látszik, ogy a DELTA-leírás alkalmazása a jogszabályokat követő állam- épezet leírására csakis akkor lehet sikeres, ha az államgépezet űködése - legalábbis eljárásjogi értelemben - valóban hasonlít gy operációs rendszer működéséhez. Mindenesetre lánctalpak sikorgása helyett zümmögnie kell.

bár a DELTA tervezői száműzték a SIMULA nyelvet megnyomorító számítógépet, de ellenezték azt, hogy a DELTA nyelv nagyon messzire ltávolodjék a korabeli számítógépek valóságától, egy a későbbiekben egtervezendő, de akkor még csak képzeletbeli GAMMA nyelvtől [4]. Ezért folyamatosan keresték a DELTA-koncepciók számítás- technikai megvalósíthatóságát is.

A DELTA egyes tervezési elképzelései végülis nem a GAMMA nyelvben értek számítástechnikai talajt, hanem más nyelvekben, például
- a párhuzamosságra vonatkozó elképzelések az ADA nyelvben és a BETA nyelvben [6],
- az igaz logikai állításokkal történő programozás pedig a logikai programozás eszközeiben, például a PROLOG nyelv továbbfejlesztéseiben.

Sokan úgy érezték, hogy a DELTA kiürült: koncepcióit továbbfejlesztették. Ráadásul a magasszintű, párhuzamos programozást megvalósító számítógépes nyelvek (pl. az ADA, a konkurens PASCAL) megjelenésével átmenetileg lecsökkent A SZÁMÍTÁSTECHNIKA IGÉNYE, egy univerzális, számítógép- és programozási-nyelv-független rendszerleíró-nyelv iránt.

2.5. A DELTA rendszerleíró-nyelvet szép csendben elfelejtették. A DELTA mai jelentőségének megértéséhez le kell "ásnunk" egészen mélyre, a filozófiai-módszertani alapokhoz [7,8,9,10,11,12,13,14,15,16].

A számítógépes szimuláció, a rendszermodell számítógépes megelevenítésén alapuló, alapvetően EMPIRIKUS jellegű vizsgálat. Még akkor is, az, ha már pusztán a rendszermodell megalkotása is TEORETIKUS jellegű megfontolásokra épül.

A matematikát a gondolati modell TEORETIKUS vizsgálata szülte. A matematikai tudományok modellezés-tudományok. Modell-modellnyelv-elmélet készletet szolgáltatnak, kezelnek és előregyártanak.

Évszázadokon keresztül a matematikai módszerek voltak a legfontosabb rendszerelemzési módszerek. A 70-es évekre azonban nyilvánvalóvá vált, hogy a matematika nem képes lépést tartani a rendszervizsgálat igényeivel. A túl sok fajta és a túl sok komponensből álló rendszerek valósághű modelljei nem sok eredménnyel kecsegtettek a MATEMATIKAI KEZELHETŐSÉG szempontjából. Hiányozott a megfelelő MATEMATIKAI JELÖLÉS A RENDSZERÁLLAPOTOK IDŐBELI VÁLTOZÁSÁRA, A KOMPONENSEK LÉPÉSRŐL LÉPÉSRE TÖRTÉNŐ MÓDOSÍTÁSÁRA, EGYMÁSRAHATÁSÁRA. A lépésenkénti (diszkrét) változások leírására viszont jó eszközt szolgáltattak az időközben megszületett (számítógépes) algoritmikus nyelvek.

A DELTA - tudomásom szerint - elsőként egyesítette az algoritmikus nyelvek és a matematikai leírás előnyeit egyetlen "papír-nyelvben", elkerülve a számítógépes programnyelvek Prokrusztész-ágyát, utat nyitva a TEORETIKUS modellezés számára. Nincs akadálya annak, hogy a DELTA rendszerleírás matematikához közeli jelöléssel, matematikai részletet tartalmazzon. Leírhatók a matematikai definíciók is. A DELTA rendszerleírás alapján még matematikai bizonyítás, sőt matematikai elmélet felépítése is elképzelhető.

A rendelkezésre álló adatok szerint, a DELTA rendszerleíró-nyelv tervezői mégsem matematikai vizsgálatra szánták. Nem az volt a céljuk, hogy a DELTA konstrukciói teljes mértékben megfeleljenek a TEORETIKUS gondolati modellezés műveleteinek. A DELTA rendszerleíró-nyelvet nem arra "hegyezték ki", hogy a DELTA-leírások alapján
- elvégezhessék egy-egy modell jelenségeinek interpretációját, magyarázatát vagy, hogy
- bizonyíthassák egy-egy modell tulajdonságait, illetve hogy
- egy-egy modellhez megalkothassák a róla szóló elméletet.

Bár a tervezési cél más volt, a DELTA konstrukciói mégis ezeknek a követelményeknek is jelentős mértékben megfeleltek.

2.6. Nehogy azt higgye bárki, hogy mától valami teljesen új tevékenységet ajánlunk, DELTA rendszerleírásra alapozva.A komplex rendszerek vizsgálatával küszködő különböző szakemberek igen jól ismerik az "esettanulmány" fogalmát. Nem új ismeret, de nem árt megemlíteni

Így a DELTA rendszerleírás alapján TEORETIKUSan lehet tanulmányozni bonyolult rendszerek eset-kezelését, és az "esettanulmányokban" TEORETIKUSan lehet rámutatni a komponensek hibás vagy hiányos működési szabályaira.

Tudni azonban, hogy ilyenkor objektum-orientált analízisről, magyarul talán "OO rendszerelemzésről" van szó, mivelhogy objektum-orientált az egész SIMULA-DELTA-BETA paradigma.

Visszatérve a jogállam eljárásjogához, alkalmazzuk most képzeletben a teoretikus rendszerelemzés módszerét: Tegyük fel, hogy a jogállam eljárásjogát DELTA-bázisú OO rendszerelemzési szándékkal közelítjük meg.

Képzeljük el, hogy az eljárásjog leírására és a jogalanyok tevékenységeinek megadására - felhasználva a DELTA rendszerleíró-nyelv fogalmait - már elkészítettünk egy DELTA rendszerleírást. Kérdés:
- milyen jelenségek interpretációjára-magyarázatára alkalmas e DELTA rendszerleírás ?
- az eljárásjog mely tulajdonságai bizonyíthatók e DELTA rendszerleírás alapján ?
- milyen tartalmú elméletet lehet felépíteni e DELTA rendszerleírásra ?

Habár a kérdések jók, és néha már a jó kérdés is eredmény, de több eredménnyel kecsegtet az a megfogalmazás, hogy DELTA rendszerleírás alapján, DELTA-bázisú OO rendszerelemzéssel, vizsgálni kívánjuk az eljárásjog esetkezelését és esettanulmányokban rá akarunk mutatni az eljárásjog szereplőinek hibás vagy hiányos működési szabályaira.

Módszeresen kiválasztott esetek tanulmányozása esetén tevékenységünk nagyon hasonlít majd ahhoz, amit egy-egy szoftver-fejlesztés során/végén, minőségellenőrzéssel foglalkozó kollégáink végeznek a VERIFICATION TEST SET segítségével. Csakhogy mindenekelőtt szükség lenne hozzá a jogállam eljárásjogának VERIFICATION TEST SET-jére.

A kelet-európai rendszerváltozások és a európai jogharmonizációs törekvések egyaránt felpezsdítették a jogszabályalkotást. Súlyos kérdéssé vált a jogszabályrendszer hibamentessége. A hibamentesség különös hangsúlyt kap akkor, amikor új, nemzetek feletti ajánlással-szabályozással, kívánják kiváltani a "jól-bevált" hazai jogot. Célszerű tehát az új jogszabályrendszert széleskörű elemzésnek alávetni. Az elemzés azonban csak akkor lehet igazán hatékony, ha tárgya nem az eredeti tárgy, hanem egy többé-kevésbé "formalizált" rendszerleírás.

A fizika nyelve a matematika. A jogtudomány nyelve azonban se nem a matematika, se nem egy rendszerleíró- illetve számítógépes-nyelv. Így a jogi folyamatok elemzése során kétszeres fordítás szükséges:
- a jogi folyamatot le kell fordítani a rendszerleírónyelvre, majd
- a rendszerleírónyelven leírt modell elemzését követően, a rendszerelemzés eredményeit vissza kell fordítani a jogi folyamat nyelvére.

A formalizálás eszköze csakis olyan rendszerleírónyelv lehet, amely egyesíti magában a TELJES MATEMATIKAI jelölési apparátust, a számítógépes nyelvek OSZTÁLY, OBJEKTUM, FOLYAMAT és ALGORITMUS leírási képességével.

Habár a DELTA kiindulási alapként elfogadható, elégtelensége mégis nyilvánvaló. Éppen az lehet egy kutatás egyik feladata, hogy a jogtudomány jól kiválasztott területein, az elkészített DELTA-rendszerleírások alapján, rávilágítson a DELTA nyelv hiányosságaira és konkrét konstrukciókat dolgozzon ki a DELTA kiegészítésére, módosítására.

4. Elemzésre mindeddig konkrét közigazgatási ügyek kapcsán került sor. A közigazgatási szervnek kérelemre vagy hivatalból igazgatási eljárásban kellett (volna) elbírálnia valamit. A közigazgatási szerv azonban vagy hallgatott vagy kötelezettségét jogszabálysértő módon teljesítette. Hibásan értelmezte a jogot, elmulasztotta a határozathozatalt, határozata nem felelt meg az alaki előírásoknak, határozat helyett tájékoztatást adott, hatásköre hiányát állapította meg, de elmulasztotta átteni az ügyet, nem vizsgálta hatáskörét, elmulasztotta a számára kötelező felügyeleti eljárást, védte a mundér becsületét, stb. A közigazgatási szervek packázásai miatt összesen 7 közigazgatási pert indítottam.

A bíróságok azonban keresetemet, érdemi, megfellebezhetetlen határozat hiányában, általában idézés kibocsátása nélkül, elutasították. Fellebbezéseimben kimutattam, hogy a bíróságok alkotmánysértő jogszabályokat alkalmaztak továbbá határozataikat rendszeresen alkotmánysértő jogértelmezéssel alapozták meg. Egy tanulmányjellegű indítványomban ezt az Alkotmánybíróságnak is leírtam. Fellebbezéseim határozott kérelmet tartalmaztak arra vonatkozóan, hogy a másodfokon eljáró bíróságok, eljárásuk felfüggesztése mellett indítványozzák az AB eljárását a megnevezett alkotmánysértő jogszabályok illetve a kimutatott alkotmánysértő élő jog megsemmisítésére, egyes esetekben az alkotmánysértő jogalkotói mulasztás megszüntetésére. A bíróságok alkotmányossági kérelmeimet, minden esetben - következetesen - indoklás nélkül mellőzték.

A közigazgatási perekben született bírósági határozatok alapján megdöbbentő kép alakult ki egy olyan eljárásjogi hibasorozatról, melynek egyik következménye az, hogy:

"az illetékes felügyelet",
mulasztásos jogsértésének megszüntetését,
a független magyar bíróság,
közigazgatási perben hozott itélettel,
nem képes kikényszeríteni,
a "magyar Csernobilban".

Íme itt van a jogállam eljárásjogát ellenőrző VERIFICATION TEST SET 1. számú esete.

félreértések elkerülése végett azért nem árt tisztázni, hogy az eredmény ABSZTRAKT, vagyis
"magyar Csernobil"-ről még nem tudunk, de lehetséges olyan, a "magyar Csernobil" nem valamilyen atomenergetikai eseményre ...tal, hanem bármilyen hatósági felügyelet alá tartozó tevékenység "eseményére", pl.:

- bank-tevékenység,
- értékpapír-tevékenység,
- adóztatási+vám tevékenységek,
- nukleáris tevékenységek,
- vegyi- és robbanóanyaggyártás, szállítás, kezelés,
- hulladék szállítás és kezelés,
- (gyógyszerészeti-, orvosi-, mezőgazdasági-, stb.) labóratóriumi tevékenységek, stb.,

megfelelő vizsgálat esetén feltehetően ugyanilyen eredményre jutnának Kelet-Európa bármely, sőt Nyugat-Európa több országában is - ha egyáltalán megvizsgálnák a vizsgálandót.

Nem árt kihangsúlyozni azt sem, hogy megtörtént az első lépés is, amikor az Alkotmánybíróság - más ügyben ugyan -, de megállapította az Országgyűlés jogalkotói mulasztását a 72/1995. (XII.15.) AB számú határozatban.

Hivatkozások:
==========

[1] The Development of the SIMULA Languages.
ACM SIGPLAN Notices. Vol. 13. No. 8., August 1978.
[2] O.J.Dahl - B.Myhrhaug - K.Nygaard :
A SIMULA 67 nyelv definiciója.
Budapesti Műszaki Egyetem, Budapest, 1985. ISBN 963 421 438 X.
[3] Standard SIMULA.
Databehandling, Programsprâk - SIMULA.
SIS, Svensk Standard SS 63 61 14.
[4] SIMULA Newsletter.
Association of SIMULA Users.Oslo-Edinburgh-Stockholm 1973-1996.
[5] Holbaek-Hansen, E. - Händlykken, P. - Nygaard, K.:
System Description and the DELTA Language.
NCC, 1975. DELTA Project Report No. 4.
[6] Madsen, O.L. - Pedersen, B.M. - Nygaard, K. :
Object-oriented programming in the BETA programming language
Addison Wesley Publishing Company, 1993.
[7] Gáspár A.-Csáki P.-Visontay Gy.:
A szimulációs módszer és a SIMULA 67 nyelv.
NJSzT Rendszerelméleti Konferencia'79, Sopron.
Rendszerek szimulációja,3-36.o.
[8] Gáspár A.:
A SIMULA 67 és a számítástudomány.
Neumann Kongresszus,1979.
[9] Gáspár A.:
Vitaindító.
Bolyai János Matematikai Társulat, Csillag Pál Szimpózium, 1975.
Kézirat.
[10] Gáspár A.:
A modellezéstudományok jelenéről és jövőjéről.
BME Karközi Alkamazott Matematikai Munkaközössége, 1976/13.
Kézirat gyanánt.
[11] Stoff, V.:
Modell és filozófia. Kossuth, Budapest, 1973.

[12] Kocsondi, A.:
Modell-módszer. Akadémiai, Budapest, 1976.
[13] Fehér, M. - Hársing, L.:
A tudományos problémától az elméletig. Kossuth, Budapest, 1977.
[14] Gáspár A. - Pálvölgyi L. - Valló Á.:
A neurokibernetika, a modellezés néhány metodológiai-filozófiai problémája. ELTE TTK Filozófiai Közlemények, 1975. II.
[15] Gáspár A.:
A gépi gondolkodás lehetőségének tagadásával kapcsolatos szemléleti korlátokról. Magyar Filozófiai Szemle, 1976/1.
[16] Pálvölgyi L.:
A modellezés lehetőségeiről a pedagógiában. Akadémiai, Bp. 19[illegible]
[17] Gáspár A.: SIMULA and the Constitutional State or else Science of Law Supported by the System Analysis. Proceedings of 20th SIMULA User Conference, 1994, Prague. Association of SIMULA Users & TIMING Prague, 1994. pp. 151-16[illegible]
[18] Gáspár A.: A rendszerelemzéssel támogatott jogtudomány HARANG BT - BME Műszaki Jogtudományi Osztály, 1994. Kézirat.
[19] Gáspár A.: Rendszerelemzéssel támogatott jogtudomány. Neumann Kongresszus, 1995.

IQÜMT
IQSOFT
ÜZLETI MEGOLDÁSOK TECHNOLÓGIA

Szerző: Németh Miklós

Verzió: 1.0.3 (1996.05.11)
1.1.0 (1996.09.14)

Intelligens Software Rt.

1142 Budapest Teleki Blanka u. 15-17.
Tel: 251-5449 Fax: 220-5598
E-mail: iqsoft@iqsoft.hu
WWW: http://www.iqsoft.hu

TARTALOM

1. BEVEZETÉS 2

2. ELEMZÉSI ÉS TERVEZÉSI MÓDSZERTAN 4

2.1 OBJECT MODELING TECHNIQUE 4
2.1.1 Az objektum modell *4*
2.1.2 A dinamikus modell *5*
2.1.3 A funkcionális modell *5*
2.2 PLATINUM PARADIGM PLUS 5
2.3 IQSOFT BUSINESS COMPONENT GENERATORS (IQBCG) FOR P+ 7

3. FEJLESZTÉSI TECHNOLÓGIA 7

3.1 CENTURA SQLWINDOWS 8
3.2 MICROSOFT VISUAL C++ 8
3.3 PLATINUM OBJECTPRO 8
3.4 IQSOFT BUSINESS CLASS LIBRARY FOR SQLWINDOWS 9
3.5 IQSOFT BUSINESS CLASS LIBRARY FOR C++ 9
3.6 IQSOFT BUSINESS CLASS LIBRARY FOR OBJECTPRO 9

4. ADATBÁZIS-TECHNOLÓGIA 10

4.1 ORACLE7 10
4.2 CENTURA SQLBASE 10
4.3 MICROSOFT SQL SERVER 10
4.4 OBJECT DESIGN OBJECTSTORE 11

5. PROJEKTVEZETÉS ÉS MINŐSÉGBIZTOSÍTÁS 11

6. OKTATÁS, TÁMOGATÁS, TANÁCSADÁS 12

6.1 OKTATÁS 12
6.1.1 SQLWindows, SQLBase *12*
6.1.2 Módszertan *13*
6.1.3 ORACLE *13*
6.1.4 Projektvezetés *13*
6.2 TÁMOGATÁS ÉS TANÁCSADÁS 13
6.2.1 Támogatás *13*
6.2.2 Tanácsadás *14*

7. SZAKIRODALOM 14

1. Bevezetés

IQSOFT Üzleti Megoldások Technológia (**IQÜMT**) olyan módszereknek és eszközöknek az együttese, amelyek alkalmasak feladat-kritikus ügyfél-kiszolgáló alkalmazások készítésére. IQÜMT lefedi a teljes fejlesztési folyamatot, az elemzéstől az implementációig a projektvezetést és az adatbázis-kezelést is beleértve. IQÜMT az IQSOFT egyik fő fejlesztési technológiája, ami egy ideje a cég ügyfelei számára is rendelkezésre áll.

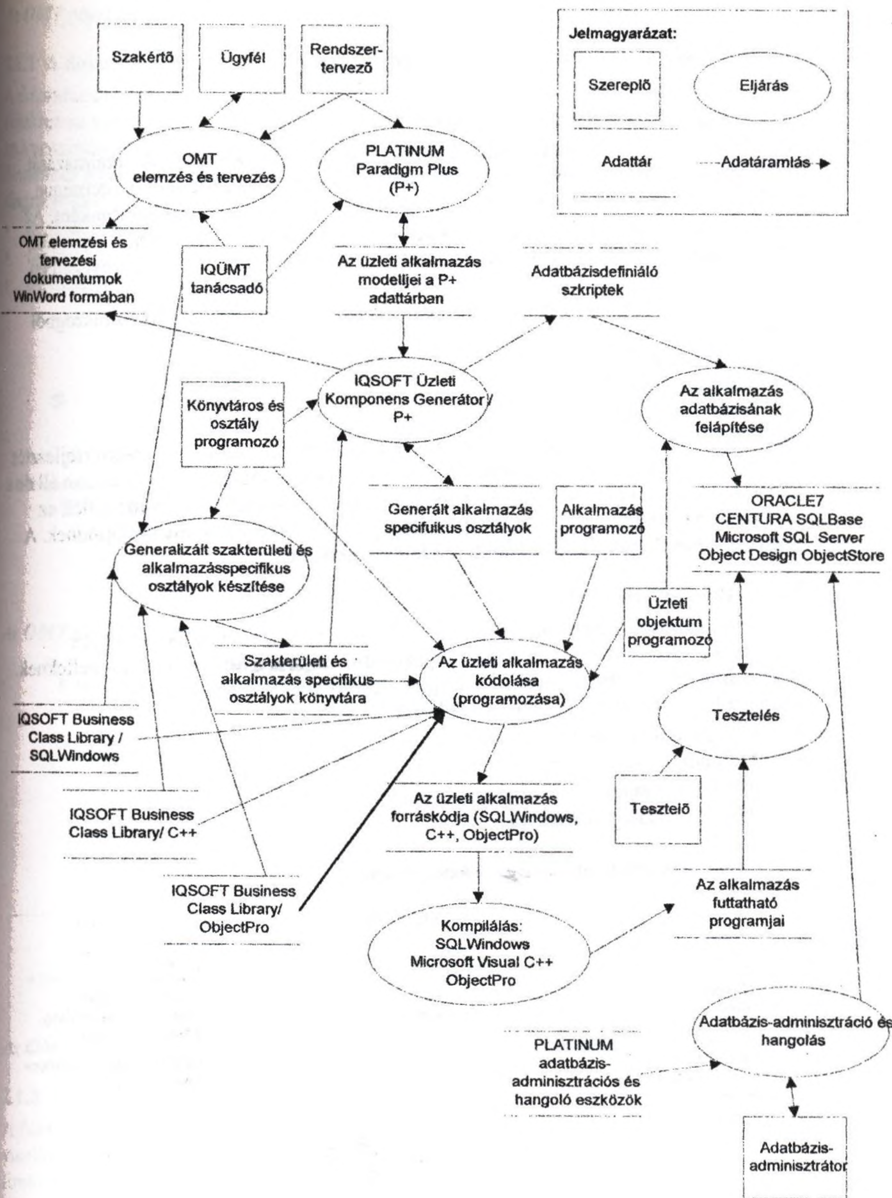
Az IQÜMT meghatározó komponensei a következők:

- Object Modeling Technique (OMT) - objektum-orientált elemzési és tervezési módszertan
- PLATINUM Paradigm Plus (P+) - objektum-orientált CASE eszköz
- IQSOFT Business Component Generators for Paradigm Plus
- CENTURA SQLWindows - 4. generációs OO implementációs környezet (Windows (16 és 32-bit), Sunsoft Solaris)
- PLATINUM ObjectPro - 4. generációs gépi kódra fordító OO implementációs környezet (Windows (16 és 32-bit), HP/UX, Sunsoft Solaris, IBM AIX)
- Microsoft Visual C++ - 3. generációs OO implementációs környezet
- IQSOFT Business Class Library (IQBCL) for SQLWindows
- IQSOFT Business Class Library (IQBCL) for ObjectPro
- IQSOFT Business Class Library (IQBCL) for C++
- ORACLE7 RDBMS
- CENTURA SQLBase RDBMS
- Microsoft SQL Server RDBMS
- Object Design ObjectStore ODBMS
- ARTEMIS International Project Mangement Methodology (pm2)
- Microsoft SourceSafe

Az IQÜMT a következő fő területeket fedi le:

- elemzési és tervezési módszertan (OMT)
- program-fejlesztési technológia (Paradigm Plus OOCASE, CENTURA, ObjectPro, C++, IQSOFT generátorok és objektum-könyvtárak)
- adatbázis-technológia (relációs és objektum-orientált)
- adatbázis-adminisztráció és hangolás (PLATINUM termékek)
- projektvezetés, minőségbiztosítás
- oktatás/támogatás/konzultáció.

Az IQÜMT elsősorban az **objektum-orientált** (OO) technológiára támaszkodik a fejlesztési életciklusnak mind a elemzési/tervezési mind az implementációs fázisaiban.



Az IQÜMT adatáramlási diagramja

3

2. Elemzési és tervezési módszertan

2.1 Object Modeling Technique

Az IQÜMT szorgalmazza formális elemzési és tervezési módszertanok alkalmazását. Az IQÜMT a James Rumbaugh és társai által definiált Object Modeling Technique (OMT) módszertant javasolja és támogatja elemzési és tervezési módszertanként. Az OMT bizonyítottan hatékony és széles körben alkalmazott OO elemzési és tervezési módszertan. Az OMT-t sikeresen alkalmazták különféle üzleti és ipari környezetben.

Az OMT a rendszert három különböző de egymással kapcsolatban álló szemszögből modellezi:

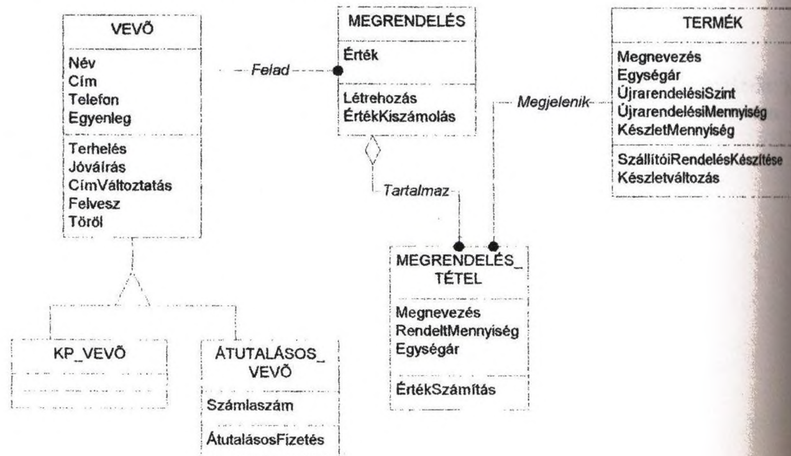
- objektum modell
- dinamikus modell
- funkcionális modell

Mindegyik modell a rendszert ortogonális nézetekre osztja szét egy a rendszerfejlesztés egészére egységes jelölést alkalmazva. A három modell egymással kapcsolatban áll de a modellek közötti kapcsolatok behatároltak és meghatározottak. Ezek a modellek az elemzési fázisban keletkeznek, és a további fejlesztési fázisokban továbbfejlődnek. A rendszer teljes leírása mind a három modellt megköveteli.

2.1.1 Az objektum modell

Az objektum modell a rendszer statikus szerkezetét írja le; bemutatja a rendszer objektumainak struktúráját és alapját képezi a dinamikus és funkcionális modelleknek. A Paradigm Plus "Object Modell" szkriptje riportot készít az objektum modell elemeiről. Az objektum modell a következőkből áll:

- osztálydiagramok (Class Diagram)
- objektumdiagramok (Object Diagram)
- alrendszer-diagramok (Subsystem Diagram)
- adatszótár (Data Dictionary)

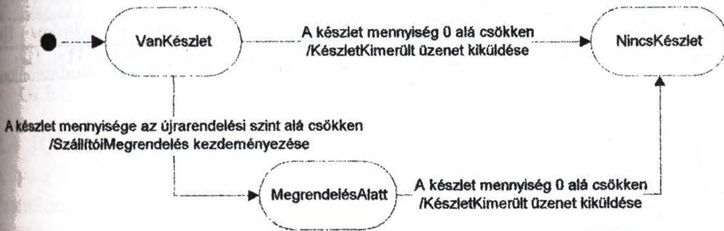


Az OMT objektum modellje az alkalmazás statikus szerkezetét írja le

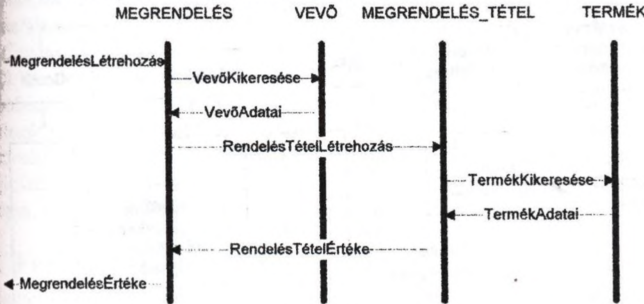
2.1.2 A dinamikus modell

A dinamikus modell leírja a rendszer vezérlési, időtől függő viselkedését. Minden objektumra vonatkoztatva bemutatja az eseményeket és a megengedett eseményszekvenciákat. A Paradigm Plus "Dynamic Modell" szkriptje riportot készít a dinamikus modell elemeiről. A dinamikus modell grafikus reprezentációja a következő diagramokkal lehetséges:

- állapotdiagramok (State Diagram)
- forgatókönyv diagramok (Scenario Diagram)
- eseményfolyam diagramok (Event Flow Diagram)



Az OMT állapotdiagramjai az osztályok dinamikus viselkedését írják le



Az OMT forgatókönyv diagramok az osztályok interakcióit írják le

2.1.3 A funkcionális modell

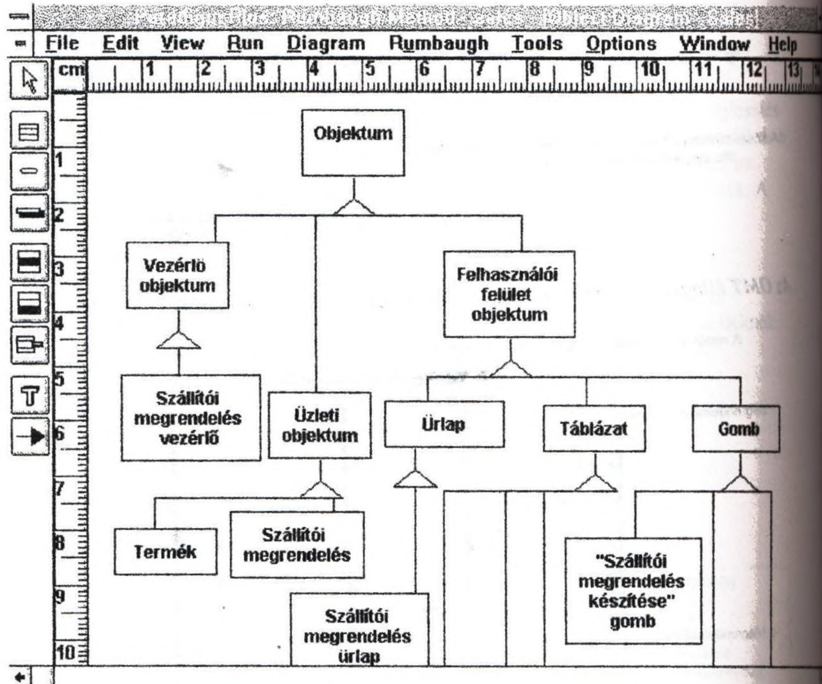
A funkcionális modell leírja a rendszer funkcionális, transzformációs (átalakító) viselkedését. A Paradigm Plus "Functional Modell" szkriptje riportot készít a funkcionális modell elemeiről. A funkcionális modell grafikus reprezentációja a következő diagramokkal lehetséges:

- adatáramlási diagramok (Data Flow Diagram)
- objektum-interakció diagramok (Object Interaction Diagram)

2.2 PLATINUM Paradigm Plus

Az OMT - vagy bármely más módszertan - használata CASE (Computer Aided Software Engineering) eszköz nélkül meglehetősen nehéz. Az OMT-t számos CASE

eszköz támogatja. Az IQÜMT a PLATINUM Paradigm Plus (P+) OO CASE eszközének használatát javasolja. A P+ fő előnye az, hogy ***saját programnyelvvel*** rendelkezik (ProtoScript), ami lehetővé teszi a P+ környezet bővítését és hatékony (saját, speciális) kódgenerátorok (SQL, C++, ObjectPro, SQLWindows, stb.) készítését. A PLATINUM a P+-hoz egy sor ProtoScript-ben írt ***kódgenerátort*** szállít (SQL, C++, Smalltalk, ObjectPro, ObjectStore, ADA, stb.). A ***reverse-engineering*** (sőt a round-trip engineering) támogatására is rendelkezésre állnak eszközök a P+-ban egyes nyelvekhez (C++, Smalltalk, ObjectPro).
A P+ fontos (központi) szerepet játszik az IQÜMT-ben, mivel a szoftverfejlesztési projekt a P+-ban tárolt projekt köré szerveződik. A P+ alkalmas a projekt ***konfigurációmenedzselési*** feladatait is ellátni akár saját konfigurációkezelő mechanizmusai akár külső konfigurációkezelő rendszer (Intersolv PVCS, Microsoft SourceSafe) segítségével.



Paradigm Plus olyan OO CASE eszköz, ami az OMT-t erőteljesen támogatja

Az alábbiakban vázlatosan felsoroljuk a P+ szolgáltatásait:

- Vállalati objektum-tárház kezelés: A P+ projektekben tárolja az objektumok tulajdonságait és kapcsolatait. A fejlesztés ezen tárházakban tárolt információk alapján folyik. A P+ a diagram-szerkesztő, objektum-tallózó, mátrix és táblázatos szerkesztők segítségével lehetővé teszi a vállalati szakemberek számára az objektum-tárház különféle szempontú karbantartását és áttekintését. Az objektum-tárházat a P+ az Object Design Inc. ObjectStore ODBMS-ével tartja karban.
- Többfelhasználós (projekt) objektum-tárház hozzáférés: Az objektum-tárházak az ObjectStore szolgáltatásainak köszönhetően a hálózaton szétosztottak lehetnek: Novell NetWare, Windows for Workgroups, TCP/IP, stb.

- Többplatformos támogatás: A P+ a következõ platformokon áll rendelkezésre: MS Windows 3.x, NT 3.x, 95, Solaris 2.x, SunOS 4.1.x, HP-UX 9.x, IBM OS/2 2.x, AIX 3.2.5, DEC Alpha OSF/1 2.x, AT&T NCR SVR4 3.x, SGI IRIX 5.x.
- Projektek közötti objektum-megosztás: Ez a szolgáltatás nélkülözhetetlen a vállalati központi objektum-könyvtárak hatékony kezeléséhez.
- Saját vagy külsõ (PVCS, SourceSafe, stb.) konfigurációkezelés
- Metaosztály és objektum szintû hozzáférés-védelem.
- ProtoScript nyelven keresztül való hozzáférés az objektum-tárházhoz
- Elõregyártott szkriptek: kódgenerátorok (C, C++, Ada, Smalltalk, PowerScript, Forte, ObjectPro), adatbázis-definíciós szkriptek generálása (ORACLE, SQL Server, Informix, ANSI SQL, SYBASE, ObjectStore, Versant), CORBA IDL szkriptek, reverse engineering (C++, Smalltalk, ObjectPro), riportgenerátorok, konzisztencia-ellenõrzõ szkriptek.
- Import/export: Paradigm Plus IMP (CSV), EIA-CDIF(CASE Data Interchange Format); diagram exportálási formák: Windows BMP, Windows Metafila (WMF), PostScript EPS, Interleaf ASCII, FrameMaker MIF.
- OLE 2 szerver és kliens szolgáltatás

2.3 IQSOFT Business Component Generators (IQBCG) for P+

A P+-t számos elõre elkészített (gyári) generátorral szállítják, amit a P+ saját ProtoScript nyelvén írtak. Ezek a generátorok egy sor alkalmazás számára megfelelõk de speciálisan jóminõségû üzleti alkalmazások számára finomítást igényelhetnek. Az IQÜMT-ben az IQSOFT bõvítette és továbbfejlesztette a P+ generátorokat az IQBCG-vel. Ezek a generátorok az IQÜMT által támogatott adatbázis-kezelõk (ORACLE7, CENTURA SQLBase, Microsoft SQL Server, ObjectStore) számára (SQL) adatbázis-definíciós (DDL) szkripteket illetve ObjectPro, C++ és SQLWindows forrásmodulokat generálnak. Az SQL DDL szkriptek az IQÜMT adatbázis-definíciós szabályait követik. A C++ és SQLWindows forrásprogramok az IQSOFT Business Class Library for C++, SQLWindows és ObjectPro elõírásainak megfelelõen generálódnak. Az SQL adatbázisok és a C++, SQLWindows, ObjectPro programok ugyanazon családhoz tartozó generátorok által készülnek, ez a lehetõ legmagasabb fokú konzisztenciát biztosítja. Az IQSOFT Business Class Library közvetlen támogatása magas fokú fejlesztési hatékonyságot és megbízhatóságot ad. Ez az integrált technológia az IQÜMT alapja, és ez teszi lehetõvé feladat-kritikus üzleti alkalmazások készítését.

3. Fejlesztési technológia

Általánosságban az implementációs technológiának két szintje van:

- a magasabb szintû 4. generációs eszközöket használ, amelyek gyorsabb prototípuskészítést és fejlesztési eljárást tesznek lehetõvé
- az alacsonyabb szintû 3. generációs nyelveket alkalmaz, amelyek maximális rugalmasságot és teljesítményt biztosítanak, de lassabb fejlesztési folyamatot.

Az IQÜMT mindkét szint számára ajánl eszközöket. Mindkét szint implementációs eszközei támogatják az objektum-orientált programozást. Az OMT-vel elemzett és megtervezett, P+ adattárakban rendelkezésre álló alkalmazás SQLWindows-zal, ObjectPro-val vagy C++-szal impementáljuk a IQBCG és IQBCL felhasználásával.
Egy feladat-kritikus üzleti rendszer egyszerre kíván magas színvonalú megbízhatóságot és teljesítményt, továbbá a sikerhez gyors és interatív megvalósítás úgyszintén nélkülözhetetlen. Az IQÜMT megoldásokat kínál mindkét cél eléréséhez: megbízhatóság, teljesítmény és gyors megvalósítás.

3.1 CENTURA SQLWindows

Az SQLWindows a CENTURA (korábban GUPTA) cég terméke, és az üzleti szektorban széleskörben alkalmazzák. Az SQLWindows 4. generációs, OO, vizuális fejlesztõeszköz, ami magas fokú termelékenységet biztosít. Az SQLWindows-zal való implementációs eljárás némely tapasztalataink szerint négyszer gyorsabb mint a C++-ban végzett programozás.
Az SQLWindows elsõsorban a Microsoft Windows platformokon áll rendelkezésre (Windows 3.x (16-bites SQLWindows), Win32 (NT és Windows95) platformokon a 32-bites CENTURA TeamDeveloper), de Solaris változata is elkészült már.

Az SQLWindows nagytermelékenységû környezetet biztosít üzleti alkalmazások fejlesztéséhez.

3.2 Microsoft Visual C++

A C++ sztenderd nyelv és - számos hibája ellenére - a C++-t tartják az objektum-orientált programozás de-facto szabványos nyelvének. A Microsoft Visual C++ C és C++ nyelveknek egy elsõsorban Windows-os implemántációs változata. Az IQÜMT ügyfél oldali szabványos platformja a Microsoft Windows család, amit a Microsoft Visual C++ támogat a legjobban. Az IQÜMT 1997-tõl teljesen OLE2 és CORBA szabvány szerinti objektum-modellt fog támogatni. Az OLE2-automatákra alapozott objektum-orientált rendszer-architektúra támogatása jelenleg is az IQÜMT stratégiája. Az OLE2-automatákra épülõ rendszer biztosítja a legzökkenõmentesebb Windows integrációt.

3.3 PLATINUM ObjectPro

Az ObjectPro a 4. generációs OO nyelvek legújabb nemzedékéhez tartozik és fõ erõssége abban rejlik, hogy a 4. generációs eszközökre jellemzõ magasokú termelékenységet egyesíti a 3. generációs eszközök hatékonyságával (program futási sebesség). Az ObjectPro mindezt úgy éri el, hogy egy teljes OO nyelvet támogató

vizuális, integrált fejlesztõi környezetet biztosít, ahol a programozók interpretatív környezetben tesztelhetik készülõ programjaikat, majd a program elkészültekor gépi kódra fordítják az alkalmazást és így adják át integrációs tesztelésre illetve végfelhasználói termelésbe. Az ObjectPro által támogatott OO nyelv teljes, ami lehetõvé teszi rendkívül rugalmas és hatékony objektum-könyvtárak készítését. Az ObjectPro több platformot is támogat: Windows 3.1, Windows 95, Windows NT, Solaris, HP/UX, AIX.

3.4 IQSOFT Business Class Library for SQLWindows

Az SQLWindows hatékony fejlesztõi környezet és gyors implementációt tesz lehetõvé. A legnagyobb hatékonyság azonban csak újrafelhasználható komponensek alkalmazása révén lehetséges. Az IQÜMT magas szintû újrafelhasználható üzleti komponenseket kínál, amelyek összefoglaló neve IQSOFT Business Class Library (IQBCL). Az IQBCL azonban nem csupán különféle osztályok laza halmaza, hanem egy kész de bõvíthetõ alkalmazási keretrendszer, ami felhasználói felület, adatbázis-hozzáférési és vezérlési(adminisztrációs) osztályokat tartalmaz. Az IQBCL felhasználói felülete a Multiple Document Interface (MDI) szabványt támogatja. Az IQBCL adatbázis-hozzáférési mechanizmusa masszív alapot biztosít újrafelhasználható alkalmazás-specifikus üzleti objektum-gyûjtemények létrehozásához.

3.5 IQSOFT Business Class Library for C++

A Visual C++ csomagban rendelkezésre álló Microsoft Foundation Classes (MFC) a Windows fejlesztõk számára elemi objektum-orientált keretrendszert biztosít. Az MFC egy általános környezet és nem kezeli az üzleti alkalmazások számos területét. Az MFC alkalmazása üzleti rendszerek kifejlesztésére nem megoldhatatlan, de jelentõs erõfeszítést igényel. Az IQÜMT az IQSOFT Business Class Library (IQBCL) for C++-t ajánlja üzleti alkalmazások alapobjektum-készleteként. Az IQBCL olyan C++ osztályok halmaza, ami az üzleti alkalmazások mindennapos feladataira specializáltak, míg az MFC általános keretrendszer. Az IQBCL tartalmaz néhány Visual C++ saját varázslót (custom wizard) a programozási munka egyszerûsítésére. Az IQBCL sokkal karcsúbb és egyszerûbb osztálykönyvtár mint az MFC, mivel nem törekszik arra, hogy a Windows környezetet teljesen elfedje a fejlesztõk elõl. Az MFC-nek az a törekvése, hogy teljesen elfedje a Windows API-t nem szerencsés, mivel a programozók a Windows (API) ismerete nélkül úgysem értik meg az MFC nagy részét. Az IQBCL ezzel szemben üzleti alkalmazások írásához néhány igen hatékony osztályt vezet be, amelyek leegyszerûsítik a programozók munkáját, de ehhez nem kell óriási rendszereket megtanulniuk és megmozgatniuk (pl. debuggolás). Az IQBCL üzleti objektumai teljesen OLE2-automatákra alapulnak, és így más OLE2-kompatibilis rendszerek számára (Excel, Access) is elérhetõk.

3.6 IQSOFT Business Class Library for ObjectPro

Az ObjectPro hatékony fejlesztõi környezet és gyors implementációt tesz lehetõvé. A legnagyobb hatékonyság azonban csak újrafelhasználható komponensek alkalmazása révén lehetséges. Az IQÜMT magas szintû újrafelhasználható üzleti komponenseket kínál, amelyek összefoglaló neve IQSOFT Business Class Library (IQBCL). Az IQBCL azonban nem csupán különféle osztályok laza halmaza, hanem egy kész de bõvíthetõ alkalmazási keretrendszer, ami felhasználói felület, adatbázis-hozzáférési és vezérlési(adminisztrációs) osztályokat tartalmaz. Az IQBCL adatbázis-hozzáférési mechanizmusa masszív alapot biztosít újrafelhasználható alkalmazás-specifikus üzleti objektum-gyûjtemények létrehozásához.

4. Adatbázis-technológia

Az üzleti alkalmazások jellemzõ közös vonása az, hogy fokozott adatbázis-kezelési igényeket támasztanak. Az IQÜMT hangsúlyozza az adatbázis-technológia jelentõségét. Az IQÜMT relációs és objektum-orientált adatbázis-kezelõket ajánl az üzleti alkalmazások adattárolási és lekérdezési feladataihoz. A feladat-kritikus üzleti alkalmazások a legmagasabb fokú adatmegbízhatóságot, teljesítményt, konkurenciakezelést és adatvédelmet igénylik az adatbázis-kezelõ alrendszertõl. Az IQÜMT csak olyan adatbázis-kezelõ rendszert támogat mely a fenti követelményeket megfelelõ szinten eleget tesz, ezek jelenleg a következõk: ORACLE7, CENTURA SQLBase, **Microsoft SQL Server.** Mindhárom adatbázismotor használja az SQL-t, tárolt eljárásokat és triggers. Az IQSOFT Business Class Library mindhárom adatbázismotort támogatja.

Az IQÜMT az Object Design ObjectStore ODBMS-t ajánlja komplex adatszerkezetekkel rendelkezõ rendszerek adatbázis-kezelõjeként.

Az IQÜMT használja a CENTURA SQLNetwork middleware-t (és ezen keresztül az ODBC-t), amely lehetõvé teszi a fejlesztõknek tetszõleges adatbázisok elérését: szöveges állományok, DBASE, Microsoft Access, SYBASE SQL Server, Rdb, IBM DB2, SQL/400, Tandem NonStop SQL, stb. Az IQSOFT nem támogatja azon fejlesztõket, akik feladat-kritikus üzleti alkalmazásokat DBASE vagy más tranzakció-kezelést nem biztosító adatbázis-kezelõkkel akarnak megoldani. A meglévõ DBASE fájlok (DBFs) elérését azonban az ODBC-n keresztül támogatja az IQÜMT azzal a céllal, hogy az új alkalmazások együtt tudjanak mûködni meglévõ rendszerekkel.

1996-97-re tervezi az IQSOFT, hogy üzembe állítja az Object Designs cég ObjectStore objektum-orientált adatbázis-kezelõ rendszerét, és az IQÜMT-t felkészíti az ObjectStore-ral való együttmûködésre.

4.1 ORACLE7

Az ORACLE7 egyike az IQÜMT által ajánlott adatbázis-kezelõknek fõleg olyan esetekben, amikor nagy teljesítményû, erõsen konkurens, skálázható üzleti alkalmazást kell kifejleszteni. Az ORACLE7 kizárólag többfelhasználós környezetekben használható hatékonyan. Ha egyfelhasználós környezetet kell támogatni, akkor a CENTURA SQLBase a megoldás. Az ORACLE gyakorlatilag mindenféle platformon rendelkezésre áll (Windows NT, OS/2, UNIX-ok, VMS, mainframe, stb.). A Personal ORACLE7 csak fejlesztési célokra alkalmas, termelési adatbázisnak kevésbé.

4.2 CENTURA SQLBase

A CENTURA SQLBase megbízható adatbázis-kezelõ kis vagy közepes vállalatok illetve vállalati részlegek számára. Az SQLBase lapszintû zárolási mechanizmusa miatt erõsen konkurens környezetben csak tapasztalt fejlesztõk számára alkalmazható. Az SQLBase különösen akkor megfelelõ, ha az alkalmazást egyfelhasználós környezetben (pl. ügynökök, vállalati vezetõk noteszgépein) is mûködtetni kell. Az SQLBase-nek van a legkisebb számítógép-erõforrás igénye. Az SQLBase valamennyi népszerû PC platformon (Windows 3.x (egyfelhasználós), Windows95, Windows NT, OS/2, NetWare) rendelkezésre áll.

4.3 Microsoft SQL Server

Windows NT-n az ORACLE7 és SQLBase adatbázis-kezelõknek lehet alternatívája.

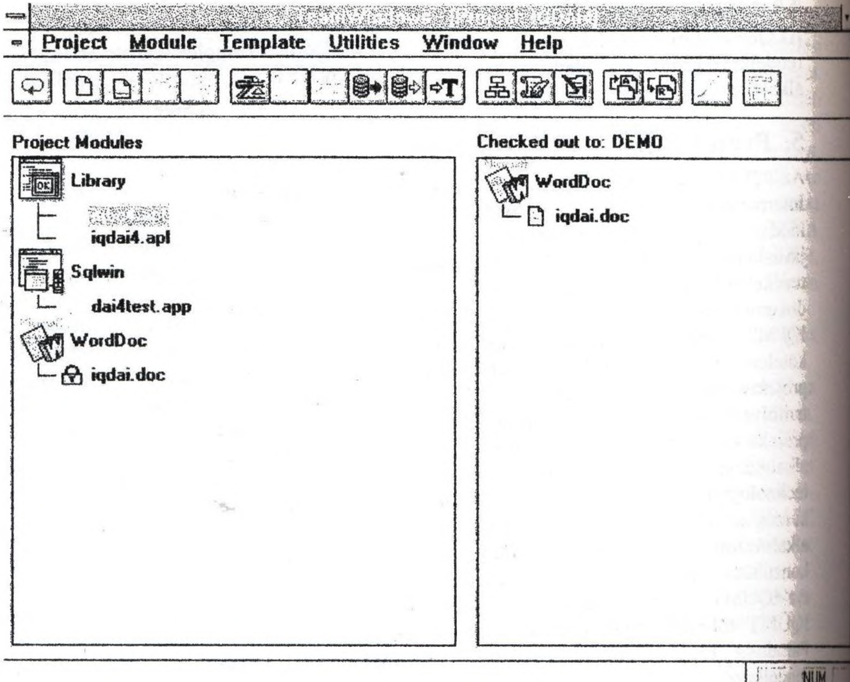
4.4 Object Design ObjectStore

Az ObjectStore a legelterjedtebb ODBMS, amit elsősorban komplex adatstruktúrákkkal rendelkező rendszerek adatbázis-kezelőjeként alkalmaznak, de fokozatosan terjed üzleti alkalmazások világában is.

5. Projektvezetés és minőségbiztosítás

Az IQÜMT része egy teljes és formalizált projektvezetési módszer is: az ARTEMIS International Project Management Methodology (pm2 [ejtsd. péem négyzet]). A IQPM2 módszer segítségével, amelyet az IQSOFT fejlesztett ki, teljes és gyakorlatias javaslatokat tartalmaz vezérfonálul a projekt vezetése során az alkalmazott lépésekre, tevékenységekre és a feladatokban résztvevőkre vonatkozóan. Több kötetnyi dokumentum-minta és űrlap teszi a minőségbiztosítási feladatokat egyszerűbbé. A IQPM2 alkalmazása lefedi a projekt minőségbiztosítási igényeit is. A IQPM2 nem kötelező része az IQÜMT-nek de erősen ajánlott. Ha a fejlesztőnek van saját formalizált projektvezetési módszere, akkor az IQÜMT-vel összhangot kell teremteni. Egy IQPM2 tanfolyam az IQSOFT-nál mindenképpen megtérülő befektetés a nagylélegzetű projektekbe kezdő menedzsereknek. Az IQÜMT-ben azonban mindenképpen alkalmazni valamilyen projektvezetési módszert. Az IQSOFT által tartott "OO technológiát alkalmazó projektek vezetése" című tanfolyam egy javasolt bevezető kurzus az IQÜMT-t alkalmazó projektek vezető munkatársai számára. A ténylegesen alkalmazott projektvezetési technológiát az ügyfél és az IQÜMT-t szállító cég konzultánsai közösen is kidolgozhatják.

Az IQÜMT-ben minőségbiztosítás elengedhetetlen része a konfigurációkezelés. Az IQÜMT többféle módon is támogatja a projekt konfigurációkezelési feladatait. A Paradigm Plus CASE eszköz jól használható konfigurációkezelés támogatással rendelkezik. A CENTURA TeamWindows-a vagy Team Developer-e az SQLWindows projektek számára ad verziókezelés támogatást. Az IQÜMT támogatja továbbá a Microsoft SourceSafe és Intersolv PVCS verziókezelő és konfigurációkezelő eszközöket is.



A CENTURA TeamWindows kifinomult verziókezelõ és projekt-adminisztráló rendszert biztosít.

6. Oktatás, támogatás, tanácsadás

Az IQSOFT az egyik független, vezetõ technológiai központja a vállalati és más szoftverházak fejlesztõi számára Magyarországon. Az IQSOFT nem csak nem csak technológiát és azokhoz tartozó eszközöket, komponenseket ajánl az IQÜMT-ben, hanem oktatást és támogatást is az IQÜMT elsajátításához. Általában három - hat hónapnyi együttmûködés elegendõ az IQSOFT és a fejlesztõ között az IQÜMT elsajátítására.

6.1 Oktatás

Az IQÜMT elsajátításának elsõ ajánlott lépései IQÜMT legalapvetõbb elemeinek tanfolyamokon történõ megtanulása. A tanfolyamok egy részének tartalmát az IQSOFT szakemberei dolgozták ki. Az IQSOFT nem rendelkezik elkülönülõ oktatói csoporttal; az oktatók gyakorlott fejlesztõk és projekt vezetõk, akik a technológiát gyakorlatból is jól ismerik, és a tanfolyamokon segítenek megoldani az ügyfelek konkrét problémáit is.

6.1.1 SQLWindows, SQLBase

- **Kliens-szerver rendszerek fejlesztése SQLWindows-ban** - 5-napos bevezetõ tanfolyama következõ témákkal: kliens-szerver architektúra, SQL, SQLBase, többfelhasználós konkrenciakezelés, az SQLWindows vizuális objektumai, alkalmazások írása SQLWindows-ban, stb.

- **SQLBase és SQLWindows haladóknak** - 5-napos haladóknak szóló tanfolyam a következő témákkal: OLE, VBX, DDE, MDI, Drag-And-Drop, objektum-orientált programozás, QuickObjects, magyar nyelv támogatás az SQLBase-ben, stb.
- **Az IQBCL használata** - 4-napos tanfolyam az IQSOFT Business Class Library for SQLWindows használatának megismertetésére

6.1.2 Módszertan

- **Vállalati információs rendszerek fejlesztése** - 2-napos szuper-haladó tanfolyam szoftvermérnököknek és a vállalati fő technológusoknak, ami bemutatja miként lehet ügyfél-kiszolgáló alkalmazásokat fejleszteni az IQÜMT-vel.
- **OO technológiát alkalmazó projektek menedzselése** - 3-napos tanfolyam projektvezetőknek és rendszertervezőknek, ami tárgyalja az objektum-orientált fogalmakat és a projekt menedzselés alapjait. A tanfolyam gyakorlatban is bemutatja a konfigurációkezelő eszközök jelentőségét és használatát: CENTURA TeamWindows, Microsoft SourceSafe és Intersolv PVCS.
- **Az OMT alkalmazása** - 3-napos bevezető tanfolyam az OMT-vel való elemzés és tervezés megismertetésére.
- **A Paradigm Plus használata** - 2-napos tanfolyam a Paradigm Plus használatáról

6.1.3 ORACLE

Az IQSOFT az ORACLE cég tanfolyamait hivatalos oktatóközpontként tartja az ORACLE magyarra fordított tanfolyami anyagainak tematikája szerint.

6.1.4 Projektvezetés

Az IQSOFT tanfolyamokat tart az ARTEMIS International projektvezetés elméleti és gyakorlati módszeréről.

6.2 Támogatás és tanácsadás

Az IQÜMT komplex, sokkomponenses technológia, amely megfelelő támogatást kíván a technológia-szállítótól. A tanfolyamok megfelelőek arra, hogy a fejlesztők alapvető és haladó szintű információkat szerezzenek az IQÜMT alkalmazásáról. A fejlesztők egy része, aki korábban is magas szintű technológiákat alkalmazott (CASE, SSADM, RDBMS, 4GL, etc.) gond nélkül tér át az új IQÜMT technológiára számottevő támogatási igény nélkül. A fenti technológiákban járatlan cégek számára azonban célszerű támogatást kérni az IQÜMT szállítójától.
A támogatás itt azt jelenti, hogy a támogató segít az ügyfélnek (fejlesztőnek) elsajátítani az IQÜMT adott komponensét. A konzultációk során a tanácsadó aki gyakorlott megoldás-szállító mérnök segíti fejlesztőket feladataik megoldásában. A tanácsadó a projektvezetők vagy fő technológusok számára is közvetlen segítség lehet a projekt során. A támogatást nyújtó szakember személye nem lényeges, de a tanácsadó az egész projekt során állandó kapcsolatban marad a fejlesztőkkel.
Az IQSOFT mint az IQÜMT szállítója támogatási és tanácsadói szolgáltatásokat is nyújt különböző módokon.

6.2.1 Támogatás

Az IQSOFT támogatási tevékenysége az alábbi IQÜMT komponenseket fedi le:

- A Paradigm Plus és az IQBCG használata
- Programozás SQLWindows-ban és az IQBCL használata
- Programozás C++-ban és az IQBCL használata
- Az ORACLE7, CENTURA SQLBase használata, technikái
- *Az ORACLE7, CENTURA SQLBase adatbázisok hangolása*

6.2.2 Tanácsadás

Az IQSOFT az IQÜMT-vel kapcsolatosan a következő fajta tanácsadási szolgáltatásokat nyújtja:

- Az IQÜMT projektvezetése - a tanácsadó segít a helyi projektvezetőnek az egész projekt során az IQÜMT céghez való bevezetésében és technológiai problémák megoldásában.
- Az OMT alkalmazása - a tanácsadó segít a rendszertervezőknek az OMT alkalmazásában az adott szakterület elemzési és tervezési feladatainak megvalósításához.

7. Szakirodalom

- Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W. 1991 *Object-Oriented Modeling and Design*. Prentice Hall.
- Goldberg A., K. Rubin. *Object Behavior Analysis*. New York:ACM, September 1992
- Wilkinson N. *Using CRC Cards*. 1995 SIGS Books.
- Spurr K, Layzell P, Jennison L and Richards N 1994 *Business Objects: Software Solutions*. John Wiley & Sons.

IQBCL SQLWindows Class Library és prototípus generátor

Horváth Attila

4D Soft Bt

Az ügyfél-kiszolgáló architektúrájú rendszerek sikerével és terjedésével szinte párhuzamosan vált egyre népszerűbbé az objektum-orientált fejlesztési technológia, amelyet számos jól használható fejlesztőrendszer támogat. Ezen objektum-orientált fejlesztőrendszerek egyike a **CENTURA** (GUPTA) cég **SQLWindows** terméke. Az objektum-orientált technológia kifejezetten arra ösztönöz, hogy alkalmazásainkat teljes egészében jól definiált osztály-objektumokból építsük fel. Az így készült alkalmazás karbantarthatósága sokkal nagyobb mértékű, és ezzel az alkalmazás megbízhatósága ill. stabilitása megnő. Ehhez fejlesztettünk ki az IQSOFT Rt-vel közösen egy olyan osztálykönyvtárat, amely nagy hatékonysággal használható ügyviteli, adatbázis-kezelés irányultságú alkalmazásokban. Az IQBCL osztálykönyvtárat, és annak elődeit nem csak a két cég használja több éve fejlesztési feladatainak megoldására, de most már más SQLWindows fejlesztők számára is elérhető az ehhez szükséges szakmai támogatással (oktatás, konzultációk) együtt.

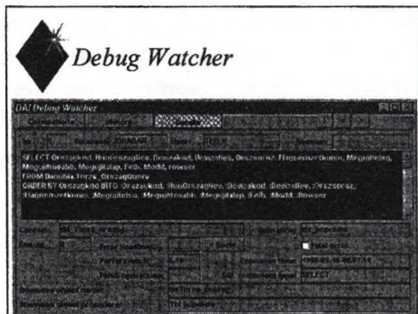
Az IQBCL nem csak osztályok laza halmaza, hanem egy jól definiált felhasználói felületet is magában foglaló keretrendszer, amely egy adatbázis-orientált alkalmazás minden aspektusát lefedi. Az IQBCL a **Multiple Document Interface** (MDI) szabványnak megfelelő felhasználói felülettel rendelkezik. Az adatbázis-hozzáférési mechanizmusa megbízható alapot biztosít újrafelhasználható alkalmazás-specifikus üzleti objektum gyűjtemények létrehozásához.

Az IQBCL felépítése megfelel a három rétegű alkalmazás fejlesztés (three-tier architecture) követelményeinek:

- felhasználói felület (presentation),
- üzleti objektumok (business rules),
- adatbázis interfész (database)

Adatbázis interfész

- Tranzakciós objektumok
- SqlXxx eljárások helyett metódusok
  gTra.do(hWndForm, 'Insert into Aruforg.Cikk ...')
- Beépített hibakezelés
- SQL Debug Watcher
- Profiler



Az adatbázis interfész (DAI - Data Access Interface) önállóan is alkalmazható objektum csoport, melyben az un. **tranzakciós objektumok** végzik az adatbázis kezelési műveleteket. Ezen objektu-mok

metódusai szolgálnak az SQLWindows-ban meglévő SqlXxxx típusú eljárások kiváltására, bővítésére. Olyan eljárások ezek (login, connect, do, query, tbl_populate, ...) amelyek hatékonyan elősegítik a biztonságos adatbázis kezelést. A metódusok saját hibakezelő mechanizmussal is rendelkeznek, így a felhasználónak kellemesebb, programozott formában jelenhetnek meg az adat-bázis hibaüzenetei. Az adatbázis interfész metódusok (megfelelő paraméterezés esetén) naplózzák a program által kiadott adatbázis műveleteket, amelyeket a program futása közben, és utána is (egy text file-ból) megvizsgálhatunk. A metódusok mérik az általuk elvégzett műveletek idejét, amelyet műveletenként, és összesített statisztikákban is lekérdezhetünk. Ezek nagymértékben megkönnyítik a program adatbázis interfészének a belövését, debuggolását, ill. az idő tényezőre történő optimali-zálását.

Profiler

Profiler Table

Entry Point	[illegible]	[illegible]	Average	Min	Max	Group1	Group2
BO.BUILD.SELECTSTATEMENT2	1,09	4	0,27	0,11	0,55	BO	BO.BUILD
BO.BUILD.UPDATESTATEMENT2	0,44	2	0,22	0,22	0,22	BO	BO.BUILD
BO.OP2.TBLPOPULATE2	7,76	2	3,88	1,40	6,26	BO	BO.OP2
BO.OP2.UPDATE2	4,11	2	2,05	0,71	3,40	BO	BO.OP2
BO.OP3.QUERY3	3,57	4	0,89	0,55	1,54	BO	BO.OP3
DAI.CON.CONNECT	21,99	13	1,69	0,27	17,96	DAI	DAI.CON
DAI.CON.DISCONNECT	2,26	10	0,23	0,17	0,28	DAI	DAI.CON
DAI.OP.COMMIT	0,33	1	0,33	0,33	0,33	DAI	DAI.OP
DAI.OP.DO	3,56	2	1,78	0,49	3,07	DAI	DAI.OP
DAI.OP.QUERY	1,98	4	0,50	0,33	0,93	DAI	DAI.OP
DAI.OP.TBLPOPULATE	6,92	2	3,46	1,21	5,71	DAI	DAI.OP
DAI.UTIL.AUDIT	0,60	9	0,00	0,00	0,39	DAI	DAI.UTIL
DAI.UTIL.DEBUG	8,01	64	0,13	0,00	2,64	DAI	DAI.UTIL

Üzleti objektumok

- DAI-ra épülő funkcionális osztályok iq_business_object
- Egységes eljáráshívási felület
- Interfész -, transzlációs lista

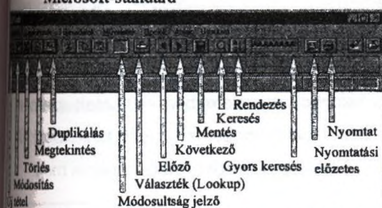
```
gboCikk.beolvas(..'Cikkszam = :dfCikksz, Ar = :dfAr',
        'Cikkszam = C.Cikkszam, Ar = A.NAr' ..)

'Select C.Cikkszam, A.Nar, .. from Cikk C, Ar A
where C.Cikkszam = A.Cikkszam ...
into :dfCikksz, :dfAr'
```

Az IQBCL-ben egy külön réteget képeznek az adatbázis interfészre épülő **üzleti objektumok**. Az üzleti objektumok az objektum-orientált elemzéssel és tervezéssel (OOA/OOD) előállított alkalmazási területre (domain) jellemző **perzisztens** osztályok. Az üzleti objektumok biztosítják a felhasználói felület objektumai számára az adatbázis-hozzáférést. A felhasználói felület objektumai az üzleti objektumok publikus eljárásain (interfészén) keresztül látják a rendszer adatait és teljesen el van előlük rejtve az adatbázis fizikai implementációja. Az IQBCL környezetben az üzleti objektumok az adatbázis interfész tranzakciós objektumain keresztül hajtják végre az adatbázis műveleteket.

Az üzleti objektumok metódusai **egységes eljáráshívási interfésszel** rendelkeznek, amely kiküszöböli az SQL utasítások szintakszisbeli különbözőségeit. Így ugyan olyan paraméterekkel lehet meghívni egy író vagy egy olvasó metódust. A változó hosszú paraméterek kezelésére az eljárások un. **interfész listát** használnak, amely egy vesszővel szeparált megfeleltetéseket tartalmazó sztring. Az interfész szimbólumok egy **transzlációs lista** segítségével képződhetnek le tényleges adatbázis objektum nevekre, így a felhasználói felület által használt interfész, és a mögötte lévő tényleges adatbázis eltérhet egymástól. Ezáltal az adatbázisban bekövetkező változások is jobban lokalizálhatók az üzleti objektumokra.

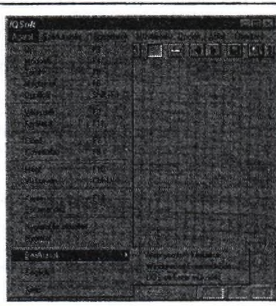
MDI keret ablak

Microsoft-standard



MDI menük

- Adatok
- Szerkesztés
- Törzsadatok
- Műveletek
- Opciók
- Ablakok
- Útmutató



Az IQBCL felhasználói felülete a **Multiple Document Interface** szabványt támogatja. Ezért az alkalmazás középpontjában egy a **Microsoft standarnak** megfelelő MDI ablak áll. Az alkalmazásban létrejövő MDI child top level képernyő objektumokon végezhető műveletek mind az MDI eszközlécéről, vagy menüjéből indíthatók. A felhasználók számára tehát elegendő a Word ill. Excel-hez hasonló MDI keret használatának a betanítása, nem kell minden egyes adatbeviteli, lekérdezési képernyőn külön kontrolokat, funkcióbillentyűket megtanulni. Az IQBCL felhasználói felületének a kialakításánál ügyeltünk arra, hogy minden művelet ne csak egérrel az eszközlécről, vagy menüből, hanem környezetérzékeny funkció billentyűk segítségével a billentyűzetről is aktivizálható legyen, ami ügyviteli alkalmazásoknál különösen fontos.

A menük használata is a Microsoft standardot követi. Az „**Adatok**", „**Szerkesztés**", ill. „**Ablakok**", „**Útmutató**" egységes menük között helyezkedhetnek el az alkalmazás specifikus „**Törzsadatok**", „**Műveletek**" és „**Opciók**" menük, melyek az alkalmazás törzsadat karbantartó, tranzakció kezelő műveleteit, ill. a szükséges opciók beállítását valósíthatják meg.

A továbbiakban a felhasználói felület fontosabb beágyazó, ill. beágyazott osztályait, ill. azok funkcionalitását fogom bemutatni.

„Szerkesztő" táblázatok

„Kisméretű" táblák kezelésére

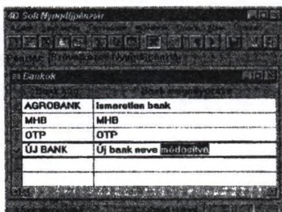
Új tétel felvétele az eszközsorból, vagy a menüből kezdeményezve

Párbeszéd a tábla és az MDI között

Explicit ill. implicit commit

„Szerkesztő" táblázatok

Módosítás, közvetlenül a kívánt adatra kattintva



A legegyszerűbb top level képernyő osztály az un. **„szerkesztő" táblázatok** osztálya. Ez a kevés oszlopot, és általában kevés sort tartalmazó kisméretű adatbázis táblák kezelésére szolgál. Az adatokat táblázatos formában jeleníti meg, felvitelük, módosításuk a táblázat megfelelő sorában történik. A tábla párbeszédet folytat az MDI kerettel, és a beágyazott oszlopaival, így a mezőkben

végzett módosulást az MDI eszközlécen megjelenő piros kereszt jelzi, ill. az MDI-ről kiváltott műveletet a tábla az oszlopokkal együttműködve végrehajtja. Az elindított tranzakciót a felhasználó explicit módon (F10) és implicit módon egy másik sorra kattintva is lezárhatja.

Ezen egyszerű funkció megvalósításához a programozónak rendkívül egyszerű SQLWindows kódot kell előállítania. Lényegében nem kell mást tennie, mint a **megfelelő objektumot** felhelyeznie a **megfelelő osztályból**. Azáltal, hogy a tábla iq_mdi_table osztályból származik, és az egyik oszlopa iq_pk_col típusú, már meghatároztuk, hogy ez az oszlop bizonyos műveletekben kitüntetett szerepet fog játszani a tábla és az oszlopok közötti párbeszéd során. A táblára vonatkozó egyéb funkciókat (pl. milyen sorrendben populálódjon) dinamikusan hívott eljárások felüldefiniálásával valósíthatunk meg.

Az SQLWindows kód

Megfelelő objektum a megfelelő osztályból

- Kommunikáció az MDI kerettel
- Tranzakció-kezelés
- Automatikus SQL

Dinamikus eljárások

```
◆ iq_mdi_table: tbl_Bank
  ◇ Description:
  ◇ Named Menus
  ◇ Menu
  ◆ Tool Bar
  ◆ Contents
    ◆ iq_pk_col: BankKod
    ◆ iq_col: Nev
    ◆ ccolSzamlaSzam: CsekkSzla
    ◆ iq_user_col: Felh
    ◆ iq_timestamp_col: ModD
    ◆ iq_rowid_col: rowid
  ◆ Functions
    ◆ Function: call_bo_func
    ◆ Function: build_order_by
    ◆ Function: pass_code_to_caller
  ◇ Window Parameters
  ◇ Window Variables
  ◇ Message Actions
```

Automatikus SQL generálás

Insert into Aruforg.Cikk
(Cikkszam, ...) values (:dfCikkszam, ...)

Cikkszam = :dfCikkszam

Cikkszam AAAAAA

◆ iq_field: dfCikkszam

Az elvégzendő műveletek paramétereinek összeállításához az IQBCL beágyazó osztályok az automatikus SQL generálás technikáját alkalmazzák. **AutoSQL** alatt valami olyasmit értünk, hogy azáltal, hogy egy beágyazott elem felkerül egy beágyazó objektumra, ezáltal automatikusan részt vesz valamilyen folyamatban. Ez itt nem más, mint a megfelelő üzleti objektum metódus paraméter interfész sztringjének a dinamikus összeállítása. Az adatbeviteli objektumok attribútumaik alapján (interfész elem neve, saját bind változó neve) állítják össze az interfész lista rájuk eső részét.

Az IQBCL egyik alapkövét képezi az **optimisztikus konkurencia kezelés**. Ez azt jelenti, hogy az osztálykönyvtár alapértelmezés szerint nem használ semmilyen zárolási mechanizmust. Ehhez egy sorverzió azonosítót használ, amelyet a sorral együtt beolvas, és a módosítás ill. törlés ennek a felhasználásával történik. A CENTURA SQLBase esetén ez annak a **Rowid** oszlopa, míg Oracle ill. bármilyen más adatbázis-kezelő esetén minden táblának explicite tartalmaznia kell egy numerikus **Rowver** oszlopot, melyben az IQBCL gondoskodik a sorverzió megkülönböztetéséről.

Optimisztikus konkurencia kezelés

Egyedi sor verzió azonosító
SQLBase : Rowid
Oracle, ... : Rowver (mesterségesen előállítva)

```
Update Aruforg.Cikk Set ...
   where Rowid = :rowid
Update Aruforg.Cikk Set ...
   where Cikkszam = :Cikkszam and
         Rowver = :rowver
```

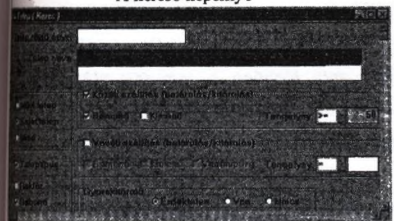
A tranzakció folyamata

Adatok előzetes ellenőrzése :
..**validate_form**(DM_Insert)
Insert műveletsztring előállítása, (..buld_sql())
és a művelet végrehajtása :
call_sql_func(DM_Insert)
A felvitel után, COMMIT előtti teendők :
..**about_to_commit**(DM_Insert)
A COMMIT művelet:
db_dml.commit()
Rowid visszaolvasása :
..**refetch_rowid**( )
A sikeres COMMIT hírüladása:
..**transaction_end**(DM_Insert, TR_Commit)

Az IQBCL beágyazó osztályai a **tranzakció folyamatát** is egységesen kezelik. A tranzakció elindításakor egy dinamikusan hívott eljárás lehetőséget ad az adatok előzetes ellenőrzésére. Ezt követő-en kerül sor az üzleti objektum metódusa számára az interfész sztringnek az előállítására, és az üzleti objektum eljárásán keresztül a művelet végrehajtására. Az adatbázis művelet után, de még a COMMIT előtti teendők elvégzésére (pl. a felvitt sorhoz tartozó detail sorok felvitelére) egy másik dinamikus eljárást hívódik meg. Ha ezen eljárások mindegyike sikeres volt, akkor az osztály COMMIT-álja a tranzakciót a db_dml tranzakciós objektumon, majd visszaolvassa a sorverzió azonosítót a további módosításhoz, majd egy informális dinamikus eljárással tudatja a programo-zóval, hogy sikeres volt-e a tranzakció (pl. lehet nyomtatni a felvitt bizonylatot).

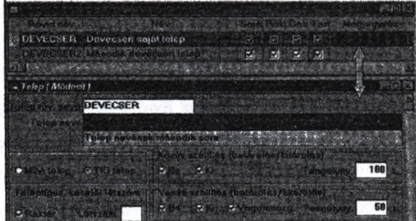
„Kereső" táblázatok

A kereső képernyő



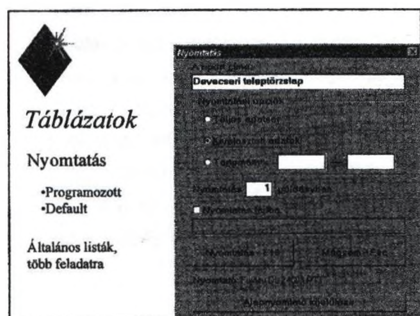
„Kereső" táblázatok

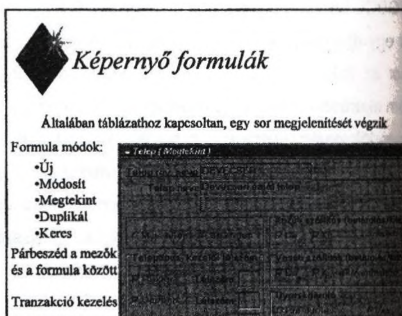
Az adatok módosítása formulán



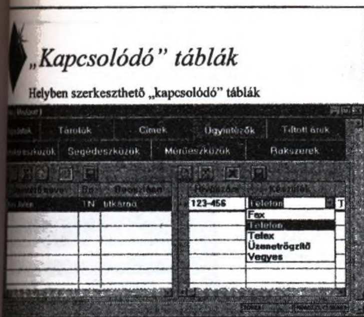
A sok oszlopot és sok sort tartalmazó adatbázis táblák kezelésére használhatók az un. **„Kereső" táblázatok.** Kliens-szerver architekturában a nagy adatbázis táblák esetén nem megengedhető, hogy a felhasználó csak úgy, találomra keresgéljen az adatok között. Ezért a tábla adatainak feltöltése előtt egy **kereső képernyő** jelenik meg, ahol a felhasználó megadhatja a keresett sor(ok) jellemzőit. Egy Query By Example jellegű feltétel összeállítása után, már csak a keresett adatokat jeleníti meg a tábla. Itt az adatok szerkesztése már nem a táblázat sorában, hanem egy, a táblázathoz szinkronizál-tan kapcsolódó képernyő formulán történik. Így a táblázat végzi az adatok vertikális megjelenítését, míg a formulában „kiterítve" kezelhetők az adatok. A táblázathoz egyszerre több nyitott képernyő formula is tartozhat.

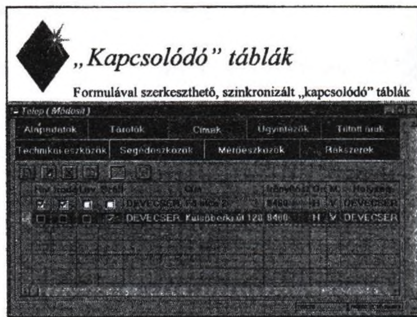
Az IQBCL osztálykönyvtárban a táblázatok mindegyike egy közös ősből származik. A táblázatok a tranzakció-kezelést végezhetik soronként, vagy az összes módosítást egyben, mint egy bizonylat tételei esetében. A leválogatott tételek tetszőleges oszlopra lerendezhetők, a rendezett oszlopon a kezdő betűsorozat begépelésével gyorskeresés végezhető. Az eredményhalmaz oszlopai között válogathat a felhasználó, hogy melyeket jelenít meg, ill. tüntet el. (Ez nagyobb lekérdezési funkciók esetén rendkívül hasznos lehet.) A táblázatok fejlett nyomtatás támogatással rendelkeznek. A nyomtatáshoz maga a táblázat szolgáltathatja az input adathalmazt. Mivel a felhasználó tetszőleges szempontok szerint leválogathatja a táblázat adatait, így rendkívül generikus nyomtatási funkciók valósíthatók meg.





Az IQBCL alkalmazásokban a **képernyő formulák** általában egy táblázathoz kapcsoltan, egy sor megjelenítését végzik. A megjelenítendő, szerkesztendő sorról a táblázat értesíti a formulát a sor primary key sztringjén keresztül. (Így a képernyő formulán történő előre, hátra lapozgatásnál is a táblázat szolgáltatja az adatokat, ezért nem érdekes, hogy az adott adatbázis-kezelő támogatja-e a visszafelé történő fetch-elést.) A táblázaton kiváltott művelettől függően a formulák különböző **üzemmód**okban jelenhetnek meg. Erről az üzemmódról a formula a beágyazott objektumait is értesíti, amelyre azok az üzemmódtól, és az objektum jellegétől függően eltérően reagálhatnak. Például „Megtekint" üzemmód esetén a beágyazott objektumok egyszerűen letiltják magukat. Az adatok felvitelét, módosítását megvalósító tranzakció-kezelés a táblázatoknál ismertetett módon történik, de a végén a formula erről a táblázatot is értesíti, hogy az is a módosított, aktuális adatokat jeleníthesse meg.





A „**kapcsolódó" (detail) táblák** megvalósítására az IQBCL osztálykönyvtárban a formulába
gazott Child Table Window típusú táblákat használhatjuk. A formula jeleníti meg a „fej", a „kapcso-
ló" tábla pedig a „tétel" adatokat. Mivel a táblázatok közös ősből származnak, ezért a „kapcsolódó"
blák a Top Level táblákhoz hasonlóan lehetnek helyben szerkeszthetők, és képernyő formulával
nkronizáltan együttműködők is. Ha a „fej" adatbázis táblához több „tétel" jellegű tábla is tartozik,
kor az ezeknek megfelelő „kapcsolódó" táblákat a formulában egymás fölé helyezzük el, és egy
iógomb jellegű opciógomb sor gondoskodik a kiválasztott „kapcsolódó" tábla megjelenítéséről. A
pcsolódó" táblán végezhető műveleteket az MDI eszközlécére hasonlító parancsgomb sor
gítségével aktivizálhatjuk.

Az IQBCL **adatbeviteli objektumai** mind közös ős osztályból származnak. Alapvető adatellenőr-
zési, ill. kötelező adatmegadás ellenőrzési funkciókkal rendelkeznek. Az AutoSQL folyamatban az
ribútumaiknak megfelelően vesznek részt. A státusz sorban egy egyszerű magyarázó szöveg
egjelenítésére képesek. A fő formula üzemmódoknak megfelelően állítható az adatbeviteli
bjektumok szerkeszthetősége. Az adatbeviteli objektumok egy lehetséges csoportosítása:

- Adatbázis specifikus osztályok: iq_pk_field, iq_fk_field, iq_rowid...
- Adatbeviteli, megjelenítési technikai osztályok: iq_radio, iq_do_field, iq_hidden_field...

z egyes alkalmazásokban ezekből örököltethetők a specifikus user interface osztályok.

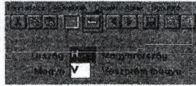
Az ügyviteli alkalmazások esetén kritikus kérdés az **idegen kulcsok** (foreign key) kezelése. Egy
yen objektumtól minimális elvárás:

- A beírt kód ellenőrzése
- Választék biztosítása (lookup)
- A kódhoz kapcsolódó adatok megjelenítése

Az IQBCL felhasználóbarát módon kezeli a több mezőből álló összetett kulcsokat tetszőleges
számú a kódhoz kapcsolt mezővel. Az ellenőrzést, ill. a kapcsolt adatok megjelenítését természe-tesen
a megfelelő üzleti objektumok végzik.

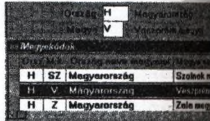
Idegen kulcsok kezelése

- Közvetlen adatbevitel esetén ellenőrzés
- Választék lehetőségének a jelzése
- Aktivizálás esetén lookup tábla felhívása



Idegen kulcsok kezelése

- Kiválasztás után adatok átvétele



Párbeszéd a tábla és a mező között

A foreign key objektumok az MDI eszközlécén jelzik, ha a fókusz beléjük érkezik, és onnan lookup tábla aktivizálható. A lookup táblából a kiválasztott sor adatai egy speciális „postaládán" keresztül jutnak el a foreign key objektumhoz.

Az IQBCL osztálykönyvtárban az eddig ismertetett tábla típusok mindegyike lehet lookup tábla. A tábla aktivizálása előtt a kód mezők egyfajta „elő kereső" formulaként viselkednek, azaz a lookup tábla adatai a kód mezők által összeállított Query By Example típusú keresési feltétel mellett jelen-nek meg. Egy speciális kontroll hatására a (normálisan editálhatatlan) kapcsolt mezők alapján is végezhet keresést a felhasználó. (Egy számla felvitele során például nem biztos, hogy ismeri a part-ner kódját, de a nevét már feltehetően igen.) A lookup tábla ráadásul egy másik alkalmazásból is aktivizálható, ekkor a DDE-n keresztül történnek az adatok átadása-átvétele.

Az IQBCL osztálykönyvtárhoz egy **prototípus generátor** is rendelkezésre áll. Ez a CENTURA cég **Component Developer Kit** (CDK) osztálykönyvtárának a felhasználásával készült. A CDK lehetővé teszi az SQLWindows fejlesztő eszköz külső programból történő programozott vezérlését, így kiváló lehetőséget nyújt program generátorok fejlesztésére. Az IQBCL prototípus generátor végül is nem csinál mást, mint a **megfelelő objektumot** generálja (inicializálja) a **megfelelő IQBCL osztályból.** A generátor semmilyen funkcionalitást nem generál, mivel azt a generált objektumok a megfelelő osz-tályokból öröklik. Az így előállított prototípus abban különbözik más prototípusoktól (pl. a CENTURA cég által kiadott QuickObject-ekkel készített prototípusok), hogy ez az elkészítendő éles alkalmazás alapját képezi, mivel ugyanazokat az osztályokat használja, így miután bemutattuk a felhasználónak a generált alkalmazást, nem kell kidobni, és újrakezdeni az egész fejlesztést, hanem már csak fino-mítani, a tényleges tudással kell felruházni a prototípusunkat. Az IQBCL prototípus generátor további előnye, hogy alkalmazása révén a generált kód példát mutat az IQBCL használatában kezdő progra-mozóknak az osztálykönyvtár objektumainak a használatához.

A generálás alapja lehet az adatbázis rendszertáblái, ill. egy CASE eszköz repository-ja. Ennek feltérképezése alapján a generátor előállítja a kiválasztott táblához tartozó üzleti objektum vázát, a „szerkesztő", vagy „kereső" táblát a képernyő formulájával, a szükséges dinamikus eljárás deklará-ciókkal. A beágyazott objektumok alapvető attribútumait (név, típus, mezőhossz...) inicializálja. A mezők sorrendjét, számát a generálás során meghatározhatjuk. A generátor feltérképezi, és ennek megfelelően - kellő beavatkozási lehetőség mellett - legenerálja a foreign key és detail tábla objektumokat is.

Objektum-orientált folyamat-vizualizáció

Fóris Tibor, dr. Szirmay-Kalos László, Márton Gábor

Budapesti Műszaki Egyetem, Villamosmérnöki és Informatikai Kar,

Folyamatszabályozási Tanszék

Absztrakt

A cikk egy objektum-orientált fejlesztőrendszert mutat be, amely az olyan folyamat-vizualizációs rendszerek létrehozásához ad hatékony eszközt (módszert) ahol a megjelenítendő illetve irányítandó rendszer nagy számú komponensből áll és az ezek között fennálló kapcsolatok topológiája is bonyolult. A kezelhetőség érdekében az alkalmazás fejlesztőnek főleg grafikus editorokkal és olyan fogalmakkal kell dolgoznia, amelyek lehető legközelebb állnak az illető szakterület ismerőihez. A képernyőképek kialakításával párhuzamosan történik a modell leírása. A rendszer komponenseinek deklaratív specifikációja alapján a kód generátor egy C++ osztályt generál. A megfelelő metódusok felüldefiniálásával a működés specifikálható. Ezen osztályok és a megrajzolt sémák alapján a fejlesztő rendszer automatikusan hozza létre a teljes futtatható rendszert.

Kulcsszavak: folyamat-vizualizációs fejlesztő rendszerek, objektum-orientált tervezés, model-view-controller paradigma

Bevezetés

Folyamat-vizualizációs rendszer alatt egy olyan programcsomagot értünk, amely egy (ipari) folyamat állapotát jeleníti meg grafikus módon és beavatkozási lehetőséget biztosít a felhasználó számára.

Tekintettel arra, hogy a megjelenítendő (irányítandó) folyamat csak bizonyos állapotváltozóinak alakulásáról áll rendelkezésre mért információ, és ezeknek változása általában nem értelmezhető direkt módon a felhasználó által, a folyamat-vizualizációs rendszernek modelleznie kell magát a folyamatot. A grafikus felhasználói felület ennek a belső modellnek az állapotát jeleníti meg.

Az általunk kidolgozott módszer olyan folyamat-vizualizációs rendszerek kialakítására szolgál ahol a megjelenítendő és/vagy irányítandó rendszerek nagy számú komponensből állnak (több ezer, több tíz ezer). A komponensek kapcsolatai bonyolult topológiájú rendszert alkotnak, ráadásul a rendszer gyakran változhat komponensek hozzáadásával vagy eltávolításával. A rendszert alkotó komponens típusok száma viszonylag alacsony (több tíz). Ilyen típusú rendszerek gyakran fordulnak elő a valóságban, például ilyenek a digitális és analóg hálózatok, áramellátó rendszerek, közvilágítási hálózatok, vasút, emberi agy, stb.

Figyelembe véve a felsorolt tulajdonságokat, megállapítható hogy egy a céloknak megfelelő rendszernek a következő követelményeket kell kielégítenie:

1. A rendszer konfigurálásakor olyan fogalmakkal kell dolgozni illetve olyan információkat kell felhasználni amelyek közel állnak a konkrét terület szakértőihez illetve amúgy is rendelkezésre állnak, mint például (mérnöki) tervrajzok és a komponensek működését leíró dokumentáció.
2. A modell és a megjelenítés konzisztenciáját automatikusan biztosítani kell.
3. A rendszer használata során a programozási feladatokat minimalizálni kell.
4. Mivel általában real-time rendszerekről van szó, a válaszidő, a sebesség döntő fontosságú.

A folyamat-vizualizációs rendszerek tervezése során a felhasználónak egyrészt a rendszer modelljét kell specifikálnia, másrészt le kell írnia az információ megjelenítésének módját.

Jelen fejlesztőrendszerben a megjelenítés és a modell definiálása párhuzamosan, grafikus szerkesztőprogramok segítségével történik és ez általában egy vagy több képernyőkép (*séma*) megadását jelenti. A sémák definiálásakor nemcsak az információ megjelenítésének módját kell meghatározni és azokat az eszközöket amelyek segítségével a felhasználó beavatkozhat az ellenőrzött folyamatba, hanem a modell objektumokat illetve a közöttük lévő kapcsolatokat is specifikálni kell. Mivel a modell és a megjelenítés leírása párhuzamosan történik, a kettő konzisztenciája automatikusan biztosított.

A megjelenítés és a modell leírásán kívül a modell működését is definiálni kell. Megfigyelhető, hogy a valós folyamatok leírhatók úgy mint különálló komponensek egymással kommunikáló hálózata. Ha ezen komponensek (objektumok) viselkedését a környezetüktől független módon írjuk le úgy mint válaszreakciók a külvilágból érkező hatásokra (üzenetekre), amelyek belső állapotváltozásokat idéznek elő, illetve a környezetre visszaható tevékenységekként nyilvánulnak meg, akkor a teljes rendszer viselkedése leírható a komponensek és a közöttük levő kapcsolatok leírásával.

Egy teljes folyamat-vizualizációs rendszerhez szükséges maga a futtatható program amely tartalmazza a konkrét alkalmazási terület modell objektumainak viselkedését és az általános keretrendszert, illetve a konfigurációt amely a megjelenítés és a modell leírását tartalmazza. Megállapítható, hogy amennyiben a modell működési elve nem változik, a fizikai rendszerben történő változások, újrafordítás nélkül, a konfiguráció megváltoztatásával követhetők.

A létrehozott vizualizációs program

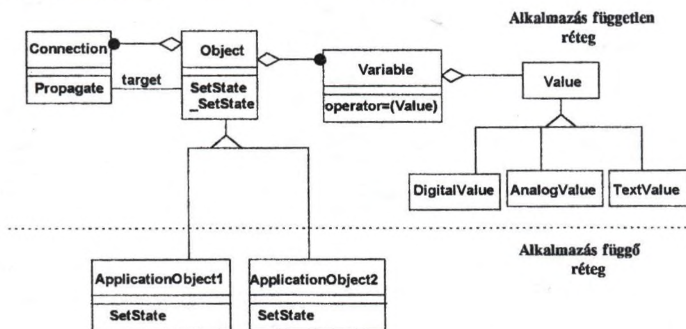
Alapvetően a futtatható program az MVC (Model-View-Controller) paradigma szerint felépített osztály hierarchiát valósít meg. Mint ismeretes, ebben a megközelítésben a felhasználói interfész és a modell teljesen elkülönül. A koncepció a feladatokat három rétegre, szintre osztja el:
1. Modell - a tulajdonképpeni problématér - nem tartalmaz megjelenítési elemeket.
2. View - a felhasználó által látott illetve az interaktivitást biztosító felület.
3. Controller - feladata a megjelenítési és modell rétegek működésének összekapcsolása.

A fejlesztőrendszer tartalmazza a vizualizációs program kernelét amely leírja az objektumok általános viselkedését, az üzenetek propagálásához szükséges keretet és az MVC rétegek konténer osztályait, majd ehhez a kernelhez szerkeszti hozzá az örökléssel kialakított új osztályokat.

A rendszer elindulásakor, a felolvasott konfiguráció alapján, létrehozza a megfelelő MVC osztályokat azaz elvégzi a példányosítást úgy a beépített mint az alkalmazás során definiált osztályokra. A példányosítással egyidőben a program felépíti a modell objektumai közötti kapcsolatokat illetve a megfelelő MVC objektumok közötti csatolásokat.

A Modell réteg

A modell szerepe a valós folyamat szimulálása, amely leírható úgy mint egymással kommunikáló objektumok hálózata. Egy objektum viselkedésének definiálása úgy történik, hogy meghatározzuk a válaszreakciókat (válaszüzeneteket) a külvilágból érkező üzenetekre, az objektum belső állapotától függően. Az objektumok belső állapotát állapotváltozóinak (*Variables*) értéke határozza meg. Az általánosságot biztosítandó, ezek a változók dinamikus típusúak azaz aktuális típusuk megegyezik az utoljára felvett értékkel. Az objektumhoz beérkező üzenet a *_SetState* függvény meghívását jelenti az illető objektumra, ezért a tulajdonképpeni viselkedést ennek a függvénynek a tartalma hordozza. A beérkező üzenet hatására az objektum megváltoztatja belső állapotát és/vagy üzeneteket küldhet más, kapcsolódó objektumoknak. Annak érdekében hogy egy modell objektum viselkedése megfelelő legyen bármely struktúrában, a modell objektumok viselkedését a struktúrától függetlenül kell leírni. Esetünkben ez azt jelenti hogy a kommunikáció más objektumokkal un. kommunikációs pontokon (*Connections*) keresztül történik. (ezen kommunikációs pontoknak az összerendelése a kernel feladata). A modell alkalmazás függő, illetve független részeit a következő objektum modell szemlélteti:



1. ábra: A vizualizációs program modell rétegének objektum-modellje

Egy áramellátó rendszerben például objektum lesz a vezeték. Ennek a nyilvánvalóan két kapcsolódási ponttal rendelkező objektumnak egyetlen egy állapotváltozója van amely azt írja le hogy a vezeték feszültség alatt van-e vagy sem. A vezeték viselkedése is egyszerűen

megfogalmazható: az egyik végéről érkező üzenetet (amely azt jelzi hogy a vezeték feszültség alatt van vagy sem) a másik vég felé kell propagálni.

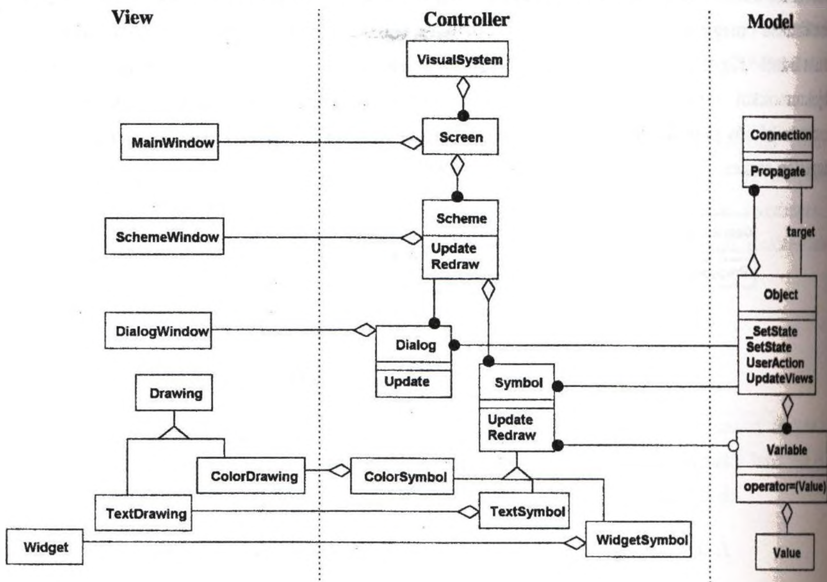
A View réteg

A megjelenítés a modell aktuális állapotát tükrözi. A view-t alkotó objektumok a *ColorDrawing* vagy *Widget* osztályok példányai. A ColorDrawing grafikus primitívek gyűjteménye. A grafikus primitívek színének változtatásával vagy eltüntetésével a megjelenítés dinamikus volta biztosított. A Widget osztály annak a hierarchiának az alaposztálya, amely az implementációtól függő widget készletet takarja. Úgy a ColorDrawing mint a Widget objektumokat a megfelelő kontroller objektumok vezérlik (lásd 2. ábra).

A Kontroller réteg

A kontroller réteg objektumai a modell és a view objektumok közötti interfészt valósítják meg. A legfontosabb kontroller osztály a *VisualSystem* amelyet a főprogram példányosít. A VisualSystem-hez tartoznak a különböző képernyők (*Screen*), egy képernyőhöz pedig több séma (*Scheme*) tartozhat. A Scheme objektum tartalmazza a *Symbol* és *Dialog* típusú kontroller objektumokat amelyek az adott sémán található view elemek megfelelő vezérlői. (lásd 2. ábra)

A modell, view és kontroller rétegek közötti kapcsolatokat a 2. ábra szemlélteti:

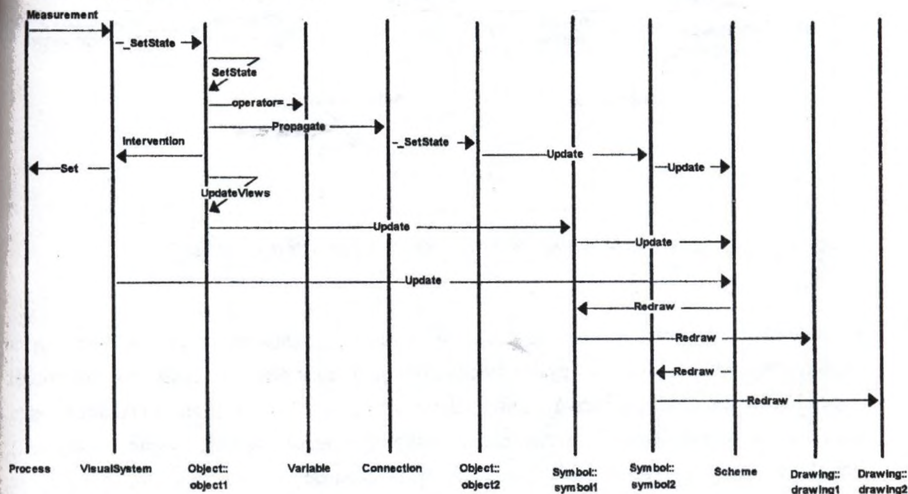


2. ábra: Modell, View és Kontroller rétegek kapcsolata

A folyamat-vizualizációs program dinamikus működése

Amennyiben a folyamatban valamilyen változás történik, a fő kontroller objektum (*VisualSystem*) üzenetet kap (*Measurement*). A megfelelő modell objektumot kikeresve a kontroller az üzenetet a *_SetState* függvény meghívásával továbbítja. Az általános reakcióért felelős *_SetState* meghívja az objektum-specifikus *SetState* függvényt. A *SetState* függvényben az objektum megváltoztatja belső állapotát az *operator=* hívással és/vagy *Propagate* hívással üzenhet más modell objektumoknak. Amennyiben egy modell objektum megváltoztatja az állapotát, a változóihoz valamint magához az objektumhoz csatolt *Symbol* kontrollerek *Update* üzeneteket kapnak hogy frissítsék a megjelenítést. A frissítés ténylegesen csak a *Measurement* által elindított üzenetlánc befejezése után, a *Redraw* üzenet hatására történik meg.

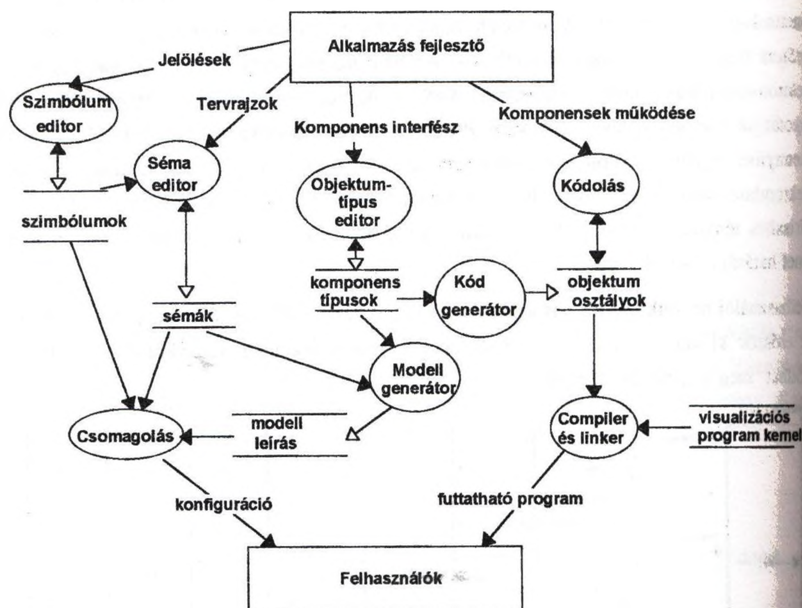
A felhasználói beavatkozás azonos üzenetláncot vált ki azzal a különbséggel, hogy a VisualSystem-nek először ki kell keresnie a cél objektumot a view objektumok lekérdezésével, csak azután hívódhat meg a *_SetState* függvény.



3. ábra: A folyamatban bekövetkező változás hatására lejátszódó üzenetlánc

A folyamat-vizualizációs fejlesztőrendszer

Az ismertetett folyamat-vizualizációs programot létrehozó fejlesztőrendszer működését a 4. ábra adatfolyam diagramja szemlélteti:



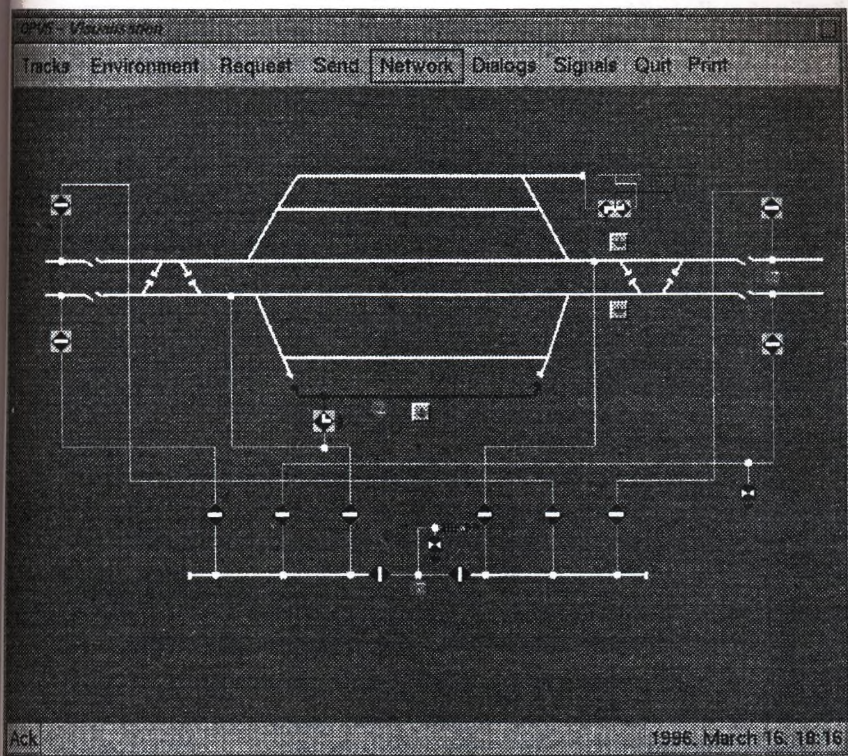
4. ábra: A folyamat-vizualizációs fejlesztési folyamat

A **szimbólum-editorral** grafikus szimbólumok dinamikus változatait hozhatjuk létre. Azon megfigyelésre építve, hogy a gyakorlatban előforduló alkatrészek lehetnek fix alkatrészek (kapcsoló, műszer, lámpa, szivattyú, transzformátor, stb.), melyek adott számú csatlakozási ponttal rendelkeznek, illetve lehetnek csatlakozást biztosító alkatrészek (vezeték, vágány, cső, stb.), melyek a fix alkatrészeket kötik össze, a szimbólumokat is két csoportra osztjuk: fix szimbólumokra és összekötő szimbólumokra.

A **séma-editor** felhasználásával a sémákat a szimbólum editor által definiált fix és összekötő szimbólumokból, valamint widget elemekből építhetjük fel. Az egyes szimbólumokhoz azt is definiálni kell, hogy mely modell objektumot jelenítik meg. Ezen információ, valamint a fix és összekötő szimbólumok által meghatározott topológia alapján a **modell-generátor** automatikusan generálja a modellt felépítő objektumok listáját és ezen objektumok által felépített rendszer struktúráját is.

Az alkalmazástervezés másik ágán az egyes komponens típusok interfészét és belső állapotát adjuk meg az **objektum típus editor** segítségével, melyből a **forráskód-generátor** egy forrásnyelvű C++ osztályt hoz létre. Ezt az osztályt kell kiegészíteni a komponenstípus viselkedését leíró definíciókkal, amely mindösszesen az osztály *SetState* függvényének üresen hagyott törzsének kitöltését igényli. A kiegészített C++ osztályokat a rendszer a vizualizációs program kerneléhez szerkeszti, elkészítve a futtatható vizualizációs programot. Ez a vizualizációs program az indítás során felolvassa a szimbólumokat, sémákat, objektumokat és a rendszer struktúrát leíró fájlokat, és ennek megfelelően építi fel a grafikus interfész (view), kontroller és modell rétegeket.

A létrehozott vizualizációs program egy jellegzetes sémaképe az 5. ábrán látható.



5. ábra: Egy létrehozott vizualizációs program egyetlen sémájának képe

Tapasztalatok

A fenti elvek alapján alakítottuk ki az OPV5 kísérleti keretrendszert. A megvalósítás során az analízis és tervezés fázisaiban objektum-orientált CASE eszközt használtunk, az implementáció C++ nyelven UNIX/OSF-MOTIF környezetre készült el. Figyelembe véve a hálózati

szolgáltatásokat, a rendszert úgy alakítottuk ki, hogy a modellt a hálózatra kapcsolódó gépeken elosztottan tárolhatja.

A javasolt fejlesztőeszközt különböző, egymástól igen távol eső alkalmazási területen teszteltük, úgy mint: elektromos energia elosztó rendszerek, vasúti közlekedés felügyelet, közvilágítási hálózatok, digitális hálózatok, ellenállásokat és kapcsolókat tartalmazó analóg hálózatok, stb. A fejlesztési módszer objektum-orientált szemlélete, miszerint a viselkedést a komponensek szintjén adjuk meg - függetlenül a rendszer struktúrájától, könnyű és gyors fejlesztést biztosított, melyben a fejlesztőnek az alkalmazás természetes fogalmaival kellett foglalkoznia.

Köszönet nyilvánítás

Ezen kutatási projektet az OTKA/F 015884 támogatta.

Irodalomjegyzék:

[1] *FIX DMACS - System development*, Display development - Intellution Inc. 1992-1994
[2] *Vision -Process Visualisation system*, DIVICON Ltd.
[3] *Visual Designer - Intelligent Instrumentation*
[4] *PVSS - Prozess-Visualisierungs- und Steuerungssystem*, EDV-Technik Mühlgassner GesmbH
[5] *Sammi, Graphical framework for real-time command and control*,
Kinesix/Scientific Software-Intercomp
[6] *Points to consider in evaluating Dynamic Data Visualisation Tools*
URL: http://www.telsa.hl.com.au
[7] *Powerful Tools to Monitor and Control Live Processes*
URL: http://www.dvcorp.com/mktg
[8] P. Mégard: *Criteria for Selecting a good GUI Development Tool*
URL: http://www.ilog.fr/Products/Views
[9] R.A. Earnshaw, N. Wiseman: *An Introductory Guide to Scientific Visualization*
Springer-Verlag, 1992
[10] W. Lalonde, J. Pugh: *Inside Smalltalk* (Volume II)
Prentice Hall, 1990
[11] G.E. Krasner, S.T.Pope:
A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80
Journal of Object-oriented Programming, August/September, 1988
[12] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen: *Object-oriented Modeling and Design*
Prentice-Hall, 1991
[13] A. Jaaksi: *Implementing Interactive Applications in C++*
Software -Practice and Experience, Vol. 25(3), 271-289 (March 1995)
[14] N. Knolle: *Why Object-oriented User Interface Toolkits are better*
Journal of Object-oriented Programming, Vol. 2, 1989
[15] B. Shneiderman: *Designing the User Interface*
Reading Mass., Addison Wesley, 1986

SOM/DSOM technológia

Nyikes Tamás, IBM Magyarország

Bevezetés

A következőkben összefoglalom a SOM/DSOM objektum technológiával kapcsolatos alapkoncepciókat. Még mielőtt rátérnénk a SOM/DSOM tárgyalására, érdemes elidőzni egy keveset az OMG által kidolgozott CORBA specifikációnál, mely a SOM/DSOM alapjait is jelenti.

Objektum-orientált programozási irányok

Napjaink számítástechnikájának egyik leggyorsabban növekvő ága az objektum technológia és objektum-orientált nyelvek felhasználása. Nehéz figyelmen kívül hagyni a költségcsökkentési mutatókat, melyek szoftverkomponensekkel való fejlesztésből adódnak. A komponensekből felépülő szoftver rendszerek könnyebben karban tarthatók és ráadásul a komponensek újból fel is használhatók a fejlesztés során.

Az objektum-orientáltság alapelveit a szűk szakma már több mint 20 éve nagyon jól ismeri. Azonban csupán napjainkban mondható el, hogy azok az eszközök, nyelvek, melyek teljes mértékben kihasználják ezen elveket elérhetők lennének az átlagos fejlesztők számára is.

Nem szándékozom az objektum-orientáltság alapelveit ismertetni. feltételezem, hogy a bezártság, öröklődés és a többi alapelv jól ismert a hallgatóság számára. Ha egy alkalmazást úgy tekintünk mint egymással kölcsönhatásban lévő objektumok egy halmaza, ahol az objektumok interfészei és a bezártság jól definiált, akkor a különálló komponensek egyenként kifejleszthetők, tesztelhetők és karbantarthatók. Ha még ezen kívül a komponensek előzetesen alaposan meg lettek tervezve, akkor más alkalmazásokban is felhasználhatók. Általában még az is előfordulhat, hogy egy jól megtervezett komponens, ugyanazon alkalmazáson belül is többször felhasználható.

Osztott objektumok

Az objektum-orientált technológia mellett napjaink másik népszerű technológiája a kliens-szerver technológia, vagy ennek újabb továbbfejlesztése, a hálózat középponttú számítástechnika. Ezen trendek természetes következménye az osztott objektumok támogatásának igénye. A felhasználó szemszögéből tekintve a kliens-szerver, vagy hálózat középpontú számítástechnika adja az erőforrások eléréséhez a közeget, míg az objektum technológia gondoskodik arról, hogy az alkalmazásokat kezelhető módon fejleszthessük ki.

Bár a gondolat, hogy különálló, jól elkülöníthető programozási egységeink legyenek (objektumok) hamar elvez minket az osztott funkció fogalmához, a legtöbb hagyományos objektum-orientált programozási nyelv és környezet egyfolyamatos, tehát nem osztott környezetben lett megvalósítva.

Másrészről az alkalmazások jelentős részénél az egyfolyamatos modell egyszerűen nem kielégítő. Az osztott objektumok adják meg a lehetőséget egy alkalmazás számára, hogy a különböző komponensek különböző folyamatokban fussanak, akár fizikailag más helyen is. A hálózat-középpontú számítástechnikának az egyik definíciója éppen az, hogy az alkalmazások, alkalmazás komponensek a hálózat tetszőleges részéről, a fizikai lokáció pontos tudta nélkül, vehessenek igénybe szolgáltatásokat.

Mivel a legtöbb hálózatos környezet heterogén rendszerekből épül fel, nyilvánvaló követelmény, hogy szabványokat érvényesítsünk abból a célból, hogy a különböző platformokon található, különböző programnyelvekben kifejlesztett objektumok együtt tudjanak működni. Ezek a szabványok lehetővé teszik, hogy a programfejlesztők szabványos felületeket használhassanak, melyeken keresztül objektum metódusokat hívhatnak meg, úgy hogy a hívást a cél objektum platformtól és objektum modelltől függetlenül tudja fogadni.

Ezeket a szabványokat az OMG (Object Management Group), mely az osztott objektum-szabványok kidolgozásával és elfogadásával foglalkozó számítástechnikai konzorcium, hozza létre. Az általuk kifejlesztett CORBA (Common Object Request Broker Architecture) ilyen szabványok egy gyűjteménye, melyet arra terveztek, hogy a fent említett követelményeket kielégítse.

CORBA

Jelen tanulmány elsősorban a CORBA V1.2-őn alapul. Bár a CORBA 2-es verziója már megjelent, és sok új, fontos specifikációt tartalmaz, a tárgykör szempontjából lényegesen új elemek nem találhatók benne.

Az OMG a világ legnagyobb szoftverfejlesztési konzorciuma, több mint 430 tagja van, melyek között vannak szoftvereladók, szoftverfejlesztők és felhasználók is. 1989-ben alapították azzal a céllal, hogy az objektum technológiát elméletének és gyakorlatának elterjedését, és osztott objektum-orientált rendszerek megvalósítását segítse. A fő cél, hogy olyan közös architektúrális keretrendszereket tudjon biztosítani, melyek alapján olyan objektum-orientált alkalmazások fejleszthetők, melyek széles körben hozzáférhető interfész definíciókon alapulnak.

Az első és legfontosabb specifikáció, melyet az OMG publikált a CORBA, mely a objektum igénylési bróker (ORB) fogalmán alapszik. Az ORB szabványos felületeket nyújt, melyen keresztül egy kliens

metódusokat hívhat meg egy objektumon, objektum modelltől független módon. Az ORB fogadja a metódus hívási kéréseket, majd szabványos felületen keresztül eléri a cél objektum modellt, és a kérdéses metódusokat végrehajtja a cél objektumokon.

Két fő interfész létezik a kliens-programozó részére, melyeken keresztül metódusokat tud hívni. Az egyik egy statikus felület, melynél szükséges a cél objektum-osztály és metódus aláírások ismerete a fordítási időben. A másik egy dinamikus felület (Dynamic Invocation Interface, DII), mely segítségével dinamikusan építhetjük a metódusokat, és hívhatjuk őket a futtatás során. A két interfész bármelyikét is használják a kliens-programozók, az ORB biztosítja, hogy a hívások szemantikája ekvivalens mindkét esetben.

Az ORB kiszolgáló oldalán egy objektum adapter áll rendelkezésre, mely a tényleges metódus hívásokat kezeli le a cél objektumon. A CORBA-ban a kiszolgáló folyamatot implementációnak hívják. A kliens oldalon többféle elérési út áll rendelkezésre ahhoz, hogy egy metódust meghívhassunk. Részletesebb információ található az OMG CORBA specifikációjában.

Az ORB belső működése (és a kliens és szerver közötti kommunikáció részletei) nincsenek specifikálva a CORBA-ban. Ez az oka annak, hogy a különböző gyártók által megvalósított ORB-ok nem feltétlenül tudnak kommunikálni egymással.

Amit a jelenlegi szabvány biztosít az alábbiakban van összefoglalva:

Interface Definition Language (IDL)

Egy nyelvfüggetlen szintaxis, mely arra szolgál, hogy az objektum osztályok interfészeit definiálhassuk. Amennyiben az IDL-t használjuk, úgy objektum modelltől független objektum osztály interfész hozható létre. Ez esetben az objektum modell lehetséges, hogy egy adott nyelvvel (C++, SmallTalk, stb) teljesen egybe van kapcsolva, de lehet tőle független is.

C kötések

A jelenlegi CORBA specifikáció megadja, hogy egy C programozó milyen szintaxist kell, hogy kövessen ahhoz, hogy metódusokat hívhasson meg olyan objektumokon, melyek osztály felülete az IDL által van definiálva. A későbbi CORBA verziók más nyelvek esetén is kitérnek majd a szintaxis definícióra. A szintaxis definiálásával a CORBA biztosítja a forrásnyelvi kompatibilitást a kliens oldalon. Ily módon tehát az a kliens, amely e szabványoknak megfelelően végzi a metódusok meghívását, egyszerű újrafordítás után használható más platformokon is, ahol rendelkezésre áll

valamilyen CORBA kompatibilis ORB. Ezek a C kötések jelentik a statikus felületet, amelyről már tettünk említést.

Dynamic Invocation Interface (DII)

A DII tulajdonképpen egy API gyűjtemény, mely segítségével metódus hívásokat indíthatunk, melyek létrehozása dinamikusan a futási időben történik. Ezek az interfészek minden ORB esetében ugyanazok, mely megvalósítja a DII-t. Az objektum implementációk (célobjektumok) nem tudják, hogy a rajtuk végrehajtott metódusok dinamikusak vagy statikusak voltak-e. Mindkét esetben a célobjektumon ugyanolyan metódusok hajtódnak végre.

Felület tárház

A CORBA szól az felület tárház fogalmáról és egy sereg felületről, melyek el tudják azt érni. A felület tárház tartalmazza ugyanazt az információt egy osztályról, ami az osztály IDL-jében van definiálva. Bármely komponens, a kliens program, vagy az ORB elérheti az osztálydefiníciókat futás közben is. Ez nagyon nagy lehetőségeket jelent, például egy osztály felületéről dinamikus információt nyerhetünk, és ezen információ alapján hozhatunk létre metódusokat, melyeket a futási idő alatt meg is hívhatunk. Egy osztott környezetben az ORB-nak képesnek kell lennie arra, hogy egy metódus hívást az összes paraméterével be tudjon csomagolni a továbbítás céljából. A fogadónak képesnek kell lennie a kibontásra. Az ORB ezeket a műveleteket a felület tárház segítségével valósítja meg.

Alap objektum adapter (Basic Object Adapter, BOA)

Az objektum adapter szerepe, hogy egy felületet adjon az ORB és az objektum implementáció között. A CORBA specifikál egy ilyen adaptert, melyet BOA-nak hív és ez benne kell, hogy legyen minden ORB megvalósításban. A BOA felületei IDL-ben vannak megírva, így különböző nyelvi megvalósítások esetén is használhatók. A BOA által nyújtott legfontosabb szolgáltatások:

- objektum referenciák generálása és értelmezése
- a hívó fél azonosítása, engedélyeztetése
- az implementáció (szerver folyamat) aktivalizálása, deaktivalizálása
- az egyedi objektumok aktivalizálása és deaktivalizálása
- metódusok meghívása vázokon keresztül.

Implementáció tárház

Az implementáció tárház egy regisztrációs mechanizmus objektum kiszolgáló implementációk számára. Rendszerint a feltöltése az alkalmazás installációjánál történik, de dinamikusan frissíthető , ha szükséges. Az implementáció tárház nem keverendő össze a felület tárházzal. Az implementáció tárház a létező kiszolgálókról tartalmaz információt, míg a felület tárház az osztály felületekről.

ORB felület

Az ORB felület egy szabványos interfész, melyen keresztül az ORB olyan funkcióit érjük el, melyek közösek a klienseken és az implementációkon

A CORBA specifikáció szolgáltatja az alapokat az osztott objektumokból álló rendszerek esetén. A fentiekben tárgyaltakon túl sok egyéb kérdés merülhet fel, mint például biztonság, tranzakciós szolgáltatások, objektum perzisztencia, stb. Ezekre a témakörökre az OMG szabványai és anyagai térnek ki részletesen.

Osztott objektumok összefoglalása

Tehát az osztott objektumok teszik lehetővé, hogy kihasználhassuk az objektum technológia által nyújtott előnyöket kliens-szerver, vagy hálózat-középpontú rendszerek esetében is. A számítástechnika, az OMG konzorciumon keresztül folyamatosan fejleszti azokat a szabványokat, melyek biztosítják az alkalmazások átvihetőségét és egymással való együttműködését különböző platformokon, programozási nyelvekben és objektum modellekben.

A szabványok azonban semmit sem érnek, ha nincsenek termékek, melyek megvalósítják őket. SOM és a DSOM keretrendszer a CORBA-nak egy közel teljes megvalósítása. A következő fejezetben áttekintjük, miként is történik a megvalósítás.

System Object Model (SOM)

Ez a fejezet egy gyors betekintést nyújt a SOM technológiába, melyet majd az osztott SOM technológia követ. A SOM tulajdonképpen szoftver objektumok csomagolására szolgáló technológia. Ez a technológia egyedülálló abban, hogy

- kiváló objektum technológia
- programozási nyelvtől való függetlenséget biztosít
- osztály könyvtárak verzióról verzióra történő bináris kompatibilitást valósít meg.

Kiváló objektum technológia

A SOM teljes támogatást nyújt a közismert objektum-orientált fogalmakra:

- öröklődés (többszörös öröklődés is!)
- többalakúság
- objektum osztályok futási idejű objektumként történő reprezentációja.

Többszörös metódus feloldási módszerek állnak rendelkezésre sokféle funkcionális és teljesítmény jellemzőkkel. Például ha egy objektum osztálya és az adott metódus amit meg szeretnénk hívni azon az objektumon már fordítási időben ismeretesek, egy nagyon gyors offset feloldási technika használható. Azonban, ha az osztály és metódus azonosítására csak futási időben van lehetőség, akkor a SOM ezt az esetet is le tudja kezelni, de ekkor leválasztási metódus feloldást használ.

A SOM alapú objektum osztályok futási időben is dinamikusan linkelhetők. A leválasztási technika és más dinamikus módszerek segítségével a SOM nagyon rugalmas programok írását teszi lehetővé.

Programozási nyelvtől való függetlenség

A SOM használata nem korlátolódik egyetlen programozási nyelvre. A SOM metódusok elérése olyan rendszer kötéseken keresztül történik, melyek gyakorlatilag minden nyelven rendelkezésre állnak, bármilyen platformon. Nyelvfüggő kötések generálhatók fejléc állományok (header files) formájában, melyek leírják a metódusok meghívásának és az objektum adatok elérésének a szintaxisát. Ez a szintaxis konzisztens a különböző operációs rendszer platformokon, ami biztosítja a forrásszintű hordozhatóságot.

A SOMObject Developer Toolkit biztosítja, hogy C és C++ nyelvek esetén ezeket a kötéseket le lehessen generálni. Más nyelvekre vonatkozó kötések később jelennek meg az adott programozási nyelv fejlesztőitől. Például a VisualAge SmallTalk jelen pillanatban támogatja, hogy más nyelven írt SOM objektumokat érjünk el, mintha azok SmallTalk objektumok lennének.

Objektum osztályok, melyeket egy adott nyelven fejlesztünk ki, részosztályra bonthatók egy másik nyelven, és a kliensek egy harmadik nyelvből is elérhetik a mindkét osztályból származó objektumokat. Azok az objektum modellek, melyek egy adott nyelvhez kötődnek, nem csupán a nyelvhez kötődnek, de az esetek nagy részében egy adott gyártó fordítójához is, és lehetséges, hogy annak egy adott verziójához is ragaszkodnak. Azzal, hogy a SOM nyelvtől független, eltűnik a fordítótól, illetve annak verzióitól való függőség is. Így a kliens oldali fejlesztő, aki alosztályokat hoz létre nincs rákényszerítve, hogy egy adott nyelvi környezetet használjon.

Azáltal, hogy egy adott osztály könyvtárat nem csak egy programozási nyelv programozói használhatnak, megnő azoknak a programozóknak a száma, akik részt vehetnek a fejlesztésben. Tehát az objektumok újrafelhasználásának a valószínűsége is növekszik, ami az alapja a gyors fejlesztési módszereknek (RAD).

Osztály könyvtárak verzióról verzióra történő bináris kompatibilitása

Objektum modell megvalósításoknál (mint a C++ esetén) az objektum osztályok fordítási időben létező fogalmak, egységek. Tehát a fordítási idő alatt az egy osztályhoz rendelt metódus táblák és adatstruktúrák statikusan definiáltak. Ha egy ilyen osztályt odaadunk egy programozónak, hogy alosztályokat hozzon létre vagy más módon elérje az osztályt, akkor ahhoz a szinthez van kötve, amit az osztálykönyvtár a fordítási időben képviselt. Ha később az eredeti osztálykönyvtár fejlesztője úgy dönt, hogy elkészíti az osztálykönyvtár új verzióját, amely javításokat tartalmazhat az eredeti kódhoz, vagy új metódusokat és osztály adatokat, akkor a kód amely felhasználta az osztály könyvtárat (esetleg alosztályok formájában) újrafordítandó.

Emiatt nagyon nehézkes az osztálykönyvtárak fejlesztése, melyeket más programozók használhatnak. A SOM megalkotásánál az egyik legalapvetőbb szempont volt e fogyatékosságok kiküszöbölése, és bináris kompatibilitás elérése. Tehát az osztálykönyvtár új verziójának megjelenésével nincs szükség újrafordításra. Az alapelv nagyon egyszerű:
Ha a kliens nem kíván meg forrásszintű változtatást (új metódus használata) akkor a bináris kódját ne kelljen megváltoztatni.

Sokféle transzformáció képzelhető el, mellyel az osztálykönyvtár készítő gazdálkodhat anélkül, hogy megsértené a verzióról verzióra történő bináris kompatibilitás (RRBC, Release-to-Release Binary Compatibility) fogalmát. Ilyenek például az implementációhoz új metódusok vagy adatok adása, az osztályhierarchiában egy metaosztály leszármaztatása, vagy új osztály hozzáadása a hierarchiához.

SOM összefoglalás

A gazdag objektum technológiai elemek, nyelvfüggetlenség és RRBC elhárította az akadályokat az objektum technológia széleskörű felhasználása elől. Ezen technológiai elemek felhasználásával a szoftverfejlesztők most már nekikezdhetnek, hogy üzleti alkalmazásoknál is használható osztálykönyvtárakat fejlesszenek. Az ezekből előálló keretrendszerek, részek más alkalmazások alapjául szolgálhatnak. A cégek megvásárolhatják ezeket az osztálykönyvtárakat és keretrendszereket, és felépíthetik saját alkalmazásaikat a saját maguk által választott nyelvi környezetben, anélkül, hogy az általuk fejlesztett kódot újra kellene fordítani, amikor az osztálykönyvtárak új verziói megjelennek.

Ezáltal a számítástechnika mint egész, az objektum technológia két ígéretét kapja meg, a kód újrafelhasználhatóságát és a komponensekből történő fejlesztést.

Osztott SOM áttekintés (DSOM)

A DSOM keretrendszer az SOM osztályok egy halmaza, melyek egy CORBA-nak megfelelő ORB-ot alkotnak a SOM objektumok szétosztásához. Jelenleg a SOM és a DSOM OS/2, AIX, OS/400, MVS és Windows alapú rendszereken érhető el. Később további platform-támogatások is megjelennek majd, ami a SOM/DSOM technológiát a legheterogénebb osztott objektum technológiává teszi.

Az IBM törekvése, hogy a későbbi OMG szabványokat, mint amilyen például az Együttműködési Szolgáltatások, is megvalósítsa. Ezáltal, akik jelen pillanatban olyan platformmal rendelkeznek melyen a DSOM nem áll rendelkezésre, amint megjelennek az OMG Együttműködési Szolgáltatásai, akkor azoknak is lehetőségük lesz az együttműködésre.

Lássuk, hogyan valósítja meg a DSOM a CORBA specifikációkat.

CORBA specifikáció	SOM/DSOM megfelelés
IDL	A SOM objektum osztályok CORBA-nak megfeleltetett IDL-ben vannak definiálva
C kötések	A SOM fordító feldolgozza az IDL kódot, majd C kötéseket generál, melyek a CORBA által definiált C leképezéseknek felelnek meg. (Továbbá rendelkezésre állnak C++ kötések és további nem konformis C kötések is)
DII	SOM támogatja a CORBA által definiált interfészt a felület tárházhoz. Továbbá, a SOM fordító fel is tudja tölteni a tárházat, amikor az IDL állományokban lévő osztály definíciókat dolgozza fel.
Felület tárház	A SOM fordító feldolgozza az IDL kódot, majd C kötéseket generál, melyek a CORBA által definiált C leképezéseknek felelnek meg. (Továbbá rendelkezésre állnak C++ kötések és további nem konformis C kötések is)

Alap objektum adapter (BOA)	A SOM a CORBA által definiált BOA-t megvalósítja, melyet SOMOA-nak hív. A SOMOA tulajdonképpen a BOA funkcióit tartalmazza néhány SOM specifikus kiegészítéssel.
Implementáció tárház	SOM támogatja a CORBA által definiált interfészt a felület tárházhoz. Továbbá, a SOM fordító fel is tudja tölteni a tárházat, amikor az IDL állományokban lévő osztály definíciókat dolgozza fel.
ORB felület	DSOM a CORBA által definiált felületeket használja, hogy ORB szolgáltatásokat érjen el, mint amilyenek az objektum referencia műveletek is.

1. tábla SOM/DSOM CORBA megfelelése

A DSOM kereszt-folyamat szintű elérést biztosít a SOM objektumok számára. Ezek a folyamatok ugyanazon, vagy fizikailag különböző gépen is lehetnek. Továbbá, a rendszerek ugyanazt, vagy akár különböző operációs rendszert is futtathatnak. Például egy OS/2 alkalmazás elérhet egy AIX rendszeren található objektumot és viszont.

Amikor két folyamat ugyanazon a fizikai gépen található, az ORB egy folyamatközti kommunikációs eszköz biztosítja a metódus hívások és eredmények átadását a folyamatok között. Amikor viszont a két folyamat fizikailag különböző gépen található, egy hálózati protokoll gondoskodik az átadásról. Az előbbi esetet hívjuk munkaállomás DSOM-nak, azt utóbbit pedig munkacsoport DSOM-nak.

A munkaállomás DSOM esetében a kliens programok más folyamatokban futó objektumokon is meghívhatnak metódusokat. Legalább három folyamat vesz részt a kommunikációban. A kliens folyamat tartalmazza az alkalmazás logikát, amely az objektum eléréséhez szükséges. A metódusok proxy objektumokon hívódnak meg, mely objektumok szintén ugyanabban a folyamatban találhatók mint a kliens. A proxy objektumon történt metódus hívás eredményeképpen folyamatközti kommunikáció során a metódus hívási üzenet eljut a tényleges objektum folyamatához. Ekkor a metódus végrehajtódik és az eredmény hasonló mechanizmuson keresztül kerül vissza a kliensig.

A harmadik folyamat az, amely a DSOM démont (somdd) futtatja. Ez a démon ellenőrzi a szerver folyamatok létezését és újakat indít, amennyiben szükséges. A DSOM démon elengedhetetlenül szükséges ahhoz, hogy a kliensek és a kiszolgálók kommunikálni tudjanak egymással.

A munkacsoport DSOM nagyon hasonló a munkaállomás DSOM környezethez, bár ekkor a kiszolgáló folyamatok egy másik rendszerben működnek. A kliens program és az osztály implementációk abszolút nem tudnak arról, hogy munkaállomás, vagy munkacsoport DSOM-ot használnak-e. A fő

különbség az, hogy az üzenet, amely a metódus nevét és paramétereit viszi nem folyamatközti kommunikáció útján kerül át, hanem valamely támogatott hálózati protokoll segítségével. A DSOM démon, amelyről már szóltunk a munkaállomás DSOM esetén, itt csak a kiszolgáló rendszereken kell, hogy fusson.

A támogatott protokollok a TCP/IP, NetBIOS és Netware IPX. Létezik egy generikus socket interfész, melyet a DSOM a rendszerek közti kommunikációra használ. Futási időben ez a generikus socket interfész leképződik a megfelelő hálózati protokollra. Az alkalmazásfejlesztő szempontjából ez transzparens, tehát nem kell ismernie a protokollt, amelyen az alkalmazás majd futni fog. A generikus socket IDL-ben van definiálva, Sockets osztály név alatt. Egy másik protokoll támogatásához a kommunikációs programozó a Sockets-ből egy másik alosztályt hoz létre, és átírhatja annak összes metódusát, hogy megfeleljen e másik protokollnak.

A futási idő protokollja konfigurációs beállítással érhető el, csupán egy környezeti változót kell átírni.

Összefoglalva, a DSOM objektum proxy-k segítségével éri el, hogy az osztott objektumok lokációja rejtve legyen a kliens elől. A program, vagy programozó nem tud róla, hogy munkaállomás, vagy munkacsoport DSOM lesz használva.

Elosztott objektum-orientált rendszerek tervezése

dr. László Zoltán, Molnár István

(laszlo@fsz.bme.hu, molnar@fsz.bme.hu)

Budapesti Műszaki Egyetem

Villamosmérnöki és Informatikai Kar

Folyamatszabályozási Tanszék

Az objektum-orientált rendszerek szerkezete sok esetben kialakítható a Model-View-Controller (MVC) paradigma alapján. Jelen cikk röviden bemutatja az MVC elvű architektúrát, majd megadja annak egy kiterjesztését a folyamatirányító rendszerekre vonatkozóan. Ezt követően tárgyalja a modell egy lehetséges értelmezését elosztott rendszerek esetében. Ennek alapján a szerzők tervezési módszert dolgoztak ki elosztott objektum-orientált rendszerek fejlesztésére. A javasolt módszer előnye, hogy dinamikusan képes változtatni az elosztott rendszer struktúráját a folyamatban végbemenő változásoknak megfelelően.

Kulcsszavak: objektum-orientált tervezés, elosztott rendszerek, Model-View-Controller (MVC) paradigma

Bevezetés

Az MVC paradigma alapgondolata, hogy egy objektum-orientált rendszer felépíthető két fő részből. A rendszer egyfelől áll egy Modellből (M), amelynek célja a valós világ objektumaiak szimulálása, másfelől tartalmazza a Modellnek a megjelenítését (vizualizációját). Ezen utóbbi rész további két elemre bontható, a látványért felelős View(V)-ra és a Modell illetve a View közötti kapcsolatot biztosító Controller(C)-re. A rendszereknek

ez a megközelítése nagyban leegyszerűsíti és áttekinthetővé teszi a megjelenítést felhasználói szinten.

Amennyiben a V és C elemek értelmezését általánosítjuk oly módon, hogy a V és C jelenthet tetszőleges - például folyamat-interfészen keresztül történő - beavatkozást a folyamatba és adatgyűjtést a folyamatról, akkor az MVC elv alkalmazható folyamatirányító rendszerek esetében is.

Az MVC elvű architektúrák kiterjeszthetők az elosztott rendszerek esetére is. Ekkor az M és a VC elemek közötti közvetlen (gépen belüli) kapcsolatot az elosztott rendszer komponensei közötti (hálózati) kapcsolat váltja fel, amely komoly implementációs és hatékonysági problémákat vet fel. A felmerülő problémák egy lehetséges megoldása, hogy az M-et továbbra sem megosztva, de annak a VC elemek szempontjából fontos részeit az elosztott rendszer komponenseire átmásoljuk.

A modell és másolatainak egységes kezelése céljából kifejlesztettünk egy objektum-orientált tervezési módszert, ami egy olyan összetett objektumon alapul, amelyben a modell és a másolat szerep különválasztása futási időben történik. Ez a tervezési módszer lehetővé teszi, hogy az elosztott rendszerben a modell a technológiát követve áthelyeződjön.

Jelen cikkben ismertetjük a tervezési módszer alapgondolatát és az alkalmazása során gyűjtött tapasztalatainkat.

Az MVC paradigma

Napjaink program-rendszereivel szemben támasztott alapvető követelmény olyan felhasználói felület megléte, amely képes a működés könnyen áttekinthető megjelenítésére és a beavatkozások hatékony kezelésére. A felületek szabványosított, ismételten hasznosítható elemekből történő felépítése szükségessé teszi a felhasználói felület és a program egyéb részeinek különválasztását. Erre a célra fejlesztették ki az MVC paradigma szerinti architektúrákat.

Az MVC paradigma egy megközelítése a felhasználói felület és a program funkcionalitása közötti szabványosított kapcsolatnak.

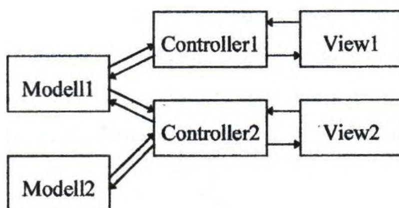
Az MVC architektúra három komponense:

1. modell — A modell része az alkalmazásnak, amely a valós világot reprezentálja. Olyan objektumok gyűjteménye, amely a problématér elemeit képezi le. A modell független a felhasználói felülettől.

2. view — A view vagy nézet a modellnek egy a felhasználó számára látható képe.

3. controller Megteremti a kapcsolatot a view és a model között. Lehetővé teszi a modell változásainak konzisztens megjelenítését, valamint a felhasználó beavatkozásainak érvényrejutását.

A controller és a view közötti összerendelés egy-egy típusú, azaz minden egyes megjelenő nézethez kapcsolódik egy kontroller objektum. A modellek és a controllerek között több-több kapcsolat is fennállhat, azaz egy modell objektumnak több megjelenése lehet, több helyről lehet működésébe beavatkozni, de egy megjelenítő egység akár több modell állapotát is tükrözheti.

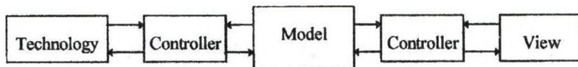


1. ábra MVC objektumok kapcsolódása

Az MVC modell kiterjesztése

Mint említettük, az MVC filozófiát elsődlegesen az emberi felhasználói felületek (user interface) kialakítására hozták létre. Az elvet sikerrel alkalmazhatjuk abban az esetben is, ha a modell viselkedését nem csak felhasználói felület, hanem tetszőleges folyamat-perifériák befolyásolják. Ez esetben a view és a controller logikai szerepét megtartjuk, de az őket realizáló speciális objektumok a folyamat-perifériákon keresztül a technológiával tartanak kapcsolatot. Lényegében a hagyományos ernyőképet lecseréljük a technológiának szóló üzenetekre, illetve a billentyűzetet és egeret helyettesítjük a technológiától származó jelekkel.

Összességében mind a felhasználó, mind a technológia oldaláról a modellt módosítjuk, majd a módosítások a controllerek és view-k működésének megfelelően megjelennek. A 2. ábrán látható módon épül fel a felhasználó és a technológia között a kapcsolat.

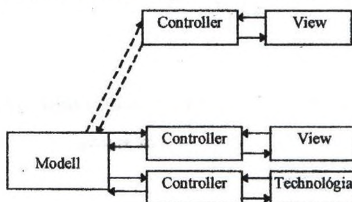


2. ábra MVC kiterjesztése

Elosztott megvalósítás

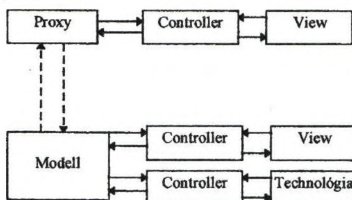
Elosztott folyamatirányító rendszerek esetében a következő két problémával állunk szemben. Egyrészről a technológia elosztottsága következtében annak modellje is elosztott, azaz a technológia egyes elemeinek modelljét - célszerűségi okokból - az elosztott rendszernek a technológiához legközelebb eső egységén képezzük. Ez a megközelítés teszi lehetővé, hogy az elosztott rendszer degradálódása esetén a technológia helyileg kezelhető marad. A technológia és modellje között egy-egy kapcsolat van.

Másrészről egy technológiai elem modelljét az elosztott rendszer több különböző egységén is megjeleníteni és a beavatkozást lehetővé tenni kívánjuk. Ez azt jelenti, hogy egy modellhez távoli helyekről is view és controller objektumokat akarunk kapcsolni.



3. ábra A VC egy kapcsolódási módja

Ez az elgondolás megfelel az MVC paradigmának, azonban probléma a modell és a távoli kontroller közötti kapcsolat realizálása, ugyanis a közöttük levő kommunikációs csatorna igénybevételét lehetőség szerint minimalizálni kell. Ezen feladat egy lehetséges megoldása a modellnek a megjelenítés szempontjából lényeges részének kihelyezése a távoli egységre. Ezt a speciális tulajdonságokkal rendelkező másolatot proxynak nevezzük.



4. ábra A proxy helye az elosztott rendszerben

A proxy objektum bár sok funkciójában megegyezik a modellel, hiszen ugyanannak a technológiának a megtestesítője, működésükben azonban eltérnek egymástól.

Ezen okok miatt a tervezésnél a következő alapelveket kell figyelembe venni:

Nem használhatjuk az eredeti modell objektumot a távoli gépen, hiszen a technológia itt nem áll rendelkezésre. A technológiai információk közvetlenül nem jutnak el a proxyhoz, hanem a modell dolgozza fel azokat és a megjelenítéshez szükséges információkat adja csak át a proxynak. A proxy tehát nem modellez, hanem képviseli a modellt a távoli gépen.

Nem tudunk közvetlenül beavatkozni a technológia felé. A proxyn keresztül kell üzennünk a modellnek, aki kezdeményezi a tényleges beavatkozást.

Nem számíthatunk a kommunikációs csatorna hibátlan működésére, következésképp a proxynál jelezni kell tudnunk a modellel való kapcsolat meglétét vagy lebomlását, azaz a modell és a proxy közötti konzisztenciát.

Lehetővé kell tennünk, hogy egyetlen modellhez több proxy is kapcsolódhasson, ami magában foglalja az egyidejű, párhuzamos beavatkozások helyes kezelését.

A modell, proxy fejlesztése

A tervezés és megvalósításra három módszer vehető figyelembe:

- Teljesen független fejlesztés, amikor a modell és a proxy objektumokat egymástól elválasztva fejlesztjük. Ez a megoldás jelentősen csökkenti a fejlesztés hatékonyságát és az elkészült megvalósítás javíthatóságát, karbantarthatóságát.
- Közös ősből történő öröklés, amikor a modell és a proxy objektumok egy közös ősnek a leszármazottjai. Ez a megoldás hatékonyabb megvalósítást tesz lehetővé a független fejlesztésnél, de a modell, proxy tulajdonságból fakadó igényt, miszerint azonos funkciók szerepelnek mindkét objekumban, melyeknek csak bizonyos kritikus szakaszai térnek el, nehezen teszi megvalósíthatóvá.
- Azonos kód, elválás csak a kritikus részeken. Ebben az esetben a modell és a proxy kódja ugyanaz, csak a kritikus helyeken, ahol különböznek a szerepek, vizsgáljuk valódi mivoltát az objektumnak. Ez a megoldás teszi lehetővé a leghatékonyabb megvalósítást.

Rendszerünk megvalósítása során a modell és a proxy együttes fejlesztését választottuk. Megvizsgáltuk az objektumok modell és a proxy specifikus tulajdonságait, s ennek alapján megállapítottuk, hogy az azonos kód módszere optimalizálja a fejlesztési, karbantartási munkát és a konfigurációs menedzsmentet.

A kritikus szakaszok szétválasztásának megvalósítására az IsProxy() függvényt dolgoztuk ki, amely a rendszer konfigurációjának megfelelően képes meghatározni a modell illetve proxy szerepét az objektumnak.

Az IsProxy statikus kiérékelődését feltételezve, egy objektumról a rendszer indulásakor eldől, hogy ő modellként vagy proxyként működik a rendszerben. Felkészíthetjük rendszerünket az IsProxy függvény dinamikus kiértékelésére, amikor is minden egyes kritikus szakasznál a rendszer aktuális konfigurációjának, állapotának függvényében döntünk a viselkedésről. Ez a módszer lehetővé teszi, hogy a modell, proxy szerep eldöntése futási időben történjen, így szerepük dinamikusan változzon, azaz olyan technológiai jellegű objektumok legyenek modellezhetőek, amelyek az elosztott rendszerben változtatják helyzetüket, vándorolnak.

Összefoglalás

Cikkünkben tervezési módszert adtunk elosztott objektum-orientált folyamatirányító rendszerek megvalósítására. Bemutattuk az MVC paradigma kiterjesztését elosztott rendszerekre és a modell, proxy fejlesztés hatékony módját.

Az itt megadott metodológia helyességét és gyakorlati alkalmazhatóságát támasztja alá, hogy segítségével jelenleg egy országos méretű rendszert valósítunk meg.

Két alapvető irányban tervezzük a továbbfejlesztést: a modell-proxy viszony szemantikájának formális megfogalmazása és az IsProxy dinamikus működésének kísérleti vizsgálata.

MediNet: elosztott, integrált egészségügyi információs rendszer

Kovács András, Nick János
HiCare Kft.

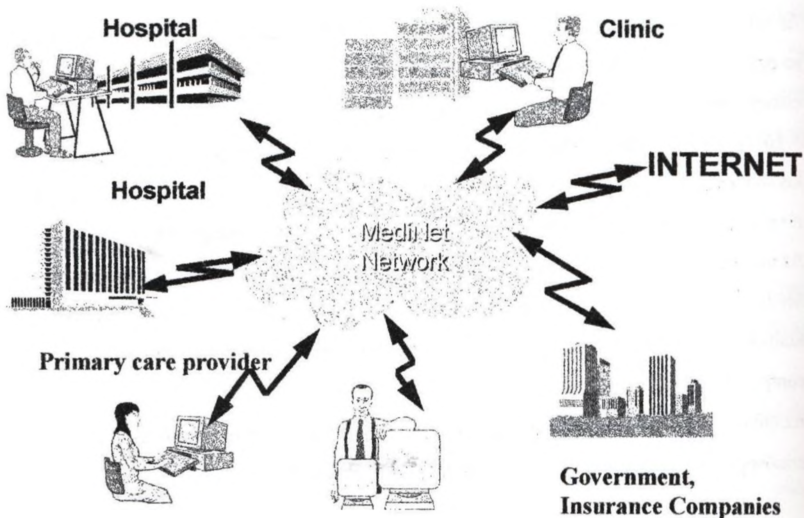
Bevezetés

Az informatika hatákony alkalmazása ha nem is gyógyulást, de lényeges javulást ígér az egészségügy két akut probléjára, a kiadások drasztikus növekedésére és a sok estben nem megfelelõ minõségû szakmai munkára.
A HiCare Kft-ben folyó MediNet projekt olyan integrált egészségügyi rendszer kialakítását tüzte célul maga elé, amely IP alapú hálózaton integrálja az egészségügy résztvevõit: háziorvosokat, rendelõintézeteket, kórházakat és felügyeleti szerveket.
Az elosztott rendszer fejlesztése korszerû objektum orientált technológiával folyik. A MediNet a legújabb komponens alapú kliens-kiszolgáló technológiára, a CORBA szabvány szerinti elosztott objektum technológiára épül (distributed object computing) [1].
Az elõadás a projekt célkitûzéseit és az alkalmazott technológiát ismerteti. A minõségbiztosítás egyes kiemelt elemeivel [2] foglakozik.

A MediNet project

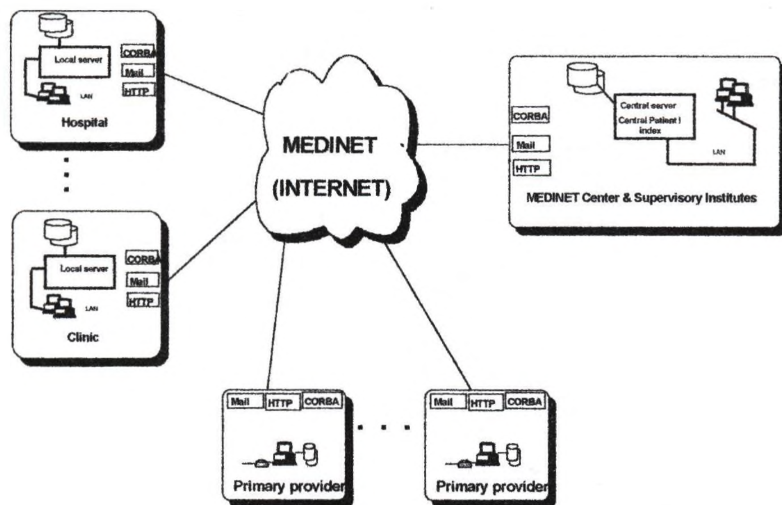
A MediNet projektnek egy olyan CORBA alapú integrált egészségügyi rendszer elemeinek a kifejlesztése a célja, amely IP alapú hálózaton összekapcsolja az egészségügy résztvevõit.
Az eddig informatikai szempontból elszigetelt résztvevõk - szigorú információ elérési és továbbítási mechanizmusok mellett - hozzzáférhetnek a páciensekrõl rendelkezésre álló klinikai információkhoz, amely növeli a szakmai munka minõségét (pl. a teljes kórtörténet összeállítható), javítja a kisebb kórházakban ápoltak esélyegyenlõségét (pl. szakértõi központok állíthatóak fel és szakértõi véleményezésre a leletek átküldhetõek) továbbá a szakmai hatákonyság javítása mellett csökkenthetõk a költségek (pl. a szakértõ nem utazik, a korábban, más szolgáltatónál készült diagnosztikai vizsgálatokat nem kell megismételni, stb). További, de nem az

elosztott-együttmûködõ jellegbõl adódó elõny, hogy a gazdasági és szakmai modulok integrálása a gazdálkodás hatákonyságát jelentõsen növeli.
A rendszer részét képezi egy központi (és többszörözött) információs katalógus "Master Patient Index", amely nyilvántartja, hogy az egyes páciensekrõl az integrált rendszerben hol találhatóak információk, pl. melyik kórházban, háziorvosnál stb.



1.ábra A MediNet rendszer

A MediNet rendszer egyik alapeleme a multimédia alapú elektronikus páciens rekord , amely azt jelenti, hogy páciensrõl keletkezett, bármilyen megjelenési formájú információt, legyen az konvencionális struktúrált adat, kép, hang, vagy video, a rendszer tárolja, és bármikor visszanyerhetõek, tanulmányozhatóak. Az keletkezett információ alapvetõen a keletkezés helyén tárolódik, ha máshol válik szükségessé, akkor a hálózaton azonnal elérhetõ, vagy késõbbi - pl. éjszakai áttöltésre elõjegyezhetõ.
A MediNet különbözõ tipusú információ elérési/továbbítási mechnizmust biztosít felhasználóinak: e-mail , WEB és CORBA. (lásd a 2. ábrát).
A MediNet ki fogja elégíteni mind az európai (CEN 251, amely hazánkban is bevezetésre kerülhet) mind az amerikai egészségügyi informatikai szabványokat (HL7, DICOM, stb).



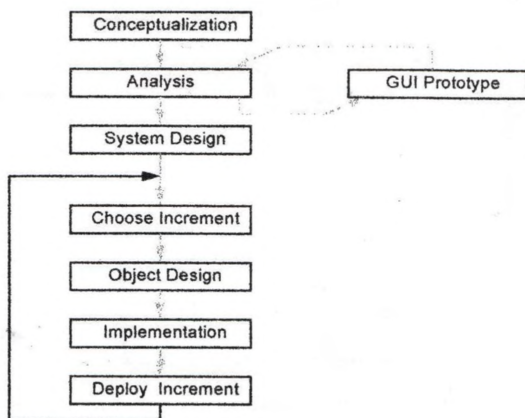
2. ábra MediNet kommunikációs lehetõségei

A biztonságos adatelérés és továbbítás, azaz az illetéktelen hozzáférés megakadályozása és a beteg személyiségi jogainak biztosítása alapvetõ feladatok. A MediNet rendszerben minden tipusú kommunikációs mechanizmus a HiCare DigiKey [3], PGP alapú security rendszerével lesz védve. A DigiKey beépítésre kerül a HTTP, az e-mail és a CORBA kommunikációs mechanizmusokba.

2. Az alkalmazott technológia

A MediNet rendszer teljes mértékben objektum orientált technológiával fejlesztett projekt. Az OO technológia lehetõvé teszi iteratív és növekményes - 3. ábra - fejlesztési életciklus alkalmazását.
Ennek megfelelõen az egyes alrendszerek fejlesztése egymással különbözõ fázisban lévõ párhuzamos ágakban történik. Az osztályok és interfészeik definiálása áll az analízis ésa tervezés középpontjában. Az osztályok végigvonalnak a teljes fejlesztési életcikluson. Az OO technológia alkalmazásával megszûnik az módszertani, technológiai szakadék a fejlesztési életciklus különbözõ fázisai - analízis, tervezés, stb - között, amely korábban a struktúrált módszertant és technológiát jellemezte.

A projekt programozási nyelve, a legelterjedtebb szabványos programozási nyelv, a C++., melynek használata biztosítja a hatékony kódot, a portábilitást. A C++ standard



3. ábra. Fejlesztési életciklus

jellege miatt a hatékony minõségbiztosításhoz, azaz a minõségi szoftver elõállításához elngedhetetlen tesztelõ, kódellenõrzõ és mérõszámgeneráló programok a piacon széles választékban állnak rendelkezésre.

Adatbáziskezelõként a nagyteljesítményû, piacvezetõ ODBMS-t, az ObjectStore (Object Design Inc) használjuk. Az ObjectStore a leghatényabb C++ ODBMS, amely C++ nyelvi környezettel szorosan, de traszparens módon van integrálva.

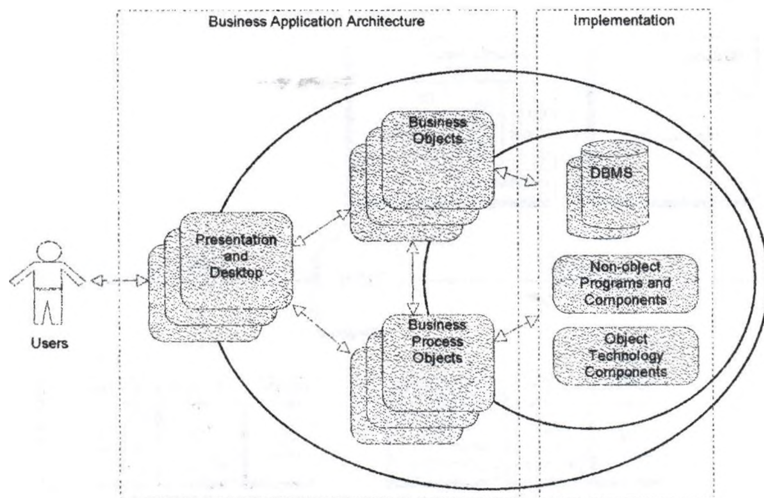
Az elosztott objektum komponensek az IONA Technologies Orbix CORBA kompatibilis Object Request Broker (ORB) technológiájával kommunikálnak egymással. Az alkalmazott programozási nyelvnek megfelelõen az Orbix C++ változatát használjuk. A már hozzáférhetõ CORBA szolgáltatások[4] - pl, Naming, Event, stb. - beépülnek rendszerbe. Továbbá elveiben - mert még szabványosítás alatt áll - követjük a CORBA Business Object Management arhitektúráját.

A WEB-en keresztüli elérés az ObjectStore-hoz kapcsolódó fejlesztõ/futtató környezet, az ObjectForms segítségével történik[5].

A MediNet a projektben igen fontos szerep jut az OO CASE és szoftver konfiguráció menedzsment eszközöknek. A Platinum Technologies Paradigm Plus CASE eszközével történik az OO analízis és tervezés OMT + Use Cases módszertanok alkalmazásásával. A CASE eszköz a repozitorijában tárolt információk alapján legenerálja az interfészek C++ kódját továbbá az ObjectStore C++ adatbázis séma definícióit és a CORBA IDL interfészt.

Szoftver konfiguráció menedzsment (SCM) nélkül, a MediNet meretû projektek kézben tartása már óriási erõfeszítéseket követelne; a modulok különbözõ verzióinak és különbözõ base-line-oknak a rendszerezett tárolása és nyilvántartása szinte lehetelen lenne. SCM nélkül minõségi szoftver fejlesztés nem képzelhetõ el [2].

A MediNet architektúrája a CORBA Business Object Management architektúráját követi (4. ábra)



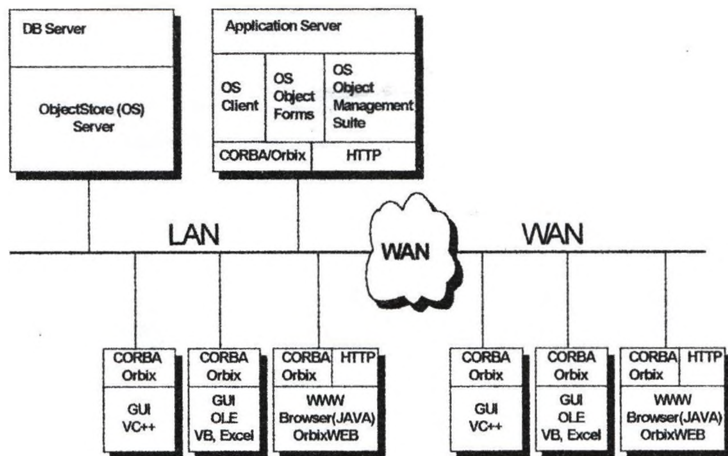
4. ábra CORBA Business Object Management Arhitecture

A kliensek Windows NT és Windows 95, a szerverek (alkalmazásszerverek) Windows NT platformokon futnak (a késöbbi verziókban UNIX gépeken is). Az általános a GUI fejlesztés a Microsoft MFC és az arra lapozott kiegészítõ osztálykönyvtárakkal MS Visaul C++ integrált fejlesztõ környezetben történik.

A MediNet általános kórházi és rendelőintézeti arhitektúráját és a felhasznált szoftver elemeket az 5.ábra vázolja.

Az alkalmazott Orbix ORB lehetővé teszi a CORBA és OLE/ActiveX komponensek közötti kommunikációt, ezért az OLE/ActiveX technológiát használó eszközök (Pl. Visual Basic, Delphi, Excel, stb) képesek lesznek elérni a Medinet komponens objektumokat (business objects). Könnyen írható VB-ben tetszőleges lekérdezést/feldolgozást megvalósító alkalmazás, pl: valamilyen demográfiai, járványügyi riportot kiértékelést készítő program.

Az OrbixWeb segítségével ugyancsak elérhetőek a CORBA objektumok JAVA környezetből. Ez lehetőséget biztosít pl. olyan alkalmazások készítése számára, ahol a kliensek JAVA-ban vannal megírva és használják az alkalmazásszervereken futó CORBA objektumok szolgáltatásait.



5. ábra Medinet Arhitektúra

Összefoglalás

A MediNet egy olyan elosztott egészségügyi információs rendszer, amely a hálózaton keresztül integrálja az egészségügy résztvevõit a háziorvosoktól a kórházakig és felügyeleti szervekig.

A MediNet az aktuális valódi illetve de facto iparági számítástechnikai (CORBA, C++,...) illetve egészségügyi információs szabványoknak, ajánlásoknak (CEN TC251, HL7,...) megfelelõen készül.
Az alkalmazott szoftver technológia teljesen egységes, homogén, amely lehetõvé teszi az OO technológia kompromisszumok nélküli alkalmazását.

Az ObjecStore ODBMS korlátozások nélküli OO modellezést biztosít és nem lép fel az OO implementáció és az RDBMS-ben történõ objektum tárolás esetén jelentkezõ un. impedancia illesztetlenség, ami az objektumok táblákra történõ leképezésébõl származik. A leképezés illetve változások esetén ezek karbantartása idõigényes, számottevõ teljesítménycsökkentõ hatása van és sok hiba forrása.
A MediNet CORBA technológiára alapozása a közeli jövõben lehetõvé teszi majd, hogy a MediNet szabványos komponens alapú elosztott alkalmazás legyen, amely standard CORBA kompatibilis rendszer és alkalmazás keretredszer [1], valamint az egészségégügyi vertikális domain alkalmazási komponens objektumokbol (business object) épüljön fel, a CORBA/CORBAMed szabványnak megfelelõen.

IRODALOM

[1] Kovács A.: A kliens-kiszolgáló rendszerek új generációja: CORBA alapú elosztott rendszerek, l. Országos Objektumorientált Konferencia, 1996
[2] Nick J.: A szoftber konfiguráció menedzsment szerepe az OO szoftver fejlesztés minõségbiztosításában. I. Országos Objektumorientált Konferencia, 1996
[3] HiCare Security - DigiKey, URL:hchp.compuline.com
[4] Object Management Group: CORBAservices
[5] Nyilas I., Újfalussy G.:WWW objektumok kezelése: Web fejlesztés ObjectForms fejlesztõ eszközzel, I. Országos Objektumorientált Konferencia, 1996
[6] Kovács A, Nick J, Újfalussy G.: Az ObjecStore ODBMS szerepe a vállalati információs rendszerekben, l. Országos Objektumorientált Konferencia, 1996

Vasúti menetrendek újraütemezése szimulációs úton
I. OO input-adatbázisok
(Gáspár András, Harang BT, Budapest)

Az előadás témája a HARANG Bt-nek a MÁVTI Kft.-vel kötött szerződésében merült fel. A probléma ismertetése, a megoldás alapelveinek és alapfogalmainak tisztázása után, az előadás a szimulátort kiszolgáló OO adatbázisokra kíván koncentrálni. A szimulátort csakis annyiban érinti majd amennyiben az adatbázisok éppen a szimulátor kiszolgálását célozzák.

1. A probléma

A vasúti hálózat egyes részein jelentős kapacitásváltozást hoznak létre a következők:
- a fejlesztések vagy a visszafejlesztések,
- a felújítások hiánya (pl. lassújelek) vagy a felújítások alatti vágányzárak,
- az "események" (pl. nagyobb balesetek).

A kapacitásváltozás miatt a hatályos menetrendeket módosítani kell, azokat számítógéppel újra kell generálni és az erőforrások foglaltságáról és használatáról szóló eredményeket olyan output-adatbázisokba kell elhelyezni, melyek felhasználásával külön műveletekben listák, rajzok, animációk, elemzések, stb. készülhetnek.

2. A vasúti erőforrások

A vasúti hálózat erőforrásai általában kizárólagos használatúak. Kivételt képeznek a térközös közlekedésre alkalmas pályák illetve azon állomási vágányok, melyeken a foglalt vágányra járatás megengedett.

A vonatok erőforrásokkal történő "kiszolgálásának" rendjét, részben a jelzési utasításokat rögzítő F1-es utasítás, részben pedig a forgalmi szabályokat rögzítő F2-es utasítás adja meg. Az F2-es utasítás szerint a vonatokat csökkenő prioritási sorrendben kell kiszolgálni.

3. Az input- és az output-menetrend

Az új menetrendet output-menetrendnek nevezzük. Az új menetrend elkészítése során mindig egy input-menetrendből indulunk ki, amelyről tudjuk, hogy a vonatközlekedési terv, az időadatok, a csatlakozások, stb. formájában, forgalmi követelmény-rendszert hordoz.

Az input-menetrend azonban csak terv, vele szemben a vasúti szabályzatok elsőbbséget élveznek, így az input-menetrend legtöbbször végrehajthatatlan. A vasúti szabályzatok szigorú betartásával olyan kisebb-nagyobb késések jönnek létre, amelyek, ha eleve be lettek volna építve a menetrendbe, akkor a menetrend végrehajtható lenne. Feladatunk éppen ennek a végrehajtható menetrendnek az előállítása, melyet a továbbiakban output-menetrendnek nevezünk.

4. Kapacitás-szűkítés / bővítés

Kapacitás-szűkítés esetén az input-menetrend adott. Például vágányzár esetén, a vágányzár nélküli forgalom eredeti menetrendje. Az eredeti menetrend, túlterheli a vizsgált vasúti részhálózat, vágányzár alatt is rendelkezésre álló erőforrásait. Ha e menetrendet vágányzár esetén is végre akarnák hajtani, akkor a vonatforgalmat csak igen jelentős késésekkel lehetne lebonyolítani. Mind az eredeti (input) menetrend torzulásait, mind pedig a forgalmi döntéseket tartalmazó vágányzári (output) menetrend adatait, számítógéppel kell előállítani - forgalmi elemzésre és utas-tájékoztatásra előírt formátumokban.

pacitás-bővítéskor azonban az input-menetrendet is elő kell állítani. ldául egyvágányú állomásközök kétvágányúvá átépítésekor vagy nagy- bességű közlekedésre alkalmas pályák megépítésekor, a régi menetrend yes vonatai gyorsabban közlekednek, illetve a menetrend extra vona- kkal egészül ki.

. A vizsgált részhálózat

vizsgálatba, csak a vizsgálati célnak megfelelő, azon legkisebb rész- lózatot vonjuk be, amelyen kívül a kapacitásváltozás hatása már el- nyagolható. Ugyancsak részhálózatot vizsgálunk az áteresztőképesség gállapításához. Az utóbbira példa lehet a Budapest körüli körvasút hány állomását érintő vizsgálat.

. Perem-állomások

vizsgált részhálózat "szélső" állomásait, perem-állomásnak nevezzük. perem-állomásokon, egyes vonatokat, a valóságos közlekedésüktől elté- ő módon írunk le. A perem-állomások vonat-"források" illetve "nyelők". vizsgált részhálózatból "eltűnő" (távozó) vagy abban "feltűnő" belépő) vonatok vonatok számára, extra-műveleteket kell bevezetnünk:
"érkezés utáni eltűnés",
"indulás előtti feltűnés",
"áthaladás közbeni eltűnés", és
"áthaladás közbeni feltűnés".

. A "feltűnés" illetve "eltűnés" műveletek paraméterei

Nem elhanyagolható az az időtartam, ameddig az "eltűnő" vonat, a perem- állomásra megérkezése után is, fenntartja a vágány foglaltságát. Ugyan- csak nem hanyagolhatjuk el a "feltűnő" vonat indulás előtti vágány fog- lalását se. Ezen időtartamok tehát a műveletek paraméterei lesznek.

Egyvágányú állomásköz esetén, nem elhanyagolhatók se a követési, se az irányváltást is tartalmazó minimális idők. Követési időtartam például a két egymásutáni "eltűnés" közötti idő. Az "eltűnést" követő újabb"feltű nésig" eltelt idő, különösen nagy lehet, mivel figyelembe kell venni
- az "eltűnéstől", a nem vizsgált szomszéd állomásra érkezésig terjedő időtartamot, de
- a szomszéd állomástól a "feltűnésig" terjedő időtartamot is.

Így tehát ezen idők is a "f(eltűnés)" paraméterei lesznek.

Szükség lesz azonban a vonatpárokat egymáshoz rendelő paraméterre is. Ugyanis a vizsgált részhálózatból, a valóságban, mondjuk 6 órás késés- sel "eltűnő" nemzetközi express, nem sok időt nyerhet menetközben vagy külföldi végállomásán. Várható tehát, hogy párja, ellenirányú vonat- ként, más vonatszámmal, de ugyancsak kb. 6 órás késéssel lép majd be a vizsgált részhálózatba. A "feltűnő" vonat paramétereként tehát azt kell megadni, hogy mikor várható a vonat legkorábbi "feltűnése", egy adott másik vonat "eltűnése" után.

Itt kell megemlíteni azt a problémát is, ami abból ered, hogy a vonat- forgalom periódikus folyamat, 24 órás ciklussal. Emiatt a menetrendben korábbi órában "tűnhet fel" egy vonat, mint párjának "eltűnési" idő- pontja, mivel párja még az előző nap "tűnt el". Ilyenkor, a szimuláció során, késésmentes "feltűnést" feltételezünk, de a szimuláció eredmé- nyét csakis akkor fogadjuk el, ha a szimuláció visszaigazolja kiinduló feltevésünket, a késésmentes "eltűnés" lehetőségét is.

8. A vizsgált részhálózat kijelölése

A vizsgált állomások és a vizsgálatba bevont vonatok kijelölése a forgalomtervező felelőssége. A ténylegesen vizsgálandó állomások számára terhelést jelentő "teljes" vonatforgalmat le kell írni. Egyes állomások azonban főként perem-állomás szerepet töltenek be, vonatforgalmuk megadása csupán a ténylegesen vizsgálandó állomások terheléseinek aránylag pontos leírására szolgál.

9. Implementálási nyelv

Az output-menetrendet és az output-adatbázisokat szimulációs ütemezéssel állítjuk elő. A vasút OO valóságát - a szimulátor programján kívül - kb. 20 hatalmas méretű input-adatbázis írja le. Lásd az 1. ábrán. A szimulátor implementálási nyelve CA-Clipper 5.2, melyben OO felületet kellett implementálni a felhasználó számára. Az input-adatbázisok tartalmát a szimulátor igényeihez, azaz a szimulációs modellhez kell igazítani. A szimulátort kiszolgáló input-adatbázisok tartalmának meghatározásához elkerülhetetlen a szimulációs ütemezés OO jellegének, a szimulátor működési algoritmusának megértése.

10. A képzeletbeli terepasztal

A szimulátor egy képzeletbeli terepasztalon a vasúti szabályok szerint lejátssza a vonatközlekedést. A szimulátor tehát ismeri a vasúti szabályzatokat. Például a vonatok erőforrásokkal történő "kiszolgálásának" rendjét.

A képzeletbeli terepasztal a vasút valóságában elképzelhetetlen kísérletezést tesz lehetővé. "Kikapcsolhatjuk az áramot", azaz megállíthatjuk a vonatokat és az idő folyását. Sőt vissza is léphetünk az időben úgy, hogy a vonatokat "visszahúzzuk" egy korábbi időpontban elfoglalt poziciójukra. Ha ezen időpontban, a korábbitól eltérő döntést hozva, ismét "bekapcsoljuk az áramot", akkor megvizsgálhatjuk a vonatközlekedés alternatív lefolyását (lefutását).

11. A konfliktusok

Az erőforrás-allokáló aktív komponensek a vonatok, a tolatási műveletek és a vágányzárak. Prioritásukat a vasúti szabályok határozzák meg. A T1 időpontban történt allokáció a T2 időpontban (T2 >= T1) különböző konfliktusokat okozhat, nevezetesen a foglalt erőforrások miatt létrejött prioritási konfliktusokat illetve patt-helyzetet.

12. Az újrajátszás

Az algoritmus, a szó szoros értelmében BACKTRACK algoritmus, mivel a vonatokat konfliktus esetén "visszahúzza" a síneken, egy korábbi időpontban elfoglalt poziciójukra.

Konfliktus esetén visszalépünk az időben, éspedig arra az időpontra, amikor a konfliktust okozó allokációs döntés megszületett, más döntést hozunk, és lejátszuk az alternatív döntésen alapuló alternatív vonatközlekedést.

A szimulátor mindezt úgy valósítja meg, hogy elemzi a döntések adatbázisát, megállapítja a konfliktust okozó döntést, elkerülendőnek minősíti azt, és elrendeli a forgalom újrajátszását.

A kombinatorikai robbanás veszélye
=================================

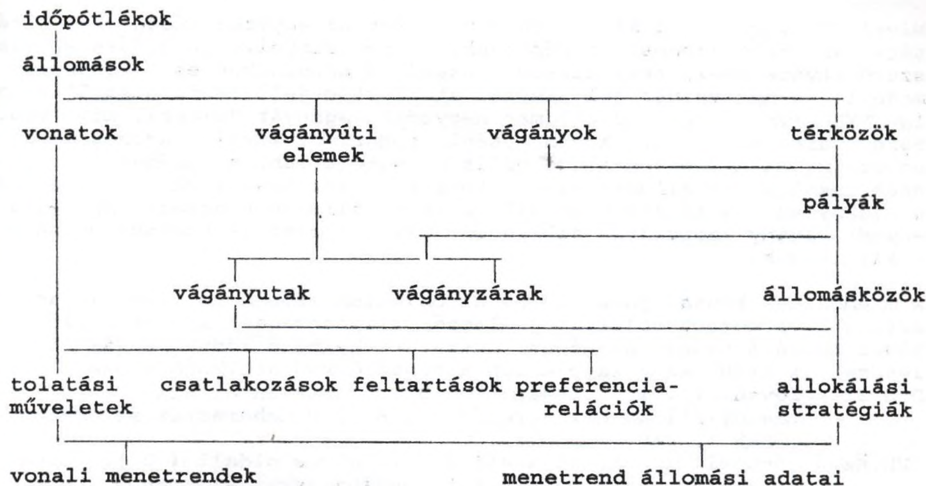
mulálni azért kell, mert optimum-kereső módszerrel kombinatorikai banás jönne létre. Szerencsére a forgalom-tervezőnek nincs szüksége imumra, csak megengedett megoldásra. Sőt, vasutas gondolkodással eladat követelményei továbbenyhíthetők.

ajdonképpen ezt szolgálja már az input-menetrend is, amelyet "kicsit érő kapacitás" mellett megvalósíthatónak tartunk. Éppen ezért, nem teljesen új output-menetrend előállítását tűzzük ki célul, hanem az ut-menetrend megvalósíthatóvá módosítását. A szimulátor úgy keresi a megvalósítható menetrendet, hogy a konfliktusban résztvevő legebb prioritású vonat számára megtiltja az újrajátszás során, a fliktust okozó allokációt, amivel az adott vonatot általában feltja, KÉSLELTETI.

ancsak vasutas gondolkodáson alapul a feladat részekre bontása is, módon, hogy először csak a legnagyobb prioritású folyamatok forrás-igényét ütemezzük, majd ezt megismételjük egyre csökkenő oritással.

Az input-adatbázisok
====================

forrásokat írnak le az állomások, a vágányok, a vágányúti elemek, a lyák és a vágányutak adatbázisai. Az említett adatbázisok megadják az forrás típusát, topológiai és geometriai adatait. Külön adatbázis ja elő a vonatok megállási- és indítási időpótlékait. További adatbázk tartalmazzák az input-menetrendet, a vonatközlekedési tervet.Külön atbázisban találhatók a vágányzár alatt kieső erőforrások, a tolatási veletek, továbbá a vágány- és pálya-allokálási illetve a tolatásemezési stratégiák. Lásd az 1. ábrán.



1. ábra: Adatbázis- és objektum-hierarchia

int már szó volt róla, az input-adatbázisok tartalmát a szimulátor gényeihez, azaz a szimulációs modellhez kell igazítani. E követelmény

sokszor egyszerű absztrakciót jelent. Például a szimulátor számára egyaránt "vágányúti elem" lehet a váltó, a sorompó vagy a "tiltott egyidejű menet". Néha azonban olyan új fogalmakat kell definiálni, amit a vasúti tudományok eddig nem ismertek. Esetleg azért, mert nem is szorosan vett vasúti fogalmak, hanem a vasúti hálózat vizsgálatának, rendszerelemzésének fogalmai. Példaként említhetjük erre a "perem-állomás" fogalmát.

A szimulációs modellt végülis különböző tudományok ismereteire alapozva kell kialakítani - gondosan ügyelve a kombinatorikai robbanás elkerülésére. E követelmény természetesen az adatbázisok fogalmaiban és tartalmában is megjelenik.

15. Objektum-tár

A vasúti objektumok megnevezésére objektum-azonosítókat használunk. Célul tűzzük ki azt, hogy a teljes vasúti hálózatról szóló információkat, fokozatosan kialakítandó, naprakész, objektum-tárakban őrizhessük, és azt, hogy a forgalomtervező, igényei szerint vehesse elő az objektum-tárból, az éppen vizsgálandó tetszőleges vasúti részhálózatot és perem-állomásait, továbbá az input-adatbázisokat. Ennek megoldására elengedhetetlen az, hogy minden vasúti objektumnak egyértelmű azonosítója legyen.

16. Beszélő kódok

Objektum-azonosítóként könnyen megjegyezhető vagy emlékezetbe vésheto (mnemonikus) kódokat célszerű alkalmazni. Példa lehet rá az állomások hívójelszerű 3 karakteres azonosítója: KEL = Bp.Kelenföld, FCS = Bp. Ferencváros, SUK = Bp.Keleti pu., DUH =Dunaharaszti; vagy a vágányok legfeljebb 5 karakteres (római) száma: III, IV.

Mivel IV. vágány sok állomáson van, ezért az egyértelműség biztosítására összetett azonosítót képezünk, a programnyelvi jelölések értelemszerű átvételével, nevezetesen a számítástechnikában és a rendszermodellezés nyelveiben jól ismert, attributum-jelölés felhasználásával. Így "KEL.IV", Kelenföld állomás negyedik vágányát jelenti, míg "FCS.IV" Ferencváros állomásét. A pont utáni azonosítót lokális azonosítónak nevezzük. Az állomásoknak globális azonotója van, a vágányoknak azonban csak lokális. Az állomás azonosítója az egész vasúti hálózaton egyedi, a vágány-azonosító azonban csak az adott állomáson egyedi. Ha mégis egyedi vágány-azonosítás szükséges, akkor összetett azonosítót kell alkalmaznunk.

A mnemonikus kódnál jobb az, ha az objektum-azonosító elárulja az általa hivatkozott objektum jellemző tulajdonságait is. Az ilyen kódot beszélő kódnak nevezzük. Összetett beszélő kódra példák az lesznek az alábbiak - zárójelben a beszélő kód árulkodó része:

DUH.B1 = fővonali (1), bejárati (B), páratlan oldali (1), vágány-allokálási-stratégia (B), Dunaharaszti állomáson (DUH), vagy

DUH.K2 = fővonali (2), kijárati (K), páros oldali (2), pálya-allokálási-stratégia (K), Dunaharaszti állomáson (DUH).

Minden objektum-típus azonosítója beszélő kód. Az egyszerű azonosítók 3 - 7 karakter hosszúak, maximális méretük függ az objektum típusától. Az egységes kezelés érdekében, objektum-azonosítóval látjuk el a vasúti objektumokon kívül, a szimulációt vezérlő adatrekordokat is.

Jelölési szabályok
==================
alakított jelölési szabályok többcélúak. Felfoghatók úgy is, mint (halk) javaslat a vasúti jelölések "OO-szabványosítására". Közei céljuk azonban sokkal prózaibb, nevezetesen az, hogy a szoftver etei (pl. a hibaüzenetek) külön magyarázat nélkül is megérthetők enek. A beszélő kódok nélkül azonban szinte megvalósíthatatlan e a szimulátor öndokumentáló nyomkövetése és állapotjelentése, a ulációs pillanatképek elkészítése. Súlyos gondot jelentene a helyyos rajzok feliratozása és az animáció is. Gondoljunk csak a felira"áthatásának" problémájára, amely még a rövid beszélő kódok alkalása esetén se kerülhető meg.

újonnan kialakított jelölési konvenciók egy része, a vasútnál régóta leskörűen alkalmazott beszélő kódok magától értetődő általánosítása. dául a páros számú váltók az állomás kezdőpont felöli oldalára esnek, áratlan számúak pedig a végpont felé. A páros számú vonatok a kezdőt felöl a végpont felé tartanak, míg a páratlanok éppen fordítva. A út - a fővonalakat összekötő szárnyvonalak kezdő és a végpontjának adása után - még a szárnyvonalakon is, betartja a paritás-helyes mozást.

Pályaszámozás
==============
szimulációhoz kialakított modell azonban a paritás-helyes számozást pályák számozására is kiterjeszti. Egy-egy adott állomásról kivezető gy oda bevezető összes pályát, úgy kell beszámozni, hogy egy adott goritmus szerint a pályákat be kell járni.

számozási algoritmus úgy van definiálva, hogy a kezdőpont felé eső lyák páros, a végpont felé eső pályák páratlan számot kapjanak, sőt pálya száma, fővonali, kétvágányú állomásköz esetén, a pálya alapvető emmódját is elárulja.

pályák beszámozását követően, könnyen képezhetők beszélő kódok a nat-vágányutakra is. Például U2III, az adott állomás 2-es pályáját, a II. vágánnyal összekötő vágányút azonosítója. A vágányút azonosítója, adott állomáson belül, egyértelműen azonosít.

pályaszámozás az állomásra "lokális". A pályák a vasúti hálózaton gyértelmű azonosítójukat mindig a pálya kezdőpont felöli végén lévő llomáson nyerik el, az állomás-azonosító és a pályaszám kombinációjáól. Az állomásköz egész hálózaton egyértelmű azonosítóját pedig a névadó" pálya azonosítójából örökli.

9. Egy objektum-egy képernyő
============================
képernyős bevitel során minden adat megadását azonnali ellenőrzés követi, hogy a megadott adat definiált-e és elfogadható-e. Ezért az adatbázisokat csak meghatározott sorrendben lehet feltölteni, az 1. ábra szerint felülről lefelé. Nem lehet megadni például a még definiálatlan állomás vágányait, vágányúti elemeit, allokálási stratégiáit, pályáit, stb.

```
Vágányutak
Az állomás neve:                       DUH
Az objektum azonosítója:               U2III
A pálya azonosítója:                   SOR3          Pályakód: 2
Az állomási fővágány azonosítója:      III
Mely vágányúti elemek tartoznak a vágányútba?
( Felsorolás a kijáró vonat útvonala mentén ! )
E8    E6    E4    E2

Mely vágányúti elemek maradnak fogva a bejárat után ?
( Felsorolás a bejáró vonat útvonala mentén )

Bejárat képzési időtartama [ perc ] ?          3.00
Kijárat képzési időtartama [ perc ] ?          3.00
```

2. ábra Egy vágányút-objektum tartalma a képernyőn

Az előírt sorrendű bevitel miatt, a más objektumokra történő hivatkozás mindig kiválasztás egy már előzőleg definiált készletből. Ha például a 2. ábra szerint egy új vágányutat akarok definiálni, akkor a szoftver megkérdezi az állomás nevét, de úgy, hogy a képernyőn megjeleníti az összes ismert állomás azonosítóját és abból kell választani. Ezt követően látszólag szabadon adhatom meg a vágányút azonosítóját, azonban a beszélő kód azonnal szintaktikai és szemantikai ellenőrzésre kerül. Például U2III megadása esetén, hibajelzést kapunk, ha nem definiált a kiválasztott állomás 2-es pályája és III. vágánya. A felsorolásokban is csak, a már definiált objektumokból választható ki a soronkövetkező

20. Az adatfordító

A szimulátort kiszolgáló adatbázisok hibamentességének ellenőrzése az adatfordító feladata. Az adatfordító - a szimuláció végrehajthatósága érdekében - teljeskörű szintaktikai és szemantikai ellenőrzést végez a szoftver input-adatbázisain. A lehetséges hibaüzenetek száma 100 fölött van.

21. Más célú vasúti-adatbázisok átvétele és kiegészítése

A nagyméretű input-adatbázisok feltöltése hatalmas feladat. Ha létezik felhasználásra alkalmas, számítógépes adathordozón tárolt, vasúti adatbázis, úgy azt át kell venni. Átvételre alkalmas adatbázisok például a szolgálati helyeket és a szolgálati menetrendet tartalmazó adatbázisok. De természetesen ezek az adatbázisok se tartalmazzák a tervezett új állomásokat és a tervezett új vonatokat. Sőt a régi állomások egyedi azonosítóját se. Az átvételt megvalósító szoftver kiszűri, összevonja, kijavítja és kiegészíti a felhasználásra alkalmas adatokat.

A vasút egyes adatbázisai nincsenek számítógépes adathordozón rögzítve. A rögzített adatbázisok viszont a szimulációs modell szempontjából hiányosak.

23. Makró-nyelv

A vasúti adatbázisok átvétele után, az adatfordító általában nagytömegű hibát jelez, mivel az átvett adatbázisok alapján csak hiányos objektumokat lehet előállítani.

vonat-objektumok például nem tartalmazzák se a bejárati vágány-
lokálási stratégiát, se a kijárati pálya-allokálási stratégiát, se az
kezések utáni tolatási műveleteket, se az indulások előtti tolatási
veleteket. A vonat-objektumok kiegészítése előtt létre kellene hozni
megfelelő, stratégiákat illetve tolatási műveleteket leíró objektu-
kat is.

egy fővonal vizsgálatában 150-250 vonat szerepel, akkor az átvett
nat-adatbázis tolatásokkal kiegészítése érdekében legalább 300-500,
, tolatási műveletet leíró objektumot kellene létrehozni és legalább
0-250 vonat-objektumot kellene kiegészíteni.

vonat-objektumok kiegészítése megtörténhetne képernyős adatgyűjtéssel
- egyesével, objektumonként. Az adatmegadás hibáinak elkerülésére
onban elengedhetetlen annak biztosítása, hogy a forgalomtervező
yidejűleg lássa a képernyőn a hivatkozott stratégiát, a hivatkozott
latási műveletet és a hivatkozó vonat-objektumot. Windows-ban elbe-
élve, arra lenne szükségünk, hogy a képernyőn nyitott ablakunk legyen
yszerre több adatbázisra és egyetlen billentyű lenyomásával tehessük
ithatóvá az azonosító segítségével referált objektum tartalmát. Az
talunk használt Clipper azonban nem támogatja az ablakkezelést (az
lakváltást), ráadásul a szoftvert is az egy objektum - egy képernyő
ndben implementáltuk.

isfelől viszont célszerű biztosítani az alapadatok feltöltésének egy-
erű megismételhetőségét, a kismértékben módosított adatokkal megismét-
sre kerülő szimuláció számára. Nem lehet tehát megelégedni az egymás-
z igen hasonló, nagyszámú képernyős művelettel. Se az egy objektum-
y képernyős adatgyűjtéssel, se a Windows-szerű több ablakos adatgyűj-
ssel, az adatbázis-ablakok közötti egyszerű átkapcsolással. Helyette
lszerűbb a vasutasok szaktudásának és döntéseinek file-okon rögzítése,
forgalomtervező fogalmaihoz közelálló adatleíró célnyelven. Olyan
élnyelven, mely lényegében makró-definiciókból és makró-hívásokból áll.
zután az átvett adatbázisok file-okról történő kiegészítését már
lipper-menűről vezérelhetjük.

4. A SIMULATION osztály beágyazása a Clipperbe

vasúti szimulátort célszerű lenne SIMULA nyelven implementálni fel-
éve, hogy az IBM PC-n elérhető lenne egy megfelelő SIMULA-mplementáció,
ely

- elegendően gyors és
- direkt hozzáférést biztosít az operációs rendszerhez továbbá a Clipper-adatbáziskezelőhöz, és
- megfelelően definiált továbbá hatékony CHECKPOINT/RESTART lehetőséggel van felszerelve.

inthogy ilyen implementáció nem létezik, a vasúti szimulátor megvaló-
ítása során választani kell [4] a következők közül:

- a SIMULA-implementációk továbbfejlesztése, vagy
- a Clipper-be beágyazott SIMULATION osztály között.

korábban vázolt ütemezési algoritmus szerint, a kombinatorikai rob-
anás veszélyét súlyosbítja az, hogy a SIMULA nyelv az irreverzibilis
dőkezelést támogatja, ezért CHECKPOINT/RESTART hiányában, az egész
olyamatot mindig újra kellene szimulálni az éjféli 0 óra 0 perctől
ezdve.

A [4] előadás kifejti a SIMULA nyelv SIMULATION osztálya Clipperbe ágyazásának alapelveit és nehézségeit. A mérési eredmények alapján az a legjobb, ha minden SIMULA-osztályhoz egy-egy Winchesteren tárolt adatbázist rendelünk, melynek rekordjai az objektumok adatai. Az egyszintű kvázi-párhuzamos-rendszer (QPS) könnyen megoldható monitor-technikával, továbbá a RETURN és a CASE-utasítással, valamint a célnak megfelelően kialakított Clipper eljárásokkal. Az időtengely (SQS) az események ütemezett időpontja (EVTIME attributuma) szerint indexelt adatbázis lesz. Összetett kulccsal megoldható gondot jelent azonban az azonos időpontra ütemezett események "fürtje".

Hivatkozások:
=============

[1] Gáspár A.-Csáki P.-Visontay Gy.:
A szimulációs módszer és a SIMULA 67 nyelv.
NJSzT Rendszerelméleti Konferencia'79,Sopron.
Rendszerek szimulációja,3-36.o.

[2] Gáspár A.:
A SIMULA 67 és a számítástudomány.
Neumann Kongresszus,1979.

[3] Gáspár A.:
A SIMULA 67 szimulációs bázisnyelv alkalmazásairól.
ALKALMAZÁS'89. Az NJSzT IV. Országos Kongresszusa.
Számítástudományi eredmények szekció. III.kötet 199-208.o.

[4] A.Gáspár: Embedding SIMULATION to Clipper.
22nd Conference of ASU on "Object Oriented Modelling+Simulation"
Clermont-Ferrand, Franciaország, 1996.

Köszönettel tartozom Csáki Péternek lektori munkájáért.

TARTALOM

Öröklődés és konkurencia az objektum orientált programozásban
Blum László (Veszprémi Egyetem), dr. Kozma László (ELTE) 3

Common Object Request Broker Architecture (CORBA)
Molnár István (BME), Moskovits Péter (BME) 11

A kliens kiszolgáló rendszerek új generációja: a CORBA alapú elosztott rendszerek
Kovács András (Hi Care Kft.) 16

Objektum-orientált és strukturált módszertanok összehasonlítása
Hontvári József, Frigó József (TRIÁD Kft) 24

Objektum-orientált módszertanok kapcsolódása
Frigó József, Hontvári József (TRIÁD Kft) 30

Objektum-orientált tervezés alkalmazása rugalmas gyártórendszereknél
Sándorné Kmecs Ildikó (MTA SZTAKI) 32

A szoftver konfiguráció menedzsment szerepe az objektum-orientált szoftverfejlesztés minőségbiztosításában
Kovács András, Nick János (Hi Care Kft) 38

Az ODMG - 93 szabvány
dr. Juhász István (KLTE) 44

Objektum-Orientált adatbáziskezelő rendszerek
Moskovits Péter (BME) 51

Objektum-orientált adatbázisok logika alapú megközelítése
Hajas Csilla (KLTE) 59

A világ vezető objektum-orientált adatbáziskezelője: az ObjectStore
Ertner Péter (IQSoft) 65

Az ObjectStore ODBMS szerepe a vállalati szintű vállalati információs rendszerekben
Kovács András, Újfalussyné Nagy Gyöngyvér, Nick János (Hi Care Kft.) 67

Relációs adatbázisok elérése objektum-orientált módon: az ObjectConnect termékcsalád
Takáts Tamás (AXIS Kft.) 74

Alkalmazás fejlesztői eszköz a nagyobb hatékonyságért
Mihaletzky Géza (NEXT Software Kft). 78

Objektum-orientált programfejlesztés a Powersoft legújabb eszközeivel
Pokó István (AXIS Kft). 86

Integrált vállalatirányítási szoftvercsomag testreszabása CORBA alapú ObjectBroker segítségével
László István (DIGITAL Magyarország Kft) 88

Kódgenerálás az ObjectTeam CASE eszközzel
dr. Balogh Kálmán (Informix Technology Center Hungary Kft). 96

WWW objektumok kezelése: Web fejlesztés ObjectForms fejlesztő eszközzel
Nyilas István, Újfalussyné Nagy Gyöngyvér (Hi-Care Kft.) 98

A Software thrugh Pictures, mint objektum-orientált CASE eszköz
Frigó József, Kelen András, Hontvári József (TRIÁD Kft) 105

Vezetői Információs Rendszerek fejlesztése objektum-orientált módszerrel
Fejér Gábor (SAS Institute) 108

Fully Integrated Banktechnikal Information System megvalósítása ORACLE fejlesztő eszközön, különös tekintettel a SERVER / REZIDENS / KLIENS architektúra alkalmazására.
Róna György, Egervári Zoltán (R&E Systems) 113

A procedurális paradigmától az eseményvezérelt szemléletmódig - COBOL nyelven
Huba Zoltán (HUNGÁRIA Számítástechnikai Kft) 120

Objektum-orientált szimuláció Javaban
Simon Géza (BME) 128

A C++ nyelv lehetőségei és korlátai
Vég Csaba (KLTE) 135

Rendszerelemzés és jogtudomány, avagy alkalmas-e a magyar jogrendszer arra, hogy...
Gáspár András (Harang Bt.) 143

IQÜMT: Üzleti Megoldások Technológiája
Németh Miklós (IQSOFT) 153

IQBCL SQLWindows Class Library és prototípus generátor
Horváth Attila (4D SOFT Bt) 167

Objektum-orientált folyamat vizualizáció
Fóris Tibor, Szirmay- Kalos László, Márton Gábor (BME) 175

SOM/DSOM
Nyikes Tamás (IBM) 183

Elosztott objektum-orientált rendszerek tervezése
Molnár István, BME (dr. László István) 193

MEDINET: elosztott, integrált egészségügyi információs rendszer
Kovács András, Hi Care Kft (Nick János Hi Care Kft 199

Vasúti menetrendek újra ütemezése szimulációs úton - I. Objektum-orientált input adatbázisok
Gáspár András, Harang Bt 206